体验技术专用版

嗨翻

BLOGE

中文版

面向 *BLOGE* 编排引擎的循序渐进学习路线

BLOGE Project

Version 0.9.8-RC1

目录

学习阶段 阶段介绍 5

第一阶段——看见工作流

Graph 形状、依赖与数据流

目标：先看见责任与依赖，再增加控制和规模。

第 1 章——BLOGE 是什么，以及它为什么存在 6
第 2 章——你的第一个 Graph 16
第 3 章——用依赖关系思考 26
第 4 章——流动的数据 34
第一阶段回顾——入门（第 1–4 章） 44

学习阶段 阶段介绍 46

第二阶段——可维护编写

分支、韧性、Operator 边界、DSL 与评审反馈

目标：把可读流程变成另一位工程师能评审的改动。

第 5 章——做决策的分支 47
第 6 章——韧性设计 61
第 7 章——设计良好的 Operator 73
第 8 章——把个人 DSL 草稿变成团队资产 87
第 9 章——工具链工作流 94
第二阶段回顾——可维护编写（第 5–9 章） 107

学习阶段 阶段介绍 110

第三阶段——组合与时间

子图、迭代、挂起与崩溃恢复

目标：让工作跨集合、外部时间与进程重启继续成立。

第 10 章——用子图复用 111
第 11 章——批处理与迭代 122
第 12 章——等待世界的回应 139
第 13 章——持久执行 147
第三阶段回顾——组合与时间（第 10–13 章） 162

学习阶段 阶段介绍 164

第四阶段——生命周期模型

Session、状态机、嵌套所有权与受限 Agent

目标：让每一个长生命周期决策只有一个明确 *owner*。

第 14 章——多轮 Session 165
第 15 章——状态机 177
第 16 章——组合 Session 与状态机 188
第 17 章——把非确定性 Agent 放进确定性生命周期 198
第四阶段回顾——生命周期模型（第 14–17 章） 205

学习阶段 阶段介绍 207

第五阶段——技术信心

测试、可观测性、Spring 接线与容量边界

目标：让每个技术 *claim* 都匹配真正支持它的证据。

第 18 章——测试你的 Graph 208
第 19 章——生产环境中的可观测性 222
第 20 章——Spring 与生产环境接线 232
第 21 章——单 JVM 里的调度与复杂度 242
第 22 章——分布式运行与容量验证 247
第五阶段回顾——技术信心（第 18–22 章） 252

学习阶段 阶段介绍 254
第六阶段——业务承诺
受控 Scenario、Source-bound Evidence、治理与归因
目标：证明经批准的业务承诺，同时不夸大证据能力。

第 23 章——业务正确性不等于测试通过 255
第 24 章——第一条贷款业务正确性验证 260
第 25 章——Policy、Fixture 与 Assurance 267
第 26 章——Durable 与 Temporal 验证 273
第 27 章——合同敏感性：Suite 能拒绝错误答案吗? 279
第 28 章——有限属性验证与有用反例 283
第 29 章——从报告到 Source-bound Evidence 288
第 30 章——从 Case 事实到精确 Requirement Contribution 293
第 31 章——Project、Reactor 与 Claim Capability 297
第 32 章——变化归因、Maven 与发布决策 301
第六阶段回顾——业务证据 307

学习阶段 阶段介绍 309
第七阶段——选型与毕业项目
约束优先的选型与可运维毕业作品
目标：只在理由充分时使用 *BLOGE*，并交付别人能质疑的结果。

第 33 章——迁移与对比 310
第 34 章——实验 323
第七阶段回顾——毕业 336
附录 A ——内置函数 337
附录 B ——Session 速查表 346
附录 C ——Streaming 348
附录 D ——状态机 353
附录 E ——智能体扩展 360
附录 F ——决策表 371
附录 G ——DSL 工程参考 379
附录 H ——Durable 与规模化运行参考 381
附录 I ——Spring 生产参考 383

导读 {-}

《嗨翻 BLOGE》是一套围绕 BLOGE 编排引擎展开的循序渐进学习材料。它从 graph、node、依赖与数据流讲起，再逐步进入持久执行、可观测性与生产接线等主题，并 始终配合仓库中的真实示例来理解。

一句话理解 **BLOGE**： BLOGE 是一个 Java 优先的编排引擎：你把业务步骤描述成 graph，引擎负责执行顺序、并行、重试、超时、分支和恢复。

> *BLOGE* 由一个轻量核心加上可选扩展组成：`bloge-agent-ext`（*LLM* 驱动的 *agent* 编排与工具调度）、`bloge-session-ext` + `bloge-session-durable`（可崩溃恢复 的多轮 *session*），以及 `bloge-state-ext` + `bloge-state-durable`（可崩溃恢 复的状态机）。这些内容在后续章节中介绍。

三个关键特征

- 流程是可见的 ——工作流的形状是显式的，不再散落在各个 service 方法里。
- 执行由依赖驱动 ——你声明谁依赖谁，引擎判断谁现在可以运行。
- 韧性是内建的 ——重试、超时、降级、等待和持久进度都属于模型本身，而不是 手写胶水代码。

这本书适合谁

- 主要读者 ——初次接触编排引擎、希望理解 BLOGE 如何建模、执行和恢复业务 工作流的 Java 开发者。
- 次要读者 ——正在将 BLOGE 与 Temporal、Camunda、Spring Batch 或手写 编排方案做对比评估的开发者。

如何阅读这本书

1. 按顺序阅读。每一章都会建立在前一章的心智模型之上。
2. 亲自运行示例。文中的代码都来自仓库中的真实文件，建议打开、运行、修改并 观察结果。
3. 语法细节回到参考文档查。这一套教程教你的是如何思考；规范文档定义的是 什么语法合法。

七个学习阶段

- 第一阶段——入门：建立 graph、node、依赖和数据流的心智模型。
- 第二阶段——可维护编写：把 DSL 修改变成可读、可测、可评审的工程工作。
- 第三阶段——组合与时间：练习复用、迭代、挂起与崩溃恢复。
- 第四阶段——生命周期模型：建模 session、状态机、嵌套所有权与受限 agent 循环。
- 第五阶段——技术信心：让测试、观测、Spring 接线与容量证据匹配精确 claim。
- 第六阶段——业务承诺：用受控输入、source-bound evidence、治理与归因验证经业务负责人批准的合同。
- 第七阶段——选型与毕业项目：先判断 BLOGE 是否适合，再交付最小且有理由的毕业项目。

学习阶段

第一阶段——看见工作流

Graph 形状、依赖与数据流

目标：先看见责任与依赖，再增加控制和规模。

章节

01

第一阶段——看见工作流

第 1 章——BLOGE 是什么，以及它为什么存在

Graph 形状、依赖与数据流

目标：先看见责任与依赖，再增加控制和规模。

本章建立对 BLOGE 是什么、它为什么存在，以及 它最核心的结构特征 的整体认识。

1.1 学习目标

1. 用一句尽量少术语的话解释 BLOGE 是什么。
2. 说出让 BLOGE 区别于临时拼装编排代码的三个特性：可见、有结构、有韧性。
3. 从概念层面解释什么是 *graph* ——由业务步骤组成、数据流与控制流都显式可见的有向无环图。
4. 解释为什么并发、重试和失败传播应该由引擎负责，而不是由业务开发者手写在服务代码里。

1.2 前置条件

- 具备基础 Java 知识（不需要很资深）。
- 已安装可用的 JDK 25+ 与 Maven 3.6+ （参见入门指南）。

1.3 源示例

文件	展示内容
`order-process.bloge`	一个完整的订单处理教学示例：包含扇出、汇合、分支与韧性策略
`bff-dashboard.bloge`	一个 BFF 聚合图：五个并行拉取节点，带有超时、重试和降级

1.4 30 秒认识 BLOGE

从整体上看，BLOGE 是一个面向业务工作流的编排引擎。它并不取代业务代码本身，而是把“步骤之间如何依赖、哪些步骤可以并行、失败后如何处理”这类运行逻辑从零散实现中提取出来，组织成一个显式的 graph。

1.4.1 BLOGE 的定义

BLOGE 是一个 Java 优先的编排引擎。它把业务工作流表示成由 node 和 dependency 组成的 graph，并由 engine 负责执行顺序、并行、重试、超时、分支、等待和恢复，因此原本散落在 service 代码中的运行逻辑被收束为一套显式的结构。

BLOGE 从 DAG 图模型进一步扩展到更高级的编排模式：多轮 session（`bloge-session-ext` + `bloge-session-durable`）、显式状态机（`bloge-state-ext`

- `bloge-state-durable`），以及 LLM 驱动的 agent 循环与工具调度（`bloge-agent-ext`）。这些能力在后续章节中讲解——当前阶段，graph 模型是一 切的基础。

`BLOGE` 是 **Biz Logic Orchestration Graph Engine** 的缩写。这个名称本身已经给出了系统的基本边界：它面向业务逻辑，处理的是 编排 问题，以 *graph* 作为执行模型，并由 *engine* 负责运行。

在这个意义上，BLOGE 既不是单纯的 DSL，也不是单纯的流程图工具。它同时包含工作流模型与执行引擎；流程的形状、依赖关系和失败策略被声明出来之后，引擎按此完成调度、并发控制和韧性处理。

1.4.2 本书用来理解 BLOGE 的三个设计特征

本书使用三个设计特征比较 BLOGE 与临时拼装的编排代码。它们分别对应可读性、执行组织方式和运行期行为的声明方式，是帮助理解设计取向的框架，不是独立的性能或适用性结论。

1. 可见 ——流程本身以 graph 的形式直接呈现出来，业务路径、分支位置和上下游关系都可以在模型中看到，而不需要通过阅读多层 service 代码来还原。
2. 有结构 ——依赖关系属于工作流模型的一部分。哪些 node 可以并行、哪些 node 必须等待，并不由代码书写顺序决定，而是由 graph 中声明的依赖关系决定。
3. 有韧性 ——重试、超时、降级、等待和持久进度等运行期策略直接附着在工作流上，因此失败处理也成为模型的一部分，而不是散落在各处的临时胶水逻辑。

1.4.3 BLOGE 不是什么

明确 BLOGE 的边界，同样有助于理解它的定位。下面三种看法都不准确，因为它们都只抓住了表面形式，而没有抓住它作为编排模型与执行引擎的整体性质。

1. 它不是顺序代码的另一层外壳。如果只是把一串方法调用换成另一种写法，执行关系仍然主要由书写顺序决定；而在 BLOGE 里，真正决定执行形状的是 graph 中声明出来的依赖关系。
2. 它不是单纯的流程图工具。 graph 在这里不是静态说明图，而是可被引擎读取、调度和执行的工作流模型。
3. 它不是要求业务团队手工补齐调度与恢复机制。并发、重试、超时、降级和等待等运行期行为不需要在每个业务步骤外层重新拼装，而是由引擎依据模型统一处理。

最简洁的心智捷径可以记成一句话：**BLOGE** 把工作流意图变成显式 **graph**，再把运行层面的负担交给引擎。

1.5 为什么不是手写编排

只知道 BLOGE 是什么还不够，你还需要知道它为什么值得存在。真正的问题不是“能不能把流程写在 service 里”，而是当流程越来越长、越来越依赖外部系统、越来越需要恢复和排障时，手写编排会迅速失控。

1.5.1 为什么这很重要

很多业务系统一开始都很简单：

1. 取用户
2. 取商品
3. 计算价格
4. 做授信检查
5. 创建订单（或者拒绝）

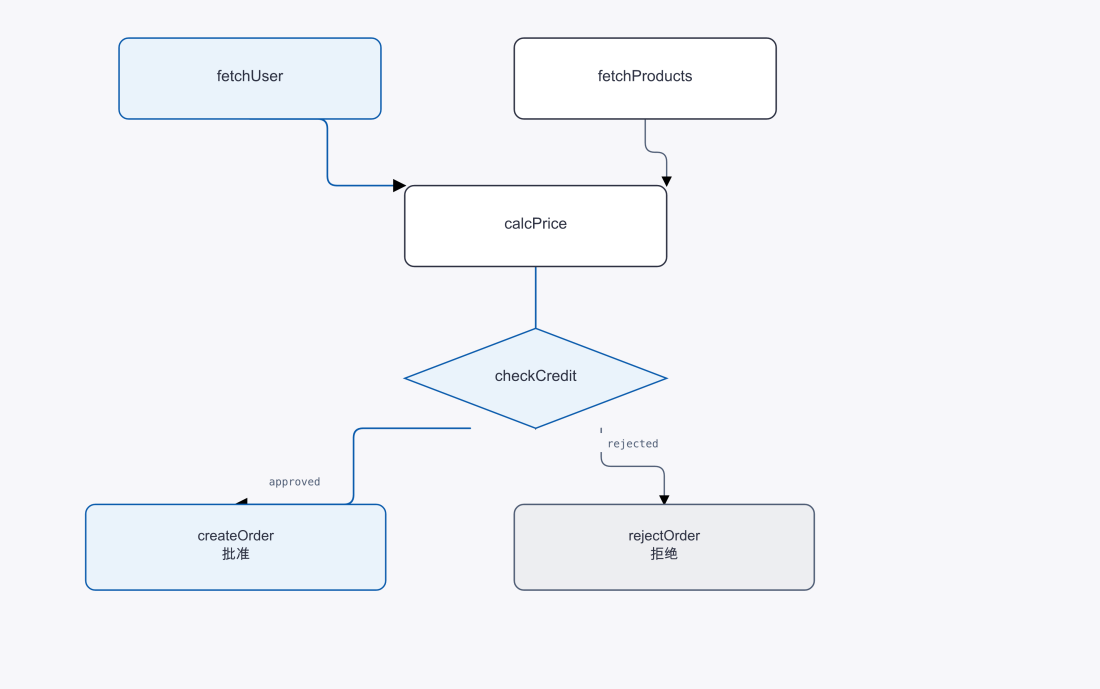
在普通服务代码里，你通常会把它写成顺序方法调用。这样一开始没问题——直到问题出现：

- 步骤 **1** 和步骤 **2** 本来互不依赖，却被顺序执行，平白浪费总耗时。
- 步骤 **4** 可能出现瞬时失败，于是你开始在业务代码里写重试循环，基础设施逻辑和领域逻辑缠到一起。
- 没人能“看见”整个流程。一旦线上出问题，排障的人必须先读代码，才能还原到底发生了什么。

这正是 BLOGE 存在的原因。当你先明确 BLOGE 是一个“执行显式 graph 的引擎”之后，这个价值就更容易理解：你声明流程的形状——谁依赖谁、失败时怎么办——然后把调度、并发和韧性策略交给引擎处理。

1.5.2 心智模型

把 BLOGE graph 想成一张交给引擎的流程卡片：



每个方框都是一个 **node**，每条箭头都是一个 **dependency**。引擎读到这张卡片后，就能知道：

- `fetchUser` 和 `fetchProducts` 彼此没有依赖 → 可以并行运行。
- `calcPrice` 同时依赖两者 → 必须等两边都完成后再启动。
- `checkCredit` 的输出驱动一个 **branch** ——引擎根据结果选择正确的下游节点。

你描述的是 谁依赖谁；引擎决定的是 什么时候运行谁。

1.5.3 第一个可运行示例

下面是从 `order-process.bloge` 里摘出的一个简化片段：

```
graph orderProcess {

  node fetchUser : FetchUserOperator {
    input { userId = ctx.userId }
    timeout = 3s
    retry = { attempts: 2, backoff: 200ms, strategy: exponential }
  }

  node fetchProducts : FetchProductsOperator {
    input { productIds = ctx.productIds }
    timeout = 5s
  }

  node calcPrice : CalcPriceOperator {
    depends_on = [fetchUser, fetchProducts]
    input {
      user     = fetchUser.output
      products = fetchProducts.output
    }
  }

  node checkCredit : CreditCheckOperator {
    depends_on = [fetchUser, calcPrice]
    input {
      userId = fetchUser.output.id
      amount = calcPrice.output.total
    }
    retry = { attempts: 3, backoff: 100ms, strategy: jitter }
    fallback = { approved: false, reason: "credit service unavailable" }
  }

  branch on checkCredit.output.approved {
    true  -> createOrder
    false -> rejectOrder
```

```
  }
}
```

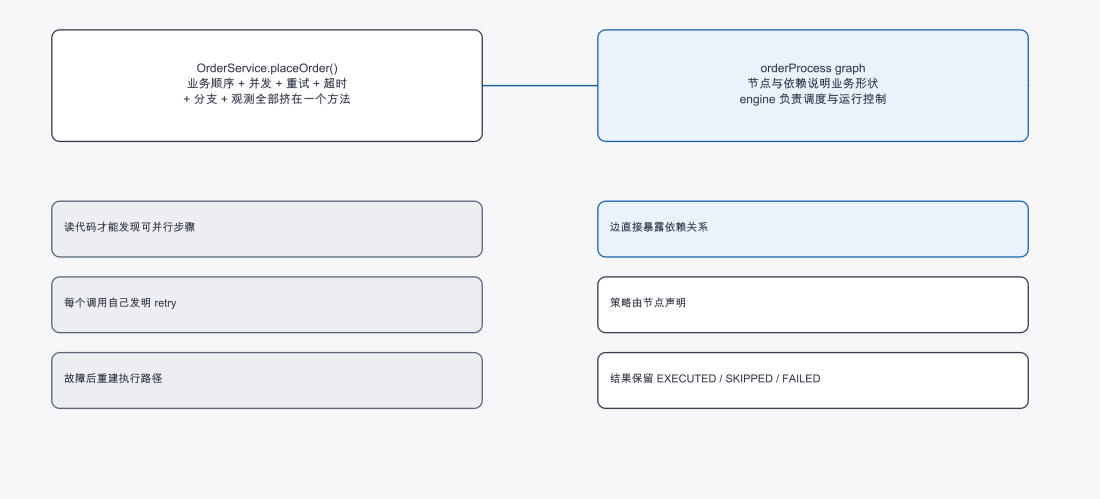
按顺序读这段 DSL：

1. 两个互不依赖的拉取节点 ——引擎会并发执行它们。
2. 一个计算节点 同时引用两个上游输出——因为出现了 `fetchUser.output` 和 `fetchProducts.output`，依赖关系会被自动推断出来。
3. 一个带重试和降级的授信检查节点 ——韧性策略直接声明在节点上。
4. 一个 **branch** ——它不是写在 Java 代码里的 `if`，而是 graph 里的第一类路由结构。

> 关键认识：你没有手写线程池，没有手写重试循环，也没有在业务代码里堆 *try/catch* 去兜底。引擎负责这些事。

1.6 跟着一笔订单：移动责任，而不是移动方法

设想 `OrderService.placeOrder()` 最初只有五次清楚的方法调用。半年后，它还要创建异步任务、重试信用检查、给库存查询加超时、捕获多种异常、记录指标，并决定创建还是拒绝订单。这个方法依然“能跑”，但业务故事和 所有运行机制已经挤在同一个责任边界里。



1.6.1 把图读成一次责任迁移

BLOGE 的价值不是给方法调用画几个方框，而是让每个问题都有一个可见的 owner：

问题	可见 **owner**
有哪些工作?	`fetchUser`、`calcPrice`、`createOrder` 等 node
哪些工作必须先完成?	依赖边
瞬时失败后怎么办?	node 上的 retry 与 timeout 策略
哪条业务路径没有被选中?	结果状态 `SKIPPED`
业务操作实际上做了什么?	Operator 输出与外部证据

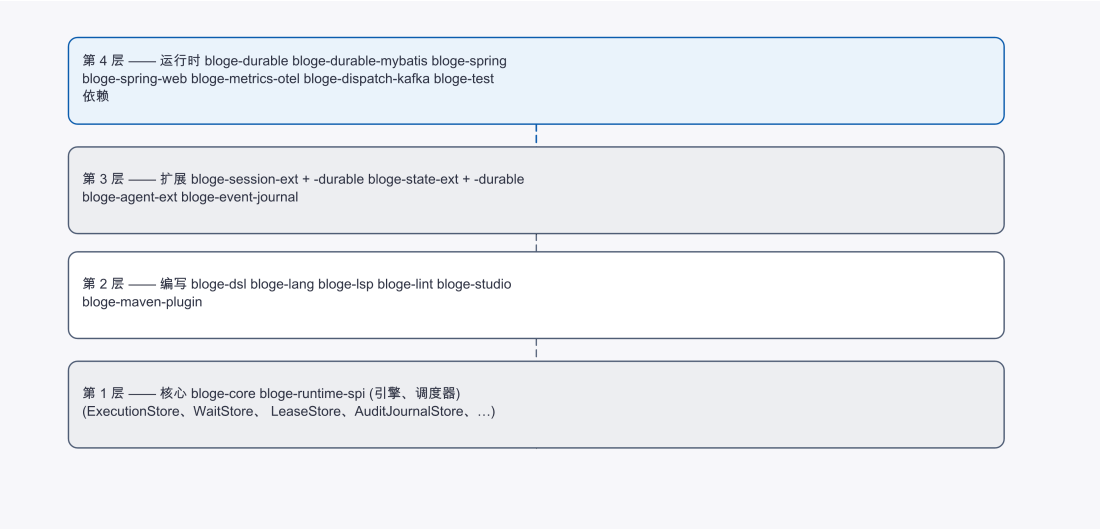
这种拆分会直接改变评审方式。增加库存检查，改的是 graph 形状；修改计价 公式，改的是 `CalcPriceOperator`；调整重试间隔，改的是 node 策略。评审者 不必再从一个控制流方法里重新发现三种不同决策。

1.6.2 先记住第一条边界

显式 graph 只能让已经声明的编排关系可检查；它不能证明 Operator 扣款 金额正确，也不能证明流程满足业务需求。第 18 章与第 23—32 章会逐层补上 更强证据。此时只要抓住一个具体收益：执行形状不再藏在偶然形成的 Java 控制流里。

1.6.3 模块地图——现在你已经知道为什么要看它

订单故事给四层模块各自找到了工作：核心负责运行 graph；编写工具让声明 可读；扩展增加更大的交互模型；运行时集成保留并观察 execution。



第 **1** 层——核心

模块	作用
`bloge-core`	graph model、调度器、listener/interceptor API 与 `GraphEngine`
`bloge-runtime-spi`	execution、checkpoint、wait、timer、lease、audit 与 inbox store 合同

第 **2** 层——编写

模块	作用
`bloge-dsl`	`.bloge` parser、compiler 与 code generator
`bloge-lang` / `bloge-lsp`	共享语言模型与编辑器诊断
`bloge-lint` / `bloge-studio`	可复现规则与可视化 graph 编写
`bloge-maven-plugin`	Operator 元数据生成

第 **3** 层——扩展

模块	作用
`bloge-session-ext` / `bloge-state-ext`	多轮 session 与显式状态机
`bloge-agent-ext`	具有能力边界的 model/tool loop
`bloge-event-journal`	只追加的 execution 与 agent 事件

第 **4** 层——运行时与集成

模块	作用
`bloge-durable` / `bloge-durable-mybatis`	恢复与持久执行状态
`bloge-spring` / `bloge-spring-web`	Boot wiring、Web 错误与运维控制台
`bloge-metrics-otel` / `bloge-dispatch-kafka`	telemetry 与远端 worker 派发
`bloge-test` / `bloge-verification`	graph 测试与业务正确性证据

先使用 `bloge-core` 与 `bloge-dsl`。只有当故事走到对应问题时才增加一层；后面的章节会在这些时刻回到这张地图。

1.7 拆开来看

概念	含义
`graph`	工作流容器——一个 DAG
`node`	一个处理步骤，会执行某个 operator
`input { …}`	把上游输出和上下文值装配为当前节点的输入
`ctx.userId`	来自 *graph context* 的值——执行 graph 时传入的初始数据
`fetchUser.output`	上游节点输出；只要引用它，就会形成隐式依赖
`timeout`	节点的执行时限
`retry`	失败时自动重试，并带退避策略
`fallback`	在重试仍失败时使用的静态降级结果
`branch on`	基于节点输出做条件路由，只激活其中一条下游路径

1.8 常见陷阱

1.8.1 [X] 把 node 当成普通函数调用来理解

```
// 不要这样想:
User user = fetchUser(ctx.getUserId());
```

```
List<Product> products = fetchProducts(ctx.getProductIds());
Price price = calcPrice(user, products);
```

在命令式代码里，代码的顺序 就是执行顺序。在 BLOGE 里，依赖图 才是执行顺序。如果两个节点互相不引用对方输出，引擎就有自由把它们并发起来。

要用依赖思维，而不是顺序思维。

1.9 引导式重写

看看 `bff-dashboard.bloge` 这个例子。它为一个仪表盘并行拉取五类数据：

```
graph bffDashboard {

  node fetchProfile : FetchProfileOperator {
    input { userId = ctx.userId }
    timeout = 2s
    retry = { attempts: 1, backoff: 100ms }
  }

  node fetchOrders : FetchOrdersOperator {
    input { userId = ctx.userId }
    timeout = 3s
    fallback = { recentOrderIds: [], totalOrders: 0 }
  }

  // ··· fetchRecommendations, fetchNotifications, fetchLoyalty ···

  node aggregate : AggregateOperator {
    depends_on = [fetchProfile, fetchOrders, fetchRecommendations, fetchNotifications, fetchLoyalty]
    input {
      profile         = fetchProfile.output
      orders          = fetchOrders.output
      recommendations = fetchRecommendations.output
      notifications   = fetchNotifications.output
      loyalty         = fetchLoyalty.output
    }
  }
}
```

可以思考下面几个问题：

1. 有多少个节点会并行运行？（答案：五个 *fetch* 节点都会并行，因为它们之间没有依赖。）
2. 如果 `fetchOrders` 失败或超时会发生什么？（答案：`fetchOrders` 没有重试策略，只有 *fallback*；因此会直接使用 `{ recentOrderIds: [], totalOrders: 0 }` 这份降级结果，*graph* 仍能继续执行。）
3. 如果你新增第六个 fetch 节点，并让它依赖 `fetchProfile.output`，它还能与其他 fetch 节点一起并行

吗？（答案：不能，它必须先等 `fetchProfile` 完成。）

1.10 脑力检查

1. 让 BLOGE graph 区别于临时编排代码的三个核心特征是什么？（可见、有结构、有韧性。）
2. 两个节点是否并发执行，是由开发者逐行控制，还是由引擎决定？（由引擎根据依赖图决定。）
3. 在 `order-process` 例子里，如果 `checkCredit` 在三次重试后仍然失败，会发生什么？（会使用 *fallback* 值：`{ approved: false, reason: "credit service unavailable" }`。）

1.11 练习

打开 `order-process.bloge`。

1. 新增一个 **`fetchInventory`** 节点，让它与 `fetchUser`、`fetchProducts` 并行执行，并给它 `timeout = 2s`。
2. 把它的输出接入 **`calcPrice`**，为 `calcPrice` 增加一个新的输入绑定。
3. 预测新的执行形状：哪些节点会并行？哪些节点会等待？

这里暂时不必急着编译，重点在于先把“如何读 DSL、如何改 DSL”的基本方式看清楚。真正的运行过程放在第 2 章。

1.12 实验验收卡

- 预期与观察：依赖与韧性策略在 DSL 中可见，读者能标注执行形状。
- 失败与恢复：让节点依赖不存在的输出；撤销错误边并重画 DAG。
- 证明边界：只证明设计意图可审查，不证明可编译、可运行或业务正确。
- 练习合同：订单 DSL；只加 inventory 节点和一条绑定；交付依赖图；每条边有来源且无环即停止。

1.13 回顾

- BLOGE 用一个可见、有结构、有韧性的 graph 替代手写编排。
- **graph** 是由 **node** 组成的 DAG，每个 node 运行一个 **operator**。
- 引擎根据依赖关系（显式声明，或通过 `.output` 推断）来调度节点。
- 韧性策略（timeout、retry、fallback）声明在节点上，而不是埋在业务代码里。
- **branch** 根据节点输出进行控制流路由。

1.14 下一步

第 2 章——你的第一个 Graph继续展开最小 graph 的编写、编译与执行过程。

1.15 参考链接

- 项目 README ——模块概览与快速开始
- 入门指南 ——完整环境与运行说明
- 核心架构 ——引擎内部结构
- `order-process.bloge` ——完整示例源码
- `bff-dashboard.bloge` ——完整示例源码

章节
02

第一阶段——看见工作流

第 2 章——你的第一个 Graph

Graph 形状、依赖与数据流

目标：先看见责任与依赖，再增加控制和规模。

本章通过一个最小可运行示例说明 *BLOGE graph* 的编写、编译与执行过程，包括 *DSL* 方式和 *Fluent Java API* 方式。

2.1 学习目标

1. 写出一个能够被编译器接受的最小 `.bloge` 文件。
2. 从 Java 里加载并执行这个文件。
3. 用 Fluent Java API 构建同一个 graph。
4. 理解 `ctx`、`input`、`output` 与 `timeout` 在运行时分别意味着什么。

2.2 前置条件

- 第 1 章——BLOGE 是什么，以及它为什么存在
- JDK 25+ 且启用 `--enable-preview`，Maven 3.6+ （参见入门指南）

2.3 源示例

文件	展示内容
`hello-world.bloge`	最小可运行示例：一个 graph、一个 node、一个输入绑定
`two-node-chain.bloge`	两个节点组成的链路，演示显式依赖
`input-bindings.bloge`	Conformance fixture：最基础的输入绑定语法

2.4 为什么这很重要

第 1 章已经交代了 BLOGE 的定义、核心特征，以及它为何不同于手写编排。本章继续把这些概念落到一个最小可运行示例上，使 graph 的编写、装载和执行过程都落到具体结果上。

一个单节点 graph 已经足以把四个最基础的运行时概念放到同一条执行路径里：

- 数据如何进入 graph（*context*）。
- node 如何拿到数据（*input block*）。
- 引擎如何调用你的代码（*operator*）。
- 执行完成后如何读结果。

后面的所有内容，都会建立在这四个概念之上。

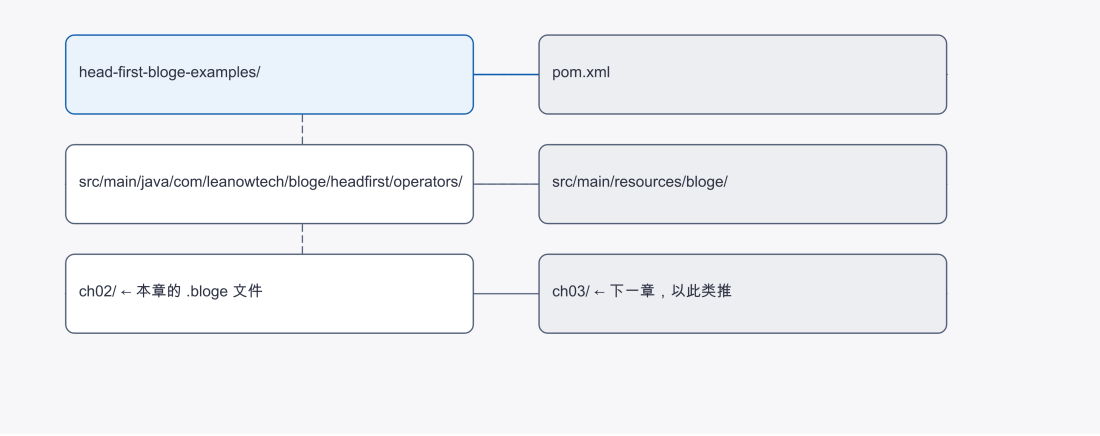
2.5 环境准备

如果你想动手跟着本章做，请使用现在与本书同仓库的 companion 项目。

1. 先安装 standalone companion module 依赖的核心 BLOGE artifact：
   ```
   mvn -q -pl bloge-core,bloge-dsl,bloge-test -am install -DskipTests -Dspotbugs.skip=true
   ```
2. 运行 companion 示例测试：
   ```
   mvn -f head-first-bloge-examples/pom.xml test
   ```
3. 打开第 2 章 companion 文件：
   - `head-first-bloge-examples/src/main/resources/bloge/ch02/hello-world.bloge`
   - `head-first-bloge-examples/src/main/resources/bloge/ch02/two-node-chain.bloge`
   - `head-first-bloge-examples/README.md`

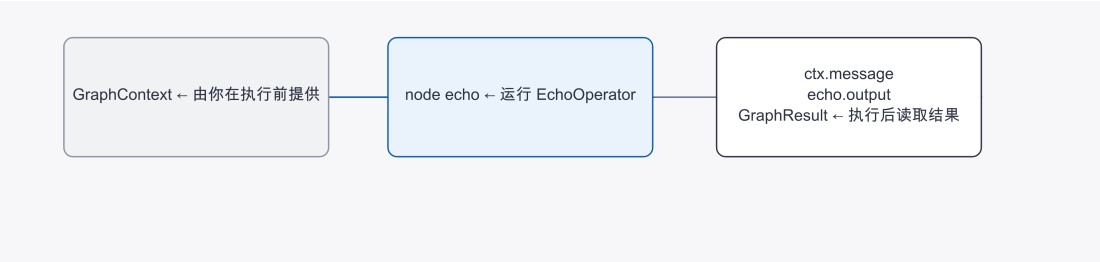
不要只读本章片段。最好把本书和 companion 项目并排打开，边看边运行，这样例子始终和可执行的东西绑在一起。

2.5.1 项目布局



每一章的 `.bloge` 文件都放在 `src/main/resources/bloge/chNN/` 下。`src/test/java/` 里的测试类会编译或执行这些文件。随着你继续阅读本书，记得运行对应章节的 companion 示例，让理解始终落到真实产物上。

2.6 心智模型



1. 你先创建一个 **GraphContext**，把初始值放进去。
2. 引擎解析 **input bindings** ——比如 `ctx.message` ——从 context（或上游输出）取值。
3. 每个 node 的 **operator** 收到组装好的输入，执行逻辑并返回输出。
4. 执行结束后，你从 **GraphResult** 里读取结果。

2.7 第一个可运行示例

2.7.1 DSL 写法

这就是 `hello-world.bloge` 的内容：

```
graph helloWorld {

  node echo : EchoOperator {
    input {
      message = ctx.message
    }
    timeout = 1s
  }
}
```

这已经是一个完整且合法的 BLOGE graph。逐项拆开看：

语法	含义
`graph helloWorld`	声明一个名为 `helloWorld` 的 graph
`node echo`	声明一个名为 `echo` 的节点
`: EchoOperator`	指定这个节点执行的 operator
`input { message = ctx.message }`	把 context 里的 `message` 绑定到节点输入字段 `message`
`timeout = 1s`	节点必须在 1 秒内完成

从 Java 里加载并运行它：

```
// 1. 注册你的 operator
OperatorRegistry registry = new DefaultOperatorRegistry();
registry.register("EchoOperator", (input, opCtx) -> input);

// 2. 加载 DSL 文件
GraphLoader loader = new GraphLoader(registry);
Graph graph = loader.load(Path.of("src/main/resources/bloge/hello-world.bloge"));

// 3. 构造上下文
GraphContext ctx = new GraphContext();
ctx.put("message", "hello");

// 4. 执行
GraphEngine engine = GraphEngine.builder().registry(registry).build();
GraphResult result = engine.execute(graph, ctx);
```

2.7.2 Java API 写法

同一个 graph，也可以用 Fluent Java API 构造：

```
Graph graph = Graph.builder("helloWorld")
    .node("echo", (Map<String, Object> input, OperatorContext opCtx) ->
        Map.of("message", opCtx.graphContext().get("message", String.class)))
        .timeout(Duration.ofSeconds(1))
    .build();

GraphContext ctx = new GraphContext();
ctx.put("message", "hello");

GraphEngine engine = GraphEngine.builder().build();
GraphResult result = engine.execute(graph, ctx);
```

两种写法最终生成的是同一个运行时 graph。DSL 更声明式、更适合外部编排；Java API 更适合在代码里做类型安全组合。很多团队会先用 DSL 做原型，再把稳定下来的 operator 逻辑收进 Java 类。

2.8 先读懂三种结果，再增加配置

第一个 graph 不能以“能够编译”作为终点。你还应该能从一份很小的结果里 看出：执行是否成功，以及两种常见失败停在哪个阶段。这样，后面的 Builder 选项才是在解决问题，而不是一份需要背诵的 API 清单。

2.8.1 结果 1——确实拿到了值

RC1 基础探针以 `message=hello` 运行 `hello-world`，并检查结果，而不是把“没有抛异常”当成成功：

```
graph: hello-world    status: SUCCESS
node:  echo           status: EXECUTED
output.message:       hello
```

这里有三个独立事实：graph 已结束、`echo` 确实执行、输出值符合预期。graph 变大以后，这三类断言仍应保留。

2.8.2 结果 2 与 3——执行根本没有开始

阶段	可观察结果	修复最小 **owner**
编译	node 没有 Operator 绑定，诊断指向声明位置	修正 `.bloge` node 声明
加载	registry 中不存在 `MissingOperator`	注册能力或修正名称
执行	已绑定的 Operator 返回或抛错	检查 node 状态、输出与失败证据

编译失败和加载失败都应满足 `operator calls = 0`。重试 graph 无法修复这两类 问题，因为业务代码从未被派发。

到这里，Builder 选项才有了落点：listener 观察阶段，interceptor 包裹真实 调用，并发参数改变调度，checkpoint store 改变执行状态的保留方式。先看到 问题，再选择拥有这个问题的配置项。

2.9 拆开来看

2.9.1 加一个第二节点

看一下 `two-node-chain.bloge`：

```
graph twoNodeChain {

  node normalizeName : NormalizeNameOperator {
```

```
    input { name = ctx.name }
    timeout = 1s
  }

  node buildGreeting : BuildGreetingOperator {
    depends_on = [normalizeName]
    input { value = normalizeName.output.value }
    timeout = 1s
  }
}
```

新增了两个概念：

概念	含义
`depends_on = [normalizeName]`	显式依赖——`buildGreeting` 必须等待 `normalizeName`
`normalizeName.output.value`	读取上游节点输出里的 `value` 字段

注意：即使不写 `depends_on`，由于 `buildGreeting` 的输入里引用了 `normalizeName.output.value`，编译器也会自动推断出这个依赖。显式声明只是让意图更清楚。

2.9.2 Conformance 基线

Conformance 测试套件里有一个最小输入绑定示例 (`input-bindings.bloge`)：

```
graph g {
  node a : Op {
    input {
      x = ctx.foo
      y = "bar"
    }
  }
}
```

它说明 `input` 右侧既可以是 context 引用（`ctx.foo`），也可以是字面量（`"bar"`）。

2.9.3 只在观察到问题后增加配置

不要把所有 Builder 选项复制进第一个 graph。先保留下面的路由表，等故事走到 对应问题时再回来：

观察到的需要	配置 **owner** 与后续章节
拒绝不兼容的输入输出形状	`Graph.builder().schemaValidation(...)` 与 `.contracts(...)` ——第 8 章
观察或包裹每次调用	`GraphEngine.builder().listeners(...)` 与 `.interceptors(...)` ——第 19 章
限制同时运行的工作量	`.maxGlobalConcurrency(...)` ——第 21 章

观察到的需要	配置 **owner** 与后续章节
保留或迁移执行状态	`.executionCheckpointStore(...)`、`.checkpointCodec(...)`、`.versionMismatchPolicy(...)` ——第 13 章
解析租户范围与准入	`.tenantContextResolver(...)`、`.tenantResourcePolicy(...)` ——第 21 章
替换调度时间实现	`.schedulerTimerSupport(...)` ——第 11—12 章

`Graph.builder()` 描述一个 graph；`GraphEngine.builder()` 配置可以运行多个 graph 的 runtime。说不清要解决哪个已观察问题时，就保留默认值。

2.10 常见陷阱

2.10.1 [X] 忘了写 **operator** 绑定

```
// 错误 —— 编译器需要知道节点要运行哪个 operator
node echo {
  input { message = ctx.message }
}
```

每个 node 都必须通过 `:` `OperatorName` 的形式声明 operator。否则编译器根本不知道该调哪段逻辑。

```
// 正确
node echo : EchoOperator {
  input { message = ctx.message }
}
```

2.11 常见故障

2.11.1 缺少 **operator** 绑定——编译器错误

如果你忘了写 operator 类型：

```
node echo {
  input { message = ctx.message }
}
```

解析器会立刻停下来，报出的核心信息是：

`GraphDefinitionException: Expected ':' after node id`

如果你保留了冒号，但后面仍然没有 operator 名称，接下来还会看到 `Expected operator reference`。

修复：每个 node 都必须写成 `:` `OperatorName`。

2.11.2 operator 未注册——加载阶段错误

如果 DSL 里引用了注册表中不存在的 operator：

```
registry.register("EchoOperator", (input, opCtx) -> input);
Graph graph = loader.load(Path.of("hello-world.bloge"));
// 但 DSL 里写的是 "ShoutOperator"
```

那么在加载 graph 时就会失败，还没进入执行阶段：

```
GraphDefinitionException: Node 'shout' references unregistered operator 'ShoutOperator'
```

修复：在 `load()` 之前，把 `.bloge` 文件里出现的每个 operator 名称都注册好。

2.12 引导式重写

从 `hello-world` 这个 graph 出发，继续往前走一步：

1. 新增一个名为 **`shout`** 的节点，把 `echo` 的输出转成大写。
2. 给它设置 **`timeout = 2s`**。
3. 预测执行顺序。哪个节点先运行？为什么？

你的 DSL 大概会长这样：

```
graph helloWorld {

  node echo : EchoOperator {
    input { message = ctx.message }
    timeout = 1s
  }

  node shout : ShoutOperator {
    input { message = echo.output.message }
    timeout = 2s
  }
}
```

因为 `shout` 引用了 `echo.output`，引擎会自动推断依赖，所以必然是 `echo` 先执行，再轮到 `shout`。

2.13 脑力检查

1. 要执行一个 graph，最少需要哪四样东西？（*Graph*、*OperatorRegistry*、*GraphContext*、*GraphEngine*。）
2. 如果某个节点的输入绑定引用了 `ctx.userId`，这个值来自哪里？（来自你在执行前构造的 *GraphContext*。）
3. 同一个 `input` block 里能不能同时写字面量和上游引用？（可以——*Conformance fixture* 里 `x = ctx.foo` 与 `y = "bar"` 就是同一个 *block*。）
4. 你在 workshop 里勾画一个 workflow，想要最快的“改了就能跑”反馈循环。你会先用 DSL 文件还是 Fluent Java API？什么时候会切换到另一种？（多数团队先用 *DSL*，因为 *graph* 的形状更直观、改起来更快。当

你需要更强的代码复用、更紧的类型约束，或更深的应用级组合时，再切换到 *Java API*。）

5. 设计题：如果一个 graph 有十个节点，但只有两个节点真的需要从 context 取数据，是不是每个节点都必须写 `input` block？（不是。只有消费外部数据或上游输出的节点才需要 *`input`*。没有 *`input block`* 的节点会收到一个空输入 *map*。）

2.14 练习

1. 新建一个 `my-first-graph.bloge` 文件，定义一个 `graph greet`，包含：
   - 一个 `fetchName` 节点，读取 `ctx.name`。
   - 一个 `buildGreeting` 节点，读取 `fetchName.output.name` 并生成问候语。
2. 在 Java 里注册这两个 operator，加载这个文件，用 `ctx.put("name", "Alice")` 执行 graph，并打印结果。
3. 通过 `GraphResult` 验证 `buildGreeting` 是否确实在 `fetchName` 之后执行。

项目环境与运行细节可参考入门指南。

2.15 实验验收卡

- 预期与观察：最小 graph 返回 SUCCESS，节点为 EXECUTED，最终 output 可读取。
- 失败与恢复：拼错 operator 名得到 compile/load error；恢复名称后重跑。
- 证明边界：只证明最小装配和结果读取，不证明重试、持久化或业务正确性。
- 练习合同： hello graph；只改一个 operator 标识；交付成功和失败输出；两者可重复即停止。

2.16 回顾

- 一个最小 BLOGE graph 需要有 `graph` 块、至少一个带 operator 绑定的 `node`，以及对应的 `input`。
- **GraphContext** 负责携带初始数据；**input bindings** 负责把数据送进节点。
- **Fluent Java API** 和 **`.bloge` DSL** 最终构造的是同一种运行时 graph。
- 依赖关系既可以是显式的（`depends_on`），也可以通过 `.output` 自动推断出来。

2.17 下一步

第 3 章——用依赖关系思考进一步展开依赖图调度，以及“按依赖思考，而不是按顺序思考”为什么是 graph 设计的关键。

2.18 参考链接

- 入门指南 ——完整环境说明
- DSL 规范 ——完整语法
- `hello-world.bloge` ——源码

- `two-node-chain.bloge` ——源码
- `input-bindings.bloge` ——conformance fixture

章节
03

第一阶段——看见工作流

第 3 章——用依赖关系思考

Graph 形状、依赖与数据流

目标：先看见责任与依赖，再增加控制和规模。

承诺：读完本章后，你会知道如何把一个 *graph* 读成依赖结构，如何预测它的执行顺序，以及如何摆脱“按顺序写代码”的思维习惯。

3.1 学习目标

1. 区分 隐式依赖（由 `.output` 引用推断）与 显式依赖（`depends_on`）。
2. 看到一个 graph 定义时，能够预测哪些节点并行运行、哪些节点必须等待。
3. 理解引擎的 completion-driven scheduler 如何把 DAG 变成实际执行计划。
4. 识别那些会无意中杀死并行度的多余依赖。

3.2 前置条件

- 第 2 章——你的第一个 Graph

3.3 源示例

文件	展示内容
`order-process.bloge`	真实流程中的扇出、汇合和分支
`bff-dashboard.bloge`	五路并行扇出与不同韧性策略
`depends-timeout.bloge`	显式依赖与 timeout

3.4 为什么这很重要

第 2 章里你跑过了一个或两个节点的 graph，那足够你理解基本机制。但真实工作流往往有很多节点和更复杂的依赖关系——这时候，执行顺序已经不是“从上到下看文件”那么简单了。

如果你不理解引擎如何解析依赖关系，通常会遇到这些问题：

- 本来能并行的节点被你错误串行化。
- 引入循环依赖，结果在编译期就被拒绝。
- 做时序排查时总以为系统会“按你眼睛看到的顺序执行”，而实际上引擎从来没有承诺过这一点。

关键转变是：你声明的是“谁依赖谁”，引擎决定的是“何时运行谁”。

3.5 心智模型

3.5.1 把 **DAG** 当成一份承诺

BLOGE graph 是一个 **DAG**（有向无环图）。每个 node 是顶点，每条 dependency 是有向边。

这样读这张图：

- **A** 和 **B** 没有入边 → 它们是 *source nodes* → 引擎会立刻并行启动它们。
- **C** 同时依赖 A 和 B → 必须等 两边都完成 才能启动。
- **D** 只依赖 B → B 一完成它就能启动，即使 A 还在跑。
- **E** 同时依赖 C 和 D → 它会等待二者都完成。

这就是 **completion-driven** 的调度模型。某个节点一旦完成，引擎就会检查它的下游：是不是已经满足所有依赖？如果满足，就把该节点投入执行。

3.5.2 隐式依赖与显式依赖

形成边有两种方式：

方式	语法	适用场景
隐式依赖	在 `input` 里引用 `someNode.output`	大多数情况下都用它——编译器会自动推断依赖
显式依赖	`depends_on = [someNode]`	你需要执行顺序，但不需要消费上游数据

两者最终形成的是同一种运行时边。大部分时候更常见的是隐式依赖，因为你通常确实需要上游数据。

3.6 第一个可运行示例

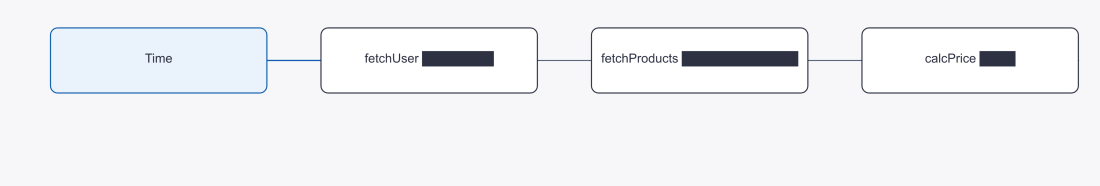
看看 `order-process.bloge` 里的依赖结构：

```
graph orderProcess {

  node fetchUser : FetchUserOperator {
    input { userId = ctx.userId }
    timeout = 3s
  }

  node fetchProducts : FetchProductsOperator {
    input { productIds = ctx.productIds }
    timeout = 5s
  }

  node calcPrice : CalcPriceOperator {
    input {
      user     = fetchUser.output
      products = fetchProducts.output
    }
  }

  // ...
}
```

依赖分析如下：

节点	依赖谁	原因
`fetchUser`	*(无)*	只读 `ctx`，是 source node
`fetchProducts`	*(无)*	只读 `ctx`，是 source node
`calcPrice`	`fetchUser`、`fetchProducts`	在输入里引用了两个 `.output`

执行时间线：



`fetchUser` 和 `fetchProducts` 同时启动；`calcPrice` 必须等二者都结束后才能开始。依赖形状已经由 graph 声明，因此业务代码不必再组装线程池或 `CompletableFuture.allOf()`；运行时 executor 的容量仍由应用配置负责。

3.7 用四帧观察 **ready set** 如何移动

“A 和 B 并行”很容易复述，却也很容易误解。调度器不会拿着光标从第一行走 到最后一行；它维护一组依赖已经满足的 node，再由完成事件推动这组集合变化。

3.7.1 时间戳到底证明了什么

图中的时间是经过取整的规范化轨迹，不是性能承诺。RC1 探针真正断言的是 下面几组先后关系：

观察	必须成立的不等式
两个源 node 同时具备资格	`start(fetchUser)` 与 `start(fetchProducts)` 都发生在释放任一源 node 之前
只完成一个源 node 还不够	另一个源 node 被阻塞时，`start(calcPrice)` 仍未设置
fan-in 在两者之后开始	`start(calcPrice) > start(fetchUser)` 且 `start(calcPrice) > start(fetchProducts)`

在 42 ms，`fetchUser` 已经结束，但 `calcPrice` 仍有一个未满足依赖。到 67 ms，第二个完成事件把计数减到零，`calcPrice` 才进入 ready set。这就是图形化 fan-in 背后的技术机制。

3.7.2 改一条边，先预测一个后果

删掉对 `fetchProducts` 的依赖，`calcPrice` 可能在 42 ms 就开始，此时商品数据 还不存在。给 `fetchProducts` 增加一条来自 `fetchUser` 的多余依赖，两个源调用 又会变成串行。因此，依赖不是绘图偏好，而是一条可执行的数据与顺序声明。

3.8 拆开来看

3.8.1 **Fan-out**

当多个 source node 只从 `ctx` 读取输入时，引擎会把它们同时启动。这就是 **fan-out**：

```
// Five independent fetches — all run in parallel
node fetchProfile       : FetchProfileOperator       { input { userId = ctx.userId } }
node fetchOrders        : FetchOrdersOperator        { input { userId = ctx.userId } }
node fetchRecommendations : FetchRecommendationsOperator { input { userId = ctx.userId } }
```

```
node fetchNotifications : FetchNotificationsOperator { input { userId = ctx.userId } }
node fetchLoyalty       : FetchLoyaltyOperator       { input { userId = ctx.userId } }
```

（摘自 *bff-dashboard.bloge*。）

3.8.2 Fan-in

当一个节点在输入里同时引用多个上游输出时，它就形成 **fan-in** ——必须等待所有上游：

```
node aggregate : AggregateOperator {
  input {
    profile         = fetchProfile.output
    orders          = fetchOrders.output
    recommendations = fetchRecommendations.output
    notifications   = fetchNotifications.output
    loyalty         = fetchLoyalty.output
  }
}
```

3.8.3 不消费数据的显式依赖

有时你只需要顺序，不需要上游输出。比如你希望 B 在 A 完成后再运行，即便 B 不用 A 的数据：

```
node a : OpA {}
node b : OpB {
  depends_on = [a]
}
```

（摘自 *depends-timeout.bloge*。）

显式的 `depends_on` 和通过 `.output` 推断出来的边，在运行时本质上是同一种东西。

3.9 常见陷阱

3.9.1 [X] 无意中把并行写成串行

```
node fetchUser : FetchUserOperator {
  input { userId = ctx.userId }
}

node fetchProducts : FetchProductsOperator {
  // WRONG — this creates an unnecessary dependency on fetchUser
  depends_on = [fetchUser]
  input { productIds = ctx.productIds }
}
```

由于 `fetchProducts` 只需要 `ctx.productIds`，这个 `depends_on` 根本没有必要。它会强行让 `fetchProducts`

等 `fetchUser` 跑完，从而损失并行度。

经验法则：只有在你真的需要上游先完成——要么为了拿它的数据，要么为了保证副作用顺序——才声明依赖。

3.10 常见故障

3.10.1 循环依赖——编译器拒绝

如果不小心让两个 node 相互读取输出：

```
node a : OpA {
  input { x = b.output.value }
}
node b : OpB {
  input { y = a.output.value }
}
```

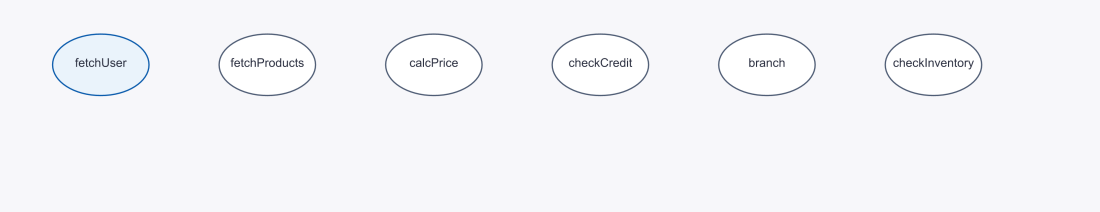
编译器会抛出 `GraphDefinitionException`，诊断中包含 `cycle`。原因不是某个 node 运行失败，而是整张依赖图已经无法做拓扑排序，因此任何业务代码都不应 开始执行。

修复：让数据只沿一个方向流动。若两个 node 看起来都需要对方输出，引入一个中间事实，或重新划分能力边界；不要用隐藏的共享状态绕过环。

3.11 引导式重写

以 `order-process` 为基础，假设现在有一个新需求：在创建订单之前，你必须独立地做库存检查和授信检查。

依赖图大概会变成：



思考下面几个问题：

1. `checkInventory` 能和 `checkCredit` 并行吗？（可以，只要它依赖的是 *`calcPrice`* 或更早节点，而不是 *`checkCredit`* 本身。）
2. branch 应该从哪些结果汇合？（应该同时依赖 *`checkCredit`* 和 *`checkInventory`* 的输出。）
3. 如果所有节点耗时都恰好 1 秒，最短总耗时是多少？（*3* 秒：第一层 *fetch* 并行 *1* 秒，接着 *`calcPrice`* *1* 秒，再接着两个检查并行 *1* 秒。）

3.12 脑力检查

1. 什么是 *source node*？（没有入边、只读 `ctx` 的节点。）
2. 如果节点 C 在输入里引用了 `A.output` 和 `B.output`，引擎至少会推断出哪些依赖？（*C* 依赖 *A*，*C* 也依赖 *B*。）
3. 为什么引擎要求 graph 必须是 DAG，而不允许有环？（一旦成环，就意味着某个节点在等待自己，*graph* 不可能完成。编译器会用 *Kahn* 算法检测并拒绝这种结构。）
4. 如果你给一个只读 `ctx` 的节点加上 `depends_on = [fetchUser]`，会发生什么？（它会被迫等待 `fetchUser`，即使它根本不需要那份数据——这就是无意中的串行化。）

3.13 练习

1. 打开 `bff-dashboard.bloge`。
2. 在纸上（或用 ASCII）画出它的依赖图。把 source nodes 标成 `[S]`，把 terminal node 标成 `[T]`。
3. 预测执行时间线。如果每个 fetch 都耗时 200 ms，而 aggregate 耗时 50 ms，总 wall-clock time 会是多少？（约 *250 ms* ——所有 *fetch* 并行，然后 *aggregate* 执行。）
4. 现在再加一个 `fetchWishlist` 节点，让它依赖 `fetchProfile.output.userId`。重新画图。新的 wall-clock time 会是多少？（约 *450 ms* ——`fetchWishlist` 要先等 `fetchProfile`，而 *aggregate* 还要等所有六个节点。）

3.14 实验验收卡

- 预期与观察：两个源节点同时进入 ready set，fan-in 只在两者完成后开始。
- 失败与恢复：增加一条源节点间依赖；移除后恢复并行形状。
- 证明边界：只证明依赖驱动调度，不证明外部服务延迟或线程安全。
- 练习合同：价格图；只改一条 depends_on；交付前后时间线；观察顺序符合预测即停止。

3.15 回顾

- BLOGE graph 是一个 **DAG**。节点是顶点，依赖是有向边。
- 依赖既可以是隐式的（由 `.output` 推断），也可以是显式的（`depends_on`）。
- 引擎使用 **completion-driven scheduler**：当某个节点完成后，检查哪些下游节点已经“就绪”。
- 当多个 source nodes 互不依赖时，会形成 **fan-out**。
- 当某个节点等待多个上游时，会形成 **fan-in**。
- 多余的依赖会杀死并行度；只有真的需要顺序时才声明它。

3.16 下一步

在第 4 章——流动的数据里，你会学习数据如何在 graph 中流动：输入绑定、输出引用、路径表达式、transform，以及什么时候应该用 operator、什么时候只需要不调度 Operator 的投影。

3.17 参考链接

- 核心架构 ——引擎调度设计
- DSL 规范 ——`depends_on` 与输入绑定语法
- `order-process.bloge` ——源码
- `bff-dashboard.bloge` ——源码
- `depends-timeout.bloge` ——conformance fixture

章节

04

第一阶段——看见工作流

第 4 章——流动的数据

Graph 形状、依赖与数据流

目标：先看见责任与依赖，再增加控制和规模。

承诺：读完本章后，你会理解数据如何进入、穿过并离开一个 *BLOGE graph*；同时你也会知道什么时候该用 *input binding*、什么时候该用 *transform*、什么时候必须写成 *operator*。

4.1 学习目标

1. 解释 `input` 块中可用的三类数据来源：**context** 值、上游输出 和 字面量。
2. 使用 路径表达式 访问嵌套字段，包括安全导航（`?.`）和空值合并（`??`）。
3. 用 下标访问 读取列表与映射元素（`xs[i]`、`xs?[i]`、`xs[-1]`、`m["key"]`），并在表达式结果上使用 动态成员访问（`(expr).field`、`(expr)?.field`）。
4. 区分 **transform**（不调度 Operator 的数据投影）与 **operator node**（可独立调度的业务逻辑）。
5. 在输入绑定中使用 **lambda** 表达式 做集合操作（`map`、`filter`、`reduce`）。

4.2 前置条件

- 第 3 章——用依赖关系思考

4.3 源示例

文件	展示内容
`order-enrichment-lambda.bloge`	在输入绑定中使用 lambda 做集合处理
`order-summary-index-access.bloge`	下标访问、负索引与动态成员访问
`basic.bloge` (transform)	最小 transform 块

文件	展示内容
`safe-navigation.bloge`	安全导航 `?.` 与空值合并 `??`
`lambda-map.bloge`	在输入绑定里使用 `map`
`order-process.bloge`	真实工作流里的 transform

4.4 为什么这很重要

第 3 章里你学到的是“依赖决定什么时候运行”。但依赖只决定 何时，不决定 拿到什么数据。

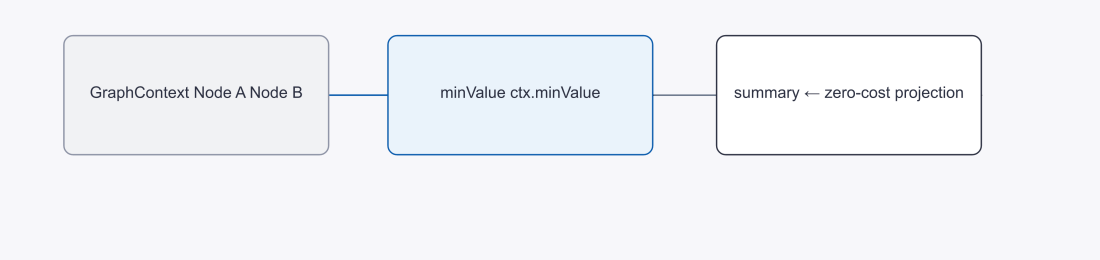
数据流设计得好，graph 会清晰、易维护；数据流设计得差，每个节点都在做额外的字段搬运和结构重塑，最后谁也说不清每一步真正的业务职责是什么。

BLOGE 为数据处理提供了分层工具：

层次	工具	调度成本	适用场景
路由	Input binding（`input { …}`）	无——在装配阶段求值	传递 context 字段或上游输出
投影	Transform（`transform { …}`）	无——虚拟节点，不调度 operator	重组、重命名字段，但不包含业务逻辑
计算	Operator node（`node …: Op { …}`）	完整调度成本	业务逻辑、I/O、副作用

选对层次，graph 才能既快又好读。

4.5 心智模型



- **Input bindings** 从 `ctx` 或 `someNode.output.field` 取值。
- **Transforms** 是虚拟的——引擎不会为它们单独开线程去调度。
- **Operator nodes** 才是真正执行代码的单元，因而会附带 timeout、retry、fallback 等运行时语义。

4.6 第一个可运行示例

4.6.1 Input bindings ——基础形式

每个 `input { …}` 都是一组 key-value 绑定。右侧是一个表达式，会在输入装配阶段被求值。

```
node calcPrice : CalcPriceOperator {
  input {
    user     = fetchUser.output          // whole output object
    products = fetchProducts.output      // whole output object
    userId   = fetchUser.output.userId   // nested field
    tax      = ctx.taxRate               // context value
    label    = "order"                   // literal string
  }
}
```

三个来源，同一个 block：

来源	语法	例子
Context	`ctx.fieldName`	`ctx.taxRate`
上游输出	`nodeName.output` 或 `nodeName.output.field`	`fetchUser.output.userId`
字面量	字符串或数字字面量	`"order"`、`42`

4.6.2 路径表达式

路径表达式用来访问嵌套结构：

```
input {
  city = fetchUser.output.address.city
}
```

这表示沿着 `output → address → city` 逐层读取。

4.6.3 安全导航与空值合并

当某个字段可能为 null 时，可以用 安全导航运算符（`?.`）避免空指针，再用 空值合并运算符（`??`）给一个默认值：

```
// From safe-navigation.bloge
node normalize : Normalize {
  input {
    profileName = ctx.request?.user?.name
    result      = fetch?.result?.value ?? "missing"
    names       = ctx.items?.map(x -> x.profile?.name)
  }
}
```

（摘自 *safe-navigation.bloge*。）

运算符	含义
`?.`	如果左边是 null，就停止继续向下访问并返回 null
`??`	如果左边为 null，就改用右边的默认值

它们可以自然组合：`a?.b?.c ?? "default"` 的意思就是“尝试读取 `a.b.c`；如果中途任意一层是 null，就返回 `default`”。

4.6.4 访问列表与映射

路径表达式访问的是 字段。要访问列表的 元素 或映射的 条目，使用下标访问运算符：

语法	含义
`xs[i]`	取下标 `i` 位置的元素；越界时抛出运行时错误
`xs?[i]`	安全下标访问；越界时返回 `null` 而不抛错
`xs[-1]`	负下标从尾部计数（`-1` 即最后一个元素）
`m["key"]`	按字符串 key 查找映射条目
`(expr).field`	在任意表达式结果上读取字段
`(expr)?.field`	在表达式结果上做安全字段访问

下标访问可与已学过的安全导航、空值合并、lambda 等自由组合，并且仍然是在输入装配阶段被求值，不引入额外的调度开销：

```
// From ch04/order-summary-index-access.bloge
node fetchOrders : FetchOrdersOperator {
  input {
    customerId = ctx.customerId
  }
}

transform summary {
  firstOrderId = fetchOrders.output.recentOrderIds[0]
  lastOrderId  = fetchOrders.output.recentOrderIds[-1]
  safeFirst    = fetchOrders.output?.recentOrderIds?[0] ?? "none"
  regionLabel  = ctx.regionMap["north-east"]
  totalOrders  = fetchOrders.output.totalOrders
}
```

常见陷阱——*`xs[i]` 还是 `xs?[i]`？*

两种写法都能取第 `i` 个元素，但在列表比预期短的时候行为完全不同：

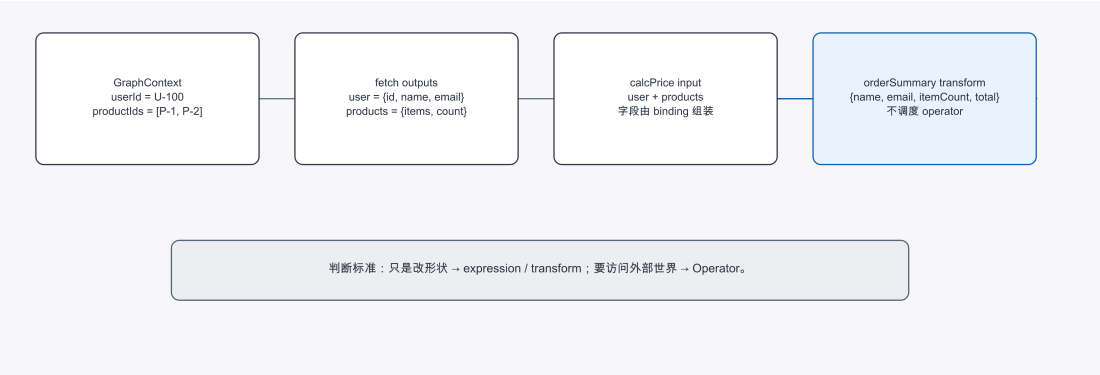
- `xs[i]` 是 严格的：越界即报错，求值立刻失败。当“空列表 / 短列表”本身就是 *bug* 时使用——例如读取刚刚创建出来的订单。

- *`xs?[i]`* 是 宽容的：越界返回 *`null`*，配合 *`??`* 给出兜底值。当“暂时没有数据”是正常情形时使用——例如读取可能尚未写入的历史记录。

默认优先选择严格写法；只有在确实需要容忍缺失数据时才换成安全写法。

4.7 跟随一笔订单经过四种数据形状

不要逐个背表达式符号，而要跟着一个业务对象走。订单从 `GraphContext` 里的 几个标识开始，经两个 Operator 补充事实，成为计价输入，最后被投影成供后续 步骤读取的小型摘要。



4.7.1 在每个边界说清楚数据形状

边界	此处可用的数据形状	存在原因
graph 入口	`{userId, productIds}`	调用方提供的稳定请求事实
查询之后	`{user}` 与 `{products}`	外部能力已经补充请求信息
计价输入	`{user, products}`	input binding 只组装计价需要的数据
摘要 transform	`{name, email, itemCount, total}`	供下游读取的低成本投影

这就是一条很小的数据血缘。当 `customerEmail` 错误时，沿图向左走：transform 读取 `fetchUser.output.email`，所以应先检查用户查询输出，而不是调试计价 Operator。graph 已经告诉你这个事实由哪个 producer 负责。

4.7.2 按 **effect** 选工具，不按代码行数

如果只是对已有值做确定性重组，使用表达式或 `transform`；如果步骤要进入 外部世界、消耗时间、可以独立失败，或者需要 retry 与 timeout 策略，使用 Operator。十行投影仍然是 transform；一行 HTTP 调用仍然是有 effect 的 Operator。

这条边界可以被检查：`orderSummary` 不应出现 Operator invocation，而 `fetchUser` 与 `fetchProducts` 必须有调用证据。这样的观察比只说 transform 这样的观察比笼统称为“零成本”更有用，因为它明确说明了引擎没有调度什么。

4.8 拆开来看

4.8.1 Transform ——不调度 Operator 的投影

Transform 是一个虚拟节点。它负责重塑数据，但不会执行 operator：

```
// From order-process.bloge
transform orderSummary {
  customerName  = fetchUser.output.name
  customerEmail = fetchUser.output.email
  itemCount     = fetchProducts.output.items.size
  total         = calcPrice.output.total
}
```

引擎不会把 transform 当成一个独立节点调度到线程上执行。它会直接在绑定解析阶段生成结果，然后以 `orderSummary.customerName`、`orderSummary.itemCount` 这样的形式暴露给后续引用。

Conformance 套件中的最小 transform 示例 (`basic.bloge`)：

```
graph g {
  node source : Op {}
  transform summary {
    greeting: String = "hello"
    result: String   = source.output.value
  }
}
```

这也说明 transform 支持类型标注（例如 `greeting: String`）。

什么时候该用 **transform**，什么时候该用 **operator**？

适合用 **transform** 的情况	适合用 **operator** 的情况
只是挑字段、改字段名	要调用外部服务
没有副作用	有真正的业务规则
不需要 timeout / retry	需要韧性策略
纯数据重组	需要独立可观测性

4.8.2 输入绑定中的 lambda 表达式

对于集合处理，BLOGE 支持直接在输入绑定中写 **lambda** 表达式。这些表达式在输入装配阶段求值，没有额外调度成本。

摘自 `lambda-map.bloge`：

```
node a : OpA {
  input {
    names = ctx.items.map(x -> x.name)
  }
```

```
}
```

表达式 `ctx.items.map(x -> x.name)` 会从 context 里的列表中提取每个元素的 `name` 字段，生成一个新列表。

再看一个更完整的例子，来自 `order-enrichment-lambda.bloge`：

```
node enrichOrders : EnrichOrdersOperator {
  depends_on = [fetchOrders, fetchProducts]
  input {
    enrichedOrders = fetchOrders.output.orders
      .map(o -> {
        orderId:     o.orderId,
        totalValue:  o.price * o.quantity,
        taxAmount:   o.price * o.quantity * 0.1,
        productName: fetchProducts.output.catalogue.associate(p -> p.id, p -> p.name)
      })
      .filter(o -> o.totalValue > ctx.minValue)
      .sortBy(o -> o.totalValue)

    totalRevenue = fetchOrders.output.orders
      .filter(o -> o.price * o.quantity > ctx.minValue)
      .reduce(0, (acc, o) -> acc + o.price * o.quantity)
  }
}
```

常见的集合操作有：

操作	语法	结果
`map`	`.map(x -> expr)`	逐元素转换后得到新列表
`filter`	`.filter(x -> condition)`	只保留满足条件的元素
`reduce`	`.reduce(init, (acc, x) -> expr)`	折叠为一个累积值
`sortBy`	`.sortBy(x -> key)`	按派生 key 排序

lambda 表达式可以生成对象字面量（`{ field: value, …}`），也可以做链式组合（`.map(…).filter(…).sortBy(…)`）。

4.9 常见陷阱

4.9.1 [X] 用 operator node 做纯数据重组

```
// WRONG — this is a transform, not business logic
node formatOutput : OutputFormatterOperator {
  input {
    customerName = fetchUser.output.name
```

```
    total       = calcPrice.output.total
  }
}
```

如果 OutputFormatterOperator 只是拿字段、改字段名，那它就在为一段纯数据投影支付完整 node 的成本（调度、超时跟踪、运行开销），而 transform 可以在不产生 Operator invocation 的情况下完成：

```
// CORRECT — projection without an Operator invocation
transform formatOutput {
  customerName = fetchUser.output.name
  total        = calcPrice.output.total
}
```

经验法则：如果没有副作用、没有 I/O、没有真正的业务规则，就优先用 transform。

4.10 常见故障

4.10.1 没有安全导航的空路径访问

下面的 binding 使用普通字段访问：

```
input {
  city = fetchUser.output.address.city
}
```

如果运行时 address 为 null，node 会在组装输入时失败，因为普通 .city 仍会继续穿过空值。BLOGE 只有在使用 ?. 时才会短路这次读取。

修复：只有当“没有地址”是合法业务状态时，才用安全导航并给出明确默认值：

```
input {
  city = fetchUser.output.address?.city ?? "Unknown"
}
```

?. 把空地址变成 null，?? 再把它替换为 "Unknown"。如果地址缺失本应 阻止下单，就不要兜底；让严格访问失败，保留真实合同错误。

4.11 引导式重写

从下面这个“用了太多 operator node”的 graph 出发：

```
graph report {
  node fetchSales   : FetchSalesOperator   { input { region = ctx.region } }
  node fetchReturns : FetchReturnsOperator { input { region = ctx.region } }

  // This node just picks fields — wasteful as an operator
```

```
  node buildSummary : SummaryBuilderOperator {
    input {
      totalSales   = fetchSales.output.total
      totalReturns = fetchReturns.output.total
    }
  }
}
```

把 `buildSummary` 改写为 transform：

```
graph report {
  node fetchSales   : FetchSalesOperator   { input { region = ctx.region } }
  node fetchReturns : FetchReturnsOperator { input { region = ctx.region } }

  transform buildSummary {
    totalSales   = fetchSales.output.total
    totalReturns = fetchReturns.output.total
    netRevenue   = fetchSales.output.total
  }
}
```

这样一来，`buildSummary` 不会增加 Operator invocation 或独立调度节点。如果还要计算 `netRevenue = totalSales - totalReturns` 这样的派生字段，可以继续写在绑定表达式里，不必额外引入 operator。

4.12 脑力检查

1. 输入绑定里有哪三类数据来源？（`ctx.*` 提供 *context* 值，`node.output.*` 提供上游输出，字面量提供常量值。）
2. `ctx.user?.profile?.name` 中的 `?.` 会做什么？（如果 `user` 或 `profile` 是 *null*，整个表达式直接返回 *null*，而不是抛空指针异常。）
3. 什么情况下应该用 transform 而不是 operator node？（当你只是重组、重命名数据，没有副作用、*I/O* 或业务规则时。）
4. 输入绑定里的 lambda 表达式会带来额外调度成本吗？（不会。它在输入装配阶段求值，不会作为独立节点被调度。）

4.13 练习

1. 打开 `order-enrichment-lambda.bloge`。
2. 新增一个输入绑定，用 `reduce` 先求和、再除以列表大小，计算平均订单金额。
3. 再新增一个 **transform**，把 `totalRevenue` 和 `ctx.budgetTarget` 组合成一个 `percentOfBudget` 字段。
4. 在脑中检查：哪些部分不会额外调度 Operator（transform + lambda），哪些情况才需要一个独立 operator？

4.14 实验验收卡

- 预期与观察：订单值经过 context、output、binding 和 transform 四种形状。
- 失败与恢复：严格路径指向缺失字段；改回正确路径或安全读取。
- 证明边界：只证明绑定和路径行为，不证明输入真实完整。
- 练习合同：订单样例；只改一条字段路径；交付成功值和失败 reason；两者可归因即停止。

4.15 回顾

- **Input bindings** 把 context、上游输出和字面量装配成节点输入；它们在装配阶段求值，没有调度成本。
- 路径表达式 用于访问嵌套结构；`?.` 提供安全导航，`??` 提供空值合并。
- **Transforms** 是虚拟节点，不产生 Operator invocation 或独立调度任务；表达式求值本身仍会消耗计算资源。
- **Lambda** 表达式（`map`、`filter`、`reduce`、`sortBy`）可直接在输入绑定中操作集合，同样没有额外调度成本。
- 要选对层次： input bindings 负责路由，transforms 负责投影，operator nodes 负责真正的业务逻辑。

4.16 下一步

在第 5 章——做决策的分支里，你会学习如何在 graph 里加入条件控制流：根据某个节点的输出，把执行路由到不同路径上。

4.17 参考链接

- DSL 规范 ——表达式语法与 transform 语法
- Operator 设计规范 ——数据变换层次与 operator 边界
- 入门指南 ——transform 与表达式基础示例
- `order-enrichment-lambda.bloge` ——源码
- `safe-navigation.bloge` ——conformance fixture
- `lambda-map.bloge` ——conformance fixture
- `basic.bloge` (transform) ——conformance fixture

章节 —

第一阶段回顾——入门（第 1–4 章）

你现在已经掌握了阅读小型 *BLOGE DAG*、预测其声明数据依赖所需的词汇和心智模型。

0.1 你学到了什么

第 **1** 章——**BLOGE** 是什么，以及它为什么存在 为你建立了 BLOGE 的三个核心特征：工作流是可见的（graph 的形状是显式的）、执行由依赖驱动（你声明边，引擎决定顺序）、韧性是内建的（重试、超时、降级属于模型本身）。

第 **2** 章——你的第一个 **Graph** 把理论变成了实践。你构建了一个 graph、运行它、检查了 `GraphResult`，亲眼看到是引擎——而不是你——决定哪些节点可以并行执行。

第 **3** 章——依赖关系思维 强化了你对依赖边的判断力。你学会了阅读有向无环图、识别不必要的串行化，以及区分控制依赖和数据依赖。

第 **4** 章——流动的数据 补全了整幅图。你了解了数据如何通过 context 进入 graph、如何通过 input binding 在节点间流动、如何通过 node output 流出。你练习了路径表达式、安全导航（`?.`）、空值合并（`??`）、transform 和 lambda 输入绑定。

0.2 你现在应该能做到

- 用一句话描述 BLOGE，不借助任何框架术语。
- 在纸上画出一个小型工作流的依赖图，并预测执行顺序。
- 编写一个包含三四个节点、显式依赖和输入绑定的简单 `.bloge` 文件。
- 读取 `GraphResult` 并安全地提取类型化输出。
- 解释 transform 和 operator 节点的区别。

0.3 这个阶段的常见错误

错误	为什么会犯	如何纠正
添加多余的 `depends_on` 边	把” 这个在那个之后执行” 和” 这个确实需要那个节点的数据” 搞混了	问自己：下游节点是否读取上游输出？如果不是，就去掉这条边。
总用 `getOutput` 而不是 `findOutput`	假设每个节点一定会完成	当节点可能被跳过或失败时，用 `findOutput` 或 `getOutputOrDefault`。
把业务逻辑放进 transform	Transform 是零开销的数据投影，不是 operator	如果代码有副作用或可能失败，就应该放进 operator。
硬编码值而不是使用 `ctx.*`	从示例中复制粘贴时忘了参数化	把环境相关或调用方相关的值移入 graph context。

0.4 核心词汇

术语	含义
Graph	由业务步骤组成的有向无环图，具有显式的数据和控制流。
Node	Graph 中的一个步骤，由一个 operator 支撑。
Operator	类型化函数 `(I, GraphContext) → O`，负责执行真正的业务逻辑。
依赖边	一条有向链接，表示” 这个节点必须等那个节点完成后才能开始”。
Input binding	从 context、上游输出或字面量解析节点输入的表达式。
Transform	纯投影节点——只做数据变形，不触发 Operator 调用或独立节点调度；表达式本身仍会消耗计算资源。
GraphResult	执行后返回的不可变记录：成功标志、输出、状态、耗时。

0.5 试一试：删除一条假依赖

把早餐画成四个 node：烧水、烤面包、冲咖啡、上桌。只添加物理上必要的依赖，然后删除一条边，解释你得到的是安全并行，还是引入了竞态。

0.6 下一步

第二阶段回顾教你在 graph 内做决策（分支）、处理故障（韧性）、设计干净的 operator，以及编写可维护、可审查的 DSL 文件。从第 5 章——做决策的分支开始。

学习阶段

第二阶段——可维护编写

分支、韧性、Operator 边界、DSL 与评审反馈

目标：把可读流程变成另一位工程师能评审的改动。

章节

05

第二阶段——可维护编写

第 5 章——做决策的分支

分支、韧性、Operator 边界、DSL 与评审反馈

目标：把可读流程变成另一位工程师能评审的改动。

承诺：读完本章后，你会知道如何根据节点的输出将执行路由到不同路径——并且完全理解那些未被选中的路径会发生什么。

5.1 学习目标

1. 编写一个 `branch on` 块，根据运行时的值将执行路由到若干目标节点之一。
2. 使用 `otherwise` 在没有任何显式 case 匹配时提供默认路径。
3. 解释引擎如何将未被选中的目标节点标记为 **SKIPPED**。
4. 区分 `branch on`（控制面路由）与 `when` 表达式（数据面值选择）。
5. 诊断缺少 `otherwise`、表达式类型错误、目标未声明等问题，再使用包容式分支（`branch mode=inclusive on ...`）让多个匹配并行运行，并解释 OR 分裂与默认 XOR 分裂的差异。

5.2 前置条件

- 第 4 章——流动的数据

5.3 源示例

文件	展示内容
`ticket-routing.bloge`	三路分支：字符串 case 加 `otherwise`
`loan-approval.bloge`	在多阶段风控流水线之后做分支，含扇出、transform 和 `otherwise`
`boolean.bloge`	最小布尔分支——`true` / `false`

文件	展示内容
`otherwise.bloge`	最小的纯 `otherwise` 分支
`ch05/multi-flag-approval.bloge`	包容式分支——多条审查轨道根据标志并行运行

5.4 为什么这很重要

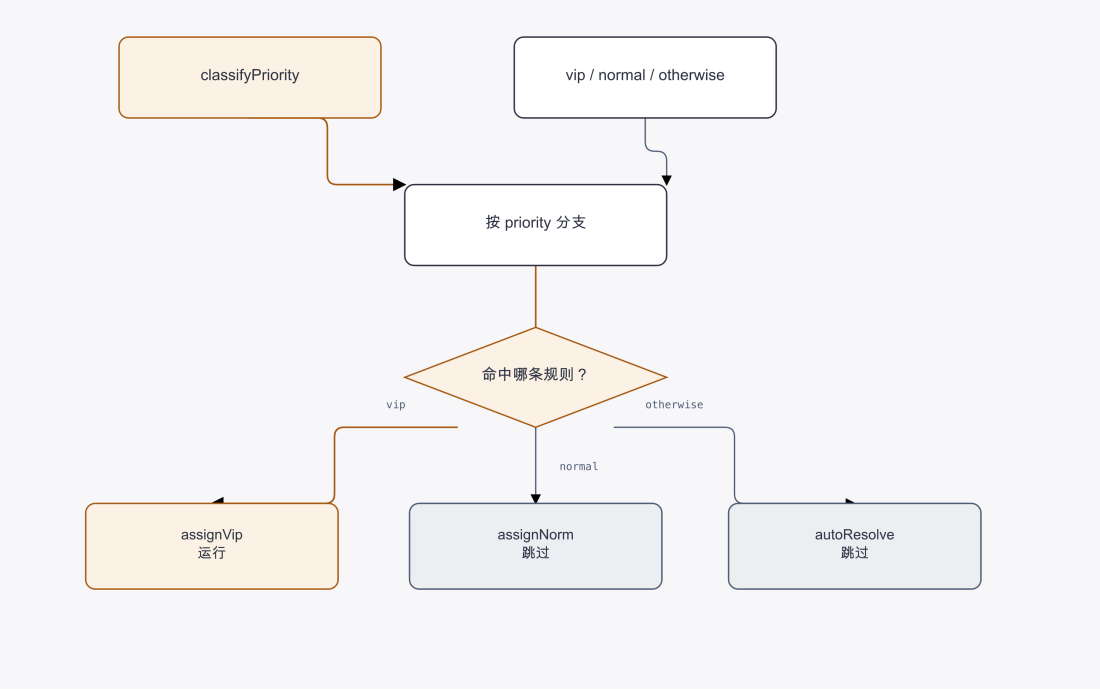
到目前为止，你的 graph 里每个节点都会执行。但真实工作流不是这样的：

- 信用检查 通过或拒绝——你不可能两条路径都走。
- 一张工单只会路由到 一个 处理程序，而不是全部。
- 贷款决策会触发批准、拒绝、或者 人工审核。

在过程式代码里你会写 `if/else` 块。在 BLOGE 里等价的是 `branch on` ——一种声明在 graph 内部的一等路由决策，而不是藏在 operator 代码里。引擎求值条件，只激活一个下游节点，跳过 其余所有节点。

这让路由决策在 graph 定义和 studio 工具中都清晰可见，而不是隐藏在 operator 的实现里。

5.5 心智模型



把分支想象成一个铁路道岔。一条轨道被选中；其余轨道被封锁。引擎的行为：

1. 等待 源节点（`classifyPriority`）执行完成。
2. 从该节点的输出中读取 条件字段（`.priority`）。
3. 按 声明顺序 逐一测试每个 **case**，直到找到匹配项。

4. 调度匹配的目标节点；将其余所有目标标记为 `SKIPPED`。

如果没有任何 case 匹配且没有 `otherwise`，所有 目标节点都不会运行。这几乎总是一个 bug ——这也是为什么当可能值的集合不是穷举已知时，你应该始终包含 `otherwise`。

5.6 第一个可运行示例

以下是 `ticket-routing.bloge` 中的分支示例：

这里保留完整的 63 行 graph，因为分支结果依赖两个并行读取、一次 fan-in 和三个 已声明目标；裁掉任一部分都会隐藏读者需要追踪的控制路径。

```
graph ticketRouting {

  node fetchCustomer : FetchCustomerOperator {
    input { customerId = ctx.customerId }
    timeout = 3s
    retry = { attempts: 2, backoff: 200ms, strategy: exponential }
  }

  node fetchTicketHistory : FetchTicketHistoryOperator {
    input { customerId = ctx.customerId }
    timeout = 3s
  }

  node analyzeSentiment : AnalyzeSentimentOperator {
    depends_on = [fetchCustomer, fetchTicketHistory]
    input {
      customer = fetchCustomer.output
      history  = fetchTicketHistory.output
      message  = ctx.message
    }
    timeout = 5s
    retry = { attempts: 1, backoff: 500ms, strategy: exponential }
    fallback = { sentiment: "neutral", score: 0.0, keywords: [] }
  }

  node classifyPriority : ClassifyPriorityOperator {
    depends_on = [analyzeSentiment]
    input {
      customer  = fetchCustomer.output
      sentiment = analyzeSentiment.output
    }
  }

  branch on classifyPriority.output.priority {
```

```
    "vip"    -> assignVipAgent
    "normal" -> assignNormalAgent
    otherwise -> autoResolve
  }

  node assignVipAgent : AssignVipAgentOperator {
    depends_on = [classifyPriority]
    input {
      customerId = fetchCustomer.output.id
      priority   = "vip"
    }
  }

  node assignNormalAgent : AssignNormalAgentOperator {
    depends_on = [classifyPriority]
    input {
      customerId = fetchCustomer.output.id
      priority   = "normal"
    }
  }

  node autoResolve : AutoResolveOperator {
    depends_on = [classifyPriority]
    input {
      customerId = fetchCustomer.output.id
      keywords   = analyzeSentiment.output.keywords
    }
  }
}
```

把它当成一个故事来读：

1. 两个并行拉取 ——客户资料和工单历史。
2. 情感分析 从两者扇入汇合，带有重试和 fallback。
3. 优先级分类 产出 `"vip"`、`"normal"` 或其他值。
4. 分支 ——`branch on classifyPriority.output.priority` ——精确选中一个处理程序。如果优先级是 `"vip"`，只有 `assignVipAgent` 运行。如果是 `"normal"`，只有 `assignNormalAgent`。其他任何值都通过 `otherwise` 落入 `autoResolve`。

关键洞察：三个处理程序节点都声明了 *`depends_on = [classifyPriority]`*，但分支确保其中只有一个实际执行。其余节点会收到 *`SKIPPED`* 状态。

5.7 用两条 Case 证明 XOR，而不是只跑 happy path

排他分支同时承诺两件事：只选择一个匹配目标，另一目标不会执行。只测试 审批通过路径，只证明了合同的一半。

5.7.1 把未选路径也当成结果来读

Case	createOrder	rejectOrder	业务含义
approved=true	EXECUTED	SKIPPED	订单已创建，拒绝代码没有运行
approved=false	SKIPPED	EXECUTED	已执行拒绝，没有创建订单

SKIPPED 不是数据缺失，也不是失败。它是 graph 已计算分支并排除该 node 的 正向证据。因此，RC1 基础探针会同时断言 `assertNodeExecuted("createOrder")` 与 `assertNodeSkipped("rejectOrder")`。

5.7.2 先把第一种决策讲窄

学习时先掌握 XOR，因为一个输入只选择一条路径。只有当业务语句确实允许 多个动作同时发生，例如"通知合规并且要求补充材料"时，才使用包容式 分支。当多个独立条件共同决定结果，而且规则集合需要做覆盖分析时，再使用 决策表。它们是更强的模型，不是二选一的花哨写法。

做一次单变量变异就能看到边界：只把 `approved=true` 改成 `false`。如果只有 选中与跳过状态交换，分支是隔离的；如果无关 node 也改变，说明决策耦合了 图中没有表达出来的其他状态。

5.8 拆开来看

5.8.1 `branch on` 块的结构

```
branch on <sourceNode>.output.<field> {
  <value1> -> <targetNode1>
  <value2> -> <targetNode2>
  otherwise -> <defaultNode>
```

```
}
```

部分	含义
`branch on`	打开条件路由块的关键字对
`<sourceNode>.output.<field>`	一个 节点输出路径 表达式——引擎在运行时读取它来做决策
`<value> -> <target>`	一个 **case**：如果条件等于 `<value>`，就调度 `<target>`
`otherwise -> <target>`	默认 **case**：如果没有其他 case 匹配，就调度这个目标

5.8.2 Case 匹配规则

编译器为每个 case 构建一个 `Predicate<Object>`，采用类型宽松比较：

Case 字面量	运行时匹配方式
`true` / `false`	如果运行时值是 `Boolean`，直接 `==`。否则，使用不区分大小写的 `String.valueOf()` 比较。
`"someString"`	字符串相等，或对运行时值调用 `String.valueOf()` 后比较。
`42`（数字）	如果运行时值是 `Number`，进行 `doubleValue()` 比较。否则，不匹配。

Case 按 声明顺序 求值——第一个匹配的胜出。

5.8.3 布尔分支

最简单的分支使用布尔值。来自 conformance fixture `boolean.bloge`：

```
graph g {
  node a : Op {}
  node b : Op {}
  node c : Op {}
  branch on a.output.approved {
    true -> b
    false -> c
  }
}
```

如果 `a.output.approved` 是 `true`，节点 `b` 运行，`c` 被跳过。如果是 `false`，`c` 运行，`b` 被跳过。

5.8.4 `otherwise` 兜底

来自 conformance fixture `otherwise.bloge`：

```
graph g {
  node a : Op {}
  node b : Op {}
  branch on a.output.status {
    otherwise -> b
```

```
  }
}
```

一个只有 `otherwise` 的分支意味着”无论值是什么，都调度 `b`”。这种写法不常见但合法——在原型开发阶段可以用作占位。

5.8.5 `SKIPPED` 的含义

当引擎选中一个分支目标时，其余所有目标节点在 `NodeStatus` 枚举中被标记为 `SKIPPED`。一个被跳过的节点：

- 永远不会执行 其 operator。
- 不产出任何输出 ——任何引用被跳过节点输出的下游节点将无法获得该数据。
- 在 `GraphResult` 和执行监听器中显示为 `SKIPPED`。

5.9 包容式分支：当多条路径都需要运行时

以上所有内容描述的都是排他式分支（XOR 分裂）——默认行为，只有第一个匹配的分支触发，其余全部跳过。`branch on` 也支持包容式模式（OR 分裂），可以让所有匹配的分支并行触发。

5.9.1 为什么需要它

假设有一笔贷款申请需要沿着多个独立维度进行审查。在评估风险标志之后，该申请可能同时需要人工审核、合规检查以及定价审查。使用排他式分支你只能选一个——但真实的工作流需要全部同时进行。

5.9.2 XOR 分裂 vs OR 分裂一览

	排他式（**XOR** 分裂，默认）	包容式（**OR** 分裂，**`inclusive`**）
触发的分支数	恰好一个——第一个匹配	所有匹配的，并行执行
`otherwise` 触发时机	没有显式 case 匹配时	零个 分支匹配时
未匹配的目标	标记为 `SKIPPED`	标记为 `SKIPPED`
DSL 关键字	`branch on ... { }`	`branch mode=inclusive on ... { }`
Java API	`.branch().on().when()...`	`.branch().on().inclusive().when()...`

5.9.3 DSL 语法

在 `on` 前设置 `mode=inclusive`：

```
branch mode=inclusive on assessFlags.reviewFlags {
  "NEEDS_REVIEW"     -> manualReview
  "NEEDS_COMPLIANCE" -> complianceReview
  "NEEDS_PRICING"    -> pricingReview
  otherwise               -> autoApprove
}
```

如果 assessFlags.reviewFlags 同时包含 "NEEDS_REVIEW" 和 "NEEDS_PRICING"，manualReview 和 pricingReview 都会并行运行，而 complianceReview 被跳过。otherwise 目标（autoApprove）仅在没有任何 case 匹配时才运行。

5.9.4 Java API

在 BranchBuilder 上使用 .inclusive() 方法：

```
graph.branch("assessFlags")
     .on("reviewFlags")
     .inclusive()                          // 启用 OR 分裂
     .when(v -> v instanceof List<?> l
               && l.contains("NEEDS_REVIEW"), "manualReview")
     .when(v -> v instanceof List<?> l
               && l.contains("NEEDS_COMPLIANCE"), "complianceReview")
     .when(v -> v instanceof List<?> l
               && l.contains("NEEDS_PRICING"), "pricingReview")
     .otherwise("autoApprove");
```

底层实现是将 Edge.Conditional 的 inclusive 字段设为 true。

5.9.5 完整示例——多标志审批

参见 ch05/multi-flag-approval.bloge 获取完整 graph。关键流程：

1. fetchApplication 获取贷款申请。
2. assessFlags 分析风险并产出审查标志列表。
3. 包容式分支 扇出到每条匹配的审查轨道：
   - "NEEDS_REVIEW" → manualReview
   - "NEEDS_COMPLIANCE" → complianceReview
   - "NEEDS_PRICING" → pricingReview
4. 如果标志列表为空，otherwise 路由到 autoApprove。

由于分支是包容式的，单笔申请可以同时触发两条甚至全部三条审查轨道。

5.9.6 何时使用包容式 vs 排他式

使用排他式（默认）的场景	使用包容式的场景
恰好只有一个结果适用（批准 / 拒绝 / 延期）	多个独立的结果可以同时适用
条件是单值枚举或布尔值	条件是一个集合或标志位的位掩码
你需要第一个匹配短路的语义	你需要向所有匹配的下游节点扇出

5.9.7 otherwise 行为差异

- 排他式：当没有显式 case 匹配时 otherwise 触发——它充当兜底默认值。
- 包容式： otherwise 仅在 零个 case 匹配时触发。如果至少有一个 case 匹配了，即使其他 case 没有匹配，otherwise 也会被跳过。

5.10 当你需要一个总是运行的节点——upstream_policy = ALL_RESOLVED

默认调度规则是：节点只在 至少一个上游 **COMPLETED** 时才运行。这对普通的 fan-in 来说是对的，但有些节点故意挂在分支下游，无论走的是哪条路径都需要运行——例如：

- 一个 结果记录器，记录哪条路径胜出
- 一个 清理 节点，在成功路径和失败路径之后都要跑
- 一个 指标 收集器，不关心业务结果

如果你写 `depends_on = [branchTargetA, branchTargetB, branchTargetC]`，而 分支只选中了其中一条路径，其他几个节点就是 `SKIPPED` ——于是你的记录器 也会被 `SKIPPED`，因为它没有任何 `COMPLETED` 的上游。

`upstream_policy = ALL_RESOLVED` 是告诉引擎:" 只要我的 所有 上游都到达 终态 (包括 `SKIPPED` 和 `CANCELLED`)，就启动我。" 这里的语义是" 等待所有上游 **resolve**(终结)"，而不是" 等待所有上游成功"。

```
node recordOutcome : RecordOutcomeOperator {
  depends_on      = [assignVipAgent, assignNormalAgent, autoResolve]
  upstream_policy = ALL_RESOLVED
  input {
    // SKIPPED 的上游没有 output —— 搭配安全导航操作符
    vip    = assignVipAgent.output?.agentId    ?? null
    normal = assignNormalAgent.output?.agentId ?? null
    auto   = autoResolve.output?.ticketId      ?? null
  }
}
```

```
.node("recordOutcome", new RecordOutcomeOperator())
    .dependsOn("assignVipAgent", "assignNormalAgent", "autoResolve")
    .upstreamPolicy(UpstreamPolicy.ALL_RESOLVED)
```

> 小测验: *upstream_policy = ALL_RESOLVED* 和 " 只把 *recordOutcome* 接在那条实际运行的路径下游 " 的差别在哪里? 提示: 想想你要画几条边，以及 加上第四条分支时会发生什么。

5.11 常见陷阱

5.11.1 [X] 对非输出表达式做分支

```
// WRONG — branch condition must be a node output path
branch on ctx.priority {
  "high" -> fastTrack
  otherwise -> normalQueue
}
```

编译器要求条件表达式是一个 节点输出路径（或 transform 字段路径）。你不能直接对 context 值做分支。如

果需要根据 context 数据进行路由，可以引入一个节点或 transform 将该值暴露为输出：

```
// CORRECT — use a transform to expose the context value
transform routing {
  priority = ctx.priority
}

branch on routing.priority {
  "high" -> fastTrack
  otherwise -> normalQueue
}
```

5.11.2 [X] 对开放值集合遗漏 otherwise

```
// DANGEROUS — what if decision is "review"?
branch on makeDecision.output.decision {
  "approved" -> approveLoan
  "rejected" -> rejectLoan
}
```

如果 makeDecision 返回 "review"，没有 case 匹配，没有任何目标节点会运行。当可能值的集合不是穷举已知时，始终应包含 otherwise：

```
// SAFE
branch on makeDecision.output.decision {
  "approved" -> approveLoan
  "rejected" -> rejectLoan
  otherwise  -> manualReview
}
```

5.11.3 [X] 混淆 branch on 与 when

branch on 和 when 共享箭头语法（->）和 otherwise，但它们是 完全不同 的：

	branch on	when
所属层面	控制面——决定哪些 节点 运行	数据面——决定一个 字段 取什么值
出现位置	graph 主体，顶层	表达式内部（input binding 或 transform）
效果	激活一个节点，跳过其余	产出一个值

当你只需要计算一个值时，不要用 branch on，应该使用 when。

5.11.4 [X] 混淆包容式分支与并行执行

```
// 错误假设 —— 这仍然是排他式分支!
branch on classifyRisk.level {
  "HIGH"   -> escalate
```

```
  "MEDIUM" -> review
  otherwise     -> autoApprove
}
```

在其他位置增加并发并不会让一个排他式分支“并行运行”。没有 `mode=inclusive` 时，只有第一个匹配的 case 触发。如果需要多个分支同时触发，必须在 branch 块上显式设置 `mode=inclusive`：

```
// 正确 —— mode=inclusive 启用 OR 分裂
branch mode=inclusive on assessFlags.reviewFlags {
  "NEEDS_REVIEW"     -> manualReview
  "NEEDS_COMPLIANCE" -> complianceReview
  otherwise               -> autoApprove
}
```

还要注意：包容式分支适用于条件值可以匹配多个 case 的情况（例如标志列表）。如果条件是一个单值字段（如字符串枚举），包容式和排他式的行为完全相同——此时优先使用排他式以保持清晰。

5.11.5 [X] 把 `branch on` 当决策表用

`branch on` 适合路由——决定哪些下游节点运行。当你用它做分类或策略查找时，结构很快会变别扭：一组输入本来只是要映射到一个输出值，却被拆成了多个分支目标节点。

这个味道很好识别：每个分支目标存在的唯一目的，就是返回同一种 shape 的结果。

```
// 别扭 —— 只是为了分类一个值，却创建了一组路由节点
branch on creditScore.output.band {
  "platinum" -> assignPlatinumTier
  "gold"     -> assignGoldTier
  otherwise  -> rejectApplication
}
```

这种设计会带来三个问题：

1. 输出分散。结果散落在多个节点输出路径里，而不是一个稳定位置。
2. 多匹配语义很弱。`branch on` 可以选一个臂或扇出多个臂，但很难清晰表达“收集所有匹配规则并返回它们”。
3. 缺少策略 **lint**。工具无法警告规则集不完备，也无法指出两条策略行冲突。

当逻辑本质是一张规则表时，使用 `decision_table`：

```
graph creditScreening {
  decision_table credit_tier(score = applicant.output.score) hit=first -> String {
    rule (score: score >= 750) -> "platinum"
    rule (score: 680 <= score < 750) -> "gold"
    otherwise -> "rejected"
  }
}
```

无论以后增加多少规则，结果始终在 `credit_tier.output.value`。

需求	优先选择
将执行路由到一个或多个下游节点	`branch on`
把输入分类成一个值	`decision_table hit=first` 或 `hit=unique`
允许多条规则命中，但要求输出一致	`decision_table hit=any`
返回所有匹配结果	`decision_table hit=collect`

完整语法、命中策略、lint 规则和错误码见附录 F ——决策表。第 9 章展示 `decision-table/missing-otherwise` 与 `decision-table/collect-otherwise` 的 lint 实战，第 34 章则把这个陷阱做成了动手练习。

5.12 常见故障

5.12.1 缺少 **otherwise** 且出现意外值

```
branch on makeDecision.output.decision {
  "approved" -> approveLoan
  "rejected" -> rejectLoan
}
```

如果 `makeDecision` 返回 `"review"`，就不会有任何 case 命中：

```
GraphResult {
  status: COMPLETED,
  nodeStatuses: {
    approveLoan: SKIPPED,
    rejectLoan:  SKIPPED
  }
}
```

Graph 会完成，但 没有任何下游工作真正发生。任何依赖 `approveLoan` 或 `rejectLoan` 输出的节点都会拿不到数据。

修复：加上 `otherwise -> manualReview`，确保至少总有一条分支会被激活。

5.13 引导式重写

看一下完整的 `loan-approval.bloge` 示例。它并行运行四项风险检查，聚合结果，然后做分支：

```
branch on makeDecision.output.decision {
  "approved"  -> approveLoan
  "rejected"  -> rejectLoan
  otherwise   -> manualReview
}
```

思考以下问题：

1. 分支之前有多少个节点运行？数一数：`fetchApplication`，然后四个并行检查（`checkCredit`、`detectFraud`、`verifyIncome`、`checkBlacklist`），然后 `aggregateRisk`，然后 `makeDecision` ——总共七个节点。
2. 如果 **`makeDecision`** 返回 **`"escalated"`** 会怎样？ `otherwise` case 捕获它并路由到 `manualReview`。
3. **`approveLoan`** 和 **`rejectLoan`** 能同时运行吗？不能——只有一个分支目标会被选中。
4. **`riskSummary`** **`transform`** 与分支的关系是什么？它将上游输出投影用于审计日志。因为它不依赖 `makeDecision`，可以独立解析——但它对选择哪条分支路径没有任何影响。

现在尝试自己修改这个 graph：

- 添加第四个 case：`"deferred" -> deferApplication`。
- 声明新的 `deferApplication` 节点，使用 `DeferApplicationOperator`，依赖 `makeDecision`。
- 预测：当决策为 `"deferred"` 时，其他三个目标节点会怎样？

5.14 脑力检查

1. 用什么关键字对打开条件路由块？（*`branch on`*。）
2. `branch on` 后面必须跟什么类型的表达式？（节点输出路径表达式，如 *`nodeId.output.field`*，或 *transform* 字段路径，如 *`transformId.field`*。）
3. 如果一个分支有三个 case 但没有 `otherwise`，当没有任何 case 匹配时会怎样？（没有目标节点运行——所有目标都被跳过。）
4. 在同一次 graph 运行中，使用默认（排他式）`branch on` 时，两个分支目标能否同时执行？（不能——每次排他式 *`branch on`* 求值只选中一个目标。）
5. `branch on` 和 `when` 的区别是什么？（*`branch on`* 是控制面路由，决定哪些节点运行。*`when`* 是数据面表达式，决定一个字段取什么值。）
6. 设计题：如果一个 graph 里有两个互不依赖的功能都需要分支，能不能在同一个 graph 中声明两个独立的 `branch on` 块？如果可以，引擎如何处理它们？（可以。每个 *`branch on`* 都会在其源节点完成时独立求值，彼此互不干扰。这在复杂 *workflow* 中很常见，因为多个决策往往分布在不同的并行路径上。）
7. 你需要给 `branch on` 添加什么关键字使其变为包容式（OR 分裂）分支？`otherwise` 的行为有何不同？（在条件表达式后添加 *`inclusive`*。在包容式分支中，*`otherwise`* 仅在零个 *case* 匹配时触发；而在排他式分支中，当没有显式 *case* 匹配时就会触发。）

5.15 练习

1. 在编辑器中打开 `ticket-routing.bloge`。
2. 添加第四个处理程序节点，名为 `escalateToManager`，使用 `EscalateToManagerOperator`。它应依赖 `classifyPriority`，并将 `fetchCustomer.output.id` 和 `analyzeSentiment.output` 作为输入。
3. 在分支中添加一个新 **`case`**：`"critical" -> escalateToManager`。
4. 画出每种可能优先级值的执行流程：
   - `"vip"` → 哪个节点运行？
   - `"critical"` → 哪个节点运行？

- `"normal"` → 哪个节点运行?
- `"unknown"` → 哪个节点运行?（提示：`otherwise`。）

5. 加分项：把分支改写成只使用 `otherwise`（不带任何显式 case）。这对 graph 的行为意味着什么？这样做有用吗?

5.16 实验验收卡

- 预期与观察：两个 XOR Case 交换 EXECUTED 与 SKIPPED。
- 失败与恢复：让所有条件都不匹配；修复阈值或 otherwise。
- 证明边界：只证明给定输入的分支选择，不证明全部业务边界。
- 练习合同：批准与拒绝 Case；只改风险阈值；交付状态矩阵；每个 Case 恰选一条路径即停止。

5.17 回顾

- `branch on` 是 控制面路由——它决定哪些下游节点执行、哪些被跳过。
- 条件必须是 节点输出路径 或 **transform** 字段路径 表达式。
- Case 使用 类型宽松比较，并按 声明顺序 测试——第一个匹配的胜出。
- `otherwise` 是 默认 **case**。当可能值的集合不是穷举已知时，始终应包含它。
- 未被选中的目标节点被标记为 **`SKIPPED`** ——它们永远不会执行，也不产出任何输出。
- `branch on`（控制面）和 `when`（数据面）共享语法风格，但在 语义上完全不同。用 `branch on` 路由执行；用 `when` 计算值。
- 包容式分支（`branch mode=inclusive on ...`）是 OR 分裂：所有匹配的分支并行触发。当条件可以同时匹配多个 case 时使用（例如标志列表）。`otherwise` 仅在零个 case 匹配时触发。

5.18 下一步

在第 6 章——韧性设计里，你会学习如何在节点上声明 timeout、retry 和 fallback 策略——让你的 graph 保持可靠，而不必在 operator 代码里堆砌基础设施逻辑。

5.19 参考链接

- DSL 规范 ——正式语法与编译规则
- 核心架构 ——`Edge.Conditional`、`NodeStatus.SKIPPED` 与引擎调度
- `ticket-routing.bloge` ——完整示例源码
- `loan-approval.bloge` ——完整示例源码
- `boolean.bloge` ——conformance fixture
- `otherwise.bloge` ——conformance fixture
- `ch05/multi-flag-approval.bloge` ——包容式分支示例

章节
06

第二阶段——可维护编写

第 6 章——韧性设计

分支、韧性、Operator 边界、DSL 与评审反馈

目标：把可读流程变成另一位工程师能评审的改动。

承诺：读完本章后，你会知道 *BLOGE* 内建的韧性链——***retry → timeout → fallback*** ——如何保护每个节点，以及 ***compensation*** 如何在 *graph* 下游失败时撤销已完成的工作。

6.1 学习目标

1. 解释韧性求值顺序，以及 为什么 retry 包裹每次尝试的 timeout，而 fallback 是最终降级手段。
2. 在节点上配置 `retry`、`timeout` 和 `fallback`，并预测 operator 抛异常时的运行时行为。
3. 根据场景选择合适的 `backoff` 策略（`fixed`、`exponential`、`jitter`）。
4. 声明 `compensate` 块来表达尽力而为的、节点级别的 saga 清理——并理解为什么这 不是 分布式事务回滚。

6.2 前置条件

- 第 5 章——做决策的分支

6.3 源示例

文件	展示内容
`error-handling-showcase.bloge`	支付节点上的 retry + timeout + fallback，附带汇总步骤
`retry-with-backoff.bloge`	基于循环的 retry，自定义指数退避并携带状态
`order-saga.bloge`	三节点 saga，每个节点带有 compensation
`retry-fallback-output.bloge`	Conformance fixture：最小节点上的 retry + fallback

文件	展示内容
`node-compensation.bloge`	Conformance fixture：单节点 compensation
`compensation.bloge`	Conformance fixture：两节点 compensation 链
ADR-007	Retry → Timeout → Fallback 顺序的架构决策记录

6.4 为什么这很重要

在第 1–4 章中，你在一些节点上见过 `retry`、`timeout` 和 `fallback`。它们看起来只是简单的注解。但实际上，这些策略的 执行顺序 决定了你的 graph 是干净地恢复还是掩盖了真正的故障。

考虑一个在高负载下不稳定的支付节点：

- 如果 fallback 在重试耗尽 之前 就运行了，你会在第一次小波动时就无声降级——没有向任何人收费，却生成了虚假的” 人工审核” 记录。
- 如果一个 timeout 包裹了 所有 重试尝试，一次慢请求就可能吃掉整个时间预算，让后续的重试根本没有时间执行。
- 如果 compensation 试图充当分布式事务，一次 compensation 失败就会阻断清理流程，让系统处于半完成状态。

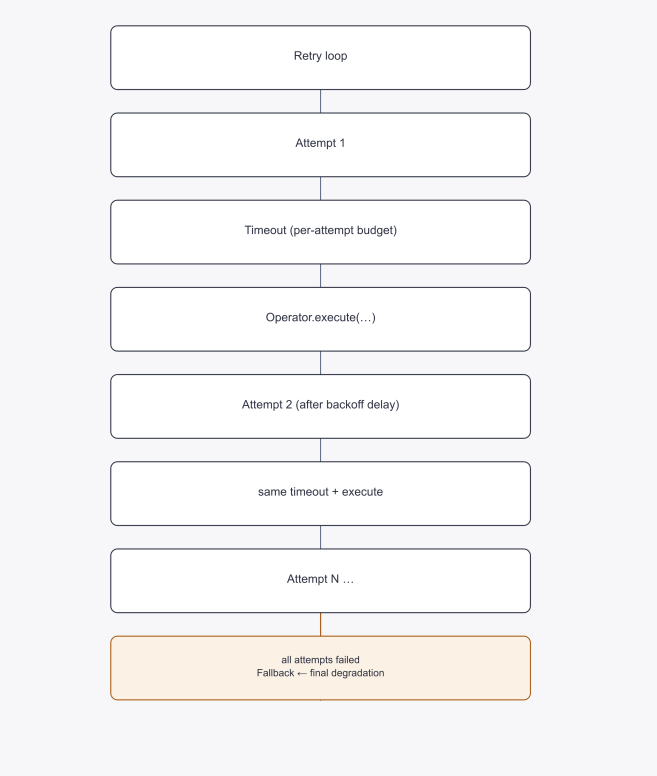
BLOGE 用两条清晰的规则消除了这些陷阱：

1. **Retry** 包裹每次尝试的 **timeout**；**fallback** 是最终兜底。
2. **Compensation** 是尽力而为的节点级清理，不是原子回滚。

一旦内化了这两条规则，你就可以把韧性声明 写在它所保护的节点旁边，并信任引擎正确地执行它。

6.5 心智模型

6.5.1 韧性链



从里向外读这张图：

1. **Operator** ——你的业务代码先执行。
2. **Timeout** ——每次 单独的尝试 都有自己的计时器。如果超时，该次尝试以 `OperatorTimeoutException` 失败。
3. **Retry** ——失败后，引擎等待退避时长然后重试，最多 `attempts` 次。每次新尝试都会启动一个全新的 timeout。
4. **Fallback** ——只有当 所有 尝试都失败（包括超时）之后，引擎才会替换为 fallback 值。这是最后的手段，不是提前退出。

关键洞察：总体挂钟时间可能超过单个 `timeout` 值，因为每次重试尝试都有自己的时间预算。例如，`timeout = 1s` 配合 `attempts: 3` 最长可能耗时 $3 \times 1\,s$ + 退避延迟。

6.5.2 Compensation ——撤销路径

同样的链条，如果用 Mermaid 渲染，会长这样：

当下游节点失败且 graph 无法继续时，引擎按 反向拓扑序 遍历已 完成 的节点，并调用每个节点的 `compensate` operator（如果声明了的话）。Compensation 是 尽力而为的：如果 `ReleaseInventory` 本身抛出异常，错误会被记录到 `CompensationResult` 中，`RefundPayment` 仍然会运行。这是 saga 风格的清理，不是分布式事务回滚。

6.6 第一个可运行示例

以下是完整的 `error-handling-showcase.bloge`：

```
graph errorHandlingShowcase {
  node chargePayment : ChargePaymentOperator {
    input {
      orderId         = ctx.orderId
      amount          = ctx.amount
      simulateFailure = ctx.simulateFailure
    }
    retry = { attempts: 1, backoff: 25ms, strategy: exponential }
    timeout = 1s
    fallback = { approved: false, note: "Gateway unavailable; queued for manual review" }
  }

  node summarizeOutcome : SummarizeOutcomeOperator {
    depends_on = [chargePayment]
    input {
      approved = chargePayment.output.approved
      note     = chargePayment.output.note
    }
    timeout = 1s
  }
}
```

走一遍 `ChargePaymentOperator` 一直抛异常的场景：

1. 第 **1** 次尝试 开始。operator 有 **1 s** 的时间来完成。它抛异常了。
2. 引擎等待 **25 ms**（基于基准值的指数退避），然后启动 第 **2** 次尝试，带有全新的 1 s timeout。它又抛异常了。
3. 配置的单次重试已耗尽。引擎激活 **fallback**：`{ approved: false, note: "Gateway unavailable; queued for manual review" }`。
4. `summarizeOutcome` 通过 `chargePayment.output.approved` 和 `chargePayment.output.note` 接收 fallback 输出——graph 以降级结果成功完成。

> 注意：*`summarizeOutcome` 没有 retry 或 fallback*。如果它失败了，*graph* 就会失败。韧性只在需要的地方声明。

6.7 看似安全的失败：超时之后，钱已经扣了

设想 `ChargeCardOperator` 带着 `idempotencyKey=order-42` 发出请求。支付服务 在 430 ms 完成扣款，但响应被延迟。500 ms 时 graph 超时，并把 node 标为 失败。“node 失败”和“扣款没有发生”是两个不同事实。

6.7.1 把执行控制与 **effect** 事实分开

观察	它能说明什么	它不能说明什么
node 状态是带 timeout 的 `FAILED`	引擎按策略停止等待	远端系统已经回滚
支付账本中存在 `order-42`	扣款 effect 已存在	graph 收到了响应
重试返回相同 charge id	provider 遵守了幂等键	provider 的所有动作都可安全重试

RC1 基础探针把这个问题缩到最小：伪账本先递增，Operator 响应再延迟；随后 node 进入 `FAILED`，账本计数仍为 `1`。这不是支付网络仿真，它只证明一个 更窄的结论：引擎 timeout 不能作为 effect 未发生的证据。

6.7.2 先设计恢复对话，再增加 **retry**

对有 effect 的 Operator，增加 retry 前先定义三件事：

1. 从业务操作推导稳定幂等键，不要使用 attempt number。
2. 提供一次 reconciliation 查询，在响应不确定时回答“这个 effect 是否发生”。
3. 当业务确实需要撤销时，提供具有独立失败处理的 compensation 动作。

retry 只有在这个合同内才安全。compensation 是一个新的 effect，不是时间 机器，而且它可以独立失败。引擎控制 attempt 和时间；integration 负责外部 账本的事实。

6.8 拆开来看

6.8.1 **Retry** 与退避策略

```
retry = { attempts: 3, backoff: 200ms, strategy: exponential }
```

字段	含义
attempts	初始执行之 后 的最大重试次数（因此 attempts: 3 意味着最多 4 次总执行）
backoff	重试之间的基准延迟
strategy	延迟如何增长：fixed、exponential 或 jitter

三种策略：

策略	行为	适用场景
fixed	每次延迟相同	简单场景，延迟可预测
exponential	延迟每次翻倍（25 ms → 50 ms → 100 ms ⋯）	对过载服务施加背压
jitter	指数退避加随机扰动	避免并行节点的惊群重试

6.8.2 Fallback

```
fallback = { approved: false, reason: "credit service unavailable" }
```

Fallback 是一个 静态值，当所有重试尝试和 timeout 都耗尽后替换节点的输出。它是最终降级手段——graph 继续运行，但使用的是替代数据。

Conformance fixture `retry-fallback-output.bloge` 展示了最小形式：

```
graph g {
  node a : Op {
    retry = { attempts: 3, backoff: 200ms, strategy: exponential }
    fallback = { score: 0 }
    output {
      result: String
    }
  }
}
```

6.8.3 Compensation

Compensation 声明在产生副作用的节点 内部：

```
node reserveInventory : ReserveInventoryOperator {
  output {
    reservationId: String
  }

  compensate : ReleaseInventoryOperator {
```

```
    input {
      reservationId = reserveInventory.output.reservationId
    }
  }
}
```

（摘自 *order-saga.bloge*。）

关键规则：

- 只有已完成的节点才会被 **compensate**。如果 `chargePayment` 从未完成，就没有什么需要撤销的。
- 反向拓扑序。引擎先 compensate 最后完成的节点，然后向前回溯。
- 尽力而为。一次 compensation 失败会被记录到 `CompensationResult` 中，但不会阻止后续的 compensation。
- 不是分布式事务。没有两阶段提交。每个 `compensate` 块独立运行。设计你的 compensation operator 时应确保幂等性。

Conformance fixture `compensation.bloge` 展示了两个节点的 compensation 链：

```
graph g {
  node pay : PaymentOp {
    input { amount = ctx.amount }
    compensate : ReversePayment {
      input { txId = pay.output.transactionId }
    }
  }
  node ship : ShipOp {
    depends_on = [pay]
    compensate : CancelShipment {
      input { trackingId = ship.output.trackingId }
    }
  }
}
```

如果 `ship` 失败，引擎会 compensate `pay`（唯一 已完成 的节点），运行 `ReversePayment` 并传入原始交易 ID。

6.9 常见陷阱

6.9.1 [X] 把 compensation 当成分布式事务回滚

```
// WRONG mental model — expecting atomic all-or-nothing
node reserveInventory : ReserveInventoryOperator {
  compensate : ReleaseInventoryOperator {
    input { reservationId = reserveInventory.output.reservationId }
  }
}
```

```
node chargePayment : ChargePaymentOperator {
  depends_on = [reserveInventory]
  compensate : RefundPaymentOperator {
    input { chargeId = chargePayment.output.chargeId }
  }
}
```

如果 RefundPaymentOperator 抛异常，库存预留 仍然 会被释放——引擎不会停止 compensate。如果 ReleaseInventoryOperator 也抛异常，你最终会有 两个 compensation 失败，而不是一次干净的回滚。

设计 **compensation operator** 时应确保幂等性，并能容忍部分失败。积极记录日志。使用 GraphResult 中的 CompensationResult 列表来检测和告警 compensation 失败。

6.10 常见故障

6.10.1 所有重试都耗尽，而且没有 **fallback**

```
node chargePayment : ChargePaymentOperator {
  retry = { attempts: 3, backoff: 200ms, strategy: exponential }
  timeout = 1s
  // No fallback declared
}
```

如果每一次尝试都失败，graph 会直接停下来：

```
GraphResult {
  status: FAILED,
  failedNode: "chargePayment",
  error: "chargePayment failed after the initial attempt and all configured retries"
}
```

所有下游节点都会被标记为 NOT_REACHED。

修复：如果业务上允许降级，就声明 fallback；如果确实属于不可恢复故障，就让失败继续向上传播。

6.11 引导式重写

从 order-saga graph 出发，添加韧性策略：

```
graph orderSaga {
  node reserveInventory : ReserveInventoryOperator {
    timeout = 3s
    retry = { attempts: 2, backoff: 100ms, strategy: jitter }
    output {
      reservationId: String
    }
```

```
    compensate : ReleaseInventoryOperator {
      input {
        reservationId = reserveInventory.output.reservationId
      }
    }
  }

  node chargePayment : ChargePaymentOperator {
    depends_on = [reserveInventory]
    timeout = 5s
    retry = { attempts: 3, backoff: 200ms, strategy: exponential }
    fallback = { chargeId: null, status: "payment_deferred" }
    output {
      chargeId: String
    }

    compensate : RefundPaymentOperator {
      input {
        chargeId = chargePayment.output.chargeId
      }
    }
  }

  node shipOrder : ShipOrderOperator {
    depends_on = [chargePayment]
    timeout = 10s
    input {
      failShipping = ctx.failShipping
    }
  }
}
```

思考以下问题：

1. 如果 `chargePayment` 在所有 3 次重试后仍然失败，会怎样？（使用 *fallback* 值 *`{ chargeId: null, status : "payment_deferred" }`*。*`shipOrder`* 接收它。没有 *compensation* 运行，因为没有节点导致 *graph* 失败——*fallback* 保住了它。）
2. 如果 `shipOrder` 随后失败（没有 fallback），哪些 compensation operator 会运行？（*`RefundPaymentOperator`* 和 *`ReleaseInventoryOperator`* 都会运行，按此顺序——反向拓扑序。*`chargePayment`* 以 *fallback* 值完成，*`reserveInventory`* 正常完成。）
3. 单独 `chargePayment` 的最坏情况挂钟时间是多少？（初始尝试：*5 s* + *3* 次重试 × *(*退避 + *5 s)*。以 *200 ms* 起始的指数退避：*≈ 5 + 0.2 + 5 + 0.4 + 5 + 0.8 + 5 = ~21.4 s*。）

6.12 脑力检查

1. BLOGE 中的韧性求值顺序是什么？（*Retry* → *Timeout*（每次尝试）→ *Fallback*。）
2. 为什么每次重试尝试都有自己的 timeout，而不是共享一个全局 timeout？（这样一次慢请求就不会消耗掉整个时间预算，导致后续重试没有时间执行。参见 *ADR-007*。）
3. Fallback 何时激活？（只有在所有重试尝试——包括它们各自的 *timeout*——都耗尽之后。它是最终降级手段。）
4. 一次 compensation 失败是否会阻止引擎 compensate 更早的节点？（不会。*Compensation* 是尽力而为的。每次失败都被记录到 `CompensationResult` 中，引擎继续处理下一个节点。）
5. 引擎按什么顺序 compensate 节点？（反向拓扑序，且仅针对主执行已成功完成的节点。）
6. 设计题：你的 graph 会并行调用三个外部支付网关，每个节点都配置了 retry + fallback。如果三个节点最后都走到了 fallback，下游聚合节点会同时收到三份“降级结果”。你会如何设计这个聚合节点，让它能区分真实数据和 fallback 数据？（聚合 *operator* 应该检查每个输入里的标记或哨兵值——例如 *fallback* 带上的 `"status": "degraded"` 字段——然后决定是带着部分数据继续完成，还是升级为 *graph* 级故障。真正要设计的是：当“全部都降级”时，产品应该算成功完成，还是应该显式失败。）

6.13 练习

1. 打开 `error-handling-showcase.bloge`。
   - 将 retry 策略从 `exponential` 改为 `jitter`，并将 `attempts` 增加到 `3`。
   - 预测：如果每次尝试在失败前恰好耗时 800 ms，1 s 的 timeout 会触发吗？（不会——*800 ms < 1 s*，所以每次尝试是因为 *operator* 异常而失败，而不是因为 *timeout*。）
2. 打开 `order-saga.bloge`。
   - 为 `chargePayment` 添加 `timeout = 2s` 和 `retry = { attempts: 1, backoff: 50ms, strategy: fixed }`。
   - 为 `reserveInventory` 添加 fallback，返回 `{ reservationId: "NONE" }`。
   - 追踪当 `reserveInventory` 的 operator 总是抛异常时会发生什么：`ReleaseInventoryOperator` 会运行吗？（不会——*fallback* 保住了 `reserveInventory`，所以 *graph* 继续执行。*Compensation* 只有在 *graph* 最终失败时才会触发。）
3. 编写一个新 graph，包含三个独立的服务调用，各自使用不同的退避策略（`fixed`、`exponential`、`jitter`）。添加一个扇入节点，要求三者全部完成。给其中两个配置 fallback。预测：在什么条件下 graph 会失败？在什么条件下会优雅降级？

6.14 实验验收卡

- 预期与观察： graph 可在超时后 fallback，但外部扣款可能已发生。
- 失败与恢复：把 effect 响应延迟到 timeout 后；用幂等键和 reconciliation 恢复。
- 证明边界：证明 node completion 与外部 effect 是两份事实，不证明 exactly-once。
- 练习合同：扣款探针；只改响应延迟；交付 graph 状态和账本计数；两份事实分列即停止。

6.15 回顾

- BLOGE 按严格顺序应用韧性策略：**Retry → Timeout → Fallback**。每次重试尝试都有自己的 timeout 预算；fallback 是所有尝试耗尽后的最终降级手段。
- 三种 退避策略 控制重试之间的延迟：`fixed`、`exponential` 和 `jitter`。
- **Fallback** 是一个静态替代值——graph 以降级数据继续运行而不是失败。
- **Compensation**（`compensate` 块）提供尽力而为的、节点级别的 saga 清理。引擎按反向拓扑序运行 compensation，仅针对已完成的节点，并捕获失败而不中断后续处理。
- Compensation 不是 分布式事务回滚。设计 compensation operator 时应确保幂等性。
- 韧性声明在 节点上，而不是埋在业务代码里——引擎拥有 retry 循环、timeout 计时器和 compensation 遍历的控制权。

6.16 下一步

在第 7 章——设计良好的 Operator里，你会学习如何编写可测试、可组合、并且与你目前所见的韧性和调度特性良好协作的 operator。

6.17 参考链接

- ADR-007 —韧性顺序 ——Retry → Timeout → Fallback 的架构决策
- 核心架构 ——引擎内部机制与调度设计
- DSL 规范 ——韧性与 compensation 语法
- `error-handling-showcase.bloge` ——完整示例源码
- `retry-with-backoff.bloge` ——基于循环的 retry 示例
- `order-saga.bloge` ——saga compensation 示例
- `retry-fallback-output.bloge` ——conformance fixture
- `node-compensation.bloge` ——conformance fixture
- `compensation.bloge` ——conformance fixture

章节

07

第二阶段——可维护编写

第 7 章——设计良好的 Operator

分支、韧性、Operator 边界、DSL 与评审反馈

目标：把可读流程变成另一位工程师能评审的改动。

承诺：读完本章后，你会知道如何判断哪些逻辑该放进 *operator*、如何声明它的 *schema* 和行为契约，以及如何避免最常见的设计错误——把每一段逻辑都做成节点。

7.1 学习目标

1. 运用单一能力原则来判断一段逻辑是该做成 operator，还是应该留在编排层。
2. 把 operator 归入三层复用金字塔：基础设施 → 能力 → 领域。
3. 声明 输入/输出 **schema**，让编译器、工具链和下游节点能更安全地校验数据。
4. 设置 幂等性 和 副作用 契约，帮助框架做出更安全的重试与调度决策。
5. 识别——并消除——那些本应是 transform 或 input binding 表达式的琐碎数据整形 operator。

7.2 前置条件

- 第 4 章——流动的数据（特别是 transform 和数据转换分层模型）
- 第 6 章——韧性设计（了解节点契约如何影响重试和超时策略）

7.3 源示例

文件	展示内容
`Operator.java`	核心 `@FunctionalInterface` ——`execute`、`idempotency`、`sideEffectType`
`OperatorContext.java`	引擎传给每个 operator 的只读 record

文件	展示内容
`SchemaAware.java`	可选接口，用于显式声明输入/输出 schema
`Idempotency.java`	`IDEMPOTENT`、`NOT_IDEMPOTENT`、`UNKNOWN`
`SideEffectType.java`	`READ_ONLY`、`WRITE`、`EXTERNAL_CALL`、`MIXED`
`OperatorMeta.java`	用于标注 layer、tags、version、owner 的注解
`OperatorLayer.java`	`INFRASTRUCTURE`、`CAPABILITY`、`DOMAIN`
`Operator Design Specification`	完整的规范文档（八个章节）

7.4 为什么这很重要

前几章教会了你如何构建 **graph** 和传递数据。但一个 BLOGE 系统的质量，最终取决于它的 **operator** 质量——它们是每个 graph 的基本构件。

如果 operator 的边界划错了，你会看到：

- 膨胀的 **graph**：一半节点只是单行字段重命名，本来用 transform 就能搞定。
- 巨石 **operator**：把三个互不相关的 I/O 调用捆在一起，既无法单独重试，也无法单独观测。
- 不透明的契约：在读源码之前，谁也不知道一个 operator 的输出长什么样。

把边界划对了，你就能解锁：

- 复用 ——一个边界清晰的 operator 可以服务于数十个 graph。
- 可观测性 ——每个 operator 都有自己的延迟、错误率和重试指标。
- 安全演进 ——类型化 schema 让破坏性变更在编译时就能被发现，而不是凌晨两点。

7.5 心智模型

7.5.1 单一能力原则

一个 operator 应该封装一项可独立度量的业务能力。问自己五个问题：

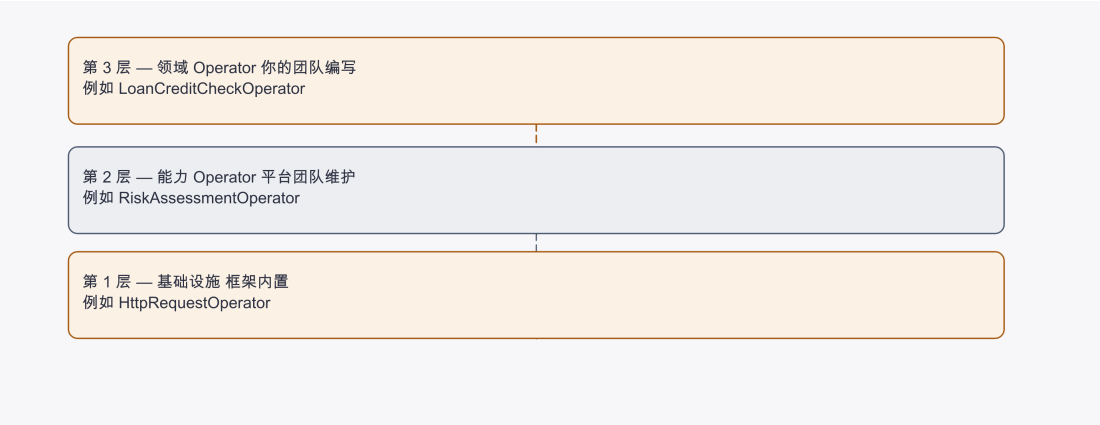
7.5.2 数据转换的四个层次

并非每段逻辑都应该放进 operator。BLOGE 提供了四个层次，各有不同的成本：

层次	工具	成本	何时使用
路由	`input { userId = fetchUser.output.id }`	零	传递或重命名字段
轻量 **transform**	`input { name = concat(a.output.first, " ", a.output.last) }`	零	拼接、类型转换、空值填充
结构适配	`transform orderSummary { …}`	零（虚拟节点）	多字段重组，且被多个下游节点复用
业务逻辑	`node …: Operator { …}`	完整成本（有调度、计时、可重试）	领域规则、I/O、副作用

经验法则：如果移除该逻辑只会破坏 数据格式兼容性——用 transform。如果移除它会改变 业务含义——它必须是 operator。

7.5.3 三层复用金字塔



每一层通过 `@OperatorMeta(layer = ⋯)` 使用 `OperatorLayer` 枚举（`INFRASTRUCTURE`、`CAPABILITY`、`DOMAIN`）来标注。用它在代码和文档中明确 operator 的复用范围。

7.6 第一个可运行示例

下面设计一个从外部服务获取用户的 operator。类型化 I/O、schema 和行为契约会明确它与框架之间的边界。

7.6.1 步骤 1 —定义类型化的输入和输出 record

```
public record FetchUserInput(String userId) {
    public FetchUserInput {
        if (userId == null || userId.isBlank())
            throw new IllegalArgumentException("userId must not be blank");
    }
}

public record FetchUserOutput(String id, String name, String email, int vipLevel) {}
```

Record 为你提供了不可变性、紧凑构造函数中的校验，以及通过 `SchemaIntrospector` 实现的自动 schema 内省。

7.6.2 步骤 2 —实现 operator

这 43 行实现保留完整，让 metadata、外部调用、幂等性、effect 分类和两端 schema 能够作为一份 Operator 合同一起审阅。

```
@OperatorMeta(
    layer = OperatorLayer.INFRASTRUCTURE,
    tags  = {"user", "query"},
    description = "Fetch a user by ID from the user service",
    owner = "user-platform",
    since = "1.0.0"
)
```

```
public class FetchUserOperator
        implements Operator<FetchUserInput, FetchUserOutput>, SchemaAware {

    private final UserServiceClient client;

    public FetchUserOperator(UserServiceClient client) {
        this.client = client;
    }

    @Override
    public FetchUserOutput execute(FetchUserInput input, OperatorContext ctx)
            throws Exception {
        // 业务数据通过 `input` 传入，而不是通过 `ctx`。
        return client.fetchUser(input.userId());
    }

    @Override
    public Idempotency idempotency() {
        return Idempotency.IDEMPOTENT;          // 可安全重试
    }

    @Override
    public SideEffectType sideEffectType() {
        return SideEffectType.EXTERNAL_CALL;    // 跨越服务边界
    }

    @Override
    public SchemaDescriptor inputSchema() {
        return SchemaIntrospector.introspect(FetchUserInput.class);
    }

    @Override
    public SchemaDescriptor outputSchema() {
        return SchemaIntrospector.introspect(FetchUserOutput.class);
    }
}
```

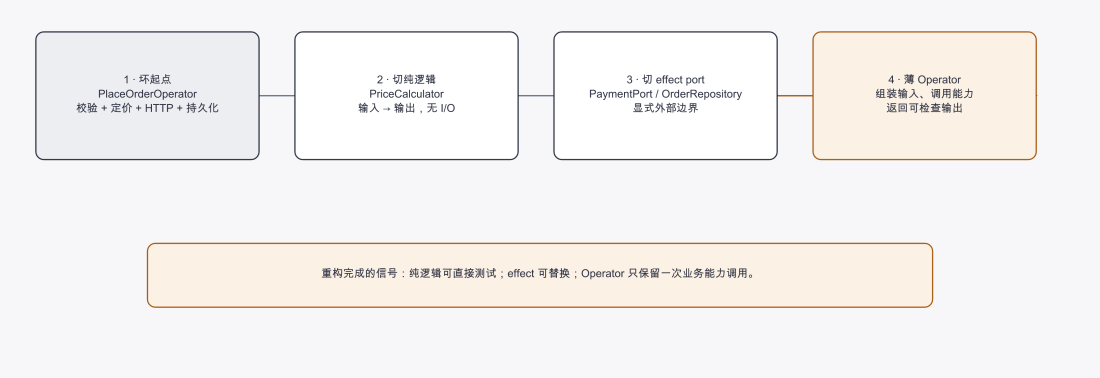
注意四点：

1. **`@FunctionalInterface`**——核心契约只有一个方法：`O execute(I input, OperatorContext ctx)`。不需要继承抽象基类，也不需要先学习生命周期钩子。
2. 类型化泛型——`Operator<FetchUserInput, FetchUserOutput>` 为编译器和引擎提供了一个具体的类型对。即使你没有实现 `SchemaAware`，框架也会自动内省泛型参数来派生 `StructuredSchema`。

3. 行为默认值 ——`idempotency()` 返回 `IDEMPOTENT`，因此框架可以认为重试是安全的。`sideEffectType()` 返回 `EXTERNAL_CALL`，表明该 operator 跨越了服务边界，应该在 graph 中配合显式的韧性设置。
4. **`OperatorContext`** 是只读的元数据 ——它携带 `nodeId`、`graphName`、`retryAttempt`、`executionId` 和 `timeSource`。所有 业务 数据都必须通过 `input` 传入。

7.7 用四刀重构一个坏 Operator

识别好 Operator 最快的方法，是修一个坏的。先看 `PlaceOrderOperator`：它校验 字段、计算折扣、调用支付、写入订单、发送邮件，还负责翻译所有异常。这个 名字描述的是一段工作流，而不是一项能力。



7.7.1 第 1、2 刀——让纯决策回到普通代码

把校验与计价提取成输入输出显式的函数或领域对象。它们不需要 `OperatorContext`、retry，也不需要 node。此时可以直接用表格驱动的单元 测试覆盖折扣边界，不必启动 graph。

7.7.2 第 3、4 刀——暴露 **effect**，再把 **adapter** 做薄

引入 `PaymentPort`、`OrderRepository` 之类的窄接口。最终 Operator 只组装 类型化输入、调用一项业务能力，并返回可检查结果：

层	负责什么	测试接缝
纯领域逻辑	校验与计价决策	输入 → 输出
effect port	支付或持久化协议	fake/受控 port 加调用证据
Operator adapter	graph 输入输出与能力调用	`GraphTestRunner` node 结果
graph	顺序、分支、retry、timeout	Scenario 与状态断言

完成标准不是“类变小了”，而是失败边界变清楚：计价错误由纯测试发现；支付 协议错误由 port contract 发现；依赖错误由 graph Scenario 发现。`SchemaAware`、元数据和 context API 用来描述这项能力，不能挽救一个仍然 混合多种责任的能力。

7.8 拆开来看

7.8.1 Operator 接口

来自 Operator.java:

```
@FunctionalInterface
public interface Operator<I, O> {
    O execute(I input, OperatorContext ctx) throws Exception;

    default Idempotency idempotency()     { return Idempotency.UNKNOWN; }
    default SideEffectType sideEffectType() { return SideEffectType.MIXED; }
}
```

成员	用途
execute(I, OperatorContext)	你必须实现的唯一方法——接收组装好的输入并返回输出
idempotency()	告知引擎重试是否安全（IDEMPOTENT）、是否禁止（NOT_IDEMPOTENT）、还是未指定（UNKNOWN）
sideEffectType()	告知框架该 operator 有何种副作用，以便调度和韧性决策保持显式

7.8.2 OperatorContext ——引擎告诉你的信息

来自 OperatorContext.java:

```
public record OperatorContext(
    String nodeId,
    String graphName,
    GraphContext graphContext,   // 请求作用域；每个节点的写入相互隔离
    int retryAttempt,           // 0 = 首次执行
    String executionId,
    TimeSource timeSource       // 永不为 null；默认 SystemTimeSource.INSTANCE
) {}
```

graphContext 用于请求级元数据（trace ID、tenant ID）。它不是存放业务数据的地方——业务数据必须通过类型化的 I 输入传递。

7.8.3 timeSource ——operator 中的可测试时间

timeSource 字段让每个 operator 都可以使用引擎的时钟，而不需要硬编码 System.currentTimeMillis() 或 Instant.now()。

为什么重要：如果 operator 直接读取墙钟，测试就无法控制它看到的时间。依赖真实时间的重试延迟断言既慢又脆弱。有了 timeSource，测试夹具注入 ManualTimeSource，operator 就能免费获得确定性时间。

经验法则：只要 operator 需要当前时间，就调用 ctx.timeSource().now() 而不是 Instant.now()。

```
@Override
public AuditOutput execute(AuditInput input, OperatorContext ctx) throws Exception {
    Instant timestamp = ctx.timeSource().now();   // [OK] 可测试
    // Instant timestamp = Instant.now();         // [X] 不可测试
```

```
    return auditService.record(input.action(), timestamp);
}
```

在生产环境中，引擎提供 SystemTimeSource.INSTANCE，它委托给 Instant.now() 和 Thread.sleep()。在测试中你用 ManualTimeSource 替换：

```
var manualTime = new ManualTimeSource(Instant.parse("2025-01-15T10:00:00Z"));

var ctx = OperatorContext.builder()
        .nodeId("audit")
        .graphName("compliance")
        .graphContext(new GraphContext())
        .timeSource(manualTime)        // 注入测试时钟
        .build();

var result = new AuditOperator(auditService).execute(input, ctx);

// 断言 operator 使用了注入的时间，而不是墙钟
assertEquals(Instant.parse("2025-01-15T10:00:00Z"), result.recordedAt());
```

这与第 18 章 中 TestGraphEngine 使用的 ManualTimeSource 相同。区别在于：这里你在 **operator** 层做单元测试，而第 18 章在 **graph** 层做集成测试。

7.8.4 SchemaAware ——显式 schema 声明

来自 SchemaAware.java：

```
public interface SchemaAware {
    default SchemaDescriptor inputSchema()  { return OpaqueSchema.INSTANCE; }
    default SchemaDescriptor outputSchema() { return OpaqueSchema.INSTANCE; }
}
```

实现 SchemaAware 是可选的。如果你不实现，框架会回退到对泛型类型参数 I 和 O 的自动内省。但推荐显式声明，因为：

- 当泛型被擦除为 Map<String, Object> 时，显式声明仍能保留 schema 信息。
- 你可以给单个字段附加描述和约束注解。
- 它能让下游 input binding 获得编译期路径校验。

在 Java operator 层，DefaultOperatorRegistry 会先检查 SchemaAware；如果你没有实现该接口，它会回退到通过 SchemaIntrospector 对 operator 泛型类型参数的内省。

7.8.5 @OperatorMeta ——目录元数据

来自 OperatorMeta.java：

```
public @interface OperatorMeta {
    OperatorLayer layer() default OperatorLayer.DOMAIN;
    String[] tags()       default {};
    String version()      default "";
```

```
    String description()  default "";
    String owner()        default "";
    String since()        default "";

    // LLM 工具元数据 ——由 bloge-agent-ext(第 17 章) 消费
    String promptHint()             default "";
    String usageExample()           default "";
    String constraintsDescription() default "";
}
```

每个 operator 都应该 至少声明 `layer`、`description` 和 `owner`。即使运行时行为取决于 operator 契约本身，这些元数据也能让意图和归属对读者和工具链保持可见。

最后三个字段是 **LLM** 工具描述。当一个 operator 作为工具暴露给 `agent` 节点 (第 17 章) 时，`bloge-agent-ext` 读取它们来构造发给模型的 function spec:

- `promptHint` ——LLM 在工具列表里看到的一行说明。
- `usageExample` ——给模型锚定行为的具体调用示例。
- `constraintsDescription` ——前置条件、副作用、速率限制。

它们也会进入 `operator-metadata.json`(第 9 章),Studio 的 operator 面板会展示同样的提示。

7.9 常见陷阱

7.9.1 [X] 为纯数据整形创建 operator

```
// 不要这样做——这个 operator 的工作不需要独立调度，可以由 transform 完成
public class FormatOrderSummaryOperator
        implements Operator<Map<String, Object>, Map<String, Object>> {

    @Override
    public Map<String, Object> execute(Map<String, Object> input,
                                       OperatorContext ctx) {
        return Map.of(
            "customerName", input.get("name"),
            "total",        input.get("price")
        );
    }
}
```

这个 operator ” 勉强” 可以独立测试，但它没有有意义的指标、没有副作用、没有领域知识、也没有复用价值。它在单一能力原则的五项检查中挂了四项。

改用 **transform**:

```
transform orderSummary {
```

```
  customerName = fetchUser.output.name
  total        = calcPrice.output.total
}
```

Transform 不增加 Operator 调度成本——不会触发 operator 调用、没有超时追踪、没有重试开销——但表达式求值本身仍有计算成本，而且整形逻辑直接可见于 graph 本身。

如何在现有 **graph** 中发现这个陷阱：

- Operator 的 `execute` 方法里没有 `if`、没有 I/O、没有外部调用。
- Operator 类没有构造函数依赖（没有 service client、没有 repository）。
- 把 operator 替换成 `return input;` 的透传，也不会改变业务结果。

如果以上任何一条成立，删掉这个 **operator**，改用 **transform** 或 **input binding** 表达式。

7.9.2 [X] 把决策表编码为普通算子

有时候，一个 operator 看起来像业务逻辑，但实现其实只是一张“条件映射到输出值”的表格——没有 I/O、没有重试、没有状态。这不是一项可独立度量的能力，而是一张藏在 Java 代码里的策略表。

以下情况请选择 `decision_table`：

- 逻辑是输入的纯函数（无副作用，无外部调用）。
- 条件是可枚举的——你可以列出所有情况，编译器或 lint 可以在列表不完备时发出警告。
- 审计线索很重要——你希望运行时有结构化错误码，并受 `decision-table/missing-otherwise` lint 规则保护。

以下情况继续使用 `Operator`：

- 节点需要执行 **I/O**（HTTP 调用、数据库查询、消息发布）或存在副作用。
- 你需要框架管理的重试、超时或降级策略。
- 逻辑涉及可变状态或无法用规则谓词表达的协调逻辑。

完整语法参考和所有错误码，请见附录 F ——决策表。

7.10 引导式重写

下面这个 graph 有两个设计问题——一个 operator 太琐碎，另一个太臃肿：

```
graph invoiceProcess {

  node fetchOrder : FetchOrderOperator {
    input { orderId = ctx.orderId }
    timeout = 3s
  }

  // 问题 1：琐碎的数据整形——应该用 transform
  node formatAddress : FormatAddressOperator {
    input {
```

```
      street = fetchOrder.output.address.street
      city   = fetchOrder.output.address.city
      zip    = fetchOrder.output.address.zip
    }
  }

  // 问题 2：巨石——把支付和通知捆在一个 operator 里
  node processAndNotify : ProcessAndNotifyOperator {
    depends_on = [fetchOrder, formatAddress]
    input {
      order   = fetchOrder.output
      address = formatAddress.output
    }
    timeout = 10s
  }
}
```

重写后：

```
graph invoiceProcess {

  node fetchOrder : FetchOrderOperator {
    input { orderId = ctx.orderId }
    timeout = 3s
  }

  // 修复 1：用 transform 替换琐碎的 operator
  transform formattedAddress {
    street = fetchOrder.output.address.street
    city   = fetchOrder.output.address.city
    zip    = fetchOrder.output.address.zip
    full   = concat(fetchOrder.output.address.street, ", ",
                    fetchOrder.output.address.city, " ",
                    fetchOrder.output.address.zip)
  }

  // 修复 2：把巨石拆成两个独立的 operator
  node processPayment : ProcessPaymentOperator {
    depends_on = [fetchOrder]
    input {
      order   = fetchOrder.output
      address = formattedAddress.full
    }
    timeout = 5s
    retry   = { attempts: 2, backoff: 500ms, strategy: exponential }
  }
```

```
  node sendNotification : SendNotificationOperator {
    depends_on = [fetchOrder]
    input {
      email   = fetchOrder.output.customerEmail
      orderId = fetchOrder.output.id
    }
    timeout = 3s
    fallback = { sent: false, reason: "notification service unavailable" }
  }
}
```

改进之处：

之前	之后	为什么重要
`FormatAddressOperator`（完整节点）	`transform formattedAddress`	零调度成本；纯数据整形不需要 timeout/retry
`ProcessAndNotifyOperator`（一个臃肿节点）	`processPayment` + `sendNotification`	独立的 SLA、独立的重试策略，而且它们可以并行执行

7.11 脑力检查

1. 说出单一能力原则中五项检查中的三项。（可独立测试、可独立观测、可独立替换、可独立复用、有清晰的领域语义。）
2. 一个 operator 的 `execute` 方法通过 `ctx.graphContext().get("items")` 获取业务数据。这有什么问题？（业务数据必须通过类型化输入 `I` 传入，而不是通过 *graph context*。*Graph context* 用于请求级元数据，如 *trace ID*。）
3. 你有一个只做字符串拼接的 operator。它该存在吗？（不该——字符串拼接是轻量 *transform*。在 *input binding* 或 *transform* 中使用 *`concat(a.output.first, " ", a.output.last)`* 即可。）
4. `SideEffectType.EXTERNAL_CALL` 告诉框架什么？（该 *operator* 跨越了外部服务边界，因此读者和集成方应将其视为有副作用的调用，并在 *graph* 中配合显式的 *timeout/retry* 策略。）
5. 如果一个 operator 没有实现 `SchemaAware`，框架如何推导它的 schema？（通过 *`SchemaIntrospector`* 对泛型类型参数 `I` 和 `O` 进行内省。）

7.12 练习

1. 审计一个现有 **graph**。打开 `order-process.bloge`。对每个节点应用五项检查。有没有节点应该改成 transform？写下你的推理过程。
2. 从零设计一个新 **operator**。假设你要实现一个 `CreditCheckOperator`，它调用外部信用评分 API。
   - 定义 `CreditCheckInput` 和 `CreditCheckOutput` 为 Java record。

- 实现 `Operator<CreditCheckInput, CreditCheckOutput>`。
- 实现 `SchemaAware` 并提供显式 schema。
- 覆写 `idempotency()` → `IDEMPOTENT`（只读查询）。
- 覆写 `sideEffectType()` → `EXTERNAL_CALL`。
- 用 `@OperatorMeta(layer = CAPABILITY, ...)` 标注。

3. **Schema** 校验检查。写好 record 后，在单元测试中使用 `SchemaIntrospector.introspect(CreditCheckOutput.class)`，验证返回的 `StructuredSchema` 包含你期望的字段。
4. 重构一个不好的 **operator**。写一个 `FormatCurrencyOperator`，接收 `Number` 返回 `String`。然后删掉它，用 transform 或绑定表达式替代。哪种方式更简洁？

7.13 实验验收卡

- 预期与观察： Operator 拆成纯逻辑、port、adapter 和 graph policy 后，失败接缝独立。
- 失败与恢复：让 adapter 返回传输错误；只替换 adapter 或 fallback。
- 证明边界：只证明责任边界可测试，不证明外部系统可靠。
- 练习合同：坏 service；每次只切一层；交付责任图和失败测试；失败归到唯一 owner 即停止。

7.14 回顾

- 一个 operator 应该封装一项可独立度量的业务能力——单一能力原则。
- 数据转换有四个层次：路由（input binding）、轻量 transform（表达式）、结构适配（transform 块）和业务逻辑（operator 节点）。使用成本最低的那个。
- 三层复用金字塔（基础设施 → 能力 → 领域）按作用范围对 operator 进行分类；用 `@OperatorMeta(layer = …)` 标注它们。
- 类型化 **I/O record** + `SchemaAware` 带来编译期校验、schema 感知的工具链和更安全的 schema 演进。
- 行为契约 ——`idempotency()` 和 `sideEffectType()` ——为框架和你的团队在重试、调度和韧性决策上提供更明确的指引。
- **`OperatorContext`** 用于元数据，而非业务数据。所有领域值通过类型化输入传递。使用 `ctx.timeSource().now()` 替代 `Instant.now()` 以获得可测试的时间。
- 最常见的错误是为纯数据整形创建 operator。如果没有 **I/O**、没有副作用、也没有领域规则——用 **transform**。

7.15 下一步

在第 8 章——把个人 DSL 草稿变成团队资产中，你将把这些想法带回 `.bloge` 编写层——将 graph 级 schema、文档注释、transform、分支和 operator 契约组合成既易读又易演进的文件。

7.16 参考链接

- Operator 设计规范 ——完整的规范文档（边界规则、schema 治理、复杂度限制、命名标准）

- `Operator.java` ——核心 operator 接口
- `OperatorContext.java` ——执行上下文 record
- `SchemaAware.java` ——显式 schema 声明接口
- `SchemaDescriptor.java` ——密封 schema 类型层次
- `SchemaIntrospector.java` ——从 Java 类型自动派生 schema
- `OperatorMeta.java` ——目录元数据注解
- `OperatorLayer.java` ——复用层枚举
- `SideEffectType.java` ——副作用分类
- `Idempotency.java` ——幂等性级别
- 核心架构 ——引擎调度与韧性内部机制

章节
08

第二阶段——可维护编写

第 8 章——把个人 DSL 草稿变成团队资产

分支、韧性、Operator 边界、DSL 与评审反馈

目标：把可读流程变成另一位工程师能评审的改动。

本章承诺：你会跟着订单 *graph* 完成五次小提交，亲眼看到一个“能编译”的个人草稿如何逐步获得契约、可诊断性、意图和复用边界，最终变成队友敢于修改的 `.bloge` 文件。

8.1 学习目标

学完本章，你能够：

1. 解释为什么零编译错误不等于 DSL 已经可维护。
2. 用 graph input、字段类型和稳定 output 把隐含假设变成团队合同。
3. 从 `invalid-expression-path` 找到错误发生在“哪条路径与哪个 schema 不一致”。
4. 判断一段说明应成为 `///` 文档注释，还是应该留在代码审查讨论中。
5. 用 import alias 把共享流程放到明确边界后面，而不是复制节点。

8.2 这一章只增加一个变量

前七章已经让订单流程能够获取用户、读取商品、计算价格并决定是否创建订单。本章不增加新的业务行为，只增加一个团队现实：另一个人下周要安全地修改这份 **graph**。

这会改变“完成”的定义。个人草稿只需作者记得 `ctx.customerEmail` 从哪里来；团队资产必须让编译器、编辑器和审查者都能追踪这个假设。

8.3 开场：绿色草稿为什么让审查者不安

第一版订单摘要只有几行：

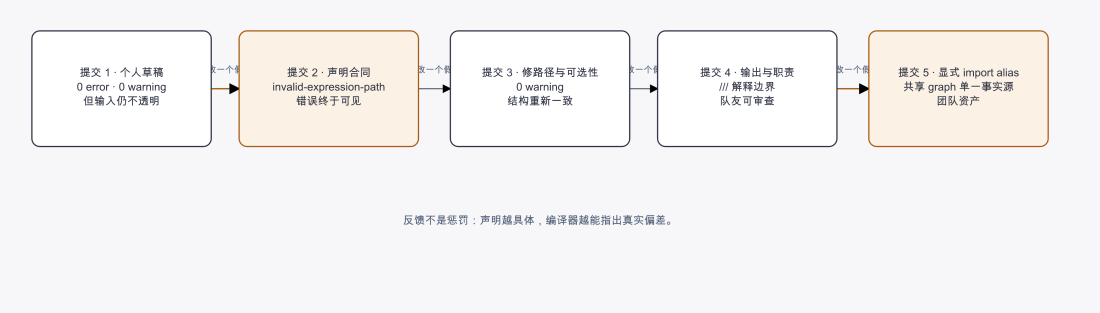
```
graph orderDraft {
  transform orderSummary {
```

```
    userId = ctx.userId
    customerEmail = ctx.customerEmail
  }
}
```

先预测：如果它编译时是 `errors=0 warnings=0`，下面哪句话成立?

说法	是否能由零告警推出
语法可被当前编译器接受	能
调用方一定提供 `customerEmail`	不能
该字段允许缺失	不能
下游知道输出形状	不能
六个月后重命名会被及时发现	不能

绿色只说明编译器没有发现违反已声明合同的地方。没有合同，很多错误根本没有成为可检查的问题。



8.4 先跑完整轨迹

本书提供五个独立快照和一个 RC1 probe。先从固定 BLOGE submodule 安装 core 与 DSL，然后运行：

```
cd submodule/bloge
mvn -pl bloge-core,bloge-dsl -am -DskipTests install

cd ../../probes/ch08-dsl
mvn test
```

2026-09-14 在 BLOGE commit `cc38fbe5` 上的关键输出：

```
commit-1 graph=orderDraft         errors=0 warnings=0 firstRule=none
commit-2 graph=orderContractDraft errors=0 warnings=1 firstRule=invalid-expression-path
commit-3 graph=orderTypedDraft    errors=0 warnings=0 firstRule=none
commit-4 graph=orderReviewable    errors=0 warnings=0 firstRule=none
commit-5 graph=orderTeamAsset     errors=0 warnings=0 firstRule=none
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

注意这不是“提交 2 比提交 1 更差”。提交 2 让一个原本不可见的假设变成可检查合同，因此编译器终于有能力指出问题。

8.5 提交 1：先让业务路径可见

第一版只保留 `userId` 和邮箱投影。它适合在个人探索中回答：“这种数据流写法是否表达得通？”

它不适合进入团队主干，因为 `ctx` 是不透明的。编译器不能区分“调用方真的提供一个顶层 `customerEmail`”与“作者把 `customer.email` 记错了”。

本次提交的审查结论不是“失败”，而是：

```
syntax accepted
input contract = opaque
safe refactor boundary = unknown
```

8.6 提交 2：声明输入合同，主动让错误浮出水面

第二次提交声明订单请求的真实形状：

```
input: OrderRequest

schema OrderRequest {
  userId: String
  customer: {
    name: String
    email: String?
  }
}
```

但摘要仍读取 `ctx.customerEmail`。编译器现在知道顶层只有 `userId` 和 `customer`，于是报告：

```
warnings=1
firstRule=invalid-expression-path
Path 'ctx.customerEmail' — field 'customerEmail' not found in graph input schema
```

这个 warning 的价值不只是“有一个拼写错了”。它给出了三段定位信息：

- 根对象是 graph input，也就是 `ctx`；
- 出错路径是 `customerEmail`；
- 对照合同是 `OrderRequest`。

恢复动作不是关闭 schema validation，而是决定现实数据究竟是顶层邮箱，还是嵌套客户邮箱，然后让 schema 与表达式一致。

8.7 提交 3：修正路径，并把可选性写进类型

本例的现实答案是 `customer.email`，且有些客户没有邮箱：

```
transform customer {
  name: String = ctx.customer.name
  email: String? = ctx.customer.email
}
```

```
transform orderSummary {
  userId: String = ctx.userId
  customerEmail: String? = customer.output.email
}
```

再次编译变为零告警。这里同时建立了两条不同关系：

- 路径关系：`orderSummary` 通过 `customer.output.email` 依赖上游 transform。
- 类型关系：`String?` 明确“缺失是合同允许的”，不再由作者口头解释。

不要为了清除 warning 把所有字段改成可选。可选性是业务事实：若订单通知必须有邮箱，就应保留 `String`，让缺失尽早失败。

8.8 提交 4：让审查者知道这段 DSL 为什么存在

路径正确仍不等于意图清楚。第四次提交增加 graph output 合同，并给 transform 一句职责说明：

```
output: OrderGraphResult

schema OrderGraphResult {
  orderSummary: OrderSummary
}

/// Projects the boundary fields; it does not make a business decision.
transform orderSummary { ... }
```

`///` 会进入 AST 并与紧随其后的元素关联，可供编辑器或文档工具读取。普通 `//` 只给源码读者看，解析器不会把它当成合同元数据。

好的文档注释回答“责任和边界”，而不是把语法再念一遍：

注释	审查价值
`/// Creates order summary`	低：名字已经说过
`/// Projects boundary fields; makes no business decision`	高：说明变换与决策的边界
`/// This is important`	低：没有可检查含义

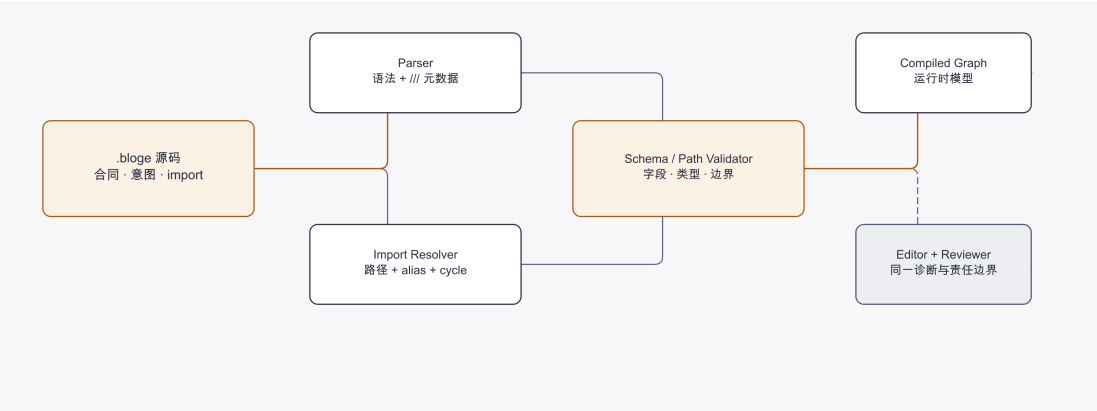
8.9 提交 5：共享行为不再复制，用 alias 建立边界

订单流程需要支付能力。复制支付节点看似最快，却让重试、金额校验和状态映射出现多个事实源。第五次提交把支付流程抽成独立 graph，并显式命名：

```
import "./payment-flow" as paymentFlow

node payment : paymentFlow {
  input {
    orderId = ctx.orderId
```

```
    amount = ctx.amount
  }
}
```

alias 不是缩写糖。它是当前文件对导入能力使用的稳定名字。编译器需要完成四件事：解析相对路径、编译被导入 graph、用其 input schema 检查当前 binding、把其 output schema 暴露给下游。



如果导入文件不存在、形成循环，或两个 import 使用同一 alias，编译应该在建立 graph 前停止。不要在运行时才猜某个 alias 指向什么。

8.10 原理：声明越具体，反馈越有意义

五次提交背后是一条反馈管线：

```
source
  → lexer/parser：语法是否合法
  → import resolver：引用指向哪份 graph
  → schema resolver：字段和类型意味着什么
  → path/boundary validator：连接是否兼容
  → compiled graph：运行时可执行模型
```

前一层失败时，后一层结论就没有基础。语法通过不能替代 import 解析；import 能找到文件也不能替代 schema 兼容；schema 兼容更不能证明业务规则正确。

这也是团队 DSL 的核心价值：把原本留在作者脑中的假设，逐层变成机器与人都能复核的反馈。

8.11 失败实验：错误 alias 不是重试问题

保持两个被导入文件不变，只让它们都使用 `as paymentFlow`。RC1 会产生稳定诊断 `duplicate-import-alias`。恢复动作是给不同能力不同名字，或删除重复事实源；重跑运行时没有意义，因为 graph 尚未成功建立。

同理，`invalid-expression-path` 的恢复动作是修合同或路径，不是给节点增加 retry。编译期反馈与运行期韧性处理的是不同世界。

8.12 这一章刻意不展开的内容

完整 schema 类型目录、bare import 的 alias 推导、import resolver SPI、schema evolution 和 `ValidatedSchema` 策略都属于参考层，不应打断五次提交的主线。

特别是 `ValidatedSchema`：它是 Java runtime 中给 schema 附加 `STRICT / COERCE / LENIENT` 验证策略的包装器，不是本章 `.bloge` 文件必须掌握的新语法。当前 DSL 编译器在结构兼容检查时会先读取其底层 expected schema；运行时策略细节将在参考附录中集中说明。

8.13 换到现实世界：从个人菜谱到餐厅出品卡

个人做菜时，“适量盐”可能够用；餐厅交接需要原料规格、过敏原、出品形状、关键原因和共用酱汁版本。对应关系是：

餐厅交接	团队 **DSL**
原料清单	graph input schema
成品规格	graph/output schema
工序卡	nodes/transforms 与依赖
为什么这样做	`///` 责任注释
共用酱汁配方	imported graph + alias
试做反馈	compiler/linter diagnostics

规范不是为了把菜谱写长，而是为了让新厨师能安全修改并知道何时偏离标准。

8.14 轮到你：做五次可审查提交

选一份你维护的 `.bloge` 文件，每次只做一种改变：

1. 保存当前可解析草稿，并记录尚未声明的输入假设。
2. 声明 graph input，让编译器暴露真实路径问题；不要先修。
3. 只修路径和可选性，记录 warning 的变化。
4. 给稳定输出与关键 transform 加职责边界，不写语法复述。
5. 只抽取一个确定共享的子流程，用显式 alias 连接。

最终提交一张审查卡：每次改了哪个假设、获得什么诊断、谁会因此更容易审查。若一次提交同时改业务分支、类型和导入，就拆开重做；否则无法知道哪项变化产生了反馈。

8.15 实验验收卡

- 预期与观察：五次提交依次暴露路径错误、修复类型并用 alias 收口，最终诊断 0/0。
- 失败与恢复：引入重复 alias 或错误路径；修复唯一声明后重跑 parser。
- 证明边界：证明 DSL 反馈可定位，不证明表达式是正确业务答案。
- 练习合同：团队 DSL；每次只改一个提交；交付五段诊断；最终 0 error/0 warning 且变化可解释即停止。

8.16 回顾

- 零告警只覆盖已声明合同；opaque 输入会制造虚假的安静。
- schema 的价值是让错误路径、缺失字段和可选性进入反馈循环。
- `///` 记录责任和边界，不能替代准确命名与结构。
- import alias 把共享 graph 变成显式依赖，不是复制节点的快捷方式。
- 语法、引用、结构合同和业务正确性是逐层递进的结论，不能互相替代。

下一章只增加一个变量：第 9 章——工具链工作流先把可维护 DSL 接入可重复的编辑、lint 与评审循环；第 10 章再引入子图边界。

8.17 精确事实入口

- `DslFeedbackLoopProbeTest.java` ——五次提交的可执行反馈。
- `01-personal-draft.bloge` 到 `05-team-asset.bloge` ——五个完整快照。
- `order-process-v6.bloge` ——远端 examples 的订单综合示例。
- `DslCompiler.java` ——编译与诊断事实。
- `FileSystemGraphResolver.java` ——相对 import 解析。
- `ValidatedSchema.java` ——运行时验证策略包装器。

章节
09

第二阶段——可维护编写

第 9 章——工具链工作流

分支、韧性、Operator 边界、DSL 与评审反馈

目标：把可读流程变成另一位工程师能评审的改动。

承诺：读完本章后，你将理解 *BLOGE* 工具链的 *TypeScript* 编写路径与 *Java* 执行路径如何协同，并且知道怎样为 `.bloge` 文件建立高效编写循环，在代码审查和上线前捕获错误。

9.1 学习目标

1. 理解 两条实现路径：TypeScript `bloge-lang` 服务 LSP/Studio 编写，Java `bloge-dsl` 与 `bloge-lint` 服务 runtime/CI。
2. 了解 诊断信息如何流转：从词法分析器、解析器、编译器和 lint 规则，经过 LSP，最终变成编辑器中的红色波浪线。
3. 使用 保存时格式化 通过 DSL 代码生成器保持 `.bloge` 文件的一致性。
4. 在 CI 中运行 **`bloge-lint` CLI**，并理解它的 Java 规则集与编辑器诊断有重叠但不完全相同。
5. 追踪 **`operator-metadata.json`** 如何从 `bloge-maven-plugin` 进入 BLOGE Studio 的 operator 面板，再用分屏视图在 DSL 文本和画布之间往返编辑。

9.2 前置条件

- 第 8 章——把个人 DSL 草稿变成团队资产 ——你需要熟悉一个完整的 `.bloge` 文件。
- 第 2 章——你的第一个 Graph ——你应该会读取编译和执行结果。
- 对 VS Code 或 IntelliJ IDEA 有基本了解。

9.3 源示例

文件	展示内容
`bloge-lang/README.md`	TypeScript 编写核心：词法分析器、解析器、编译器、代码生成器、schema
`bloge-lsp/README.md`	LSP 服务器能力与架构
`bloge-lsp/src/server.ts`	LSP 服务器入口——能力注册
`bloge-lsp/src/document-manager.ts`	五阶段分析流水线（lex → parse → compile → lint → index）
`bloge-lsp/src/features/hover.ts`	关键字、内置函数、节点、schema 和持续时间的悬停文档
`bloge-lsp/src/features/completion.ts`	上下文感知的自动补全
`bloge-lsp/src/features/formatting.ts`	通过 `DslCodeGenerator` 实现的整文档格式化
`bloge-lsp/src/lint/lint-rule.ts`	`LintRule` 接口和 `LintRunner`
`bloge-vscode/src/extension.ts`	VS Code 扩展入口——LSP 客户端接线
`bloge-vscode/syntaxes/bloge.tmLanguage.json`	用于语法高亮的 TextMate 语法
`bloge-intellij/README.md`	IntelliJ 插件——双模式架构（PSI + 可选 LSP 叠加层）
`bloge-studio/README.md`	可视化 DAG 编辑器——React 19 + React Flow
`bloge-studio/public/operator-metadata.json`	内置 operator 目录（名称、描述、I/O schema）
`bloge-maven-plugin/README.md`	从 `@BlogeOperator` 类生成 `operator-metadata.json` 的 Maven 插件
`bloge-lint/README.md`	Java CLI linter ——规则、`.blogerc.json` 配置、退出码
`ticket-routing.bloge`	本章通篇使用的示例
`missing-timeout.bloge`	被 lint 捕获的反模式

9.4 为什么这很重要

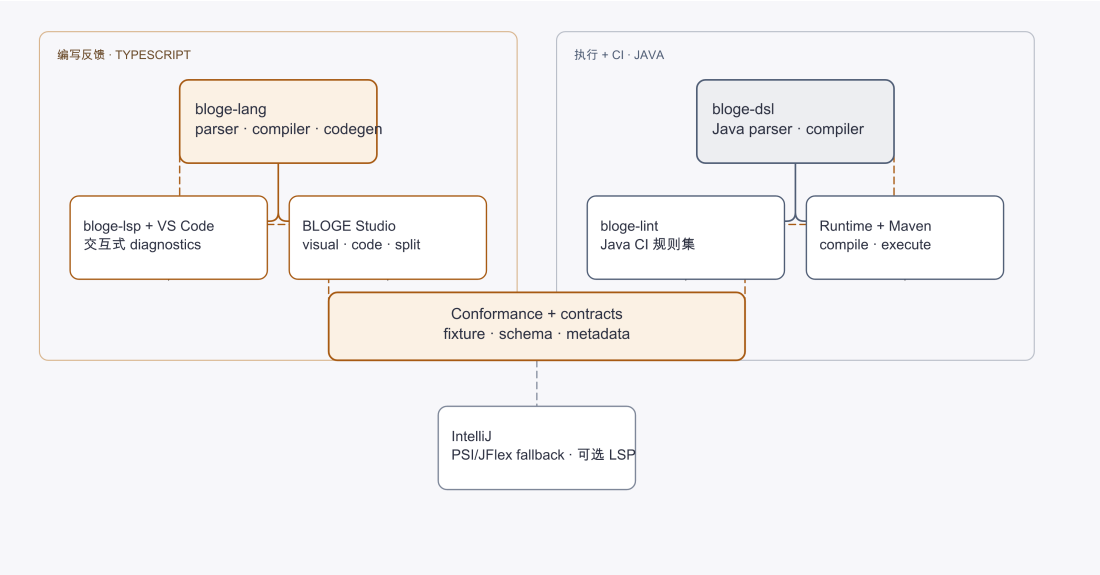
你可以在纯文本编辑器中编写语法正确的 `.bloge` 文件。但没有工具链的话：

- 节点 **ID** 中的拼写错误 要到 graph 引擎启动时才会被发现。
- 未使用的节点 和 缺失的 **timeout** 会在审查中被忽略。
- 格式漂移 使 PR 充满噪音，掩盖了真正的变更。
- **Operator schema** 不匹配（Java 代码和 DSL 之间的不一致）在运行时 `ClassCastException` 之前都是不可见的。

BLOGE 工具链的设计原则是 最快的反馈循环也是成本最低的：输入时红色波浪线就会出现，保存时格式自动规范化，CLI linter 作为最终关卡在 CI 中运行。这些界面并不共享一套实现：LSP 与 Studio 走 TypeScript，runtime 解析与 CLI lint 走 Java；conformance fixture 和导出合同负责约束两条路径的一致性。

9.5 心智模型

把工具链想象成两条由兼容性证据连接的反馈路径：



关键洞察：对齐边界是共享合同，不是共享代码。 `bloge-lang` 提供 `bloge-lsp` 与 Studio 使用的 TypeScript parser/compiler；runtime 与 Maven 走 Java `bloge-dsl`，`bloge-lint` 提供 Java CI 规则；IntelliJ Community 保留 PSI/JFlex 路径，并在支持时叠加 LSP。实现可能漂移，因此一致性需要验证，不能靠假设。

9.6 第一个可运行的示例

在安装了 `bloge-vscode` 扩展的 VS Code 中打开 `ticket-routing.bloge` 示例。你应该立即看到：

1. 语法高亮 ——`graph`、`node`、`branch`、`depends_on` 等关键字被着色；文档注释（`///`）获得文档注释作用域；像 `FetchCustomerOperator` 这样的 operator 引用被高亮为类型。
2. 诊断信息 ——如果你故意拼错一个 `depends_on` 目标（比如把 `fetchCustomer` 改成 `fetchCustmer`），保存之前就会出现红色下划线。
3. 悬停 ——将鼠标悬停在 `timeout` 上，你会看到：*"Sets the maximum execution time for a node."*。悬停在 `3s` 上，你会看到：*"Duration: 3s = 3000ms"*。悬停在 `analyzeSentiment` 上，你会看到它的 operator 引用、文档注释、依赖和 timeout。
4. 补全 ——在 `input { }` 块内，输入 `=`，LSP 会建议 `ctx`、所有节点 ID 以及 `concat`、`size`、`coalesce` 等内置函数。
5. 格式化 ——触发" 格式化文档"，整个文件会通过 `DslCodeGenerator.generate(ast)` 重新输出，规范化缩进和空格。

以上五个功能都由 `document-manager.ts` 中定义的单一流水线驱动：

```
// Phase 1: Lexing   → tokens
// Phase 2: Parsing  → AST (GraphDef)
// Phase 3: Compile  → CompiledGraph + diagnostics
// Phase 4: Lint     → LintRunner.run(ast, compiled)
// Phase 5: Index    → buildSymbolIndex(ast, uri, text)
```

这条五阶段流水线在 每次按键时 运行（通过 `server.ts` 中的 `documents.onDidChangeContent`），缓存的 `Parse`

`State` 会被 hover、completion、definition、references 和 rename 处理器复用。在这五个阶段之后，`DocumentManager` 还会刷新 import graph 的簿记信息，以保持跨文件特性同步。

9.7 跟着一个 PR 走完反馈闭环

工具链的价值，在于缩短“做出编辑”到“得到可评审事实”的距离。设想一个 PR 删除了 `missing-timeout.bloge` 中外部调用的 `timeout`。真正有用的故事不是 参观编辑器功能，而是让一个缺陷连续经过四个反馈面。

9.7.1 一个缺陷使用一个稳定身份

反馈面	读者看到什么	必须交接什么
编辑器	node 位置上的行内诊断	`ruleId=missing-timeout` 与源码位置
CLI lint	可复现的非零结果	与 CI 相同的 rule id 和 severity
Scenario 测试	代表性 Case 的运行行为	Case verdict 加 node/effect 证据
PR 评审	为什么这个编辑安全	诊断已消失，聚焦测试已通过

自动补全能帮你写语法，但不能闭合这个循环。团队真正共享的合同，是稳定的 诊断身份，以及另一名开发者可以重新执行的命令。

9.7.2 先诊断，再决定是否运行全部测试

最短的有效序列是：

`EDIT` 一处声明
→ 在源码位置 `DIAGNOSE`
→ 使用与 `CI` 相同的规则集执行 `LINT`
→ 运行覆盖变更路径的最小 `Scenario`
→ 把两类结果一起交给 `REVIEW`

如果 lint 仍失败，就回到声明；此时运行完整集成套件只会增加噪音。如果 lint 已通过但 Scenario 失败，应检查运行证据；修改 lint 配置修不好业务行为。只有两个聚焦信号都通过，才值得运行更大范围构建。

这里的声明边界很清楚：lint 证明声明式编写规则，Scenario 证明 fixture 条件 下的行为，任何一项单独成立都不能证明生产正确性。

9.8 拆解分析

9.8.1 1 —共享语言核心：bloge-lang

`bloge-lang` 是一个纯 TypeScript 库，零依赖。它导出：

- **`Lexer`** ——手写的扫描器，从 `.bloge` 源码产生 token。
- **`Parser`** ——递归下降解析器，带有优先级爬升的表达式解析。处理 import、graph 级别的 `input/output` schema、`signal_schema`、`oneOf(...)`、联合类型、`?.` 安全导航等。
- **`compile(ast, options?)`** ——前端语义分析：依赖解析、环检测、schema 校验、联合类型收窄、分支穷尽性检查。当提供了 `CompileOptions.importResolver` 时，import 会被递归解析。
- **`DslCodeGenerator.generate(ast)`** ——AST 到源码的格式化器。LSP 的格式化功能和 Studio 的导出都使用同一条代码路径。

三行代码即可使用：

```
import { Lexer, Parser, compile, DslCodeGenerator } from 'bloge-lang';

const tokens = new Lexer(source).tokenize();
const ast    = new Parser(tokens).parse();
const result = compile(ast);
// result.diagnostics → 错误、警告
// DslCodeGenerator.generate(ast) → 格式化后的源码
```

9.8.2 2 —LSP：bloge-lsp

`bloge-lsp` 将 `bloge-lang` 包装在一个通过 **stdio** 通信的 Language Server Protocol 服务器中。`server.ts` 入口注册了以下能力：

能力	处理器文件	功能
Diagnostics	`document-manager.ts`	来自 lexer + parser + compiler + lint 的实时 error/warning/info
Completion	`features/completion.ts`	上下文感知：关键字、节点 ID、schema 类型、内置函数、`oneOf(...)` 和代码片段模板
Hover	`features/hover.ts`	关键字文档、约 70 个内置函数、node/schema/transform 声明、持续时间
Go to Definition	`features/definition.ts`	基于符号索引的跳转到声明的 `nameRange`；通过 `resolveSymbolCrossFile` 支持跨文件
Find References	`features/references.ts`	节点/schema/transform 的所有引用，包括传递 importer
Rename	`features/rename.ts`	重命名声明 + 所有引用；传播到传递 importer；冲突检查
Formatting	`features/formatting.ts`	通过 `DslCodeGenerator.generate(ast)` 实现整文档格式化

能力	处理器文件	功能
Code Actions	`features/code-actions.ts`	`missing-timeout`、`missing-doc-comment`、`unused-node`、`no-duplicate-node-id` 的快速修复；以及 **Source Fix All** 批量安全修复
Semantic Tokens	`features/semantic-tokens.ts`	全文档语义高亮：graph 名称、operator 引用、节点引用、路径段、字面量、关键字——用 AST 感知的 token 类细化 TextMate 作用域

Import 解析 对 `file://` 文档是自动的——`DocumentManager` 接入了一个基于文件的 `ImportResolver`，相对于打开的文件解析路径，并在需要时追加 `.bloge` 后缀。被导入文件中的错误会作为诊断信息显示在导入文件中。

9.8.3 3 —Lint 规则

BLOGE lint 采用三层结构。知道一条规则属于哪一层，就知道它在哪里运行、能看到什么上下文，以及增加新规则时要重建哪个工件。

层	在哪里运行	看到什么	例子
Core（**15** 条）	`bloge-lint` CLI + LSP（翻译成 TS）	单个 `.bloge` 文件解析并编译后的 AST	`no-duplicate-node-id`、`no-duplicate-schema-name`、`unused-node`、`missing-timeout`、`excessive-fan-out`、`missing-doc-comment`、`no-cycle`、`no-unresolved-dependency`、`no-unresolved-branch-target`、`max-script-nodes`、`script-timeout-required`、`script-line-limit`、`loop-exit-completeness`、`cron-expression-valid`、`deadline-in-past`
编译器驱动（**3** 条）	编译器自身	符号解析和类型检查后的整图 IR	`schema-validation-strictness-mismatch`、`dynamic-subgraph-output-untyped`、`ambiguous-timer`
扩展（目前 **2** 条，可通过 **SPI** 扩展）	对应 `*-ext` JAR 在 classpath 上时被发现	扩展自身的 DSL 表面	`session-backward-transition-guard`（来自 `bloge-session-ext`）、`state-machine-structure`（来自 `bloge-state-ext`）

9.8.3.1 Core 规则——LSP 子集

LSP 内置六条交互式规则 (`bloge-lsp/src/lint/rules/`):

规则 ID	严重级别	捕获内容
`no-duplicate-node-id`	Error	节点 ID 被声明了多次
`no-duplicate-schema-name`	Error	Schema 名称被声明了多次
`unused-node`	Info	节点从未被下游引用
`missing-timeout`	Info	节点没有 timeout
`excessive-fan-out`	Info	节点有超过 5 条出边
`missing-doc-comment`	Info	节点缺少 `///` 文档注释

Java 端的 CLI linter(`bloge-lint`) 在批量 检查中包含这些核心规则，同时增加上面表里图级和脚本相关的规则，以及扩展 通过 `LintExtensionContributor` SPI 贡献的规则。

9.8.3.2 编译器驱动规则 这 3 条规则由编译器发出，不是 rule runner。你在 `.blogerc.json` 里 关不掉——它们对应编译器生成合法 `Graph` 时必须做的结构检查。它们 在 LSP 和 CLI 里和普通规则一样，作为诊断浮出来。

9.8.3.3 扩展规则 把 `bloge-session-ext` 或 `bloge-state-ext` 加进项目就同时引入了它们的 lint 贡献。你也可以通过 `LintRuleProvider`(Java SPI,CLI) 和 `BLOGE_LSP_EXTRA_RULES`(LSP) 注册自定义规则。严重级别和规则参数都可在 `.blogerc.json` 配置:

```
{
  "rules": {
    "missing-timeout": "warning",
    "missing-doc-comment": "off"
  }
}
```

9.8.3.4 决策表规则 bloge LSP 和 CLI 都内置了两条专门针对 `decision_table` 节点的 lint 规则。使用以下实战配方让决策表保持完备性和意图清晰。

规则 ID	严重度	触发条件
`decision-table/missing-otherwise`	WARNING	`first`、`unique` 或 `any` 表没有 `otherwise` 子句
`decision-table/collect-otherwise`	INFO	`collect` 表存在 `otherwise` 子句（总是追加——可能并非有意为之）

配方 **1** ——通过添加 **`otherwise`** 消除 **`decision-table/missing-otherwise`**:

```
// [WARN] decision-table/missing-otherwise
decision_table credit_tier(score = ctx.score) hit=first -> String {
  rule (score: score >= 750) -> "platinum"
  rule (score: 680 <= score < 750) -> "gold"
  // 没有 otherwise —— score < 680 时会发生什么?
```

```
}

// [OK] 修复后
decision_table credit_tier(score = ctx.score) hit=first -> String {
  rule (score: score >= 750) -> "platinum"
  rule (score: 680 <= score < 750) -> "gold"
  otherwise -> "rejected"
}
```

配方 **2** ——若 **`decision-table/collect-otherwise`** 属于有意为之，显式确认：

```
// [INFO] decision-table/collect-otherwise
// "baseline" 在每次运行时都会被追加，即使其他规则也命中了。
decision_table discounts(score = ctx.score) hit=collect -> String {
  rule (score: score >= 700) -> "loyalty"
  otherwise -> "baseline"   // 无条件追加——这是有意为之吗?
}

// [OK] 若有意为之，通过 .blogerc.json 覆盖来抑制：
// { "rules": { "decision-table/collect-otherwise": "off" } }
// 若并非有意为之，删除 otherwise 子句。
```

完整的编译期与运行时错误码列表，请见 附录 F ——决策表。

9.8.4 4 —编辑器扩展

VS Code（`bloge-vscode`）：

- `syntaxes/bloge.tmLanguage.json` 处的 TextMate 语法为关键字、文档注释、字符串、操作符、类型和内置函数提供语法高亮作用域。
- `language-configuration.json` 处的语言配置提供括号匹配、自动闭合对、注释切换（`//`、`/* */`）和代码块折叠。
- 扩展入口（`extension.ts`）将 `bloge-lsp` 作为子进程通过 stdio 启动，并将其接线为 `bloge` 语言 ID 的 LSP 客户端。

IntelliJ IDEA（`bloge-intellij`）：

- **Community** 版——纯 PSI 模式：手写的 JFlex 词法分析器、轻量级 `PsiBuilder` 解析器、`ExternalAnnotator` 委托给 `bloge-dsl` Parser 和 `bloge-lint` `LintRunner`、结构视图、代码折叠、语义高亮。无需外部进程。
- **Ultimate** / 商业版——可选 LSP 叠加层：`BlogeLspServerSupportProvider` 启动捆绑的 `bloge-lsp` Node 服务器；重叠的 PSI 提供者（diagnostics、completion、formatting）退让以避免重复结果。需要 `PATH` 上有 Node.js 或设置 `BLOGE_NODE_PATH`。

9.8.5 5 —BLOGE Studio

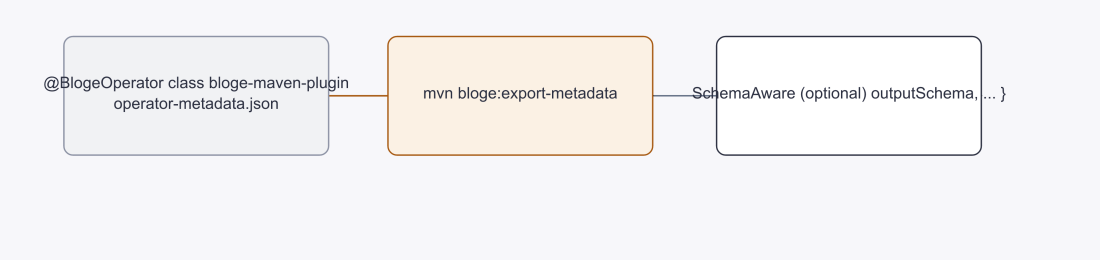
BLOGE Studio 是一个基于浏览器的可视化 DAG 编辑器，使用 React 19 + React Flow 12 构建。它以本地 `file:` 依赖的方式使用 `bloge-lang`，复用相同的解析器、编译器和代码生成器。

与你的编写工作流相关的关键功能：

- 三种视图模式：Visual（仅画布）、Code（仅 Monaco 编辑器）、Split（画布 + 编辑器并排）。Zustand store 在它们之间同步。
- **Operator** 面板：从 `public/operator-metadata.json` 加载。Operator 按层级（基础设施 / 能力 / 领域）标记，支持搜索。拖到画布上即可创建节点。
- 诊断面板：`DiagnosticsPanel.tsx` ——显示来自编译器和 lint 规则的 error、warning 和 info。点击一条诊断信息可以在画布上选中对应的节点。
- 撤销/重做：基于 DSL 快照的历史记录（最多 50 步）。
- 导出：校验 → 运行代码生成器 → 下载 `.bloge` 文件。

9.8.6 6 —Operator 元数据流水线

你的 Java operator 代码与可视化工具之间的桥梁是 `bloge-maven-plugin`：



该插件在 `process-classes` 阶段扫描带有 `@BlogeOperator` 注解的类，反射 `Operator<I,O>` 的泛型参数，对每个类型调用 `SchemaIntrospector.introspect()`（支持 Record 和 POJO），并写入一个 JSON 文件。Studio 加载此文件以填充 operator 面板，并为字段补全提供 I/O schema 信息。

9.9 内嵌运维控制台——/bloge-console

Studio 适合用来编排图。生产环境需要的是另一种视角:" 现在哪些 execution 在跑、下一个 timer 什么时候触发、哪个 operator 在抛错。" 这个视角就是 **`/bloge-console`** ——一个随 `bloge-spring-web` 一起发布的小型只读 运维 UI，挂在配置的 base path 上。

```yaml
spring:
  bloge:
    console:
      enabled: true
      base-path: /bloge-console        # 默认
      require-role: BLOGE_OPS          # Spring Security 角色门禁
```

它展示什么:

- **Executions** ——运行中 / 等待中 / 已完成 / 已失败; 点进去看单次 run 的审计日志。
- **Timers** ——当前已 arm 的持久化 timer 及下次触发时间; 支持手动 *fire now*(受 `require-role` 限制)。
- **Leases** ——当前持有者及剩余 TTL。
- **Schema** ——Flyway 的 `currentVersion`、pending/applied migrations，以及 `UP_TO_DATE`、`PENDING` 或 `FAILED` 状态。
- **Operator registry** ——所有注册的 operator、所在模块，以及 `@BlogeOperator` 元数据，便于快速发现。

控制台设计上是观测用途。它不让你编辑图 (那是 Studio 的事)，也不让 你改写 checkpoint(那要走迁移)。”fire timer” 和”release lease” 是显式的 运维恢复工具，每个动作都以调用者主体写入审计日志。

小测验*:* 什么时候用 */bloge-console*，什么时候用 *Studio?* 提示*: Studio* 写图，控制台读运行时。

9.10 常见陷阱

9.10.1 [X] 假设编辑器和 CI 检查的是相同的规则

LSP 的 lint 规则（TypeScript，在 `bloge-lsp` 中）和 CLI 的 lint 规则（Java，在 `bloge-lint` 中）是 两套独立的实现，规则集有重叠但并不完全相同。LSP 有 `unused-node` 和 `excessive-fan-out`；CLI 额外覆盖了 `no-unresolved-dependency`、`no-unresolved-branch-target`、`no-cycle` 以及若干脚本规则。如果你只依赖编辑器中的绿色波浪线，CI 仍然可能在 LSP 没有检查的规则上失败。

```
# 推送前在本地运行 Java linter:
java -jar bloge-lint.jar check src/main/resources/bloge/

# 或者在构建中配置:
# .blogerc.json 控制编辑器和 CLI 使用时的严重级别覆盖。
```

最佳实践：将 `bloge-lint check` 作为 pre-commit hook 或 CI 步骤运行，以确保在合并前应用完整的规则集。

9.11 引导式重写

拿 `missing-timeout.bloge` 反模式文件，使用工具链来修复它:

1. 在 **VS Code** 中打开（需要安装 `bloge-vscode` 扩展）。你应该在 `loadReport` 上看到一条来自 `missing-timeout` lint 规则的 info 诊断。
2. 悬停 在 `loadReport` 上，确认悬停弹窗中没有列出 timeout。
3. 添加 **timeout**:

```
node loadReport : LoadReportOperator {
  input {
    reportId = ctx.reportId
  }
  timeout = 5s
}
```

4. 格式化文档（VS Code 中按 Shift+Alt+F）。`DslCodeGenerator` 会规范化缩进。
5. 在 **`loadReport`** 上方添加 `///` 文档注释。确认 `missing-doc-comment` 诊断消失。
6. 对该文件运行 **CLI linter**:
   `java -jar bloge-lint.jar check src/main/resources/bloge/antipatterns/missing-timeout.bloge`
   添加了 timeout 和文档注释后，输出应报告零个错误。

7. 在 **BLOGE Studio** 中打开同一文件（在 `bloge-studio/` 中运行 `npm run dev`，然后使用工具栏中的"Open"）。切换到 Split 视图，观察画布和 DSL 代码同步。检查底部的诊断面板——应该是空的。

9.12 思维检查

1. LSP 的格式化功能实际委托给了哪个模块？（*`bloge-lang` 中的 `DslCodeGenerator`。参见 `formatting.ts`——它调用的是 `DslCodeGenerator.generate(state.ast)`。*）
2. LSP 是如何检测到 `depends_on` 目标拼写错误的？（*`bloge-lang` 中的 `compile(ast)` 运行依赖解析。无法解析的依赖会产生一个 `error` 级别的诊断信息，`DocumentManager` 将其转换为 LSP `DiagnosticSeverity.Error`。*）
3. 在 IntelliJ 插件中，当 IDE 是 Community 版且没有安装 Node.js 时会怎样？（插件以纯 *PSI* 模式运行——所有功能由内置的 *JFlex* 词法分析器、*PsiBuilder* 解析器和 *ExternalAnnotator* 提供。*LSP* 叠加层不会激活，因为 `com.intellij.modules.lsp` 不存在。）
4. BLOGE Studio 是如何知道在面板中显示哪些 operator 的？（它在启动时加载 `public/operator-metadata.json`。该文件可以由 `bloge-maven-plugin` 的 `export-metadata` *goal* 生成，也可以手动编写。）
5. 能否在 CLI 中禁用某条 lint 规则？如何操作？（可以——创建一个 `.blogerc.json` 文件，包含 `rules` *map*。将规则 *ID* 设为 `"off"`。例如：`{ "rules": { "missing-doc-comment": "off" } }`。参见 `LintConfig.java`。）

9.13 实验

1. 编辑器设置：安装 `bloge-vscode` 扩展（或在 `bloge-intellij/` 中运行 `./gradlew runIde` 启动 IntelliJ 插件沙箱）。打开 `ticket-routing.bloge`。验证：
   - 语法高亮覆盖了所有关键字和文档注释。
   - 悬停在 `analyzeSentiment` 上会显示其 operator、描述、依赖和 timeout。
   - 在 `depends_on` 列表中 Ctrl+ 点击 `fetchCustomer` 可以跳转到节点声明。
   - 重命名 `classifyPriority` 会更新分支条件和所有 `depends_on` 引用。
2. 故意破坏一个 **graph**：在 `ticket-routing.bloge` 中，将 `classifyPriority` 上的 `depends_on = [analyzeSentiment]` 改为 `depends_on = [nonExistentNode]`。观察出现的诊断信息。然后对该文件运行 `bloge-lint check`，对比 CLI 输出和编辑器波浪线。
3. **Operator** 元数据往返：如果你有一个包含 `@BlogeOperator` 类的 Maven 项目，运行 `mvn bloge:export-metadata`，将生成的 `operator-metadata.json` 复制到 `bloge-studio/public/` 中。启动 Studio 并确认你的 operator 出现在面板中，且 I/O schema 正确。
4. **Studio** 分屏视图：在 BLOGE Studio 中打开 `ticket-routing.bloge`。切换到 Split 视图。在代码编辑器中添加一个新节点，观察它出现在画布上。删除它，然后使用撤销（Ctrl+Z）恢复。

9.14 桥接：从工具链到验证门禁

编辑器 diagnostics 和 `bloge-lint` 可以缩短编写反馈，却不会执行经过批准的业务 Scenario。通用 customer module 进入 `0.9.8-RC1` 验证路径时，应在同一次 Maven 调用中编译 bootstrap，再让 Preview goal 读取 `.blogeverify.yaml`：

```
mvn test-compile \
  com.leanowtech.bloge:bloge-maven-plugin:0.9.8-RC1:verify
```

生成的 artifact 仍需 source binding，并把 receipt 保存在可变目录之外。本书贷款 starter 在 RC1 中会因 `com.leanowtech.bloge.starter.*` 类加载边界 fail closed，当前请运行其 JUnit Scenario 测试；不要把 `TEST_CLASSPATH_UNAVAILABLE` 写成业务失败。第 32 章——变化归因、Maven 与发布决策会解释源码安装、已知限制、命令阶梯和 claim 边界。

9.15 实验验收卡

- 预期与观察： missing-timeout 经 edit、diagnose、lint、Scenario 和 review 形成闭环。
- 失败与恢复：删除 timeout 让门禁失败；恢复后重跑最小检查集。
- 证明边界：证明工具链阻止已声明缺陷，不证明能发现未编码规则。
- 练习合同：一个 PR；只引入或修复一项缺陷；交付诊断、测试、审查记录；三者指向同一规则即停止。

9.16 回顾

- **`bloge-lang`** 是 LSP 与 Studio 使用的 TypeScript 编写核心，不是 Java runtime parser。
- **`bloge-dsl`** 是 Java runtime/Maven 解析路径；conformance fixture 与导出合同构成两套实现的兼容性边界。
- **`bloge-lsp`** 将 `bloge-lang` 包装在 LSP 服务器中，通过 stdio 提供 diagnostics、completion、hover、go-to-definition、find references、rename 和 formatting。
- **`bloge-vscode`** 提供 TextMate 语法高亮和语言配置，然后将所有智能功能委托给 `bloge-lsp`。
- **`bloge-intellij`** 在 Community 版以纯 PSI 模式运行，在 Ultimate/商业版上添加可选的 LSP 叠加层。
- **BLOGE Studio** 是基于浏览器的可视化编辑器，具有三向视图（visual / code / split）、诊断面板和由 `operator-metadata.json` 驱动的 operator 面板。
- **`bloge-maven-plugin`** 通过从 `@BlogeOperator` 注解生成 `operator-metadata.json`，在 Java operator 代码和可视化工具之间架起桥梁。
- **`bloge-lint`**（Java CLI）在 CI 中运行同类型的 lint 规则；其规则严重级别可通过 `.blogerc.json` 配置。
- LSP 和 CLI 的 lint 规则集 有重叠但并不完全相同——始终在 CI 中运行 CLI linter 作为最终关卡。

9.17 下一步

在第 10 章——用子图复用中，你会继续使用这条反馈循环，把可复用 graph 行为放进明确的父子合同。

9.18 参考链接

- `bloge-lang/README.md` —TypeScript 编写核心文档
- `bloge-lsp/README.md` —LSP 服务器文档和架构图
- `bloge-lsp/src/document-manager.ts` —五阶段分析流水线源码
- `bloge-lsp/src/lint/lint-rule.ts` —`LintRule` 接口和内置规则集
- `bloge-vscode/README.md` —VS Code 扩展文档
- `bloge-vscode/src/extension.ts` —LSP 客户端接线
- `bloge-vscode/syntaxes/bloge.tmLanguage.json` —TextMate 语法
- `bloge-intellij/README.md` —IntelliJ 插件文档和双模式架构
- `bloge-studio/README.md` —可视化 DAG 编辑器文档
- `bloge-studio/src/components/panels/DiagnosticsPanel.tsx` —Studio 诊断面板源码
- `bloge-studio/public/operator-metadata.json` —内置 operator 目录
- `bloge-maven-plugin/README.md` —用于 operator 元数据生成的 Maven 插件
- `bloge-lint/README.md` —Java CLI linter 文档
- `bloge-lint/src/main/java/.../LintConfig.java` —`.blogerc.json` 配置解析器
- `bloge-lint/src/main/java/.../BlogeLintCli.java` —CLI 入口
- `ticket-routing.bloge` —主要示例
- `missing-timeout.bloge` —反模式 fixture

第二阶段回顾——可维护编写（第 5–9 章）

章节

—

第二阶段回顾——可维护编写（第 5–9 章）

你现在可以编写带分支、带韧性的工作流，并用干净、可审查的 *DSL* 文件表达它们。

0.1 你学到了什么

第 **5** 章——做决策的分支 引入了 `branch on` 进行控制面路由。你了解了引擎如何将未选中的路径标记为 SKIPPED、`otherwise` 为什么必不可少，以及 `branch on`（路由）和 `when`（值选择）的区别。

第 **6** 章——韧性设计 展示了 retry、timeout、fallback 和 compensate 都是 graph 级声明，而非 try/catch 内联代码。你学会了将策略附加到节点上、将它们分层组合（retry → timeout → fallback），并预测策略交互时的引擎行为。

第 **7** 章——设计良好的 **Operator** 把注意力转向了 operator 边界。你了解了什么让 operator 可测试（单一职责、无隐藏状态、声明式 schema）、如何惯用地处理失败，以及何时返回丰富结果类型而非直接抛异常。

第 **8** 章——把个人 **DSL** 草稿变成团队资产 将所有内容整合到可维护的 `.bloge` 文件中。你练习了 graph 级 schema、文档注释、命名 schema、`let` 绑定和一致的文件布局，让个人草稿变成可审查的团队资产。

第 **9** 章——工具链工作流 补上了编写闭环。你让一次修改依次经过编辑器诊断、lint、测试、graph 渲染和 PR 证据，使“我能写”升级为“别人能审”。

0.2 你现在应该能做到

- 用 `branch on` 将执行路由到不同路径，并用 `otherwise` 提供安全默认值。
- 为需要的节点附加 `retry`、`timeout` 和 `fallback`，并解释其他节点为什么不需要。
- 设计一个具有清晰输入/输出契约和可测试边界的 operator。
- 编写一个包含 schema、文档注释、分支、韧性策略和 transform 的完整 `.bloge` 文件。
- 审查他人的 `.bloge` 文件并识别结构性问题。
- 跑通编辑 → lint → 测试 → 可视化闭环，并为评审附上最小充分证据。

0.3 这个阶段的常见错误

错误	为什么会犯	如何纠正
分支表达式类型不匹配	在引擎期望 enum 或 boolean 的位置用了字符串	让 `branch on` 表达式的类型与声明的 case 完全匹配。
给每个节点都加 retry	防御本能	只对调用不稳定外部系统的节点加 retry；纯逻辑节点不应该重试。
忘记 `otherwise`	自信地认为所有 case 都已覆盖	始终包含 `otherwise`——即使它路由到错误节点——以避免 graph 静默卡住。
Operator 内部依赖 graph 结构	把 graph context 内部信息传递给了 operator 逻辑	Operator 接收类型化输入和 context；它不应该导航 graph 模型。
不写 schema 文档	”类型很明显”	添加 `///` 文档注释和显式 schema 块；成本为零，却能为下一位读者节省大量时间。
把编辑器诊断当成正确性证明	红线看起来很权威	编辑器负责缩短反馈，parser 与行为测试才负责判定正确性。

0.4 新增核心词汇

术语	含义
`branch on`	控制面路由：根据运行时值将执行导向多个目标节点之一。
`otherwise`	当没有显式 case 匹配时的默认分支目标。
SKIPPED	未被选中的分支路径上的节点所获得的状态。
`retry`	韧性策略：失败时最多重新执行 N 次。
`timeout`	韧性策略：节点执行超过指定时长时取消。
`fallback`	韧性策略：主逻辑失败时替换为默认结果。
`compensate`	韧性策略：当已成功的节点需要撤销时执行清理操作。
Schema	graph 或 node 的输入/输出契约的命名或内联类型声明。
编写闭环	让同一次修改依次经过编辑、lint、测试、可视化与评审。

0.5 试一试：准备一个可评审改动

同事给支付 graph 的每个 node 都加了 retry，PR 里只有一张 DSL 截图。请给每个 node 标记“保留 retry”或“删除 retry”，再写出评审前必须补齐的三份证据。好的答案会区分不可靠 I/O 与确定性逻辑，并要求 lint 结果、聚焦行为测试，以及能看出路径变化的 graph 图。

0.6 下一步

第三阶段回顾教你组合：子图复用、批处理迭代、等待外部事件和持久执行。从第 10 章——用子图复用开始。

学习阶段

第三阶段——组合与时间

子图、迭代、挂起与崩溃恢复

目标：让工作跨集合、外部时间与进程重启继续成立。

章节

10

第三阶段——组合与时间

第 10 章——用子图复用

子图、迭代、挂起与崩溃恢复

目标：让工作跨集合、外部时间与进程重启继续成立。

承诺：读完本章后，你将知道如何把一个流水线提取为命名子图，使用 *subgraph("name")* 将其嵌入父图，控制作用域隔离，并识别何时应该使用动态子图。

10.1 学习目标

1. 解释什么是 BLOGE 中的子图（subgraph），以及它为何存在。
2. 使用 `DslCompiler.registerSubGraph()` 注册预构建的 `Graph`，并在 DSL 中通过 `node x : subgraph("name") {}` 语法引用它。
3. 区分 ** 隔离（isolated）作用域（子图的默认模式）和父级（parent）** 作用域，并预测每种模式下子图内部能看到什么。
4. 读取子图输出——理解终端节点的输出会被聚合到父节点的结果 map 中。
5. 识别何时应该使用 `dynamicSubGraph` operator 来执行运行时生成的 DSL，而不是编译时注册的子图。

10.2 前置条件

- 第 8 章——把个人 DSL 草稿变成团队资产

10.3 源示例

文件	展示内容
`loan-approval-subgraph.bloge`	两个并行子图（信用评估 + 合规检查）汇入一个带分支的核保决策
`international-shipment.bloge`	并行的清关和路线优化子图，带有扇入

文件	展示内容
`smart-ticket-handling.bloge`	情感分析子图驱动优先级分支，进入升级处理子图
`order-full-pipeline.bloge`	支付和库存子图并行运行
`dynamic-agent.bloge`	运行时 DSL 生成 + `dynamicSubGraph` 执行
`LoanApprovalSubGraphDslExample.java`	Java 接线：通过 `Graph.builder()` 构建子图，使用 `DslCompiler.registerSubGraph()` 注册
`SubGraphOperator.java`	引擎层面的实现：执行子图并聚合终端节点输出

10.4 为什么这很重要

到第 8 章为止，你已经能用 DSL 编写完整的 graph。但真实系统中有些流水线会反复出现在不同业务领域：

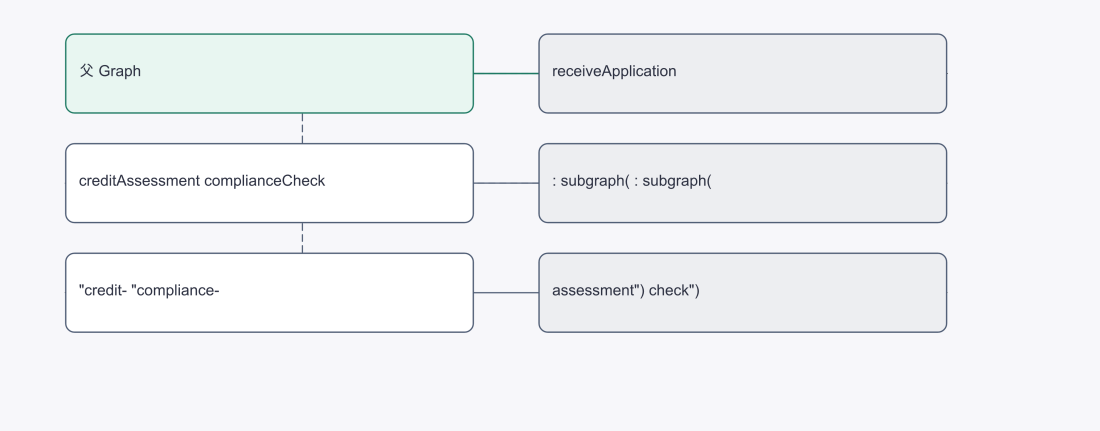
- 一条信用评估流水线（征信查询 → 收入验证 → 负债率计算 → 风险评分）会出现在贷款审批、房贷办理和信用卡发行中。
- 一条合规检查流水线（反洗钱筛查 → KYC → 制裁名单 → 合规判定）被多个监管工作流所需。
- 一条厨房调度或配送协调流水线在不同的餐饮订单类型之间共享。

没有子图的话，你只能把整段节点链复制粘贴到每个父图中，重复韧性配置、依赖和 operator 接线。当合规规则变化时，你需要更新 *N* 份副本——而且不可避免地会漏掉某一份。

BLOGE 用命名子图解决这个问题：你把一条流水线构建为一个 `Graph`，在一个名字下注册它，然后在任何父图中用 `node x : subgraph("name") {}` 引用它。引擎会将子图作为嵌套执行来运行，把父级的 `input {}` 绑定注入为子图的上下文，并将子图的终端节点输出聚合回父节点的结果。

一条流水线。一次注册。多个调用点。

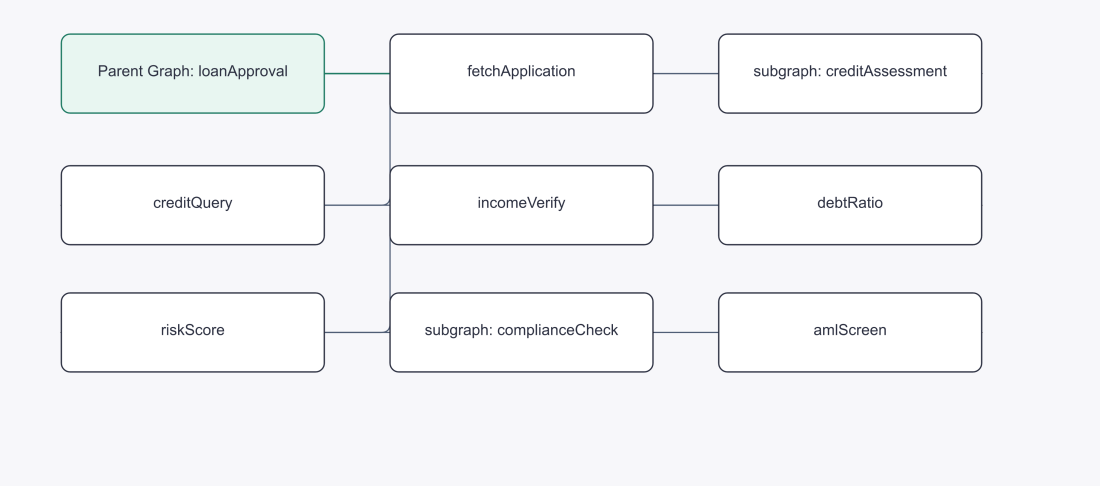
10.5 心智模型



把每个子图方框想象成一次函数调用：父图通过 `input {}` 传递参数，子图运行其内部 DAG，父图收到以节点 ID 为键的终端节点输出 map。

核心规则：

1. 输入通过 **`input {}`** 流入。这些绑定会被注入为子图的 `GraphContext`。子图通过 `ctx.*` 表达式读取它们。
2. 输出通过终端节点流出。 `SubGraphOperator` 遍历 `subGraph.terminalNodes()`，将每个终端节点的输出收集到一个 `Map<String, Object>` 中。父图通过 `subgraphNode.output.terminalNodeId` 引用它们。
3. 作用域默认为 **`isolated`**。子图不会看到父节点的输出，也不会继承父级的 `GraphContext`——除非你显式设置 `scope = parent`。



10.6 第一个可运行示例

下面是 `loan-approval-subgraph.bloge`，精简到子图接线部分：

这 71 行接线保留在同一个代码块中，因为两个并行 child 合同、parent fan-in 和 两条终止路由必须同时可见；继续裁剪会隐藏父图可见的 output 边界。

```
graph loanApprovalPipeline {

  node receiveApplication : ReceiveApplicationOperator {
    input {
      applicationId   = ctx.applicationId
      applicantName   = ctx.applicantName
      requestedAmount = ctx.requestedAmount
      termMonths      = ctx.termMonths
      employerId      = ctx.employerId
    }
    timeout = 3s
  }

  /// 运行信用评估子图：
  /// creditQuery → incomeVerification → debtRatioCalc → riskScoring
  node creditAssessment : subgraph("credit-assessment") {
```

```
  depends_on = [receiveApplication]
  input {
    applicationId   = receiveApplication.output.applicationId
    applicantName   = receiveApplication.output.applicantName
    requestedAmount = receiveApplication.output.requestedAmount
    employerId      = receiveApplication.output.employerId
  }
  timeout = 30s
}

/// 运行合规检查子图：
/// amlScreening → kycVerification → sanctionListCheck → complianceDetermination
node complianceCheck : subgraph("compliance-check") {
  depends_on = [receiveApplication]
  input {
    applicationId = receiveApplication.output.applicationId
    applicantName = receiveApplication.output.applicantName
  }
  timeout = 30s
}

node underwritingDecision : UnderwritingDecisionOperator {
  depends_on = [creditAssessment, complianceCheck]
  input {
    applicationId = receiveApplication.output.applicationId
    credit        = creditAssessment.output.riskScoring
    compliance    = complianceCheck.output.complianceDetermination
  }
}

branch on underwritingDecision.output.decision {
  "approved" -> generateApprovalLetter
  otherwise  -> generateRejectionNotice
}

node generateApprovalLetter : GenerateApprovalLetterOperator {
  depends_on = [underwritingDecision]
  input {
    applicationId   = underwritingDecision.output.applicationId
    applicantName   = ctx.applicantName
    requestedAmount = ctx.requestedAmount
    approvedRate    = underwritingDecision.output.approvedRate
    termMonths      = ctx.termMonths
  }
}
```

```
  node generateRejectionNotice : GenerateRejectionNoticeOperator {
    depends_on = [underwritingDecision]
    input {
      applicationId = underwritingDecision.output.applicationId
      applicantName = ctx.applicantName
      reason        = underwritingDecision.output.reason
    }
  }
}
```

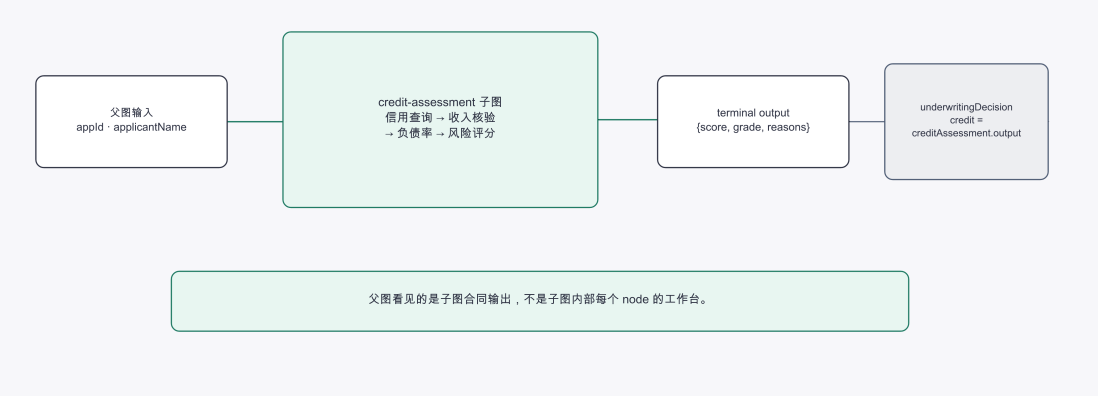
把它当成一个故事来读：

1. **receiveApplication** 验证传入的贷款申请。
2. 两个子图并行运行 ——`creditAssessment` 和 `complianceCheck` ——各自包装了一个四节点的流水线。两者都依赖 `receiveApplication`，因此引擎会并发调度它们。
3. **underwritingDecision** 从两个子图扇入。它读取 `creditAssessment.output.riskScoring` ——信用子图中终端节点的输出——以及 `complianceCheck.output.complianceDetermination`。
4. 分支根据结果路由到批准或拒绝。

> 关键认识：每个 *`subgraph("name")`* 节点在父图的 *DAG* 中看起来就像一个普通节点。引擎不关心它内部包含四个步骤——它只看到一个有依赖和输出 *map* 的节点。

10.7 一个子图，只向父图交付一个结果

贷款流程中的 `creditAssessment` 内部包含信用查询、收入核验、负债率计算和 风险评分。父图不应该绑定四个内部工作台；它只需要一份稳定的信用评估结果。



10.7.1 从外向内读取这条边界

视角	可以依赖	不应依赖
子图内部	收入、负债率等中间输出	父图 node 名或父图调度

视角	可以依赖	不应依赖
子图终态合同	`{score, grade, reasons}`	内部 node 布局
父图 `underwritingDecision`	`creditAssessment.output`	子图里的 `creditQuery.output`

companion smoke Scenario 观察的是公开结果：`creditAssessment` 与 `complianceCheck` 都执行，underwriting 消费两者输出，批准信执行而拒绝信为 `SKIPPED`。父图不需要知道分数由子图中的哪个 node 产生。

10.7.2 重构时不要泄漏工作台

把 `riskScoring` 改名为 `calculateRisk`，或在它之前插入欺诈检查。只要 terminal output 仍满足 `{score, grade, reasons}`，父图就不用变化。如果父图 直接引用子图内部 node，一次私有重构就会变成跨 graph 迁移，说明复用边界 已经失效。

动态子图应作为进阶部署技术。它改变运行时如何选择 definition，不改变这条 合同规则。先掌握静态父子输出边界，只有当业务确实需要在多个 definition 之间选择时，再增加动态选择。

10.8 拆开来看

10.8.1 `subgraph("name")` 语法

```
node <id> : subgraph("<registered-name>") {
  depends_on = [...]
  input { ... }
  timeout = ...
  scope = isolated | parent    // 可选；默认为 isolated
}
```

解析器识别 `subgraph(` 令牌序列，并将 operator 引用重写为一个内部前缀（`subgraph:<name>`）。`StandardNodeCompiler` 随后在已注册的子图 map 中查找该名称，并接线一个 `SubGraphOperator`。

10.8.2 注册子图（Java 端）

在编译父级 DSL 之前，你需要构建每个子图并注册它：

```
// 通过 Java API 构建子图
Graph creditGraph    = buildCreditAssessmentSubGraph();   // Graph.builder("credit-assessment")...
Graph complianceGraph = buildComplianceCheckSubGraph();  // Graph.builder("compliance-check")...

// 在 compile() 之前注册
var compiler = new DslCompiler(registry);
compiler.registerSubGraph("credit-assessment", creditGraph);
compiler.registerSubGraph("compliance-check", complianceGraph);

Graph mainGraph = compiler.compile(ast);
```

（摘自 *LoanApprovalSubGraphDslExample.java*，第 *274–284* 行。）

每个子图都使用标准的 `Graph.builder()` API 构建——operator、依赖、输入、重试、超时——与顶层 graph 完全一致。唯一的区别是它从 `ctx.*` 读取初始数据，而不是从父节点的输出。

10.8.3 输出聚合的工作原理

`SubGraphOperator.execute()` 运行子图，然后：

```
Map<String, Object> output = new LinkedHashMap<>();
for (String terminalId : subGraph.terminalNodes()) {
    Object terminalOutput = result.results().getRaw(terminalId);
    if (terminalOutput != null) {
        output.put(terminalId, terminalOutput);
    }
}
return output;
```

父图通过名称引用每个终端节点：`creditAssessment.output.riskScoring` 导航到该 map 中的 `riskScoring` 键。

10.8.4 作用域：`isolated` vs `parent`

作用域	默认用于	行为
`isolated`	`subgraph("name")`	子图仅接收显式的 `input {}` 绑定。不会泄漏父级 `GraphContext`。
`parent`	`loop`、`foreach`	子图继承父级 `GraphContext`，并可通过运行时注入的 `__parentOutput_*` 键引用父节点输出。

你可以覆盖默认值：

```
node nested : subgraph("childGraph") {
  scope = parent
  input { key = parentNode.output.value }
}
```

使用 `scope = parent` 时，`SubGraphOperator` 会先将父级的 `GraphContext` 合并到子图上下文中，再叠加显式输入。这对原型开发很方便，但会削弱封装性——子图现在隐式依赖于父级的上下文结构。

（参见 *DSL 规范* 和 *StandardNodeCompiler.java* 了解作用域规则和编译路径。）

10.8.5 进阶：动态子图

对于子图在编译时未知的场景，BLOGE 提供了 `dynamicSubGraph` operator。上游节点生成 DSL 文本；`dynamicSubGraph` 在运行时对其进行沙箱验证、编译并执行：

```
node plan : generateDynamicDsl {
  input { task = ctx.task, history = ctx.history }
  timeout = 3s
```

```
}

node executePlan : dynamicSubGraph {
  depends_on = [plan]
  input {
    dslSource = plan.output.dslSource
    context = { task: ctx.task, history: ctx.history, planKind: plan.output.planKind }
  }
  timeout = 15s
  retry = { attempts: 2, backoff: 100ms, strategy: exponential }
}
```

（摘自 *dynamic-agent.bloge*。）

`dynamicSubGraph` 是一个能力层 operator，通过 `CommonOperators.builder().dynamic().build()` 注册。它使用 `DslSandbox` 验证生成的 DSL，通过 `GraphComplexityValidator` 强制复杂度限制，并通过 `GraphEngine.executeNestedGraph()` 运行子图。

对编译时组合使用 **`subgraph("name")`**。仅当 **graph** 结构必须在运行时决定时才使用 **`dynamicSubGraph`**（例如 LLM 生成的计划、插件架构）。

10.9 常见陷阱

10.9.1 [X] 把子图节点的输出当作扁平值

```
// 错误 —— creditAssessment.output 是一个以终端节点 ID 为键的 Map
node decision : UnderwritingOp {
  input {
    riskGrade = creditAssessment.output.riskGrade   // ← 路径错误
  }
}
```

子图节点的输出是一个 `Map<String, Object>`，其中每个键是子图的终端节点 **ID**。如果子图的终端节点是 `riskScoring`，正确的路径是：

```
// 正确 —— 通过终端节点 ID 导航
node decision : UnderwritingOp {
  input {
    credit = creditAssessment.output.riskScoring   // ← 终端节点 ID
  }
}
```

如果子图有多个终端节点，每个都会作为单独的键出现。查看子图的结构以确定要引用哪些终端 ID。

10.9.2 [X] 把 `scope = parent` 当作默认选项

到处加 `scope = parent` 以免在 `input {}` 中逐一穿透每个值，确实很诱人。请克制。隔离作用域迫使你声明契约——子图实际需要哪些输入。这使得子图能在不同的父图之间复用。将 `scope = parent` 保留给真正的共享基础设施上下文（例如追踪 ID、租户标识）。

10.10 引导式重写

查看 `international-shipment.bloge`。它使用了两个并行子图——`customsClearance` 和 `routeOptimization`——然后在 `bookingConfirmation` 处扇入。

思考以下问题：

1. 父图如何向 **`customsClearance`** 传递数据？（通过显式的 *`input {}`* 绑定。子图读取 *`ctx.shipmentId`*、*`ctx.commodityType`* 等，因为这些值是从父图的 *input* 块注入到子图的 *`GraphContext`* 中的。）
2. **`routeOptimization.output.optimalRouteSelection`** 是什么？（它是路线优化子图中名为 *`optimalRouteSelection`* 的终端节点的输出。由 *`SubGraphOperator`* 聚合而来。）
3. 如果移除 **`customsClearance`** 上的 **`timeout = 30s`** 会怎样？（子图节点将没有时间限制。如果子图挂起——比如 *HS* 编码分类服务宕机了——父图将在该节点无限期阻塞。请始终为子图节点设置 *timeout*。）
4. 能否在另一个父图中复用 **`customsClearance`** 子图？（可以——只要你用 *`"customs-clearance"`* 注册同一个子图，并提供期望的输入：*`shipmentId`*、*`commodityType`*、*`originCountry`*、*`destinationCountry`*、*`weightKg`*。）

现在尝试修改这个 graph：

- 把 `trackingSetup` 和 `sendNotification` 提取到一个名为 `"post-booking"` 的新子图中。注册它。用一个 `node postBooking : subgraph("post-booking") {}` 节点替换这两个节点。
- 你需要传递哪些输入？父图将引用哪个终端节点 ID？

10.11 脑力检查

1. 在 DSL 中引用预注册子图的语法是什么？（*`node <id> : subgraph("<name>") { ... }`*。）
2. 如何在编译前注册子图？（在 *`DslCompiler`* 实例上调用 *`compiler.registerSubGraph("name", graph)`*。）
3. `subgraph("name")` 节点的默认作用域是什么？（*`isolated`* ——子图只能看到显式的 *`input {}`* 绑定。）
4. 父节点如何读取子图的结果？（通过 *`subgraphNode.output.<terminalNodeId>`* ——*`SubGraphOperator`* 将每个终端节点的输出收集到一个以节点 *ID* 为键的 *map* 中。）
5. 什么时候应该使用 `dynamicSubGraph` 而不是 `subgraph("name")`？（当子图的结构在运行时才能确定时——例如 *LLM* 生成的计划或插件提供的工作流。*`subgraph("name")`* 用于编译时组合。）
6. 设计题：你的 `compliance-check` 子图被三个父图复用。其中贷款审批父图需要更细的 AML 明细，而另外两个父图并不需要。你应该把子图参数化、派生一个变体，还是把这部分细节推回父图里？（优先考虑通过子图输入做参数化。如果这份额外明细会从根本上改变子图的处理流水线，就创建一个变体。不要把领域逻辑推回父图，否则就削弱了抽取子图的意义。）

10.12 练习

1. 打开 `loan-approval-subgraph.bloge` 和 `LoanApprovalSubGraphDslExample.java`。
   - 追踪数据流：`creditAssessment.output.riskScoring` 在运行时解析为什么？（一个包含 *`riskGrade`*、*`compositeScore`*、*`summary`* 键的 *map* ——即信用评估子图中 *`riskScoring`* 节点的输出。）
2. 打开 `smart-ticket-handling.bloge`。
   - 该 graph 有两个子图：`sentimentAnalysis` 和 `escalationWorkflow`。只有其中一个总是运行。是哪个，为什么？（只有 *`sentimentAnalysis`* 总是运行。*`escalationWorkflow`* 位于 *`branch on determine Priority.output.priority`* 之后——它只在优先级为 *`"high"`* 时执行。）
3. 为电商退货流水线设计一个新 graph：
   - 创建一个 `"fraud-screening"` 子图，包含三个节点：`orderLookup → returnHistoryCheck → fraud Score`。
   - 创建一个 `"refund-processing"` 子图，包含两个节点：`calculateRefund → initiateRefund`。
   - 编写一个父图：先运行欺诈筛查，根据欺诈评分进行分支，将干净的退货路由到退款处理。
   - 每个子图需要的最小 `input {}` 契约是什么？
4. 附加题：打开 `dynamic-agent.bloge`。
   - 如果 `plan` 节点生成了无效的 DSL 会怎样？（*`DynamicSubGraphOperator`* 内部的 *`DslSandbox.parseAnd Validate()`* 调用会抛出异常。如果 *`executePlan`* 配置了 *retry*，引擎会重试。如果所有尝试都失败且没有 *fallback*，*graph* 将失败。）

10.13 实验验收卡

- 预期与观察：父图只消费子图 terminal output，不依赖内部节点。
- 失败与恢复：绑定不存在的内部 output；恢复 terminal node ID 后重跑。
- 证明边界：证明子图输出合同和隔离，不证明动态子图安全。
- 练习合同：贷款子图；只改一个 output binding；交付可见 key 和失败 reason；父图只引用 terminal key 即停止。

10.14 回顾

- **`subgraph("name")`** 让你将一个命名的、预构建的 graph 作为单个节点嵌入到父图中。子图通过 `DslCompiler.registerSubGraph()` 在编译前注册。
- 输入通过 **`input {}`** 绑定流入，这些绑定成为子图的 `GraphContext`。输出通过终端节点流回——父图通过 `subgraphNode.output.<terminalNodeId>` 读取它们。
- 子图默认使用 **`isolated`** 作用域：不会泄漏父级上下文。谨慎使用 `scope = parent` ——它会削弱封装性。
- 子图是父图 DAG 中的普通节点。它们像其他节点一样支持 `depends_on`、`timeout`、`retry` 和 `fallback`。当依赖关系允许时，多个子图节点可以并行运行。
- 对于运行时生成的 graph，使用 `dynamicSubGraph` operator，它在执行时对 DSL 文本进行沙箱验证和编译。

- 子图通过 `GraphEngine.executeNestedGraph()` 运行——从父级继承监听器、持久化服务和执行 ID 链。

10.15 下一步

在第 11 章—批处理与迭代中，你将学习 `foreach` 和 `loop` 如何让你对集合和轮询周期重复执行工作——这两者在底层都会将其主体编译为嵌套子图。

10.16 参考链接

- DSL 规范 §5.9 —作用域模式编译 —`scope`、默认矩阵和编译行为
- 核心架构 —引擎内部机制和嵌套执行
- Operator 设计规范 —operator 层级分类（`SubGraphOperator` 是能力层 operator）
- `loan-approval-subgraph.bloge` —主要示例
- `LoanApprovalSubGraphDslExample.java` —子图注册的 Java 接线
- `LoanApprovalSubGraphExample.java` —使用类型化 record 的纯 Java API 版本
- `international-shipment.bloge` —并行双子图示例
- `smart-ticket-handling.bloge` —子图 + 分支组合
- `order-full-pipeline.bloge` —支付 + 库存子图
- `dynamic-agent.bloge` —动态子图示例
- `SubGraphOperator.java` —引擎层面的嵌套执行
- `DynamicSubGraphOperator.java` —沙箱化的运行时 DSL 编译
- `DslCompiler.java` —`registerSubGraph()` 和 import 解析
- `StandardNodeCompiler.java` —子图节点编译逻辑

章节
11

第三阶段——组合与时间

第 11 章——批处理与迭代

子图、迭代、挂起与崩溃恢复

目标：让工作跨集合、外部时间与进程重启继续成立。

承诺：读完本章后，你将知道如何使用 *`foreach`* 处理集合中的每一项，使用 *`loop`* 重复工作直到条件满足，并在单个 *graph* 中组合这两种构造——所有这些都无需在 *operator* 代码中编写任何 *`for`*/*`while`* 循环。

11.1 学习目标

1. 编写一个 `foreach` 块，对集合进行扇出——默认并行或顺序执行——并在主体中引用当前项和索引。
2. 编写一个包含 `until`、`max_iterations`、`delay` 和 `carry` 的 `loop` 块，表达有界迭代并在轮次之间转发状态。
3. 使用三个 loop 特有的隐式变量——`carry.<path>`、`prev.<nodeId>.<path>` 和 `loopIteration`——并解释每个变量在运行时如何解析。
4. 在下游节点中使用 `depends_on` 消费 `foreach` 或 `loop` 的聚合输出。
5. 识别 `foreach` 和 `loop` 何时可以在同一个 graph 中组合——以及何时 `stream foreach` / `stream loop` 是更好的选择。

11.2 前置条件

- 第 10 章——用子图复用——你需要理解 `foreach` 和 `loop` 的主体会被编译为嵌套子图。

11.3 源示例

文件	展示内容
`batch-order-processing.bloge`	最简单的并行 `foreach`——获取订单、验证、处理

文件	展示内容
batch-order-parallel.bloge	并行 foreach，带有下游 summarize 节点消费聚合输出
sequential-transfer.bloge	foreach ⋯sequential ——逐一执行的银行转账
status-polling.bloge	基本的 loop，带 until 和 loopIteration ——轮询直到 READY
cursor-pagination.bloge	带多字段 carry 的 loop ——API 游标分页
retry-with-backoff.bloge	基于 loop 的自定义重试，带 carry 和 loopIteration
logistics-batch-dispatch.bloge	同一个 graph 中的 foreach + loop ——调度包裹，然后轮询状态
basic.bloge (foreach)	一致性测试 fixture：最小的 foreach
sequential-index.bloge	一致性测试 fixture：带索引绑定的 foreach ⋯sequential
with-carry.bloge	一致性测试 fixture：带 carry、prev 和 loopIteration 的 loop

11.4 为什么这很重要

到目前为止，你的 graph 中每个节点最多运行一次。真实系统很少这么简单：

- 一个电商结账流程必须验证每一条订单行——而不是只验证一条。
- 一个数据同步任务必须翻页遍历 API 的所有页面，并向前传递游标。
- 一个部署流水线必须轮询外部系统，直到状态变为 READY。

在过程式代码中，你会使用 for 或 while。在 BLOGE 中，等价的构造是 foreach（处理集合中的每一项）和 loop（重复直到条件成立）。两者都是超级节点（**hyper-node**）：它们在外层 graph 中表现为单个节点，但内部会编译为一个嵌套子图，引擎会像对待其他 graph 一样对其进行调度、重试和观测。

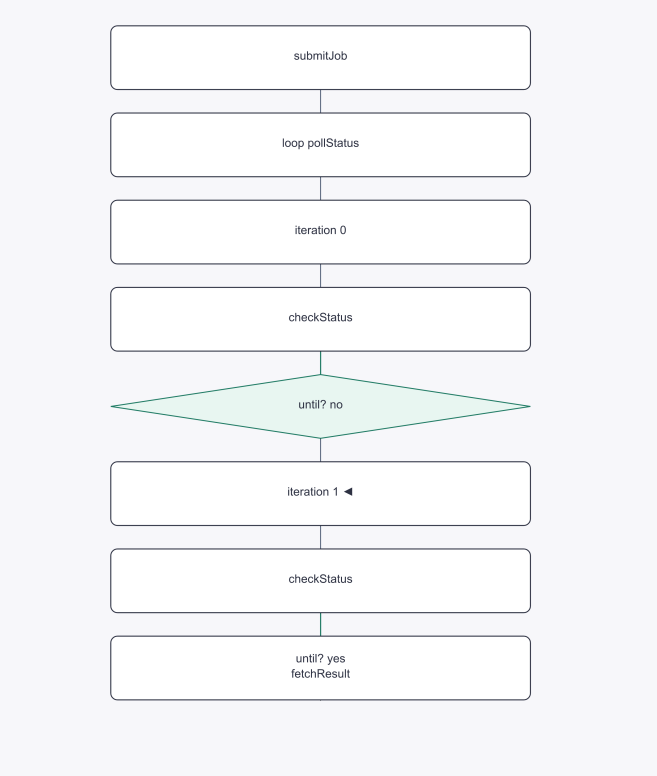
核心好处是：迭代语义——并行度、顺序、有界重试、carry 状态——都声明在 DSL 中，而不是埋在 operator 代码里。引擎拥有循环；你的 operator 只需处理单个项或单次迭代。

11.5 心智模型

11.5.1 `foreach` ——对集合进行扇出

把 `foreach` 想象成一台冲压机：引擎拿到集合，为每个元素冲出一份主体子图的副本，运行它们（默认并行，或顺序执行），然后将结果收集为聚合输出。

11.5.2 `loop` ——重复直到完成



把 `loop` 想象成一个带计数器的闸门：引擎重新执行嵌套子图，在每轮结束后评估 `until`，当谓词为 `true` 或达到 `max_iterations` 时停止。在迭代之间，引擎等待 `delay` 指定的时间，并将 `carry` 状态传入下一轮。

11.6 第一个可运行示例

下面是 `batch-order-processing.bloge` ——最简单的并行 `foreach`：

```
graph batchOrderProcessing {
  node fetchOrders : OrderFetcher {
    input {
      customerId = ctx.customerId
    }
  }

  foreach processOrders : (order, idx) in fetchOrders.output.orders {
    node validate : OrderValidator {
      input {
        order = order
        index = idx
      }
    }
    node process : OrderProcessor {
```

```
      depends_on = [validate]
      input {
        validated = validate.output
      }
    }
  }
}
```

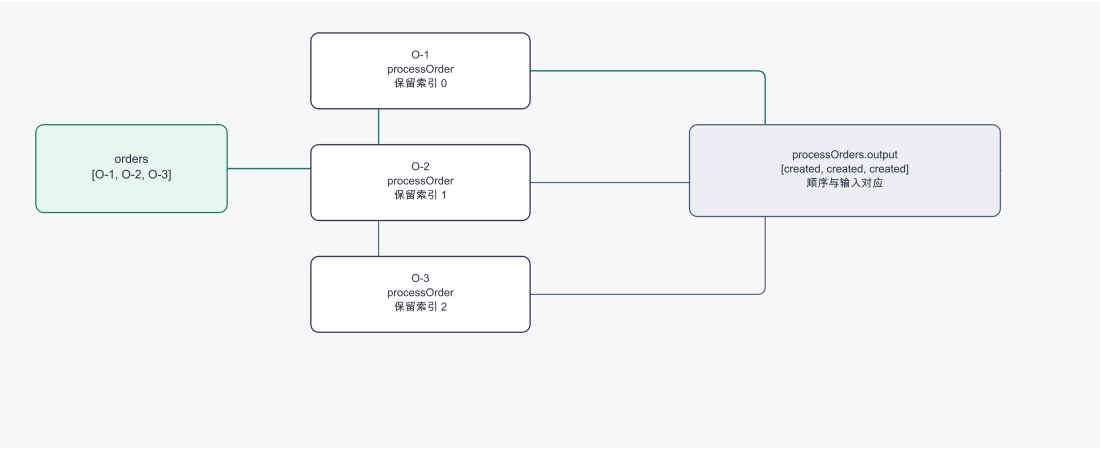
逐步解读：

1. `fetchOrders` 运行并产出 `output.orders` ——一个列表。
2. 引擎进入 `foreach processOrders`。对于列表中的每一个元素，它创建一份主体子图的副本，将 `order` 绑定到当前元素，`idx` 绑定到其从 0 开始的位置。
3. 默认情况下，所有副本并行运行。
4. 在每个副本中，`validate` 先运行；`process` 依赖于它。
5. 当所有副本完成后，`processOrders` 本身以聚合输出完成。

> 关键认识：你从不编写循环。引擎负责扇出、调度和收集。你的 *operator* ——*`OrderValidator`*、*`Order Processor`* ——只处理单个项。

11.7 完整故事 A：`foreach` 交付一张批处理 receipt

`orderBatch` 加载三笔订单，对每一项运行相同的 `processOrder` 能力，再把一组 可关联结果交给 `generate Report`。结果不是“后台发生了三次任务”，而是一张 父图可以检查的批处理 receipt。



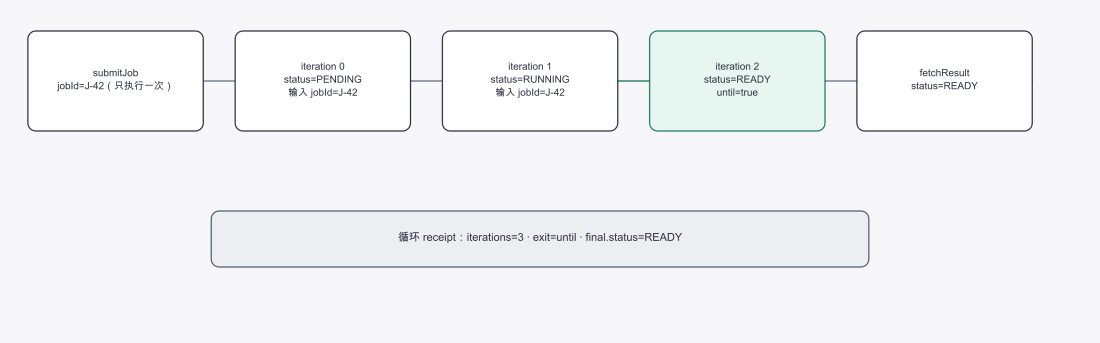
11.7.1 分开观察输入、单项结果与汇总

层级	示例观察	失败时要问什么
collection	输入包含 `O-1`、`O-2`、`O-3`	dispatch 前是否漏掉某项？
item	index `1` 对应 `O-2`	这一项执行了还是失败了？
aggregate	`totalProcessed=3`	父图是否收到完整 receipt？

companion Scenario 断言 `generateReport` 已执行且 `totalProcessed=3`。它证明 三项 fixture 在这次受控运行中到达报告；它不证明无限并行、所有失败策略下 的顺序，也不证明外部 effect 安全。

11.8 完整故事 B：`loop` 交付一张退出 receipt

轮询 loop 回答的是另一个问题。`submitJob` 只运行一次；loop 复用 `jobId=J-42` 检查状态，直到看到 `READY` 或触及 `max_iterations`。



11.8.1 `loop` 必须同时拥有记忆和停止规则

receipt 字段	示例	父图为什么需要它
`iterations`	`3`	区分立即成功与重复轮询
`exit`	`until`	区分业务条件成立与保护上限
最终输出	`status=READY`	不暴露每轮工作台也能给 `fetchResult` 供值

上一轮输出，以及 DSL 明确声明的 `carry` 字段，共同构成 loop 的记忆。本例的 `jobId` 始终来自父图输入，并没有被伪装成 carry 状态。`until` 是业务停止条件，`max_iterations` 是安全停止条件。如果达到安全上限时仍为 `PENDING`，不能因为引擎正常结束就把轮询写成 成功。

`foreach` 把一个 collection 展开为多个兄弟执行；`loop` 让一个逻辑活动跨轮 演进。后面可以组合两者，但前提是两张 receipt 都能独立读懂。

11.9 拆开来看

11.9.1 `foreach` 语法

正式语法（DSL 规范 §3.1）：

```
ForEachDef = "foreach" IDENT ":" ForEachBinding "in" Expression
             "sequential"? "{" body "}"

ForEachBinding = "(" IDENT ("," IDENT)? ")"   // 带可选索引的元组
               | IDENT                         // 裸项变量
```

部分	示例	含义
ID	`processOrders`	外层 graph 中超级节点的名称
绑定	`(order, idx)`	`order` = 当前项，`idx` = 从 0 开始的索引。两者都是可选的——`(order)` 或裸 `order` 都有效。
`in` 表达式	`fetchOrders.output.orders`	要迭代的集合
`sequential`	（关键字，可选）	逐一处理项，而非并行
主体	`{ node …}`	编译为 `<foreachId>__subgraph__` 的嵌套子图

11.9.1.1 `foreach` 内部的隐式变量

变量	AST 类型	运行时上下文键	含义
`<itemVar>`（如 `order`）	`ItemPath`	`__item__`	当前集合元素
`<itemVar>.<field>`（如 `order.amount`）	`ItemPath`	`__item__`	当前元素上的字段
`<indexVar>`（如 `idx`）	`ItemIndex`	`__itemIndex__`	从 0 开始的整数位置

（参见 *DSL 规范规则 7–8*。）

11.9.1.2 并行 vs. 顺序

在 `sequential-transfer.bloge` 中，银行转账必须按顺序处理，以便每笔转账能看到前一笔更新后的余额：

```
foreach processTransfers : (transfer, idx) in fetchTransfers.output.transfers sequential {
  node riskCheck : RiskCheckOperator {
    input {
      amount      = transfer.amount
      fromAccount = transfer.fromAccount
      toAccount   = transfer.toAccount
      index       = idx
    }
  }
  node executeTransfer : TransferExecutionOperator {
    depends_on = [riskCheck]
    input {
      transfer   = transfer
      riskResult = riskCheck.output
    }
  }
  node recordLedger : LedgerRecordOperator {
    depends_on = [executeTransfer]
    input {
      transfer  = transfer
      execution = executeTransfer.output
```

```
      index     = idx
    }
  }
}
```

唯一的语法区别是在 `in` 表达式后面加上 `sequential` 关键字。运行时引擎按列表顺序逐一执行每个项。

11.9.1.3 消费 foreach 的下游输出

一个 `foreach` 超级节点会暴露聚合输出。任何下游节点都可以通过 `depends_on` 依赖该 foreach 并读取它。摘自 `batch-order-parallel.bloge`：

```
node summarize : BatchSummaryOperator {
  depends_on = [processOrders]
  input {
    results = processOrders.output
  }
}
```

11.9.2 `loop` 语法

```
LoopDef  = "loop" IDENT "{" LoopBody "}"

LoopBody = ( "max_iterations" "=" NUMBER
           | "delay" "=" DURATION
           | DependsOn
           | Member
           | CarryBlock
           | UntilClause )*

CarryBlock  = "carry" "{" ( IDENT ":" Expression ","? )* "}"
UntilClause = "until" Expression
```

部分	示例	含义
ID	`pollStatus`	外层 graph 中超级节点的名称
`max_iterations`	`20`	硬性上限——即使 `until` 从未触发，引擎也会停止
`delay`	`2s`	迭代之间的暂停时间
`until`	`checkStatus.output.status == "READY"`	提前退出谓词——在每次迭代后评估
`carry { …}`	`cursor: fetchPage.output.nextCursor`	转发到下一次迭代的命名状态
主体	`node …`	编译为 `<loopId>__subgraph__` 的嵌套子图

11.9.2.1 `loop` 内部的隐式变量

变量	AST 类型	运行时上下文键	含义
`carry.<path>`	`LoopCarryPath`	`__carry__`	来自上一次迭代（或初始输入）的状态 map
`prev.<nodeId>.<path>`	`LoopPrevPath`	`__prev__`	上一次迭代的节点输出
`loopIteration`	`LoopIterationRef`	`__loopIteration__`	从 0 开始的迭代计数器

（参见 *DSL 规范规则 9–11*。）

11.9.2.2 基本 loop ——状态轮询 摘自 `status-polling.bloge`：

```
loop pollStatus {
  max_iterations = 20
  delay = 2s
  depends_on = [submitJob]
  node checkStatus : StatusCheckerOperator {
    input {
      jobId     = submitJob.output.jobId
      iteration = loopIteration
    }
  }
  until checkStatus.output.status == "READY"
}
```

引擎的执行流程：

1. 运行 `checkStatus`（第 0 次迭代）。
2. 评估 `until`。如果 `status == "READY"` → 退出。
3. 否则，等待 **2** 秒，递增 `loopIteration`，然后重复。
4. 经过 20 次迭代仍未匹配后，循环停止——`max_iterations` 是安全网。

11.9.2.3 带 carry 的 loop ——游标分页 摘自 `cursor-pagination.bloge`：

```
loop fetchAllPages {
  max_iterations = 100
  delay = 500ms
  depends_on = [initPagination]

  node fetchPage : PageFetcherOperator {
    input {
      endpoint = ctx.endpoint
      pageSize = ctx.pageSize
      cursor   = carry.cursor
    }
  }
  node transformPage : PageTransformerOperator {
```

```
    depends_on = [fetchPage]
    input {
      records      = fetchPage.output.records
      currentTotal = carry.totalRecords
    }
  }

  carry {
    cursor:       fetchPage.output.nextCursor
    totalRecords: transformPage.output.runningTotal
  }
  until fetchPage.output.hasMore == false
}
```

要点：

- `carry.cursor` 和 `carry.totalRecords` 在主体中被读取——它们来自上一次迭代的 `carry { …}` 块。
- 在第一次迭代时，`carry` 字段未设置（null），因此 operator 应当处理初始情况。
- `carry` 块定义了在每次迭代结束时写入的状态，供下一次迭代读取。

11.9.2.4 消费 loop 的下游输出

loop 完成后，其输出是最后一次迭代的节点输出，以节点 ID 为键：

```
node finalizeData : DataFinalizerOperator {
  depends_on = [fetchAllPages]
  input {
    totalRecords = fetchAllPages.output.transformPage.runningTotal
    lastCursor   = fetchAllPages.output.fetchPage.nextCursor
  }
}
```

11.9.3 在同一个 graph 中组合 foreach 和 loop

`logistics-batch-dispatch.bloge` 展示了两种构造协同工作：

1. 一个 `foreach` 对包裹进行扇出（路线规划 → 调度）。
2. 一个 `loop` 在 foreach 完成之后轮询聚合的配送状态，通过 `depends_on = [assignRoutes]` 连接。
3. 最终的 `dispatchReport` 节点依赖于 loop。

这展示了核心组合规则：`foreach` 和 `loop` 是超级节点——你像连接其他节点一样用 `depends_on` 连接它们。

11.9.4 关于流式变体的说明

DSL 还支持 `stream foreach` 和 `stream loop`。它们会在每个项或每次迭代的结果完成时立即向下游发送，而不是等待整个批次完成。例如，摘自 `streaming-batch.bloge`：

```
stream foreach processOrders : item in loadOrders.output.orders {
```

```
  buffer = 16
  node processItem : OrderProcessor {
    input { order = item }
  }
}
```

流式处理是一个超出本章范围的高级主题。关键要知道的是它的存在，以及 `buffer = N` 控制流式边的环形缓冲区容量。

11.10 限流与单项容错

并行 `foreach` 默认是”一次性把所有 **item** 全部启动”。在小列表上很快，但对一个只能扛 50 路并发的下游服务一下子扇出 10,000 个 item，就是灾难。有两个 `foreach` 属性可以解决，且不用动 body:

```
foreach order in orders {
  batch_size      = 25            // 同一时刻最多 25 个 item 在跑
  on_item_failure = continue       // 一个坏 item 不会拖垮整批

  node process : ProcessOrderOperator {
    input { order = item }
  }
}
```

11.10.1 batch_size

取值	含义
省略	不限流——引擎为每个 item 启动一个任务
`batch_size = N`($N \geq 1$)	引擎同一时刻最多处理 N 个 item; 一个完成就启动下一个

`batch_size` 是局部限流。全局封顶请看下面的 `maxGlobalForeachBatches`。

11.10.2 on_item_failure

取值	单个 **item** 失败时的行为
`abort_all`(默认)	把异常向上抛; 整个 `foreach` 失败——经典 fail-fast 语义
`abort_batch`	把当前在跑的 item 跑完，然后 `foreach` 标记失败; 后续 item 不再启动
`continue`	把失败记进聚合输出，继续处理剩下的;`foreach` 自身成功完成

设为 `on_item_failure = continue` 时，聚合输出保留输入顺序，并为失败 item 留下哨兵记录，下游节点可以按 `item.status` 分支:

```
node summarize : SummarizeBatchOperator {
```

```
  depends_on = [process]
  input {
    results = process.output       // List，既有成功项也有失败项
  }
}
```

11.10.3 全局封顶——maxGlobalForeachBatches 和 defaultForeachProtection

batch_size 是单个 foreach 上的设置。在繁忙的服务里你还想要一个 系统级封顶，防止某个热点租户把整个服务挤死。这两个引擎级开关对所有 图都生效:

```
GraphEngine engine = GraphEngine.builder()
    .maxGlobalForeachBatches(200)          // 跨所有图的硬上限
    .defaultForeachProtection(             // 对 .bloge 里未声明 batch_size
        ForeachProtection.builder()        //   的 foreach 的兜底
            .batchSize(50)
            .onItemFailure(OnItemFailure.ABORT_BATCH)
            .build())
    .build();
```

- maxGlobalForeachBatches: 集群级上限。触顶后新批次排队等已有的跑完。
- defaultForeachProtection: 对 .bloge 中未指定 batch_size / on_item_failure 的 foreach 应用的兜底配置。生产环境务必配置，以免 漏配某个开关就把集群打爆。

小测验: *on_item_failure = continue* 什么时候应该配人工告警? 提示:*50* 个 *item* 失败了 *49* 个，批次还显示"OK"，你就掩盖了一次真实故障。*continue* 必须搭配一个下游节点检查失败率。

11.11 常见陷阱

11.11.1 [X] 忘记 carry 字段在第一次迭代时为 null

```
// 错误 —— 在第 0 次迭代时 cursor 将为 null；operator 可能会崩溃
loop fetchPages {
  max_iterations = 50
  node fetch : PageFetcher {
    input {
      cursor = carry.cursor    // 第一次迭代时为 null!
    }
  }
  carry { cursor: fetch.output.nextCursor }
  until fetch.output.done == true
}
```

在第一次迭代时，没有上一轮的 carry ——每个 carry.<field> 都解析为 null。你的 operator 必须处理这种情况:

```
// 在 PageFetcher.execute(...) 中
String cursor = (String) input.get("cursor");
if (cursor == null) {
    // 第一页 ——从头开始
}
```

这不是 DSL 的 bug。它和循环变量在第一次迭代前未初始化是同一个道理。设计你的 operator 时，将 `null` carry 值视为" 从头开始"。

11.11.2 [X] 在 `loop` 上省略 `max_iterations`

`max_iterations` 是硬性安全网。没有它，一个不稳定的 `until` 条件可能导致循环无限运行。即使你预期 `until` 会远在上限之前触发，也要始终声明一个合理的上限。

11.12 常见故障

11.12.1 `loop` 缺少 `max_iterations`

```
loop pollStatus {
  until = checkStatus.output.status == "READY"
  delay = 5s
  // No max_iterations — the compiler rejects this
  body {
    node checkStatus : StatusCheckerOperator { … }
  }
}
```

解析器会直接拒绝这种写法：

```
GraphDefinitionException: Loop 'pollStatus' is missing required 'max_iterations'
```

修复：始终声明 `max_iterations`。选择一个能够覆盖现实最坏情况、并留有余量的上限。

11.13 引导式重写

从一致性测试 fixture `with-carry.bloge` 开始：

```
graph g {
  loop processBatch {
    max_iterations = 100
    delay = 5s
    node fetchBatch : BatchFetcher {
      input {
        cursor     = carry.cursor
        prevResult = prev.fetchBatch.result
        iteration  = loopIteration
```

```
      }
    }
    carry { cursor: fetchBatch.output.nextCursor }
    until fetchBatch.output.done == true
  }
}
```

思考以下问题：

1. 这里使用了哪三个隐式变量？（*carry.cursor* ——来自上一次迭代的 *carry* 状态。*prev.fetchBatch.result* ——上一次迭代 *fetchBatch* 输出的 *result* 字段。*loopIteration* ——从 *0* 开始的迭代计数器。）
2. 在第 **0** 次迭代时，**carry.cursor** 的值是什么？（*Null* ——还没有上一轮的 *carry*。*operator* 必须处理这种情况。）
3. 在第 **0** 次迭代时，**prev.fetchBatch.result** 的值是什么？（*Null* ——没有上一次迭代。与 *carry* 相同的规则。）
4. 如果 **fetchBatch.output.done** 永远不变为 **true**，会怎样？（循环运行 *100* 次——*max_iterations* 是硬性停止条件——然后退出。）

现在自己扩展这个 graph：

- 在 loop 主体中添加一个 `node transformBatch`，使其依赖 `fetchBatch`，对原始记录进行转换。
- 更新 `carry` 块，增加 `totalRecords: transformBatch.output.runningTotal`。
- 在 loop 外部添加一个下游 `node finalize`，消费 `processBatch.output.transformBatch.runningTotal`。

将你的结果与 `cursor-pagination.bloge` 进行对比——它遵循相同的模式。

11.14 脑力检查

1. `foreach` 的默认执行模式是并行还是顺序？（并行。添加 *sequential* 关键字可切换为逐一执行。）
2. 在 `foreach` 主体中有哪两个隐式变量可用？（项变量——如 *order*——解析为 *ItemPath*；以及可选的索引变量——如 *idx*——解析为 *ItemIndex*。）
3. 列出 `loop` 主体中可用的三个隐式变量。（*carry.<path>* ——*LoopCarryPath*。*prev.<nodeId>.<path>* ——*LoopPrevPath*。*loopIteration* ——*LoopIterationRef*。）
4. 当 `until` 永远不为 `true` 时会怎样？（循环运行到 *max_iterations* 为止，然后退出。）
5. 下游节点如何消费 `foreach` 或 `loop` 的输出？（通过声明 *depends_on = [<foreachOrLoopId>]* 并读取 *<id>.output*。）
6. 基于 `loop` 的重试和节点级 `retry` 有什么区别？（基于 *loop* 的重试让你在每次尝试中运行任意子图、携带自定义状态并计算动态退避。节点级 *retry* 是引擎自动应用于单个 *operator* 调用的声明式策略。）
7. 设计题：你有 1 000 个订单要处理。一个并行 `foreach` 会立刻扇出到 1 000 个虚拟线程，但下游支付网关只允许 50 个并发请求。你会如何处理？（引擎本身没有内建的 *foreach* 并发限流。常见做法是：先把订单切成每批 *50* 个的分组、在 *operator* 内部做限流或信号量控制，或者如果顺序本来就重要，就改用 *foreach sequential*。真正的设计取舍在于你更看重吞吐量还是顺序约束。）

11.15 练习

1. 并行 **foreach** ——批量验证 打开 `batch-order-parallel.bloge`。
   - 在 `foreach` 主体中添加第三个节点 `notifyCustomer`，使其依赖 `deductStock`。
   - 添加一个 `input` 绑定，读取 `order.orderId` 和 `deductStock.output.remaining`。
   - 预测：添加这个节点是否会改变跨项的并行度？（不会——每个项的子图仍然是独立的。新节点只在每个项的副本内增加了一个步骤。）
2. 顺序 **foreach** ——顺序很重要 打开 `sequential-transfer.bloge`。
   - 删除 `sequential` 关键字。
   - 预测：会发生什么变化？（所有转账现在并行执行——风险检查和余额扣减交织进行。这对于顺序很重要的银行转账来说是错误的。）
3. 带 **carry** 的 **loop** ——构建分页器 从头编写一个新 graph：
   - `node initSearch` 提供初始查询。
   - `loop fetchPages`，设置 `max_iterations = 50`，`delay = 1s`。
   - loop 内部：`node fetchPage` 读取 `carry.pageToken`。
   - `carry { pageToken: fetchPage.output.nextPageToken }`。
   - `until fetchPage.output.hasMore == false`。
   - loop 之后：`node aggregate` 消费 `fetchPages.output`。
   - 与 `cursor-pagination.bloge` 进行对比。
4. 组合 **foreach** + **loop** 打开 `logistics-batch-dispatch.bloge`。
   - 追踪数据流：哪个节点的输出流入 `foreach`？哪个 `depends_on` 将 `foreach` 输出连接到 `loop`？哪个将 `loop` 连接到最终报告？
   - 在 foreach 内部的 `planRoute` 上添加 `retry = { attempts: 2, backoff: 500ms, strategy: fixed }`。这会影响 loop 吗？（不会——*retry* 是按节点的，在每个 *foreach* 项的子图内部。）

11.16 实验验收卡

- 预期与观察： foreach 交付逐项 receipt，loop 交付退出 receipt。
- 失败与恢复：让 loop 永不满足 until；恢复上限或 carry 更新。
- 证明边界：证明批处理与迭代合同不同，不证明无限流自动收敛。
- 练习合同：三项批次和一个 loop；只改退出条件；交付两类 receipt；关联完整且退出原因唯一即停止。

11.17 回顾

- **`foreach`** 对集合进行扇出。每个项获得主体子图的一份副本。默认并行；添加 `sequential` 可按顺序执行。
- 在 `foreach` 内部，项变量（如 `order`）和可选的索引变量（如 `idx`）是隐式的——你在绑定中声明它们，在 input 表达式中使用它们。
- **`loop`** 重复执行一个嵌套子图，直到 `until` 评估为 `true` 或达到 `max_iterations`。`delay` 在轮次之间暂停。
- 在 `loop` 内部，三个隐式变量提供迭代上下文：`carry.<path>`（转发状态）、`prev.<nodeId>.<path>`（上一次迭代的输出）和 `loopIteration`（从 0 开始的计数器）。
- 两种构造都编译为超级节点——它们在外层 graph 中表现为单个节点，像其他节点一样通过 `depends_on` 连接。
- `carry` 块定义在每次迭代结束时写入的状态。在第一次迭代时，carry 字段为 `null` ——operator 必须处理这种情况。
- 流式变体（`stream foreach`、`stream loop`）在结果完成时即向下游发送，而不是等待整个批次完成。

11.18 下一步

在第 12 章——等待世界的回应中，你将学习 BLOGE graph 如何暂停并等待外部事件——定时器、人工审批和 webhook 回调——而不阻塞线程。

11.19 参考链接

- DSL 规范—ForEach 和 Loop 语法 —正式语法、绑定规则和执行语义
- DSL 规范—表达式规则 7–11 —`ItemPath`、`ItemIndex`、`LoopCarryPath`、`LoopPrevPath`、`LoopIterationRef`
- 隐式声明注册表 —完整的保留名称列表和运行时生命周期
- `batch-order-processing.bloge` —基本的并行 foreach
- `batch-order-parallel.bloge` —带下游 summarize 的 foreach
- `sequential-transfer.bloge` —foreach sequential
- `status-polling.bloge` —带 until 的基本 loop
- `cursor-pagination.bloge` —带 carry 的 loop
- `retry-with-backoff.bloge` —基于 loop 的自定义重试
- `logistics-batch-dispatch.bloge` —foreach + loop 组合
- `streaming-batch.bloge` —stream foreach 示例
- `basic.bloge (foreach)` —一致性测试 fixture
- `sequential-index.bloge` —一致性测试 fixture

- `with-carry.bloge` ——一致性测试 fixture

章节
12

第三阶段——组合与时间

第 12 章——等待世界的回应

子图、迭代、挂起与崩溃恢复

目标：让工作跨集合、外部时间与进程重启继续成立。

本章承诺：你会让一条 *graph* 在人工审批或 *webhook* 尚未到达时释放执行资源，读懂真实的挂起与恢复结果，并能识别错误关联键、重复事件和 `0.9.8-RC1` 的已知语义边界。

12.1 学习目标

学完本章，你能够：

1. 用业务语言解释阻塞、轮询和挂起的资源差异。
2. 从 `GraphResult` 判断 graph 是失败、完成还是仍在等待。
3. 根据外部回应的形状，在 `wait` 与 `await` 之间做选择。
4. 用一次单因素实验验证错误 correlation key 和重复事件的处理结果。

12.2 这一章只增加一个变量

前一章的循环仍发生在一次活跃执行里：取一页、处理一页、再取下一页。本章只增加一件事——下一步的前提暂时不在系统手里。

客服经理可能两小时后批准退款，支付平台可能十分钟后推送回调，仓库也可能明天才发送到货事件。graph 不能一直占着工作线程等世界回答，也不能每秒追问一次数据库“有结果了吗”。它需要记住自己等什么，然后把控制权交还给运行时。

12.3 先遇到问题：一张等待经理批准的工单

客户 `C-17` 说退款迟迟未到账。系统创建工单 `T-2048`、完成分类并通知经理，然后必须等待批准结果。规则是：经理批准后才发确认；48 小时没有回应则进入超时处置。

在看代码前，先预测三种实现会发生什么：

做法	等待期间占用	外部系统压力	重启后的难点
阻塞线程 48 小时	一个执行线程和关联状态	无	进程消失，内存等待也消失
每 5 秒轮询一次	定时任务、连接和查询	高	需要恢复轮询进度和去重
挂起并等待信号	一条等待记录	低	需要持久化身份、状态和关联条件

第三种方案不是“睡得更聪明”。它改变了控制模型：执行先到达稳定边界并返回，外部回应稍后触发另一段执行。

12.4 先跑一次：看到真实的挂起边界

书稿自带的 probe 会直接加载远端 examples 中的 `ticket-approval-wait.bloge`，为文档型 Operator 提供最小替身，然后执行两遍：第一遍停在挂起边界，第二遍收到人工审批信号后继续。

如果本机还没有安装 `0.9.8-RC1` 依赖，先从固定的 BLOGE submodule 安装：

```
cd submodule/bloge
mvn -pl bloge-core,bloge-dsl -am -DskipTests install
```

回到书稿根目录后运行：

```
cd probes/ch11-waiting
mvn test
```

2026-09-14 在 BLOGE commit `cc38fbe5` 上的关键输出如下；随机 `executionId` 和完整 Maven 日志已省略：

```
boundary.isSuccess=true
boundary.isSuspended=true
boundary.suspendedNodes={waitApproval=wait}
boundary.waitApproval=SUSPENDED
boundary.sendConfirmation=CANCELLED
resume.isSuccess=true
resume.isSuspended=false
resume.waitApproval=COMPLETED
resume.sendConfirmation=COMPLETED
```

先只读前三行：

- `isSuccess=true` 表示此刻没有节点错误，不表示业务流程已经结束。
- `isSuspended=true` 表示至少一个节点正在等待信号。
- `suspendedNodes` 同时给出节点身份 `waitApproval` 和本次需要匹配的 suspend key `wait`。

`sendConfirmation=CANCELLED` 也不是业务拒绝。它表示本轮调度在挂起边界收束，依赖等待节点的下游尚未运行。第二遍信号到达后，等待节点和确认节点才变为 `COMPLETED`。

观察结论：“成功但挂起”是合法的中间结果。只检查 *`isSuccess()`* 会把尚未完成的流程误报为完成。

Probe 源码在 `WaitingForWorldProbeTest.java`。它是书稿的可复核观察，不代替 BLOGE 自身的耐久性、并发或生产验收。

12.5 挂起到底保存了什么

把挂起想成餐厅的取餐牌。顾客不需要站在窗口占着服务员；餐厅只要记住“哪一单、等什么、结果送回哪里”。BLOGE 的对应关系是：

现实对象	**BLOGE** 中的身份	作用
一笔流程	`executionId`	找回同一次 graph 执行
一个等待点	`nodeId`	找到恢复入口
取餐牌号码	suspend/correlation key	防止回应送错流程
已完成步骤	checkpoint/status	恢复时不重做已完成工作
外部回应	signal payload	成为等待节点的输出

这五样东西缺一不可。只有 payload 而没有执行身份，系统不知道唤醒谁；只有执行身份而没有等待节点，系统不知道从哪里继续；没有已完成状态，恢复可能重复执行已经发生的 effect。

12.6 两个入口：`wait` 与 `await`

它们共享“挂起后恢复”这一机制，但回答不同问题。

12.6.1 `wait`：时间或直接信号决定下一步

工单示例的核心差异只有这一段：

```
wait waitApproval = 48h after notifyManager {
  signal_key = ctx.ticketId
  on_timeout {
    decision = "auto-approved"
    approver = "system"
  }
}
```

语义意图是：notifyManager 完成后挂起；人工处理可以直接向 waitApproval 发送结果，计时器到期也可以产生超时结果。适合明确的时间点、延迟或某个已知等待节点。

12.6.2 await：事件名称和业务键共同决定下一步

支付平台不会知道 BLOGE 的 executionId。它只会推送“订单 0-42 的支付已确认”。这时使用 payment-wait.bloge 中的事件关联：

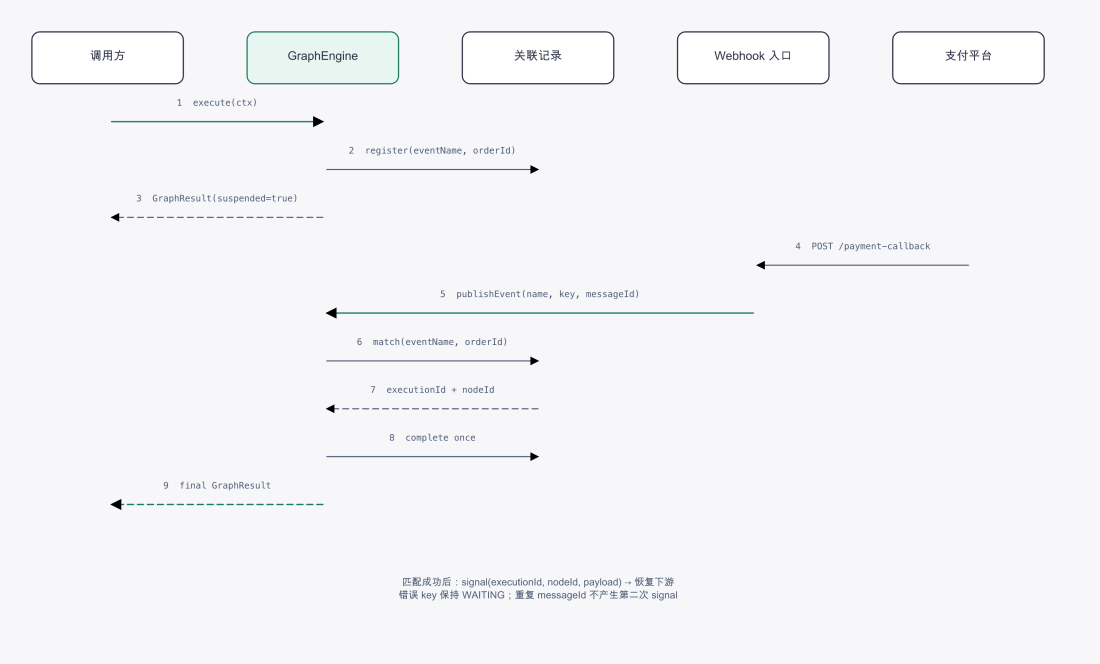
```
await awaitPayment {
  event "payment.confirmed" where orderId = createOrder.output.orderId
  timeout = 15m
  on_timeout { status = "timeout" }
}
```

Webhook handler 把外部协议翻译成运行时事件：

```
engine.publishEvent(
    "payment.confirmed",
    orderId,
    Map.of("status", "confirmed", "transactionId", transactionId),
    webhookMessageId
);
```

eventName 说明发生了什么，orderId 说明属于哪一笔业务，webhookMessageId 用于识别重复投递。不要把这三种身份压成一个字符串。

12.7 一张图串起 suspend、webhook、correlation 和 resume



读图时抓住两次控制权转移：

1. `execute` 到达等待点后写入等待事实，并把“仍在等什么”返回给调用方。
2. Webhook 到达后，`publishEvent` 先按事件名和业务键查找等待记录；匹配成功才向原 `executionId + nodeId` 发送 signal，继续下游。

因此 webhook endpoint 不是“继续执行函数”的公开入口。它只负责认证外部消息、提取稳定业务键、提供幂等消息 ID，并调用事件发布入口。真正的恢复身份由关联记录给出。

12.8 原理：恢复不是从头再跑

收到 signal 后，运行时应满足三个不变量：

1. 已完成节点保持已完成，不因恢复而无条件重做。
2. Signal payload 成为等待节点的输出，下游通过普通依赖读取它。
3. 同一等待条件只能完成一次；迟到或重复消息不能改写第一次已接受的结果。

在 probe 第二遍运行中，`Map.of("decision", "approved", "approver", "manager-7")` 被写成 `waitApproval` 的输出，因此 `sendConfirmation` 能继续读取 `decision` 和 `approver`。这就是“外部世界的回答重新进入 graph 数据流”的具体位置。

耐久化恢复还需要 execution checkpoint、wait store 和可重建的 graph definition。那是下一章的唯一新增变量；本章只建立进程内可观察的挂起协议。

12.9 单因素破坏一：把 correlation key 写错

保持事件名和 payload 不变，只把 `orderId` 从 `0-42` 改成 `0-24`：

```
var matched = engine.publishEvent(
    "payment.confirmed", "O-24", payload, "msg-901");
```

预期观察不是异常，也不是恢复错误订单，而是：

```
matched=[]
awaitPayment=WAITING
downstream.started=false
```

没有匹配项时应保持原 correlation 为 `WAITING`。恢复动作是核对 webhook 中的业务键提取规则，再用一个新的消息 ID 重放正确事件；不要修改等待记录去迎合错误消息。

这个反例防住的是“串单”。在支付、物流和审批系统里，唤醒错误流程通常比晚几分钟更危险。

12.10 单因素破坏二：重复投递同一事件

这次 key 正确，但两次投递使用相同 `webhookMessageId=msg-902`。`0.9.8-RC1` 的 `EventDeduplicationTest` 验证：相同 idempotency key 不会再次应用部分事件，也不会产生第二次 signal。

2026-09-14 的聚焦测试结果：

```
EventDeduplicationTest                Tests run: 5, Failures: 0
EventCorrelationAndOrTest             Tests run: 3, Failures: 0
SuspendSignalTest                     Tests run: 1, Failures: 0
GraphResultContractTest               Tests run: 4, Failures: 0
Total                                 Tests run: 13, Failures: 0
```

重复事件不是罕见异常，而是 webhook 的正常交付特征。幂等消息 ID 应来自发送方的稳定消息身份；用接收时间或随机 UUID 现造一个 ID，等于关闭去重。

12.11 RC1 边界：源码意图与实际行为不一致时必须停下

Probe 揭示了一个不能掩盖的差距：远端最新 examples 的工单 DSL 声明 `signal_key = ctx.ticketId`，但实际挂起输出是 `{waitApproval=wait}`。在固定的 `0.9.8-RC1` 源码中，`WaitAwaitCompiler` 解析了 `signal_key` 和 `on_timeout`，当前 `compileWait` 路径却只按 duration 构造 `WaitOperator`，没有把这两项绑定到运行时 Operator。

这意味着本章能证实：

- ticket graph 在 RC1 上能够到达 `SUSPENDED`，并可通过节点 signal 恢复。
- `GraphResult` 能区分成功、挂起和下游尚未运行。
- RC1 核心测试覆盖事件关联、AND/OR 和重复事件。

本章不能证实：

- 该 companion DSL 声明的动态 `signal_key=T-2048` 已被 RC1 执行。
- `on_timeout` 中的业务 payload 会在 48 小时后按示例自动注入。
- 进程退出后仍能恢复；这需要下一章的 durable store 和冷恢复证据。

此外，远端 examples 的 POM 仍固定 `0.3.1`，且依赖已经更名的 `bloge-core-ext`。直接构建会先在依赖解析处失败；即使覆盖为 `0.9.8-RC1` 也找不到该旧 artifact 名。不要把 examples 文件存在写成 RC1 companion 全

量通过。

> 工程判断：当 *DSL*、编译器和运行输出三者不一致时，以固定 *commit* 上的运行观察为准，记录差距并停止扩大承诺。

12.12 迁移到你的系统

在你熟悉的业务里找一个“下一步不在本系统手里”的动作，例如医院等待患者同意、物流等待海关放行、HR 等待候选人签约。产出一张四行责任卡：

项目	你要填写的内容
等待身份	`executionId` 和等待节点如何保存
业务关联键	哪个稳定字段能把回应送回正确实例
重复身份	发送方提供哪个 message/event ID
停止条件	多久超时，超时后谁有权决定结果

不要写实现代码，先让业务和平台负责人共同确认这四行。若“超时后自动批准”没有业务授权，就停在未定义状态，不要让框架替业务方发明答案。

12.13 实验：把轮询改成事件等待

选择一段现有轮询逻辑，只做以下改动：

1. 输入：一个稳定业务键、一种外部事件和一个可接受超时。
2. 修改范围：把轮询节点替换为一个 `await`，保留下游业务节点不动。
3. 运行：先发送错误 key，确认没有恢复；再发送正确 key，确认只恢复一次；最后重复相同消息 ID。
4. 产物：保存三次观察、关联键来源和超时责任人。
5. 停止条件：业务方无法确认超时结果，或发送方不能提供稳定重复身份时，停止实现并记录缺口。

12.14 实验验收卡

- 预期与观察： wait 后结果 suspended，正确 signal 恢复同一执行。
- 失败与恢复：发送错误 key 或重复事件；恢复正确 key 并幂等处理。
- 证明边界：证明 suspend/signal/resume，不证明 durable store 已配置。
- 练习合同：工单事件；只改 correlation key；交付挂起和恢复结果；错误 key 不推进、正确 key 只推进一次即停止。

12.15 回顾

- 挂起释放活跃执行资源，但保留恢复所需的身份和状态。
- `wait` 面向时间或直接 signal；`await` 面向事件名与业务键关联。
- `isSuccess=true` 与 `isSuspended=true` 可以同时成立。
- 错误 key 应保持无人匹配，重复事件应被幂等层吸收。
- 示例意图、编译器实现和实际输出必须三方对齐；不对齐时缩小承诺。

下一步只增加一个变量：第 13 章——持久执行会持久化恢复所需事实，让满足条件的进程在重启后继续同一笔执行。

12.16 精确事实入口

- `WaitingForWorldProbeTest.java` ——本章 ticket graph 挂起与恢复观察。
- `ticket-approval-wait.bloge` ——工单故事 DSL。
- `payment-wait.bloge` ——webhook 事件关联 DSL。
- `GraphResult.java` ——成功、挂起和节点状态合同。
- `EventCorrelationAndOrTest.java` ——AND/OR 恢复事实。
- `EventDeduplicationTest.java` ——重复事件与早到事件边界。

章节

13

第三阶段——组合与时间

第 13 章——持久执行

子图、迭代、挂起与崩溃恢复

目标：让工作跨集合、外部时间与进程重启继续成立。

承诺：读完本章后，你将理解 *BLOGE* 如何持久化执行状态，使 *graph* 在进程重启后恢复已接受的 *checkpoint*；当 *graph definition* 与 *durable store* 都可重建时，满足条件的另一 *worker* 可以继续执行。

13.1 学习目标

1. 解释”持久执行”在 BLOGE 中的含义——基于检查点的持久化，涵盖节点输出、挂起上下文、循环快照和执行标识，使 graph 能在崩溃后恢复。
2. 识别使持久化成为可能的运行时存储：`ExecutionStore`、`ExecutionCheckpointStore`、`WaitStore` 和 `WorkItemStore`。
3. 将内存持久化存储接入 `GraphEngine`，并在两个引擎实例之间运行挂起 → 信号 → 恢复流程。
4. 描述冷恢复的工作方式：执行标识 → graph 注册表查找 → 检查点重新加载 → 继续执行。
5. 识别检查点类型（`NODE_OUTPUT`、`LOOP_SNAPSHOT`、`FOREACH_PROGRESS`、`SUSPEND_CONTEXT`、`PARTIAL_RESULT`）及各自的写入时机。

13.2 前置条件

- 第 6 章——韧性设计 ——重试和超时基础。
- 第 12 章——等待世界的回应 ——挂起、信号和事件关联。
- 熟悉 `SuspendableOperator` 和 `OperatorResult.suspend()`。

13.3 源示例

文件	展示内容
`RuntimeGraphEngineDurabilityTest.java`	使用运行时存储的跨进程信号；两个引擎实例共享状态
`RuntimeGraphRegistryRecoveryTest.java`	从已发布的 graph 定义进行冷恢复；hash 不匹配拒绝；过期租约自动恢复
`PaymentWaitExample.java`	使用运行时存储的端到端挂起 → publishEvent → 恢复生命周期
`payment-wait.bloge`	DSL `await` 块，包含事件关联、超时和分支
`RuntimeLoopOperatorDurabilityTest.java`	循环快照检查点和从迭代中恢复

13.4 为什么这很重要

到目前为止你学到的一切——节点、依赖、分支、韧性、等待——都假设 JVM 保持存活。在生产环境中，这个假设会失效：

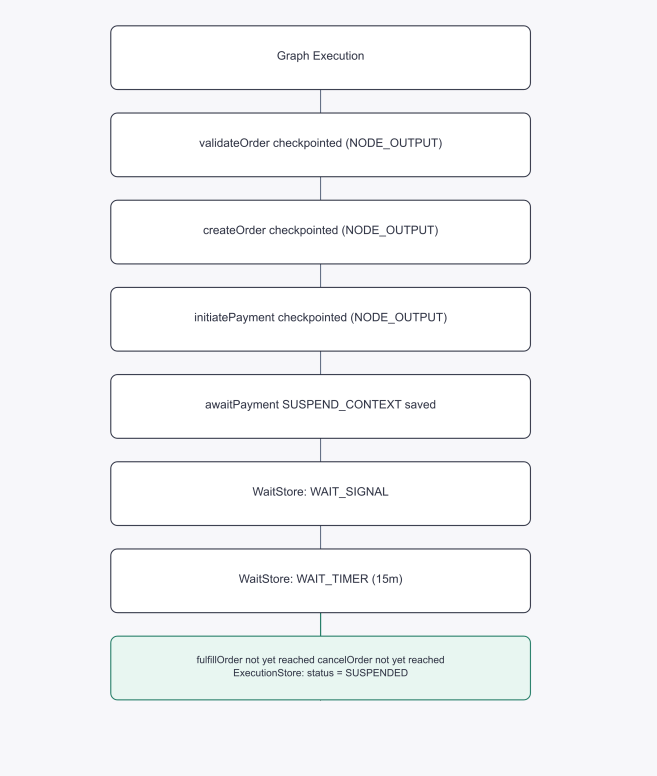
- 进程在执行过程中崩溃。没有持久化，所有进行中的节点输出都会丢失，工作流必须从头重新开始。
- 挂起可能持续数分钟、数小时甚至数天（等待支付回调、经理审批）。让一个线程阻塞这么长时间会浪费资源，且任何部署都会将其销毁。
- 水平扩展 意味着多个 worker 进程。在 Worker A 上挂起的 graph 必须能在 Worker B 上恢复，而 Worker A 可能永远不会回来。

持久执行解决了这三个问题。引擎在节点完成时写入 检查点 到持久化存储，并在节点挂起时写入 等待记录。恢复时——无论是由信号、定时器还是崩溃恢复触发——任何引擎实例都可以重新加载这些检查点，并从执行暂停的精确前沿继续。

> 核心洞察： *BLOGE* 中的持久化不是“事件溯源”或“从头重放”。引擎会 跳过 *checkpoint* 被接受的节点，只调度剩余前沿。若外部副作用已发生、但对应 *checkpoint* 尚未提交，该结果仍可能处于未知状态；幂等与对账仍是应用责任。

13.5 心智模型

把持久执行想象成 食谱书中的书签：



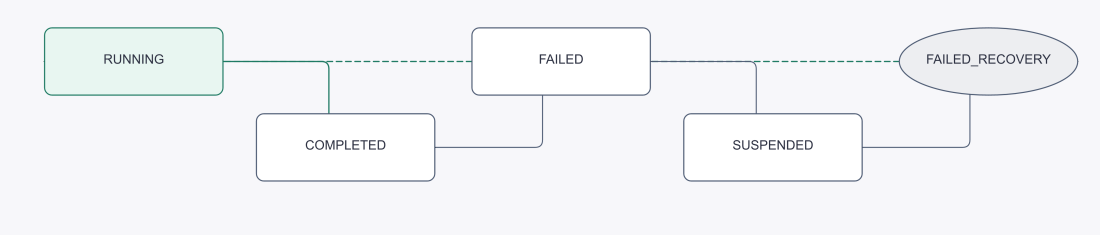
当支付事件到达时，任何引擎实例都可以：

1. 从 `ExecutionStore` 加载 `ExecutionInstance`。
2. 重新加载 三个已完成节点的 `NODE_OUTPUT` 检查点。
3. 解析 `WAIT_SIGNAL` 等待记录。
4. 从 `awaitPayment` 恢复 → branch → `fulfillOrder`。

该模型的四大支柱：

支柱	存储	记录内容
标识	`ExecutionStore`	执行生命周期、状态、租约、恢复尝试次数
检查点	`ExecutionCheckpointStore`	节点输出、循环快照、挂起上下文、foreach 进度
等待	`WaitStore`	执行为何暂停（信号、定时器、任务、事件）
工作项	`WorkItemStore`	待处理操作：定时器到期、事件匹配、任务恢复

执行生命周期如果画成状态图，会长这样：



13.6 bloge-runtime-spi 模块

在写任何代码之前，先给”缝”命名。持久化执行层被切成两个模块:

- **bloge-runtime-spi** ——只放接口。ExecutionStore、ExecutionCheckpointStore、WaitStore、TimerService、GraphRegistryStore、LeaseStore、AuditJournalStore、TaskInboxStore，加上这些 store 发出的强类型事件。
- **bloge-durable** ——引擎管道，消费这些接口: 恢复循环、哈希校验、checkpoint 编解码，以及测试用的内存实现。

数据库实现各自有模块 (MyBatis/JDBC 对应 bloge-durable-mybatis)。这个切分意味着: 如果你只是想插入一个自定义的存储，只依赖 bloge-runtime-spi 即可，无需引入引擎工件。

```
<dependency>
  <groupId>com.leanowtech</groupId>
  <artifactId>bloge-runtime-spi</artifactId>
</dependency>
```

13.7 第一个可运行的示例

本示例摘自 RuntimeGraphEngineDurabilityTest。它演示了一个 graph 在一个引擎实例上挂起、在另一个实例上完成——这正是你在生产中需要的”跨进程信号”模式。

13.7.1 步骤 1 —创建存储和 graph

```
// Shared durable stores (in production these would be database-backed)
var executionStore    = new InMemoryExecutionStore();
var checkpointStore   = new InMemoryExecutionCheckpointStore();
var waitStore         = new InMemoryWaitStore();
var timerService      = new InMemoryTimerService();

// A suspendable operator that parks execution
SuspendableOperator<Void, String> waitOp = (in, ctx) ->
        OperatorResult.suspend("ck", null, Duration.ofMinutes(10));
Operator<Object, String> completeOp = (in, ctx) -> "C";

var graphBuilder = new GraphBuilder("runtime-cross-process");
var graph = graphBuilder
        .suspendNode("w", waitOp)
        .node("c", completeOp).dependsOn("w")
        .build();
```

```
CountDownLatch suspended = new CountDownLatch(1);
AtomicReference<String> executionId = new AtomicReference<>();

ExecutionListener listener = new ExecutionListener() {
    @Override
    public void onGraphStart(String graphName, GraphContext ctx) {
        executionId.set((String) ctx.get(ReservedKeys.EXECUTION_ID));
    }

    @Override
    public void onNodeSuspended(String graphName, String nodeId, String suspendKey) {
        if ("w".equals(nodeId)) {
            suspended.countDown();
        }
    }
};
```

13.7.2 步骤 2 —在引擎 1 上执行直到挂起

```
var engine1 = GraphEngine.builder()
        .registry(new DefaultOperatorRegistry())
        .executionStore(executionStore)
        .executionCheckpointStore(checkpointStore)
        .waitStore(waitStore)
        .timerService(timerService)
        .listeners(List.of(listener))
        .build();

// 源测试把 executeWithOperators() 放到后台线程中运行，这样它就能先观察挂起，
// 再由第二个引擎实例恢复。
Thread runThread = Thread.ofVirtual().start(() -> {
    try {
        engine1.executeWithOperators(graph, null, graphBuilder.operators());
    } catch (Exception ignored) {
    }
});

// The execution is now SUSPENDED in the store
assertTrue(suspended.await(5, TimeUnit.SECONDS));
assertNotNull(executionId.get());
assertEquals(ExecutionStatus.SUSPENDED,
    executionStore.get(executionId.get()).orElseThrow().status());
```

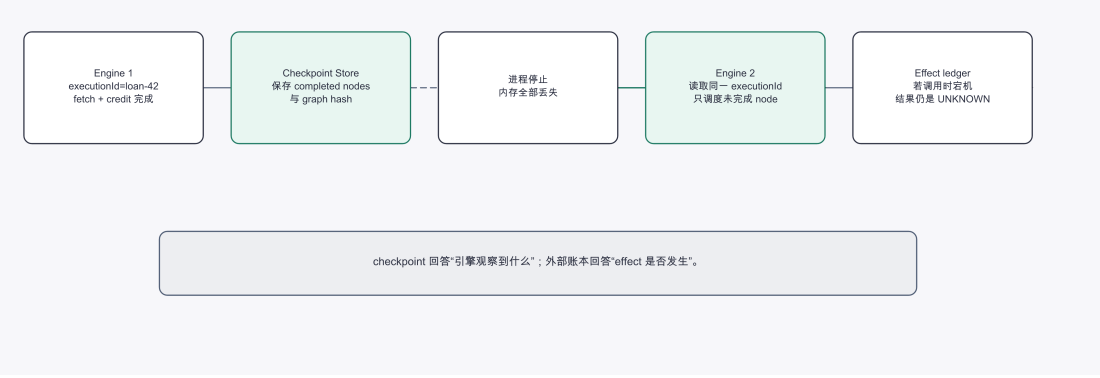
13.7.3 步骤 3 —在不同的引擎 2 上恢复

```
var engine2 = GraphEngine.builder()
        .registry(new DefaultOperatorRegistry())
        .executionStore(executionStore)          // same stores
        .executionCheckpointStore(checkpointStore)
        .waitStore(waitStore)
        .timerService(timerService)
        .build();

// Signal resumes the suspended node and continues to "c"
engine2.signal(graph, executionId.get(), "w", "resume-data");

// Execution completes
assertEquals(ExecutionStatus.COMPLETED,
    executionStore.get(executionId.get()).orElseThrow().status());
assertEquals("C",
    CheckpointCodec.DEFAULT.deserialize(
        checkpointStore.load(executionId.get(), CheckpointType.NODE_OUTPUT, "c")
            .orElseThrow().payload()));
runThread.join(1_000);
```

引擎 *1* 和引擎 *2* 是完全独立的对象。除了持久化存储外，它们不共享任何东西。这就是基本的生产模式：一个进程挂起，另一个进程恢复。

13.8 冷恢复：进程消失后，什么还能留下

warm resume 很容易被高估，因为原 engine 仍持有内存对象。真正的 durable 声明需要 Engine 1 消失、Engine 2 读取同一 `executionId`，并且已经完成的 工作仍保持完成。



13.8.1 跟随 `loan-42`，不要跟随 engine 对象

时刻	**durable** 事实	运行后果
Engine 1 写 checkpoint	`fetchApplication` 与 `checkCredit` 已完成	两者输出可以恢复
进程停止	heap 对象全部消失	内存引用不能证明任何 durable 事实
Engine 2 claim `loan-42`	graph hash 与 checkpoint 匹配	只有未完成 node 可进入调度
恢复完成	新 checkpoint 记录后续工作	同一逻辑 execution 继续前进

identity 属于 execution record，不属于任一 engine instance。派发未完成工作 之前，系统会重新加载 graph definition，并与保留状态核对。

13.8.2 **effect** 仍可能是 **UNKNOWN**

设想 Engine 1 调用了 `approveLoan`，放款方已经提交批准，但进程在 completion checkpoint 之前停止。Engine 2 只看见一个未完成 node；checkpoint 无法回答 外部 effect 是否发生。

因此，恢复仍需要第 6 章的合同：稳定幂等键、reconciliation 查询，以及明确的 retry 或人工修复策略。durable execution 能在完成记录已保留时避免再次调度 已完成的引擎工作；它不能为一个未协调的外部系统凭空制造 exactly-once effect。

13.9 拆解分析

13.9.1 检查点类型

引擎在不同的生命周期时刻写入不同的检查点类型。这些定义在 `CheckpointType` 中：

类型	写入时机	用途
`NODE_OUTPUT`	每个节点完成时	序列化的输出——恢复时跳过
`SUSPEND_CONTEXT`	operator 返回 `OperatorResult.suspend()` 时	挂起键、部分输出、graph 上下文快照
`PARTIAL_RESULT`	与 `SUSPEND_CONTEXT` 一起写入	挂起期间可见的中间输出
`LOOP_SNAPSHOT`	每次循环迭代后	`completedIterations` + `carryStateJson`，以便循环从中间恢复
`FOREACH_PROGRESS`	每个顺序 foreach 项之后	逐项进度，以便 foreach 可以跳过已完成的项
`EVENT_CORRELATION`	`await` 注册匹配器时	关联键，以便事件路由到正确的执行
`SESSION_PHASE_SNAPSHOT` / `SESSION_ROUND_SNAPSHOT`	Session 的 phase / 轮次转换时	Session 特有的状态（参见第 14 章——多轮 Session）

13.9.2 执行生命周期状态

`ExecutionStatus` 跟踪执行在其生命周期中的位置：

- `RUNNING` → 节点正在活跃执行
- `SUSPENDED` → 至少一个节点返回了 `OperatorResult.suspend()`

- `COMPLETED` → 所有终端节点成功完成
- `FAILED` → 节点在所有韧性尝试后失败
- `FAILED_RECOVERY` → 自动崩溃恢复超过了 `maxRecoveryAttempts`
- `CANCELLED` / `TERMINATED` → 被显式停止

13.9.3 等待类型

`WaitType` 记录执行 为何 被挂起：

- `WAIT_SIGNAL` —等待具有匹配键的 `engine.signal()`
- `WAIT_TIMER` —基于超时的自动恢复
- `WAIT_EVENT` —等待具有关联匹配的 `engine.publishEvent()`
- `WAIT_TASK` —等待用户任务完成
- `WAIT_PHASE_TIMEOUT` —session 的 phase 级超时已触发
- `WAIT_ROUND_TIMEOUT` —session 的 round 级超时已触发
- `WAIT_RETRY_BACKOFF` —作为持久工作项调度的延迟重试

如果 session 对你还是新主题，可以阅读第 14 章——多轮 Session，那里会把 phase / round 超时和 session 快照如何建立在同一套 durable 机制之上讲清楚。状态机的持久化则走了另一条路径：`StateMachineCheckpoint` 通过通用 execution-checkpoint SPI 接入，而不是再定义一种新的核心检查点类型；你会在第 15 章——状态机里看到它。

13.9.4 DurableManager 桥接层

`DurableManager` 是 `GraphEngine` 用于协调跨所有存储写入的内部桥接层。你不需要直接调用它——引擎在以下时机调用它：

- Graph 启动 → `initExecution()`
- 节点完成 → `saveNodeCompletion()`
- 节点挂起 → 写入挂起上下文以及对应的等待/定时器记录
- 循环迭代完成 → `saveLoopSnapshot()`
- 恢复完成 → 清理挂起上下文 + 等待记录

13.10 常见陷阱

13.10.1 [X] 假设检查点数据保留其原始 Java 类型

当你挂起并稍后恢复时，检查点载荷会被序列化为 JSON 然后反序列化回来。如果你的 operator 输出是一个 `PaymentEvent` 记录，反序列化后它将变成 `Map<String, Object>`，除非你使用了类型化的 `CheckpointCodec`。

```
// FRAGILE — will ClassCastException after resume
PaymentEvent event = results.get("awaitPayment", PaymentEvent.class);

// SAFE — handle both fresh-run and checkpoint-restored shapes
Object raw = results.getRaw("awaitPayment");
String txnId;
```

```
if (raw instanceof PaymentEvent pe) {
    txnId = pe.transactionId();
} else if (raw instanceof Map<?,?> m) {
    txnId = (String) m.get("transactionId");
}
```

这种模式在 PaymentWaitExample 中随处可见。

修复方法：要么使用 getRaw() 配合模式匹配，要么将 JacksonCheckpointCodec 的类型化反序列化接入 GraphEngine.builder().checkpointCodec(...)。

13.11 常见故障

13.11.1 schema 变更后，检查点反序列化失败

你发布了 operator 的 v2 版本，把输出类型从 record OrderOutput(String orderId, int total) 改成了 record OrderOutput(String orderId, BigDecimal total, String currency)。

某个还在运行中的执行使用的是 v1 schema 挂起并写入了检查点。恢复时加载这个检查点，就会遇到这样的失败：

```
Recovery error: cannot deserialize the checkpoint for node 'createOrder'
  because required field 'currency' is missing from the stored payload.
```

修复： schema 变更尽量保持向后兼容（新增可选字段，不要删除必填字段）。如果确实是 breaking change，就先清空在途执行，或者实现一个能处理版本迁移的 CheckpointCodec。

13.12 引导式重写

打开 RuntimeGraphRegistryRecoveryTest。测试 publishesDslGraphAndRecoversColdSignalWithoutCallerGraph 演示了 冷恢复——在调用方不传入 Graph 对象的情况下恢复：

```
// Engine 1: execute with a DSL graph — the engine publishes the
// GraphDefinition to graphRegistryStore automatically
GraphEngine engine1 = GraphEngine.builder()
        .registry(registry)
        .executionStore(executionStore)
        .executionCheckpointStore(checkpointStore)
        .waitStore(waitStore)
        .graphRegistryStore(graphRegistryStore)        // ← enables cold recovery
        .addGraphDefinitionCodec(codec)                // ← how to decode DSL
        .timerService(timerService)
        .build();

engine1.execute(graph, new GraphContext(Map.of("sessionId", "SESSION-001")));
```

```
// Graph suspends at "wait" node

// Engine 2: signal WITHOUT passing the graph object
GraphEngine engine2 = GraphEngine.builder()
        .registry(recoveryRegistry)
        .executionStore(executionStore)
        .executionCheckpointStore(checkpointStore)
        .waitStore(waitStore)
        .graphRegistryStore(graphRegistryStore)
        .addGraphDefinitionCodec(codec)
        .timerService(timerService)
        .build();

// The engine loads the graph from graphRegistryStore using the
// execution's graphVersion + graphHash binding
engine2.signal(executionId, "wait", Map.of("message", "resume"));
```

思考问题：

1. 如果 **graph** 定义在挂起和恢复之间发生了变化会怎样？引擎会比较 `ExecutionIdentity` 中存储的 `graphHash` 与已发布 `GraphDefinition` 的 hash。如果不匹配，恢复会以 `"hash mismatch"` 错误失败。这在 `rejectsColdRecoveryWhenPublishedGraphHashChanges` 测试用例中有测试。
2. 自动崩溃恢复是怎样的？当启用了 `RecoveryConfig` 并且某次执行的租约过期（worker 崩溃了）时，另一个引擎实例可以自动认领过期的执行并重新运行它。`recoveryAttempts` 计数器每次递增。如果预算耗尽，状态转变为 `FAILED_RECOVERY`。
3. **Graph** 定义存储在哪里？在 `GraphRegistryStore` 中，它持久化 `graphName + graphVersion + graphHash + source`。在生产环境中，这由 `bd_graph_definition` 数据库表支撑。

13.13 Graph 定义来源（GraphDefinitionSource）

当引擎把 graph 发布到 `GraphRegistryStore` 时，它需要知道这个定义是从哪里来的——是磁盘上的 DSL 文件、Java builder 调用，还是从 registry 导入的。`GraphDefinitionSource` 接口（`com.leanowtech.bloge.core.runtime.registry.GraphDefinitionSource`）正是用来捕获这个来源信息的：

```
public interface GraphDefinitionSource {
    String sourceId();          // 该来源的唯一标识
    String sourceType();        // 例如 "dsl-file"、"java-builder"、"registry"
    Optional<String> content(); // 如果可用，保存原始源内容
}
```

在构建 graph 时附加来源信息，用于 registry 发布：

```
var source = new FileBasedGraphDefinitionSource(
        "file:///graphs/order-process.bloge", dslContent);
```

```
Graph graph = Graph.builder("orderProcess")
    .definitionSource(source)
    // ... nodes and edges ...
    .build();

engine.publishGraphDefinition(graph);
```

三个字段服务于不同的恢复需求：

字段	用途
`sourceId()`	唯一标识来源——文件路径、classpath 资源或 registry URI
`sourceType()`	告诉 registry 定义是如何产生的（`"dsl-file"`、`"java-builder"`、`"registry"`）
`content()`	可选地存储原始 DSL 文本，这样即使没有原始文件也能重建定义

如果没有 `GraphDefinitionSource`，registry 只会存储编译后的 graph 结构，丢失回到原始构件的线索。对于需要审计追踪或多集群复制的生产系统，务必附加来源信息。

13.14 检查点编解码器 SPI（CheckpointCodec）

默认情况下，引擎将检查点载荷序列化为 JSON。对于很多工作流这已经够用了，但生产系统有时需要：

- 自定义压缩 ——降低高吞吐工作流的存储成本
- 加密 ——保护包含敏感数据（PII、财务记录）的检查点
- 兼容性层 ——处理跨部署的 schema 迁移

`CheckpointCodec` SPI（`com.leanowtech.bloge.core.checkpoint.CheckpointCodec`）让你可以替换默认序列化方式：

```
public interface CheckpointCodec {
    byte[] encode(ExecutionCheckpoint checkpoint);
    ExecutionCheckpoint decode(byte[] data);
}
```

在引擎 builder 中接入：

```
GraphEngine engine = GraphEngine.builder()
    .registry(registry)
    .checkpointCodec(new JsonCheckpointCodec())  // 自定义序列化
    .build();
```

类型化的 codec 还能消除上面” 常见陷阱” 部分描述的 `Map<String, Object>` 问题——如果 codec 知道目标类型，恢复后的检查点就能直接还原为正确的 Java record，而不是裸 map。

13.15 Operator 指纹不匹配策略

冷恢复仍会拒绝 graph definition hash 不匹配。另一个独立问题是：完成节点保存了 Operator 指纹，但当前注册的 Operator 已换成另一指纹。`VersionMismatchPolicy` 只控制这类 checkpoint 决策：

策略	行为
`WARN`	恢复 checkpoint 并记录 Operator 不匹配；这是默认值。
`RERUN`	跳过该 checkpoint，让节点重新进入调度。
`FAIL`	用 `OperatorVersionMismatchException` 中止恢复。

在引擎 builder 中明确配置：

```
GraphEngine engine = GraphEngine.builder()
    .registry(registry)
    .versionMismatchPolicy(VersionMismatchPolicy.FAIL)
    .build();
```

选择依据应是副作用语义，而不是环境标签。只有节点可安全重复、不会复制不可逆副作用时，`RERUN` 才成立。`WARN` 会让新代码接受旧输出，因此需要显式兼容性判断。`FAIL` 会停止自动恢复，把迁移或人工处置留在引擎之外。

> 生产提醒——***schema migration*** 是另一条版本轴
> 以 *`bloge-durable-mybatis`* *artifact* 自带、按方言区分的 *Flyway migration* 为事实源。*`/actuator/bloge/schema`* 会报告 *`UP_TO_DATE`*、*`PENDING`* 或 *`FAILED`*；依赖新 *schema* 的 *binary* 上线前，应先应用所需 *migration*。*RC1* 的 *V25* 只迁移旧 *session enum* 值；*recovery lease* 来自 *V17*，*audit journal* 表来自 *V9*，*lease fencing* 来自 *V26*。运维迁移清单集中在附录 *H*，不埋在叙事章节里。

13.16 思维检查

1. 使持久执行成为可能的四个运行时存储是什么？（*ExecutionStore*、*ExecutionCheckpointStore*、*WaitStore*、*WorkItemStore*。）
2. 当引擎恢复一个挂起的执行时，已完成的节点会怎样？（它们被跳过。引擎加载它们的 *NODE_OUTPUT* 检查点并填充 *NodeResults*，而不重新执行 *operator*。）
3. 为什么引擎在 graph hash 变化时拒绝冷恢复？（因为 *graph* 结构可能已经改变——新节点、删除的边、不同的分支——所以检查点的节点输出可能不再对应当前的 *graph*。恢复将产生未定义行为。）
4. 当自动恢复超过 `maxRecoveryAttempts` 时，执行转变为什么 `ExecutionStatus`？（*`FAILED_RECOVERY`* ——一个终态。）
5. 如果一个 `SuspendableOperator` 返回 `OperatorResult.suspend("key", partial, Duration.ofMinutes(5))`，会创建多少条等待记录？（两条：一条 *`WAIT_SIGNAL`* 用于手动信号恢复路径，一条 *`WAIT_TIMER`* 用于 *5* 分钟超时自动恢复路径。）
6. 设计题：你的持久工作流会处理 2–4 周的保险理赔。在这段时间里，系统会多次发版。你准备如何确保这些在途执行能够穿越部署继续恢复？（保持 *operator* 输出 *schema* 的向后兼容；使用 *`GraphRegistryStore`*

给 *graph* 做版本化，让恢复流程能找到正确的 *graph* 定义；把 *compensation operator* 设计成幂等的，这样恢复后即使某个节点重跑也安全；并且在正式发布前，用长时间运行的执行在 *staging* 中做恢复演练。）

13.17 实验

目标：构建一个两阶段持久支付流程，并模拟阶段之间的崩溃。

1. 创建一个 **graph**，包含三个节点：
   - `createOrder` ——一个返回订单 ID 的普通 operator。
   - `awaitPayment` ——一个 `SuspendableOperator`，使用 `OperatorResult.suspend("payment:" + orderId, null, Duration.ofSeconds(30))` 挂起。
   - `fulfillOrder` ——依赖 `awaitPayment`，返回一个物流 ID。
2. 接入运行时存储（使用内存实现）：

```
var executionStore    = new InMemoryExecutionStore();
var checkpointStore   = new InMemoryExecutionCheckpointStore();
var waitStore         = new InMemoryWaitStore();
```

3. 执行 graph 在 `engine1` 上。验证：
   - 执行状态为 `SUSPENDED`。
   - `createOrder` 存在 `NODE_OUTPUT` 检查点。
   - `awaitPayment` 存在 `SUSPEND_CONTEXT` 检查点。
   - 等待记录包含 `WAIT_SIGNAL` 和 `WAIT_TIMER`。
4. 创建 **`engine2`**，使用相同的存储但全新的 `DefaultOperatorRegistry`。对挂起的节点发送信号：

```
engine2.signal(graph, executionId, "awaitPayment", paymentData);
```

5. 断言：
   - 执行完成。
   - `fulfillOrder` 有一个 `NODE_OUTPUT` 检查点。
   - 所有等待记录已被清理。

进阶：启用 `GraphRegistryStore`，尝试在不传入 `graph` 对象的情况下调用 `engine2.signal()` ——仅使用执行 ID。

13.18 实验验收卡

- 预期与观察：新进程凭 identity、definition 和 checkpoint 接管。
- 失败与恢复：改变 definition version；恢复匹配版本或显式迁移。
- 证明边界：证明引擎冷恢复，不证明外部 effect 状态已知。
- 练习合同：中断 checkpoint；只改版本；交付接受和拒绝记录；完成节点不重跑且 UNKNOWN 单列即停止。

13.19 回顾

- 持久执行 意味着持久化节点输出、挂起上下文、等待记录和执行标识，使 graph 能在崩溃后存活，并由任何引擎实例恢复。
- 四个运行时存储 ——`ExecutionStore`、`ExecutionCheckpointStore`、`WaitStore` 和 `WorkItemStore` ——是基础。开发时使用内存实现；生产环境使用 MyBatis 支撑的实现。
- 恢复时，引擎 加载检查点并跳过已完成的节点。它不会重放已完成的工作。
- 冷恢复 使用 `GraphRegistryStore` 通过 `graphVersion + graphHash` 绑定重新加载 graph 定义——不需要调用方提供 `Graph` 对象。
- 引擎在 **graph hash** 变化 时拒绝恢复，防止检查点/graph 不匹配。
- 自动崩溃恢复 使用租约过期和 `RecoveryConfig`，让健康的 worker 认领过期的执行，并带有 `recoveryAttempts` 预算。
- 检查点载荷是 **JSON** 序列化的。使用 `getRaw()` 配合模式匹配，或自定义 `CheckpointCodec` 来处理全新运行与恢复执行之间的类型差异。

13.20 下一步

在 第 14 章——多轮 Session 中，你将从“持久的一次性 graph”走向“更长生命周期的交互”。你在这里学到的检查点与恢复机制，会直接成为 phase / round session 的基础，并继续延伸到第 15 章——状态机。

13.21 参考链接

- `bloge-durable` README —模块概览、存储表、快速开始、schema 迁移
- `bloge-durable-codec` README —`JacksonCheckpointCodec` 和自定义序列化
- `CheckpointCodec` SPI —codec 接口
- `ExecutionCheckpointStore` —检查点存储 SPI
- `ExecutionCheckpoint` —检查点记录，包含标识、类型、载荷和 operator 指纹
- `CheckpointType` —`NODE_OUTPUT`、`LOOP_SNAPSHOT`、`FOREACH_PROGRESS`、`SUSPEND_CONTEXT` 等
- `ExecutionIdentity` —租户、命名空间、业务键、graph 绑定、路由
- `ExecutionInstance` —生命周期记录，包含租约、版本、恢复尝试次数
- `ExecutionStatus` —`RUNNING`、`SUSPENDED`、`COMPLETED`、`FAILED`、`FAILED_RECOVERY`
- `WaitType` —`WAIT_SIGNAL`、`WAIT_TIMER`、`WAIT_EVENT`、`WAIT_TASK`、`WAIT_PHASE_TIMEOUT`、`WAIT_ROUND_TIMEOUT`、`WAIT_RETRY_BACKOFF`

- `OperatorResult` —`Completed` vs `Suspended` 密封结果类型
- `SuspendableOperator` —可挂起节点的 operator 契约
- `DurableManager` —引擎与持久化存储之间的内部桥接层
- `DurableStoreFactory` —使用 MyBatis + Flyway 的生产存储接线
- 核心架构 —引擎内部机制和包结构

第三阶段回顾——组合与时间（第 10–13 章）

章节

第三阶段回顾——组合与时间（第 10–13 章）

你现在可以用可复用的部件组合大型工作流、遍历集合、挂起等待外部事件，并在进程重启后恢复执行。

0.1 你学到了什么

第 **10** 章——用子图复用 引入了 `subgraph("name")` 语法。你学会了用 `DslCompiler.registerSubGraph()` 注册预构建的 `Graph`、将其嵌入父 graph，并控制作用域隔离（isolated vs. parent）。你还看到了 `dynamicSubGraph`（运行时动态生成 DSL）在什么场景下更合适。

第 **11** 章——批处理与迭代 展示了两个循环原语：`foreach`（遍历集合，每个元素执行一次子 graph 并收集结果）和 `loop`（带 `max_iterations`、`delay` 和 `until` 的可控重复）。你了解了批处理结果如何收集，以及 `loopIteration` 如何提供从零开始的计数器。

第 **12** 章——等待世界的回应 增加了挂起能力。通过 `await`，graph 暂停执行且不持续占用 worker thread，只有外部信号到达或超时后才恢复。你练习了按 key 关联事件，以及根据信号是否到达来分支执行。

第 **13** 章——持久执行 增加了重启恢复所需的 store 与 identity。满足条件的 worker 可以接受 checkpoint、跳过已完成节点并继续剩余前沿；尚未提交 checkpoint 的外部副作用仍需要幂等或对账。

0.2 你现在应该能做到

- 将可复用的流程提取为命名子图，并嵌入多个父 graph。
- 用 `foreach` 处理集合、用 `loop` 实现轮询循环，并为每种场景选择正确的原语。
- 用 `await` 挂起 graph、关联外部事件，并优雅地处理超时。
- 将持久化 store 接入 `GraphEngine`，并演示 挂起 → 崩溃 → 冷恢复 → 继续执行 的完整流程。
- 解释 checkpoint、执行标识和 graph 注册表之间的关系。

0.3 这个阶段的常见错误

错误	为什么会犯	如何纠正
在应该用 isolated 作用域时用了 parent	”子图只需要父 graph 的一个值”	默认使用 isolated 作用域，通过子图输入绑定显式传递所需值。parent 作用域会泄漏整个 context。
无限循环（没有 `until` 也没有 `max_iterations`）	忘记 `loop` 默认是无界的	即使 `until` 通常会先退出，也始终设置 `max_iterations` 作为安全网。
把 `await` 当成 `Thread.sleep()`	混淆了挂起和阻塞的概念	`await` 等待时不持续阻塞 worker thread；持久化 wait、timer 与 correlation 状态仍会消耗系统资源。
开发时跳过持久化 store	“以后再加持久化”	用内存 store 实现练习 API 与恢复路径，同时记住它们会随进程退出而丢失状态。
不测试恢复路径	只测试没有任何崩溃的快乐路径	至少写一个测试：执行到一半，创建新引擎，从 checkpoint 恢复。

0.4 新增核心词汇

术语	含义
子图（**Subgraph**）	一个按名称注册的预构建 `Graph`，作为单个节点嵌入父 graph。
`foreach`	遍历集合，为每个元素执行一次循环体并收集结果。
`loop`	最多重复执行 `max_iterations` 次，可选 `delay` 和 `until` 退出条件。
`loopIteration`	每个循环体内可用的从零开始的内置计数器。
`await`	挂起 graph 并释放线程，直到外部信号到达或超时。
Checkpoint	每个节点完成后对 node 输出和执行状态的持久快照。
冷恢复（**Cold recovery**）	在另一个引擎实例上，通过从持久化存储加载 checkpoint 来恢复 graph 执行。

0.5 试一试：选择正确的时间模型

一个订单流程要处理 200 个商品、最多轮询库存五次，还可能等待客户确认两天。请分别选择 **foreach**、**loop** 或 **await**，并标出哪些状态必须跨进程重启保存。再解释为什么用 sleep 循环替代 `await` 会占住错误的资源，并隐藏外部 signal 边界。

0.6 下一步

第四阶段回顾 将重心从“持久的一次性执行”转向“更长生命周期的编排模式”：session、状态机，以及嵌套生命周期所有权。从第 14 章——多轮 Session开始。

学习阶段

第四阶段——生命周期模型

Session、状态机、嵌套所有权与受限 Agent

目标：让每一个长生命周期决策只有一个明确 *owner*。

章节
14

第四阶段——生命周期模型

第 14 章——多轮 Session

Session、状态机、嵌套所有权与受限 Agent

目标：让每一个长生命周期决策只有一个明确 *owner*。

承诺：读完本章后，你将知道什么时候应该从一次性 *graph* 升级到 `session`（多轮交互会话），如何用 `ONCE` / `ROUND` 两类 *phase* 建模多轮交互，以及如何让整段交互在重启、超时和恢复之后仍然保持清晰。

14.1 学习目标

1. 区分“只挂起一次的 graph”和“跨越多次信号、多段 phase 边界的 `session`”。
2. 把 `session` 建模成一组有序的 phase，并理解 `ONCE` 与 `ROUND` 的职责边界。
3. 使用 `ctx.round.input`、`yield_on`、`until` 和 `then` 来表达真实的多轮交互。
4. 配置空闲超时、历史保留、所有权和恢复逻辑，而不丢失运行时心智模型。
5. 知道什么时候 `session` 是正确抽象，什么时候 第 12 章 或 第 13 章 里的普通 graph 仍然更简单。

14.2 前置条件

- 第 12 章——等待世界的回应 ——你需要先有挂起 / 信号的心智模型
- 第 13 章——持久执行 ——session 建立在同一套检查点和恢复基础之上
- 第 7 章——设计良好的 Operator ——多轮交互只有在 operator 边界清晰时才会保持可读

14.3 源示例

文件	展示内容
`ch13/customer-service-session.bloge`	典型 session DSL：问候 → 多轮分诊 → 处理 → 收尾
`ch13/interactive-review-session.bloge`	更小的 collect → review 轮次 → finalize 模式，用于本章引导式重写

文件	展示内容
`CustomerServiceSessionExample.java`	用 `PhaseBuilder.once(...)` / `PhaseBuilder.round(...)` 构建同一个 session 的 Java Fluent API 版本
`bloge-session-ext/README.md`	内存 session 运行时、手动恢复边界和嵌套状态机说明
`docs/session-phase-round-specification.md`	session / phase / round 的精确 DSL 与运行时契约

14.4 为什么这很重要

在 第 12 章 中，你已经学会了让一张 graph 挂起，然后在稍后恢复。这已经能解决很多真实工作：等待审批、等待支付、等待外部事件。

但有些工作流不是“一个 graph + 一次等待”，而是一段真正的对话式交互：

- 先打招呼
- 再问更多细节
- 等用户回复
- 不够清楚时继续追问
- 最后转人工或结束对话

同样的行为也可以用 `await`、分支和上下文拼装在一张 graph 中表达。代价是对话身份、轮次边界和恢复状态仍隐含在 graph 结构里，不容易检查和解释。

`session` 的价值就在这里：它把“多轮交互本身”提升为一等运行时模型。你不再把所有轮次都压进一张巨大 DAG，而是明确写出：

- 哪些步骤只执行一次
- 哪些步骤会按轮次重复
- 哪些信号属于同一段业务交互
- 这段交互该如何超时、恢复和继续推进

14.5 先读对话，再读 **Session** 模型

客户说包裹显示已送达，但自己没有收到。助手读取事实、问一个收窄问题、收到 回答，再把工单交给人工路径。Session 要保留的是这段四轮交换，而不是一个 抽象的“支持聊天”目标。

14.5.1 每个对话事实只翻译一次

对话事实	Session owner	为什么保留
工单与客户 identity	Session context	每一轮都稳定
当前客户消息	`ctx.round.input.userMessage`	只属于当前 round
助手回答与 `done`	`respond.output`	决定 yield 与退出
`action=handoff`	phase transition 条件	把 owner 从 triage 交给 solve
最终处理结果	wrap-up output	给调用方稳定的 terminal result

第 2 轮在 `respond` 后 yield，因此调用方能显示问题，而 Session 仍停留在 `triage`。第 3 轮回答成为下一次 round input。第 4 轮返回 `done=true` 与 `action=handoff`，phase 才迁移到 `solve`；它不会因为模型文本里写了“转人工”就暗中修改工单。

现在术语有了存在理由：**phase** 负责一个对话阶段，**round** 负责一次重复 交换，`yield_on` 标出外部可见暂停，`until` 说明重复何时结束。

14.6 心智模型

可以把 `session` 看作是多个 **graph** 执行的容器。

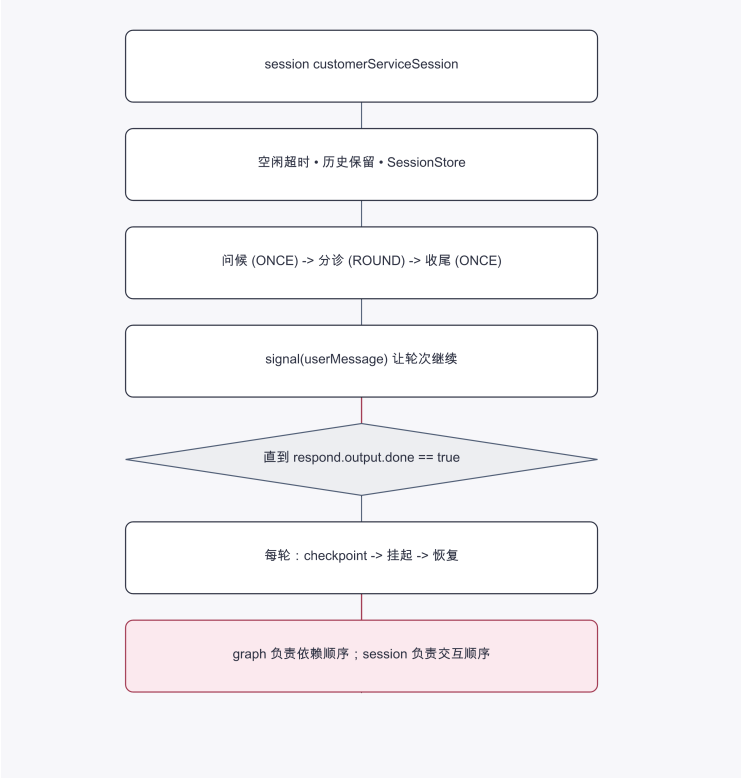
组成部分	责任
`SessionGraph`	整段交互的不可变定义
`phase`	交互生命周期中的一个命名步骤
`ONCE` phase	内部 graph 执行一次，然后发生跳转
`ROUND` phase	每次收到新的信号载荷就执行一轮，轮次之间会挂起

组成部分	责任
`SessionExecutor`	启动、发信号、恢复调用方提供的状态，并终止内存 session
`DurableSessionManager`	在需要持久化时协调 checkpoint store、定义查询、lease 与重启恢复

两类 phase 基本覆盖了大多数模式：

Phase 类型	适合什么	运行时行为
`ONCE`	问候、加载上下文、最终收尾	执行一次 graph，写入输出，解析 `then`
`ROUND`	澄清直到完成、评审迭代、菜单循环	等待信号，执行一轮 graph，计算 `until`，然后继续挂起或跳到下一个 phase

关键转变是：graph 负责一次执行内部的依赖顺序，而 `session` 负责多次执行之间的交互顺序。



14.7 第一个可运行的示例

companion 项目里的 `customerServiceSession` 是最适合照着抄的最小真实形状：

```
session customerServiceSession {
  idle_timeout = 5m
  timeout_action = "cs_session_timeout_policy"
  max_rounds = 20
```

```
  max_history = 50

  phase greeting {
    node greet : CsSessionGreeter {
      input {
        sessionId = ctx.sessionId
      }
    }
    then -> triage
  }

  phase triage {
    max_rounds = 5
    yield_on = [respond]
    round {
      node respond : CsSessionResponder {
        input {
          userMessage = ctx.round.input.userMessage
        }
      }
    }
    until respond.output.done == true
    then {
      respond.output.action == "handoff" -> solve
      otherwise -> wrapUp
    }
  }
}
```

用自然语言读它：

1. 先进入一次性的 greeting phase
2. 再进入可重复的 triage phase
3. 每次信号的 payload 都会成为新的 `ctx.round.input`
4. 只要 `done != true`，就继续停留在当前 phase
5. 一旦完成，就决定是转 `solve` 还是直接 `wrapUp`

这正是 `session` 的价值：即使交互跨越了多轮用户输入，结构依然清晰可读。

14.8 拆解分析

14.8.1 `ONCE` 与 `ROUND` 解决的是不同层面的问题

当一个步骤只应该给出一次答案时，使用 `ONCE` phase：

- 问候用户
- 加载当前账户或上下文
- 生成最终摘要

当一个步骤天然可能重复时，使用 `ROUND` phase：

- 不断追问，直到问题足够清楚
- 不断评审，直到不再需要修改
- 按菜单多轮收集输入，直到用户到达明确结果

DSL 之所以把两者区分出来，就是为了避免把“重复交互”伪装成一堆难以读懂的分支。

14.8.2 每一轮能看到什么

在 `ROUND` phase 中，运行时会在普通 `GraphContext` 之外再注入几类绑定：

- `ctx.round.input` ——当前轮次恢复时收到的 payload
- `ctx.<phaseId>.output` ——之前已经完成的 phase 输出
- `ctx.session.namespace` / `ctx.session.ownerId` ——如果提供了身份信息，就能拿到解析后的 session 身份

所以一轮真正会同时消费三类数据：

- 初始请求上下文，例如 `ctx.sessionId`
- 之前 phase 的累积输出，例如 `ctx.collect.output.*`
- 当前用户的最新回复，即 `ctx.round.input.*`

14.8.3 Session 级控制项与默认值

属性	作用域	默认值	你为什么要关心
`idle_timeout`	session	`30m`	两次信号之间允许空闲多久
`timeout_action`	session	无	通过 `TimeoutPolicyProvider` 解析的策略引用
`max_rounds`	session	`100`	所有 round phase 共用的总轮次预算
`max_rounds`	`ROUND` phase	`1`	当前 phase 自己的轮次上限——真实对话几乎总要显式设置
`max_history`	session	`0`	历史保留条数；`0` 表示无限制，不是关闭
`yield_on`	`ROUND` phase	终端节点	每轮结束后返回哪些节点输出
`on_round_failure`	`ROUND` phase	`terminate_session`	一轮执行失败时该怎么处理
`max_visits`	phase	不限	运行时通过 `then` 规则重新进入该 phase 的硬上限
`then`	phase	无	无条件或基于规则的后续跳转

这里有两个特别容易误读的默认值：

- `max_history = 0` 表示全部保留

- 没有显式设置 max_rounds 的 ROUND phase 只会跑一轮

用 ***max_visits*** 防止 ***phase*** 之间的反复跳转

max_rounds 限制的是一个 *ROUND phase* 内部循环多少次。*max_visits* 限制的是运行时通过其它地方的 *then* 规则进入该 *phase* 的次数。没有它，两个互相 *route* 的 *phase* 会无限来回跳:

```
phase classify : ROUND { ... then "needs-more-info" -> gatherInfo }
phase gatherInfo : ROUND { max_visits = 5  then "complete" -> classify }
```

一旦超过 *max_visits*，运行时会发出 *PHASE_VISIT_LIMIT_EXCEEDED*，并抛出 *SessionPhaseVisitExceeded Exception*。执行器随后通过普通错误终止路径 把 *session* 标记为失败。可被重新进入的 *phase* 都应设置 *max_visits*。

14.8.4 启动、发信号与恢复内存状态

Session 的生命周期 API 故意保持得很小:

```
SessionExecutor executor = SessionExecutor.builder(engine)
    .accessGuard(new OwnerOnlySessionAccessGuard())   // 默认
    .build();

SessionHandle handle = executor.start(
        sessionGraph,
        new GraphContext(Map.of("sessionId", "SESSION-1001")),
        SessionIdentity.of("support", "user-123")
);

executor.signal(handle.sessionId(), Map.of("userMessage", " 我想退款"), "user-123");
```

SessionExecutor 只负责当前 JVM 中活跃的内存交互。builder 可以配置 access guard、listener、timeout policy 和 snapshot callback，但不接收持久化 store。

如果应用自行保存了 session snapshot，就要先重建普通的 SessionState 和 上下文，再显式交给执行器:

```
SessionHandle restored = executor.restore(
        sessionGraph,
        restoredState,
        restoredContext
);
```

这个方法只恢复调用方提供的状态，不会主动发现 checkpoint、抢占 lease，也不负责服务重启后的批量恢复。后面介绍的 DurableSessionManager 才负责这些工作。

14.8.5 所有权是模型的一部分

session 往往会存活得足够久，以至于“谁有资格继续推动它”会成为业务问题本身。

默认情况下，运行时使用 OwnerOnlySessionAccessGuard。这意味着:

- 谁启动了 session，后续往往也决定谁有权 signal(...)

- 读取、继续推进、终止等行为都可以被调用方身份约束
- 与“只会挂起一次的 graph”相比，session 明确把身份和生命周期绑在了一起

应用必须从自己的认证请求中解析 caller id，再把它传给 `start(...)` 和后续 操作。下面这个名字相近的 Spring 属性服务于另一类 owner：

```
spring:
  bloge:
    session:
      owner-id: ${HOSTNAME}   # 当前 JVM 的 durable recovery/lease owner
```

`spring.bloge.session.owner-id` 不是用户身份表达式解析器，而是执行 durable recovery claim 的 JVM 标识；省略时，durable manager 会生成随机 UUID。认证主体到 `SessionIdentity` 的映射仍由应用负责。

14.8.6 什么时候不该使用 **session**

不要因为它“更高级”就默认用 `session`。

普通 graph 仍然更适合这些情况：

- 整个流程只有一次明显的挂起 / 恢复边界
- 业务上并没有比 node 更重要的 phase 结构
- 核心问题仍然是一张 DAG 加上持久执行

如果真正困难的是命名状态之间的迁移，而不是多轮对话，那么通常 第 15 章 才是更合适的抽象。

14.9 常见陷阱

14.9.1 [X] 忘了 **ROUND phase** 默认只跑一轮

下面这段看上去像是会不断循环：

```
phase triage {
  round {
    node respond : CsSessionResponder { }
  }
  until respond.output.done == true
}
```

但如果你没有显式设置 phase 级 `max_rounds`，它只会执行第一轮：

```
phase triage {
  max_rounds = 5
  round {
    node respond : CsSessionResponder { }
  }
  until respond.output.done == true
}
```

Session 级 `max_rounds = 20` 是全局预算，并不会覆盖 `ROUND` phase 默认值 `1`。

14.10 常见故障

14.10.1 服务重启后，新的内存执行器找不到旧 session

服务重启后，如果直接创建一个新的 `SessionExecutor`，再向旧 session id 发送 signal，就会失败，因为新的执行器里没有那段活跃状态：

```
SessionExecutor executor = SessionExecutor.builder(engine).build();
executor.signal(sessionId, payload, "user-123"); // 当前执行器里没有这个 session
```

如果由应用自行管理快照，就重建 `SessionState`，再调用 `restore(sessionGraph, state, context)`；如果需要基于 store 的重启恢复，就使用 `DurableSessionManager`，同时配置 execution store、checkpoint store 和 `definitionLookup`。这条边界避免内存执行器假装自己拥有分布式恢复能力。

14.11 引导式重写

打开 `interactive-review-session.bloge`，再回看 第 12 章 里的单次等待模式。

这次重写的关键步骤是：

1. 把一次性初始化保留为 **`ONCE` phase**。`collect` 只负责收集评审上下文。
2. 把重复的人机交互抽成 **`ROUND` phase**。`review` 每收到一次回复就执行一轮。
3. 用 **`until`** 直接表达退出条件。当结果不再是 `needs_revision` 时，结束当前 phase。
4. 最后用一个 **`ONCE` phase** 收尾。`finalize` 只写一次最终总结。

```
session interactiveReview {
  phase collect { ... then -> review }

  phase review {
    max_rounds = 3
    yield_on = [reviewRound]
    round {
      node reviewRound : DecideReviewOperator { ... }
    }
    until reviewRound.output.decision != "needs_revision"
    then -> finalize
  }

  phase finalize { ... }
}
```

这次重写的目标不是“多用几个运行时特性”，而是把边界从节点级等待提升到交互级结构。

14.12 思维检查

1. 什么时候应该继续使用普通 graph，而不是引入 `session`？
   （当整个流程只有一次明确的挂起 / 恢复边界，并不需要多 *phase* 对话结构时。）
2. session 级 `max_rounds` 与 phase 级 `max_rounds` 有什么区别？
   （前者是所有 *round phase* 共用的总预算；后者只限制当前 *`ROUND` phase*，而且默认值是 *`1`*。）
3. `ctx.round.input` 里是什么？
   （当前轮次恢复时收到的 *payload*。）
4. `max_history = 0` 代表什么？
   （历史无限制保留，而不是关闭历史。）
5. 调用 `SessionExecutor.restore(...)` 时，调用方必须提供什么？
   （*`SessionGraph`*、重建后的 *`SessionState`* 和对应的 *`GraphContext`*；持久化与 *lease* 恢复不属于纯内存执行器。）
6. `session` 与“一张只有一个 `await` 节点的 graph”本质差异是什么？
   （*session* 明确拥有 *phase* 顺序、轮次边界、*session* 级超时 / 历史 / 访问控制，以及跨多次信号的恢复语义。）

14.13 实验

目标：把一个一次性客服 graph 改造成真正的多轮 support session。

1. 从 greet、clarify、resolve 三步交互开始。
2. 把 `greet` 和 `resolve` 放进 `ONCE` phase。
3. 把澄清步骤放进一个 `ROUND` phase。
4. 配置：
   - session `idle_timeout = 10m`
   - session `max_rounds = 12`
   - phase `max_rounds = 4`
   - `yield_on`，让调用方在每轮结束后都能拿到最新回复
5. 增加一个分支：当轮次输出中 `action == "handoff"` 时转人工。

进阶：增加自定义 `timeout_action` 策略，让空闲 session 在终止前先发送提醒。

14.14 持久 Session（Durable Sessions）

上面的内容都使用纯内存 `SessionExecutor`。在生产环境中，跨越数小时甚至数天的 session 需要检查点持久化，才能在进程重启后存活。`bloge-session-durable` 模块提供的 `DurableSessionManager` 正是为此而生的。

14.14.1 Maven 依赖

```
<dependency>
    <groupId>com.leanowtech.bloge</groupId>
    <artifactId>bloge-session-durable</artifactId>
```

```
</dependency>
```

14.14.2 构建与使用

```
Map<String, SessionGraph> sessionRegistry = Map.of(
    sessionGraph.name(), sessionGraph
);

DurableSessionManager durableSession = DurableSessionManager.builder()
    .executionStore(executionStore)
    .checkpointStore(checkpointStore)
    .graphEngine(engine)
    .definitionLookup(sessionRegistry::get)
    .recoveryConfig(SessionRecoveryConfig.defaultConfig())
    .build();

durableSession.startup();
// 若本地没有 active session，signal() 会先 claim durable execution、
// 恢复 checkpoint，再投递下一轮输入。
durableSession.signal(sessionId, newTurn);
```

`DurableSessionManager` 会创建带 durable snapshot callback 的 `SessionExecutor`，session snapshot 经 checkpoint store 写入。当 `signal(...)` 在本地找不到该 session 时，manager 会 claim 对应 execution、加载最近 checkpoint，再投递 signal。

14.14.3 内容哈希与版本检测

每个 session graph 定义都有一个 `contentHash` ——对 session graph 结构计算的 SHA-256 摘要。这个哈希会存储在每一个检查点内部。当 `DurableSessionManager` 加载检查点时，它会把存储的哈希与当前定义的哈希进行比较。

如果两个哈希不匹配，说明 session 定义在检查点写入之后发生了变化。

14.14.4 SessionVersionMismatchException

当检测到哈希不匹配时，`DurableSessionManager` 会抛出 `SessionVersionMismatchException`：

```
try {
    durableSession.signal(sessionId, userTurn);
} catch (SessionVersionMismatchException e) {
    // e.sessionId()   —哪个 session 失败了
    // e.checkpointHash() —检查点中的哈希
    // e.currentHash() —当前定义的哈希
}
```

恢复策略：

策略	适用场景
快速失败（默认）	开发阶段——强制你立刻注意到定义变更
丢弃旧 **session**，重新开始	旧 session 状态可丢弃时——捕获异常后启动全新 session
迁移	需要连续性时——实现应用层迁移逻辑，把旧的 phase 状态映射到新定义

选择哪种策略取决于你的业务场景。短生命周期的客服 session 通常可以安全丢弃；长周期的入职引导流程可能需要迁移逻辑。

14.15 实验验收卡

- 预期与观察：四轮对话映射到 context、round input、yield 和 transition。
- 失败与恢复：缺失 session identity 时恢复失败；恢复同一 identity 和 checkpoint。
- 证明边界：证明 Session 生命周期，不证明对话内容正确。
- 练习合同：四轮 transcript；只改一轮输入；交付 phase 表；每轮 owner 和下一 transition 可解释即停止。

14.16 回顾

- 普通 graph 建模的是一次执行；`session` 建模的是多轮交互。
- `ONCE` phase 负责一次性工作；`ROUND` phase 负责按信号驱动重复推进。
- `ctx.round.input` 是外部信号与当前轮次 graph 之间的桥梁。
- session 级限制和 phase 级限制不是一回事，真实流程通常两者都要配。
- 纯内存执行器只恢复调用方提供的状态；基于 store 的重启恢复属于 `DurableSessionManager`。
- 当业务需要的是“对话结构”而不是单张 DAG 时，`session` 才是真正合适的抽象。

14.17 下一步

在 第 15 章——状态机 中，你将从“多轮对话”切换到“显式命名状态”。到那时，重点不再是反复跑几轮，而是描述诸如 `draft -> pendingReview -> processing -> completed` 这样的生命周期迁移。

14.18 参考链接

- `bloge-session-ext/README.md` ——内存 session 运行时与恢复边界
- `docs/session-phase-round-specification.md` ——session / phase / round 的完整契约
- `CustomerServiceSessionExample.java` ——Java Fluent API 示例
- `customer-service-session.bloge` ——本章 DSL 示例
- `interactive-review-session.bloge` ——引导式重写示例

章节

15

第四阶段——生命周期模型

第 15 章——状态机

Session、状态机、嵌套所有权与受限 Agent

目标：让每一个长生命周期决策只有一个明确 *owner*。

承诺：读完本章后，你将知道什么时候一个工作流已经超出了单张 *DAG* 的表达能力，需要用 `state_machine` 来描述；你也会学会如何建模事件驱动迁移、超时和守卫条件，并让生命周期状态显式可见，而不是藏在一堆分支和标志位里。

15.1 学习目标

1. 识别什么时候问题更适合用命名状态和迁移来表达，而不是继续堆分支 graph。
2. 定义 `initial`、普通和 `terminal` 状态，并把状态内部的本地工作放进嵌入式 graph。
3. 分清事件迁移、守卫迁移、自动迁移和超时迁移各自的职责。
4. 在上线前就设置好 `max_transitions` 和 `max_state_visits` 这类安全上限，避免坏循环变成生产事故。
5. 知道什么时候该用状态机，什么时候 graph 或 `session` 仍然更简单。

15.2 前置条件

- 第 5 章——做决策的分支 ——状态迁移解决的问题和 graph 分支不同
- 第 12 章——等待世界的回应 ——事件驱动迁移仍然依赖同样的外部信号模型
- 第 13 章——持久执行 ——状态机的检查点与超时也站在持久执行基础上
- 第 14 章——多轮 Session ——有助于对比“对话轮次”和“生命周期状态”

15.3 源示例

文件	展示内容
ch14/order-lifecycle-state-machine.bloge	draft → review → processing → completed 的订单生命周期，并带超时回退
ch14/ticket-state-machine.bloge	带 assign / escalate / resolve / close 路径的客服工单生命周期
OrderLifecycleStateMachineExample.java	使用 StateMachineBuilder 的 Java Fluent API 版本
bloge-state-ext/README.md	状态机运行时模型、超时语义和嵌套 session 说明
ReviewStateMachineWithSessionExample.java	一个“状态机内嵌 session”的预告，下一章会展开

15.4 为什么这很重要

Graph 最擅长回答的问题是：在当前依赖关系下，什么现在可以执行？

状态机擅长回答的是另一个问题：这个业务对象现在处于什么状态？哪个事件会把它推进到下一个状态？

当流程存在回退、反复进入某个状态，或者必须让非实现人员也能一眼读懂生命周期时，这个区别就会变得非常关键。

例如：

- 一个订单可以从 `draft` 进入 `pendingReview`，又因为拒绝回到 `draft`
- 一张工单可以从 `open` 进入 `triaging`，再决定去 `assigned` 或 `escalated`
- 一个暂停中的 review 状态可以因为超时自动回退到前一个状态

这些规则也可以用分支、标志位和循环编码在一张 graph 中，但生命周期会隐含在 node 名称和条件表达式里。`state_machine` 则直接表示命名状态和状态迁移。

15.5 心智模型

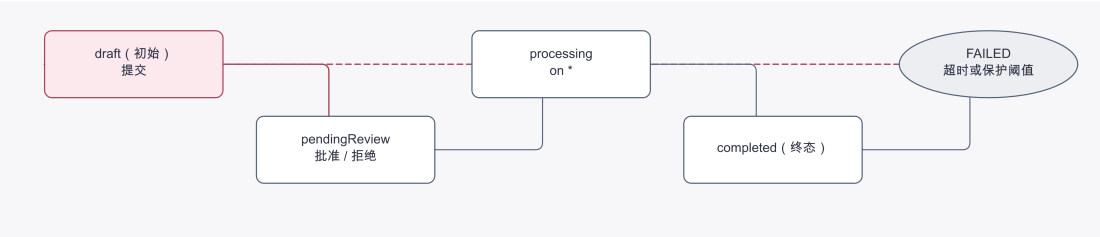
状态机本质上是一个围绕每个状态局部工作而建立的事件驱动外壳。

组成部分	责任
`StateMachineDef`	整个状态机的不可变定义
`state`	一个命名的生命周期步骤，例如 `draft`、`review`、`processing`
嵌入式 `graph`	在该状态内部要执行的本地工作
迁移	在事件、超时或自动条件满足时前往另一个状态的规则
`StateMachineExecutor`	启动状态机并递送外部事件
`StateMachineCheckpoint`	可序列化快照，用于恢复与持久化

四种迁移样式最重要：

迁移样式	语法	适合什么
事件迁移	`on approve -> processing`	外部事件应该推动生命周期前进
守卫迁移	`on * when ... -> approved`	当前状态输出本身决定接下来去哪
自动迁移	`on * -> completed`	当前状态完成后应立即前往下一个状态
超时迁移	`timeout = 24h + on_timeout -> draft`	长时间无响应时需要自动回退、关闭或失败转移

状态机负责让生命周期状态可见；状态内部的 graph 负责完成该状态下的局部工作。



15.6 第一个可运行的示例

`orderLifecycle` 是最适合入门的第一个状态机：

这 44 行 definition 保留完整，因为读者需要把全部 state、event、timeout 和终止 迁移看成同一个生命周期；拆成孤立片段会重新制造本章要消除的 ownership 歧义。

```
state_machine orderLifecycle {
  max_transitions = 25
  max_state_visits = 5
  timeout = 72h

  state draft [initial] {
    graph {
      node initOrder : InitOrderOperator {
        input {
          orderId = ctx.orderId
          customerId = ctx.customerId
        }
      }
    }
    on submit -> pendingReview
  }

  state pendingReview {
    graph {
      node reviewOrder : ReviewOrderOperator {
        input {
```

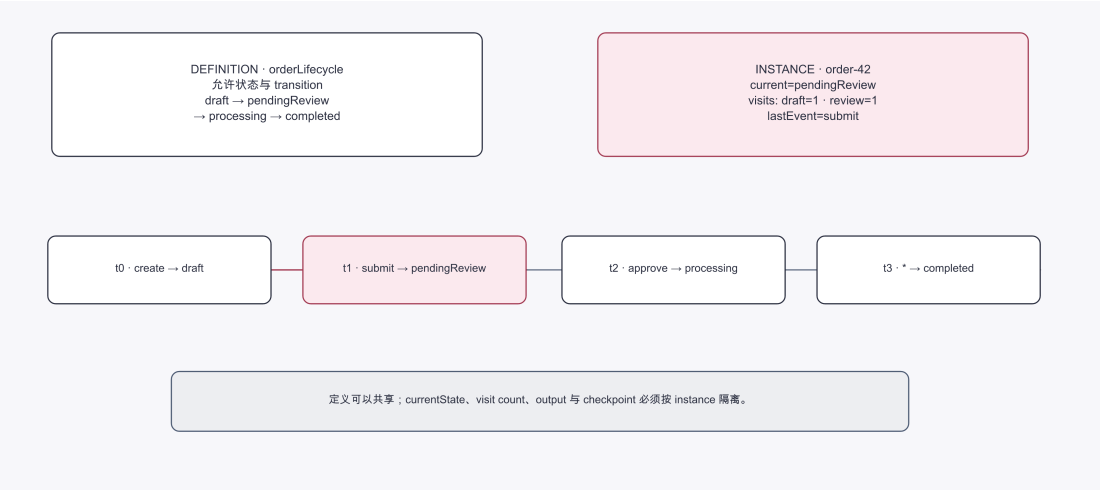
```
          orderId = ctx.draft.output.initOrder.orderId
        }
      }
    }
    on approve -> processing
    on reject -> draft
    timeout = 24h
    on_timeout -> draft
  }

  state processing {
    graph {
      node fulfillOrder : FulfillmentOperator {
        input {
          orderId = ctx.pendingReview.output.reviewOrder.orderId
        }
      }
    }
    on * -> completed
  }

  state completed [terminal] { }
}
```

用生命周期的方式读它，而不是用 DAG 的方式读它：

- 从 `draft` 开始
- 等待 `submit`
- 迁移到 `pendingReview`
- 再根据 `approve`、`reject` 或超时决定去向
- `processing` 状态内部工作结束后，自动迁移到 `completed`

15.7 共享 **definition** 不是正在运行的订单

`orderLifecycle` 定义每笔订单可以做什么。`order-42` 是一个当前停在 `pendingReview` 的 instance；另一实例可以同时停在 `processing`。混淆这两 个概念，是状态泄漏最直接的来源。



15.7.1 把一个 **instance** 读成被接受的事件序列

时间	事件	接受后的 **instance** 事实
`t0`	create	`currentState=draft`，draft visits = 1
`t1`	`submit`	`currentState=pendingReview`，review graph output 已保留
`t2`	`approve`	`currentState=processing`，transition count = 2
`t3`	wildcard completion	`currentState=completed`，terminal = true

definition 包含 state id、graph、guard、timeout 与允许的 transition。instance 包含 current state、visit/transition count、state-local output、已接受事件历史 和 checkpoint identity。definition 可以缓存共享；instance 必须按订单隔离。

15.7.2 修改 **definition** 后，要追问旧 **instance** 怎么办

在 definition 中增加 `manualReview`，不会让 `order-42` 自动移动过去。必须有 事件选中允许的 transition；durable restore 还必须判断旧 checkpoint 是否与 新 definition 兼容。因此，definition versioning 是迁移决策，不是给 live state 直接赋值。

15.8 拆解分析

15.8.1 状态负责局部工作，迁移负责生命周期规则

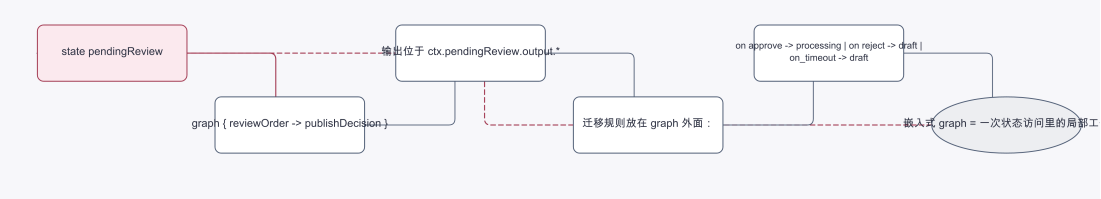
尽量保持下面的职责边界：

- `graph { ... }` 只描述当前状态内部要做什么
- `on ... -> target` 只描述什么时候切换到下一个状态

一旦这两个职责混在一起，定义就会迅速变得难懂。一个简单判断标准是：

- 如果它是在描述一次进入状态后，数据如何依赖与流动，就放进状态内部 graph

- 如果它是在描述什么时候改变生命周期状态，就放进迁移规则



15.8.2 事件迁移、自动迁移、守卫迁移和超时迁移

`state_machine` 提供了不止一种前进方式：

- 事件迁移：`on approve -> processing`
- 自动迁移：`on * -> completed`
- 守卫迁移：`on * when ctx.review.output... -> approved`
- 超时迁移：`timeout = 24h` 与 `on_timeout -> draft`

状态级超时底层会通过一个合成的超时事件来实现。若执行器使用了专门的 `TimerService`，等待中的实例可以异步过期；如果没有，超时仍然会在下一次 `execute(...)` 或 `signal(...)` 时被强制检查。

15.8.3 输出是按状态命名空间组织的

状态输出不是被全局摊平，而是挂在状态 ID 下。

这就是为什么订单示例里要这样写：

```
orderId = ctx.draft.output.initOrder.orderId
```

而不是：

```
orderId = ctx.initOrder.output.orderId
```

这种按状态命名空间组织输出的方式，会把“这个数据是哪个生命周期状态产生的”保留下来。

15.8.4 安全上限是设计的一部分，不是补丁

运行时内建了显式的失控保护：

保护项	默认值	作用
`max_transitions`	`100`	限制总迁移次数
`max_state_visits`	`10`	限制单个状态最多被访问多少次
顶层 `timeout`	无	限制整个状态机的总墙钟时间
状态 `timeout`	无	限制当前等待状态能挂起多久

对于 `reject -> draft` 这样的回退迁移，这些保护尤其重要。没有它们，错误的事件流或错误的守卫条件就可能让机器无声地永远转下去。

15.8.5 什么时候 **graph** 仍然更合适

遇到下面这些情况时，继续使用 graph：

- 本质上仍然是一条单次穿越依赖的执行路径
- 回到之前的命名状态并不是业务模型的一部分
- 不需要通过外部事件来驱动生命周期切换

如果核心问题是多轮对话，优先考虑 `session`；如果核心问题是显式生命周期状态，优先考虑状态机。

15.9 全局迁移（Global Transitions）

到目前为止，示例中的每条迁移规则都定义在某个具体状态内部。但有些事件应该在任何状态下都能被处理——取消、错误恢复或全局超时。在每个非终态状态里都重复写一遍 `on "CANCEL" -> cancelled` 既冗余又容易遗漏。

`global_transitions` 块正是解决这个问题的：

```
state_machine orderLifecycle {

  global_transitions {
    on "CANCEL"  -> cancelled
    on "ERROR"   -> error_state
    on "TIMEOUT" -> timed_out
  }

  initial_state = pending

  state pending {
    on "PAYMENT_RECEIVED" -> processing
    on "EXPIRED"          -> expired
  }

  state processing {
    on "FULFILLED"    -> completed
    on "OUT_OF_STOCK" -> backordered
  }

  state completed { terminal = true }
  state cancelled { terminal = true }
  state error_state { terminal = true }
  state timed_out { terminal = true }
  state expired { terminal = true }
  state backordered {
    on "RESTOCKED" -> processing
  }
}
```

15.9.1 全局迁移的工作方式

全局迁移会被应用到状态机中每一个非终态状态。它们的行为与状态级迁移完全一致，只是只需要写一次。

如果某个状态为同一事件定义了自己的迁移，那么状态级迁移优先。这样你就可以在某个状态需要不同处理时覆盖全局行为。

15.9.2 什么时候该用全局迁移，什么时候该用状态级迁移

用全局迁移	用状态级迁移
在任何状态都应该生效的取消事件	只在特定生命周期阶段才有意义的事件
错误兜底处理器	基于状态内部输出的守卫迁移
全局超时升级	自动迁移（`on *`）

判断标准：如果从某个状态中移除这条迁移会是一个 bug，那它就应该放进 `global_transitions`。

15.10 常见陷阱

15.10.1 [X] 把一条笔直的 DAG 硬包成状态机

如果你的流程本质上只是：

```
validate -> price -> confirm
```

那么再额外包出 `validating`、`pricing`、`confirming` 这些状态，并不会让模型更清晰，只会增加样板结构。

只有当状态名称真的能表达出 DAG 本身表达不出的生命周期含义时，状态机才值得引入。

15.11 常见故障

15.11.1 回退迁移把你烧进了 `max_state_visits`

假设 `pendingReview` 可以因为 `reject` 回到 `draft`，而某个错误的集成系统又不断发布 `reject`。这时 `max_state_visits` 的意义就会立刻体现出来：运行时会主动停下，而不是让状态机静默地永远兜圈。

修复思路不是“把上限再调大一点”，而是先追问：

- 是守卫条件写错了吗？
- 是外部事件流有问题吗？
- 还是业务生命周期本来就缺了一个中间状态？

上线前一定要专门测试这种回退循环。

15.12 引导式重写

打开 `ticket-state-machine.bloge`。

假设你最初写的是一张“大 graph + 很多 branch 节点”的版本，那么重写成状态机的思路通常是：

1. 先命名稳定的生命周期状态。`open`、`triaging`、`assigned`、`escalated`、`resolved`、`closed`。
2. 把局部工作挪进状态内部 **graph**。接单属于 `open`，分类属于 `triaging`。
3. 把事件边界挪到迁移规则。`assign`、`escalate`、`resolve`、`close` 都是生命周期事件，而不是普通 graph 分支。
4. 在“沉默有意义”的地方增加超时。例如 `resolved` 状态等待太久后自动关闭。

这样重写的价值在于：即使是刚加入项目的人，也能先通过状态名称读出生命周期，再去关心 operator 细节。

15.13 思维检查

1. 状态机比单张分支 graph 更擅长解决什么问题？
   （显式生命周期建模：命名状态、外部事件驱动和跨状态迁移。）
2. `on * -> completed` 代表什么？
   （自动迁移：当前状态内部 *graph* 完成后，立刻进入 *`completed`*。）
3. 为什么示例要从 `ctx.draft.output.initOrder.orderId` 读取数据？
   （因为输出是按状态命名空间组织的，这样数据来源的生命周期位置是显式可见的。）
4. 顶层 `timeout` 和状态级 `timeout` 的区别是什么？
   （前者限制整个状态机的总生命周期，后者限制某个等待状态最多能挂起多久。）
5. 什么时候不应该引入状态机？
   （当流程本质上仍然是一条单次 *DAG*，没有真正重要的命名状态，也不需要跨状态事件驱动时。）
6. 为什么 `reject -> draft` 这类迁移尤其依赖 `max_state_visits`？
   （因为回退迁移最容易在坏事件或坏守卫条件下形成无限循环。）

15.14 实验

目标：把一个真实审批流程建模成状态机。

1. 定义 `draft`、`review`、`approved` 和 `rejected` 四个状态。
2. 把加载数据的步骤放进 `draft`。
3. 把评审富化逻辑放进 `review`。
4. 增加这些迁移：
   - `on submit -> review`
   - `on approve -> approved`
   - `on reject -> draft`
5. 显式设置 `max_transitions` 与 `max_state_visits`。
6. 如果 review 可能长时间卡住，再为该状态加上 `timeout` 和 `on_timeout`。

进阶：再加一条 `on * when ... -> approved` 守卫迁移，让状态输出本身就能决定自动结束生命周期。

15.15 持久状态机（Durable State Machines）

内存中的状态机在进程重启后会丢失当前状态。`bloge-state-durable` 模块提供的 `DurableStateMachineManager` 会把每一次迁移都持久化为检查点。

15.15.1 Maven 依赖

```
<dependency>
    <groupId>com.leanowtech.bloge</groupId>
    <artifactId>bloge-state-durable</artifactId>
</dependency>
```

15.15.2 构建与使用

```
Map<String, StateMachineDef> definitions = Map.of(
    stateMachineDef.name(), stateMachineDef
);

DurableStateMachineManager durableSm = DurableStateMachineManager.builder()
    .executionStore(executionStore)
    .checkpointStore(checkpointStore)
    .graphEngine(graphEngine)
    .definitionLookup(definitions::get)
    .recoveryConfig(StateMachineRecoveryConfig.defaultConfig())
    .build();

durableSm.startup();
StateMachineResult waiting = durableSm.start(
    "orderLifecycle",
    Map.of("orderId", "ORD-123")
);
String executionId = waiting.instance().instanceId();

StateMachineResult result = durableSm.signal(
    executionId,
    "PAYMENT_RECEIVED",
    Map.of("providerRef", "PAY-9")
);
```

`start(...)` 会创建并 claim 一次 durable execution。状态机进入 `WAITING_EVENT` 时，manager 保存 checkpoint，并把 execution 标为 suspended。`signal(...)` 会 claim 这次 execution、加载 checkpoint、用 `definitionLookup` 选择定义恢复，然后投递 event。

调用 `startup()` 后，recovery loop 会扫描过期 claim，并从最近一次已提交 checkpoint 恢复。checkpoint 之后的工作可能被重试，因此外部 effect 仍要满足 幂等。checkpoint/status 是否原子提交，还取决于是否接入 transactional persistence coordinator；默认 direct coordinator 会分别执行两次写入。

这与第 13 章——持久执行中介绍的检查点基础设施是同一套机制，只是应用到了状态机这一层抽象上。

15.16 实验验收卡

- 预期与观察：共享 definition 不共享 current state、visits、output 或 checkpoint。
- 失败与恢复：两个实例误用同一 ID；恢复唯一身份并分别重放。
- 证明边界：证明 definition/instance 隔离，不证明事件来源可信。
- 练习合同：两个工单实例；只改 instance ID；交付并排轨迹；互不污染即停止。

15.17 回顾

- graph 建模的是依赖顺序；状态机建模的是生命周期状态。
- 每个状态都可以运行自己的局部 graph，但迁移才是这个抽象的主角。
- 输出按状态命名空间组织，让生命周期来源保持可见。
- 事件迁移、自动迁移、守卫迁移和超时迁移分别解决不同问题。
- 全局迁移可以处理任意状态下的事件——取消、错误、超时——无需重复定义。
- `max_transitions` 和 `max_state_visits` 不是多余配置，而是回退流程的安全底线。
- `DurableStateMachineManager` 持久化迁移，使状态机能在重启后恢复。
- 当你需要的是“显式状态”，而不只是更多分支时，才应该引入状态机。

15.18 下一步

在 第 16 章——组合 Session 与状态机 中，你将把两种编排模型组合在一起：有些流程是“对话里嵌一个生命周期”，有些则是“状态里嵌一个短小交互”。

15.19 参考链接

- `bloge-state-ext/README.md` ——状态机运行时概览与超时语义
- `OrderLifecycleStateMachineExample.java` ——Java Fluent API 示例
- `order-lifecycle-state-machine.bloge` ——本章生命周期示例
- `ticket-state-machine.bloge` ——更丰富的客服工单状态机示例
- `ReviewStateMachineWithSessionExample.java` ——下一章会展开的嵌套 session 预告

章节
16

第四阶段——生命周期模型

第 16 章——组合 Session 与状态机

Session、状态机、嵌套所有权与受限 Agent

目标：让每一个长生命周期决策只有一个明确 *owner*。

承诺：读完本章后，你将知道如何组合 `session` 和 `state_machine`，而不会弄丢信号所有权、超时语义和输出路径；更重要的是，你会理解这两种嵌套方向都很有用，但并不对称。

16.1 学习目标

1. 对比两种组合模式：session 在外，状态机在内；状态机在外，session 在内。
2. 决定在嵌套流程里，究竟由哪个运行时拥有生命周期、信号、恢复和超时语义。
3. 在嵌套编排中追踪输出路径，而不是把重要结构压平成一层。
4. 理解当前实现里的不对称约束，避免设计出运行时根本不支持的组合方式。
5. 选择最小但足够清晰的组合边界。

16.2 前置条件

- 第 14 章——多轮 Session ——先理解 session 的生命周期
- 第 15 章——状态机 ——先理解显式状态迁移
- 第 13 章——持久执行 ——嵌套运行时仍然站在同一套持久执行基础上

16.3 源示例

文件	展示内容
`ch15/order-session-with-state-machine.bloge`	session 在外：某个 phase 内嵌一个完整订单状态机
`ch15/review-state-machine-with-session.bloge`	状态机在外：某个状态内嵌一个短小 review session

文件	展示内容
ch15/ivr-customer-service.bloge	一个更长生命周期的 session，可作为未来嵌套模式的延展
OrderSessionWithStateMachineExample.java	session → 内嵌状态机 的 Java Fluent API 版本
ReviewStateMachineWithSessionExample.java	状态机 → 内嵌 session 的 Java Fluent API 版本
OrderSessionWithStateMachineExampleTest.java / ReviewStateMachineWithSessionExampleTest.java	证明嵌套形状能编译并运行的测试

16.4 为什么这很重要

真实业务常常同时需要两种抽象：

- 一段客户对话里，包含一个多步骤订单生命周期
- 一个显式生命周期状态里，需要执行一小段自包含的 review 交互，然后才能决定往哪里走

如果把一切都压平到一张 graph 里，模型会变得模糊；如果把一切都拆成独立服务，生命周期边界又会变得模糊。

组合的价值在于，它让业务模型本身保持可见：

- 外层运行时决定“主要生命周期”是什么
- 内层运行时只负责处理其中一个局部编排问题

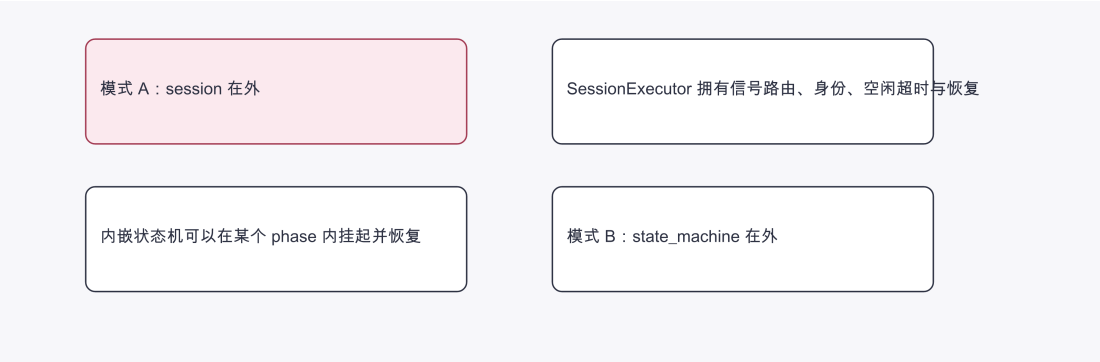
所以真正该问的问题不是“它们能不能嵌套”，而是：哪个运行时才应该拥有这段业务真正重要的生命周期？

16.5 心智模型

当前支持两种方向：

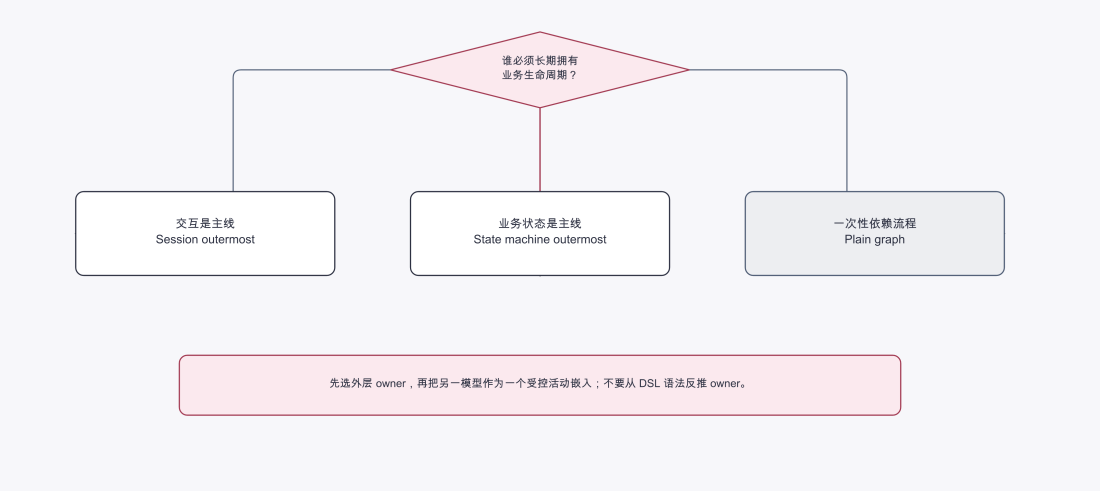
模式	外层所有者	内层所有者	最适合什么
session → 内嵌 state_machine	SessionExecutor	以合成 node 包装的状态机运行逻辑	主要是长生命周期交互，但其中某个 phase 更适合显式状态迁移
state_machine → 内嵌 session	StateMachineExecutor	以合成 node 包装的 session 运行逻辑	主要是状态生命周期，但其中某个状态需要一个短小多步骤交互

两种模式都很有用，但运行时约束并不一样。



16.6 先选外层 owner，再选语法

组合首先只问一个问题：内层活动结束后，哪个 runtime 还必须继续拥有这个 case?



主导事实	外层 **owner**	内层活动示例
客户还会继续多轮对话	Session	ordering phase 内的订单状态机
订单还会继续经历持久业务状态	State machine	review state 内的评审 Session
工作是一次性的依赖流程	Graph	不需要第二种生命周期模型

对 `orderWorkflow`，订单到达 `shipped` 后，调用方还要回到对话，所以 Session 在外。对 `reviewWorkflow`，交互评审结束后 case 必须进入 `approved` 或 `rejected`，所以状态机在外。嵌套方向是 ownership 决策，不是格式偏好。

16.7 第一个可运行的示例

更灵活的一种方向是：**session** 在外，状态机嵌在某个 **phase** 内部。

```
session orderWorkflow {
```

```
  idle_timeout = 72h
  max_rounds = 10

  phase ordering {
    state_machine orderFlow {
      max_transitions = 20
      max_state_visits = 5

      state draft [initial] {
        graph {
          node collectInfo : CollectOrderInfoOperator { }
        }
        on submit -> pendingPayment
      }

      state pendingPayment {
        graph {
          node chargePayment : ChargePaymentOperator { }
        }
        on payment_confirmed -> processing
        on payment_failed -> draft
      }

      state processing {
        graph {
          node shipOrder : ShipOrderOperator { }
        }
        on * -> shipped
      }

      state shipped [terminal] { }
    }
    then -> fulfillment
  }
}
```

用业务语言读它：

- 外层是一段订单 session
- 其中 `ordering` 这个 phase 内部，用状态机表达订单生命周期
- 内层状态机结束后，session 再继续前往 `fulfillment`

外层 session 仍然拥有持久化、发信号入口和整段交互的身份。

16.8 signal 向内传，terminal output 向外传

内层 runtime 拥有自己的 current state，所以外部事件不能直接跳到 `chargePayment` 或 `shipOrder`。它先进入外层 Session，再被翻译成状态机 signal，最后才选择 transition。

16.8.1 跟随一笔订单穿过两个 owner

方向	穿越边界的合同	错误捷径
向内	customer `submit` → Session input → 状态机 `submit` signal	webhook 直接寻址内层 node id
向内	payment callback → owning runtime → `payment_confirmed` event	直接修改 `currentState`
向外	状态机 terminal output 包含 `shipped` 与 `shipmentId`	caller 读取任意内层 workspace
向外	Session `fulfillment` phase 调用 `notifyCustomer`	让内层 Operator 发送不受治理的对话消息

这样两个方向都有 receipt：向内是 accepted signal 与 state transition；向外是 namespaced terminal output 与 Session result。signal 被拒绝时，应先检查 owner、instance id 与 allowed transition，再检查业务 node 代码。

16.9 拆解分析

16.9.1 模式 A ——session 在外，状态机在 phase 内

这正是 `order-session-with-state-machine.bloge` 使用的形状。

关键行为如下：

- 真正的运行时入口仍然是外层 `SessionExecutor`
- 外部 `signal(...)` 也是先打到 session handle，而不是直接发给内层状态机
- 当内层状态机在等待事件时，外层 session 会表现为 `SUSPENDED`

- 当内层状态机恢复运行时，外层 session 又会回到 `ACTIVE`

输出路径之所以长，是因为它保留了所有权信息：

```
finalState = ctx.ordering.output.orderFlow.stateMachine.currentStateId
shipmentId = ctx.ordering.output.orderFlow.processing.output.shipOrder.shipmentId
```

这条路径明确说明数据来自：

- 外层 session 的 `ordering` phase
- 内嵌状态机 node `orderFlow`
- 内层状态 `processing`
- 状态内部 graph 的 `shipOrder` node

16.9.2 模式 B ——状态机在外，session 在状态内

这是 `review-state-machine-with-session.bloge` 使用的形状。

`review` 状态里内嵌了一个 session：

```
state review [initial] {
  session reviewSession {
    phase collect { ... then -> finalize }
    phase finalize { ... }
  }
  on * when ctx.review.output.reviewSession.finalize.output.decideReview.decision == "approved" ->
    approved
  on * -> rejected
}
```

也正是在这里，不对称性最明显。

内嵌在状态机里的 session 目前必须同步完成。如果这个内层 session 还想等待后续外部信号，`SessionOperator` 会直接快速失败，而不会让外层状态机处于一种“半暂停”状态。

所以，这种模式适合：

- 一个短小的两阶段 review
- 一个很紧凑的 enrich → decide 交互
- 一次状态访问内就能结束的内层生命周期

它不适合长时间的人机多轮对话。

16.9.3 当前实现里的组合约束

现在的嵌套形式是故意收窄的：

- 一个 phase 最多只能有一个顶层内嵌 `state_machine`
- 这个内嵌 `state_machine` 必须是该 phase 中唯一的可执行成员
- 一个状态可以内嵌 `session`，但该 session 必须同步完成

这些限制的目的，是让生命周期所有权保持绝对清晰。

约束——避免信号死锁
嵌在 `state_machine` 里的 `session` 不能挂起等待外部信号，只能同步 完成。原因是所有权: 外层状态机执行器持有生命周期锁，不会把控制权让 给另一个信号接收者。如果嵌套 *session* 试图挂起等信号，编译器会抛出 `SM_NESTED_SESSION_AWAITS_SIGNAL` 并拒绝发布该 *graph*。
反方向——`session` *phase* 里嵌套的 `state_machine` ——可以自由挂起，因为外层 *session* 就是信号所有者。任何可能需要外部信号的场景，优先 选 模式 ***A****(session* 在最外层*)*。

16.9.4 生命周期所有权检查清单

在开始嵌套之前，先回答这些问题:

问题	如果答案偏向 `session`	如果答案偏向 `state_machine`
这段业务最重要的形状是什么?	人机对话 / 多轮交互	对象生命周期 / 状态推进
外部信号应该打给谁?	session	状态机
谁拥有空闲超时 / 访问控制?	session 运行时	状态机运行时
面板首先应该显示什么?	当前 phase / 轮次	当前状态
内层是否需要跨外部轮次等待?	外层是 session 时可以	状态机内嵌 session 时不可以

16.9.5 先选外层所有者，再决定是否嵌套

设计嵌套编排时，最稳妥的顺序是:

- 如果人们主要是按对话轮次来理解这段流程，就让 `session` 在外层
- 如果人们主要是按命名状态来理解这段流程，就让状态机在外层

只有在这一步明确之后，你才应该判断某个局部片段是否真的值得单独引入另一套编排模型。

16.10 常见陷阱

16.10.1 **[X]** 误以为两种嵌套方向是对称的

它们不是。

- session 内嵌状态机时，内层状态机可以通过外层 session 的生命周期来等待和恢复
- 状态机内嵌 session 时，内层 session 必须同步结束

如果你按“两个方向都能自由挂起”来设计，就会在很后面才发现模型本身就选错了。

16.11 常见故障

16.11.1 信号被发到了错误的运行时边界

在“session 在外”的模式里，逻辑上等待的也许是内层状态机，但外部信号仍然必须从外层 session 入口进入。
所以这是一种错误心智模型:

```
// 错误：试图直接给内层状态机发信号
stateMachineExecutor.signal(...)
```

正确做法应该是：

```
sessionHandle.signal(Map.of("event", "payment_confirmed"));
```

永远把信号打给外层所有者。

16.12 引导式重写

对照两个 companion 示例，练习在真正写 DSL 之前先决定外层所有者。

1. 先问业务问题。这更像一段对话，还是更像一个显式生命周期?
2. 先选外层运行时。对话用 session，生命周期用状态机。
3. 只把“形状不同”的那一小段内嵌出来。
   - 订单流程：外层是 customer/order session，但其中 `ordering` 更适合状态机
   - review 流程：外层是 review 生命周期，但其中一个状态需要 collect → finalize 的短交互
4. 写一条下游表达式，验证输出路径是否仍然可读。

如果输出路径已经变得难以理解，那往往意味着你的边界选错了。

16.13 思维检查

1. 在“session 在外”的模式里，外部信号应该打给谁?
   （打给外层 *`SessionExecutor`* / *`SessionHandle`*。）
2. 在“状态机在外”的模式里，内嵌 session 能不能等待下一次外部信号?
   （不能。它目前必须同步完成。）
3. 为什么 `ctx.ordering.output.orderFlow.stateMachine.currentStateId` 这样的长路径反而是好事?
   （因为它保留了嵌套边界上的所有权和来源信息。）
4. 设计嵌套流程时，第一件应该决定的事是什么?
   （哪一个运行时拥有主要业务生命周期。）
5. 什么时候“session 在外”通常是更安全的默认值?
   （当流程是长生命周期、由外部信号驱动，并且核心是人机多轮交互时。）
6. 把两种组合方向当成对称结构，最大的风险是什么?
   （你可能会设计出一个需要挂起的内嵌 *session*，但“状态机在外”的模式并不支持这种用法。）

16.14 实验

目标：为两个“看起来相似”的流程选出正确的外层所有者，并实现出来。

1. 建模一个订单流程：客户可能会在几个小时后才回复支付提示。

2. 建模一个内部审批流程：review 交互可以从现有数据立即完成。
3. 对每个流程都写下：
    - 外层运行时
    - 内层运行时（如果需要）
    - 谁接收信号
    - 一条示例输出路径
4. 各实现一份 DSL 文件。

进阶：增加一个测试，证明你可以通过外层 session handle 给“session 在外”的示例发信号，并最终抵达内层状态机的终止状态。

16.15 实验验收卡

- 预期与观察：外层 owner 决定生命周期；signal 向内，terminal output 向外。
- 失败与恢复：让 Session 和状态机同时拥有外层；恢复为唯一 owner。
- 证明边界：证明组合责任和信号方向，不证明任意嵌套无死锁。
- 练习合同：客服组合；只交换外层 owner；交付选择卡和时序图；只剩一个 owner 即停止。

16.16 回顾

- `session` 和状态机可以组合，但两种方向的约束不同。
- 外层运行时拥有信号、恢复和真正重要的业务身份。
- 当主要问题是长生命周期交互时，适合“session 在外”。
- 当主要问题是命名生命周期状态时，适合“状态机在外”。
- 长输出路径不是坏事，它保留了嵌套边界上的来源信息。
- 最稳妥的设计方式，是先选外层所有者，再判断是否真的需要引入内层编排模型。

生产环境下的持久变体。在生产环境中组合 *session* 和状态机时，应接入对应的持久管理器：使用 `DurableSessionManager`（来自 `bloge-session-durable`）实现可崩溃恢复的 *session* 生命周期，使用 `DurableStateMachineManager`（来自 `bloge-state-durable`）实现可崩溃恢复的 状态机生命周期。本章的组合模式保持不变——持久包装器在同一 *API* 表面下增加了 *checkpoint* 持久化。

16.17 下一步

在第 17 章——把非确定性 Agent 放进确定性生命周期中，同样的所有权纪律会把 Agent 限定在明确的工具、轮次与生命周期边界内。第 18 章再回到测试纪律。

16.18 参考链接

- `order-session-with-state-machine.bloge` ——session 在外示例

- `review-state-machine-with-session.bloge` ——状态机在外示例
- `ivr-customer-service.bloge` ——更长生命周期的 session 示例
- `OrderSessionWithStateMachineExample.java` ——Java Fluent API 实现
- `ReviewStateMachineWithSessionExample.java` ——Java Fluent API 实现
- `OrderSessionWithStateMachineExampleTest.java` ——内嵌状态机运行测试
- `ReviewStateMachineWithSessionExampleTest.java` ——内嵌 session 运行测试

章节
17

第四阶段——生命周期模型

第 17 章——把非确定性 Agent 放进确定性生命周期

Session、状态机、嵌套所有权与受限 Agent

目标：让每一个长生命周期决策只有一个明确 *owner*。

本章承诺：你会让客服 *Agent* 只使用白名单只读工具，把轮次、工具并发和退出原因变成可观察事实，并用确定性门禁阻止不完整回答把工单误标为已解决。

17.1 学习目标

学完本章，你能够：

1. 区分 Session 轮次、工单状态迁移和 Agent 模型轮次。
2. 把 tool list 当成 capability allowlist，而不是提示词建议。
3. 区分 `finishReason`、loop exit reason 和业务“已解决”。
4. 正确处理正常停止、工具失败、`max_turns` 和可恢复 checkpoint。
5. 在 Agent 之后放置确定性验证与人工交接门禁。

17.2 这一章只增加一个变量

客服故事已经有稳定的 `ticketId`、多轮 Session 和工单状态机。本章不再发明第四套生命周期；只在 `triaging` 阶段内加入一个非确定性的模型—工具循环。

```
Session round: 客户与系统交流了几轮
Ticket state: open → triaging → assigned / escalated → resolved
Agent turn: 模型响应 → 工具调用 → 观察 → 下一次模型响应
```

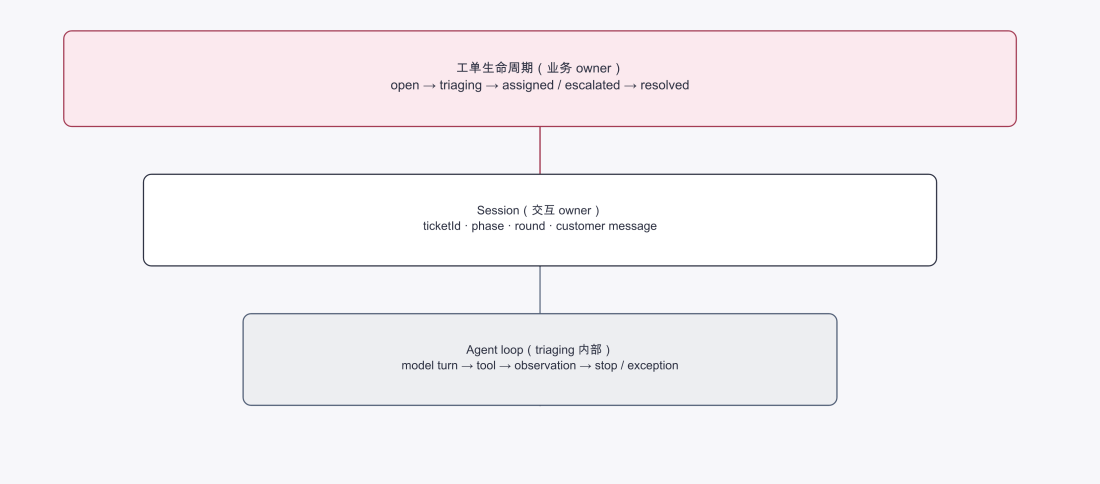
三种计数可以同时变化，但不能用一个“轮次”概括。

17.3 开场：一句“已经解决”能关闭工单吗

客户说：“上次退款还没到账。”Agent 搜索知识库后回答：“退款通常需要 3—5 个工作日，问题已经解决。”

先判断：此时 `ticket.status` 应该变成 `resolved` 吗？

不能。模型文本只是一项候选解释。系统仍不知道具体退款是否存在、是否超过 SLA、是否需要人工查账，更没有发生任何退款 effect。



17.4 最小客服 Agent：只开放两个只读工具

```
agent supportTriage {
  model = "replaceable-support-model"
  system_prompt = "Clarify the issue. Never claim a refund was issued."
  max_turns = 4
  max_tool_concurrency = 2
  memory = sliding_window(6)

  tool searchKnowledgeBase : KBSearchOperator {
    description = "Search approved support articles"
    input { query = tool_args.query }
  }

  tool readTicket : ReadTicketOperator {
    description = "Read the current ticket snapshot"
    input { ticketId = ctx.ticketId }
  }

  exit_condition = finish_reason == "stop"
}
```

注意工具参数的两个来源：`query` 由模型产生；`ticketId` 从外层受控上下文注入，模型不能换成别人的工单。这里没有 `refund`、`closeTicket` 或 `changeSla`。提示词中的“不要退款”是行为指引，tool list 中不存在退款

capability 才是执行边界。

17.5 先观察 RC1 的真实循环语义

运行编译器和 loop 聚焦测试：

```
cd submodule/bloge
mvn -pl bloge-agent-ext \
  -Dtest=AgentLoopOperatorTest,AgentDslCompileTest test
```

2026-09-14 在 commit `cc38fbe5` 上观察到：

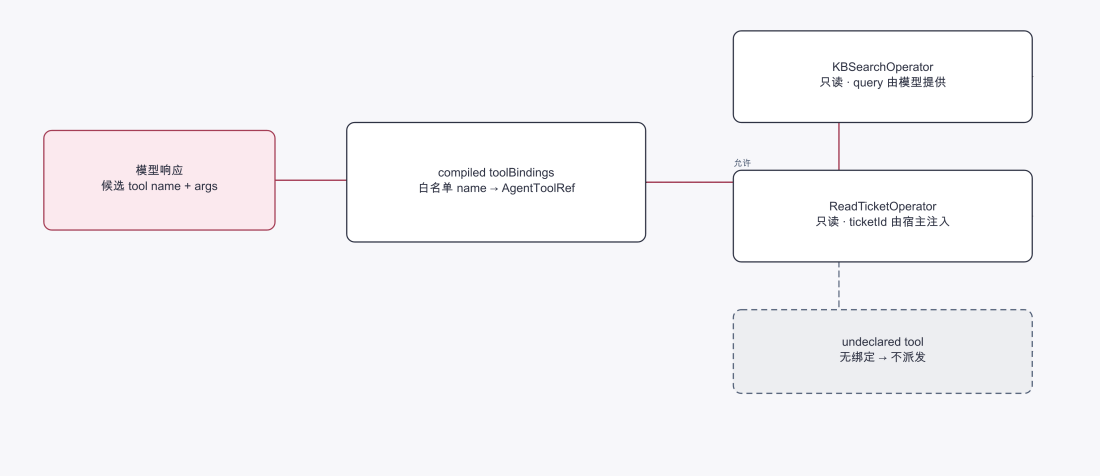
```
AgentLoopOperatorTest  Tests run: 6, Failures: 0
AgentDslCompileTest    Tests run: 5, Failures: 0
Tests run: 11, Failures: 0, Errors: 0, Skipped: 0
```

这 11 项覆盖直接 `stop`、工具调用后继续、多工具并发上限、工具失败回注、`max_turns` 异常、DSL schema 推导和嵌入式 operator。

17.6 机制：一次 turn 里发生什么

`AgentLoopOperator` 的同步路径是：

```
组装 memory
  → 调用 llmChat
    → 无 tool call 或 finish_reason=stop：返回 AgentOutput
    → 有 tool call：只在声明的 toolBindings 中派发
      → 把 tool result 追加到 memory
        → 检查 exit_condition
          → 下一 turn 或退出
```



`ToolDispatcher` 按工具名查 `AgentToolRef`，后者指向编译时生成的子图和 operator。一个未声明的工具名没有可派发绑定。宿主仍需保证 registry 里的 operator 身份和权限正确；白名单不能修复一个本身越权的 operator。

17.7 四个退出结果必须分开读

运行结果	RC1 行为	下游应如何处理
模型 `stop`，无工具调用	返回 `AgentOutput`	仍需业务门禁
`exit_condition` 为真	返回 `AgentOutput`；listener 记录 exit reason	仍需业务门禁
工具失败	错误变成 tool message，loop 可继续	检查回答是否降级、是否需人工
达到 `max_turns`	抛 `MaxTurnsExceededException`，携带 `AgentCheckpoint`	Agent 节点失败；保存/转人工，不得当最终输出

这里纠正一个常见误读：RC1 达到 `max_turns` 时不会把部分文本作为正常 **`AgentOutput`** 交给下游。部分进度存在异常携带的 checkpoint 中，包括 conversation、turn count 和最近 tool results。

正常返回的 `AgentOutput.finishReason` 是模型最后一轮给出的 reason；`exit_condition` 是 listener 的 loop-exit reason。两者都不是业务状态 `resolved`。

17.8 预算：真正受控的和还没受控的

`max_turns` 给模型调用次数设硬上限，`max_tool_concurrency` 限制单轮工具并发。`sliding_window(6)` 只控制每轮发送多少历史，不是总 token/费用预算。

RC1 的 loop 会累计 token 并发给 listener，但本章源码范围内没有“超过 total tokens 自动停止”的硬门禁。若业务要求费用上限，宿主必须在 provider、operator/interceptor 或外层控制面建立可拒绝的预算，并把拒绝映射为明确状态。

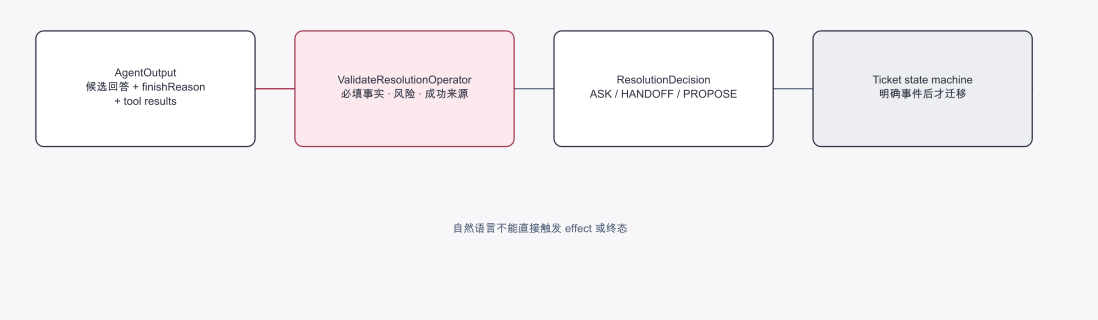
17.9 确定性下游门禁：回答可以变化，准入规则不能漂

Agent 正常返回后，`ValidateResolutionOperator` 只读取结构化事实：

```
ResolutionDecision validate(AgentOutput out, TicketSnapshot ticket) {
    if (!"stop".equalsIgnoreCase(out.finishReason())) return HANDOFF;
    if (out.content() == null || out.content().isBlank()) return HANDOFF;
    if (!ticket.requiredFactsComplete()) return ASK_CUSTOMER;
    if (ticket.refundOrSlaDispute()) return HUMAN_REVIEW;
```

```
    return PROPOSE_RESOLUTION;
}
```

这段门禁不判断回答“写得像不像人”，只判断关闭工单所需事实是否完整。PROPOSE_RESOLUTION 仍是提议；状态机接收明确事件后才迁移到 resolved。



稳定的责任链是：

```
Agent 产生候选回答
  → deterministic gate 检查结构事实
    → state machine 接收明确事件
      → 人工或系统 owner 承担最终状态
```

17.10 单因素失败实验：把 max_turns 从 4 改成 2

保持模型序列和工具响应不变：第 1 轮请求知识库，第 2 轮再次请求工具，没有 stop。AgentLoopOperatorTest.throwsWhenMaxTurnsAreExhausted 断言：

```
exception = MaxTurnsExceededException
maxTurns = 2
checkpoint.turnCount = 2
```

恢复不是把 checkpoint 中最后一句话直接展示为“答案”。先保存 checkpoint 和 exit reason，把工单保持 triaging，再由策略决定重新运行、请求客户补充或转人工。

17.11 另一个失败：工具报错但 loop 给出流畅回答

RC1 会把工具异常序列化为 tool message，让模型有机会解释或降级。测试中的 failing tool 抛出 boom，下一轮返回 Handled fallback.。

技术上 loop 成功返回，不代表业务事实已获得。确定性门禁必须知道 required fact 是否来自成功工具结果；一句流畅的 fallback 不能把“查不到退款记录”变成“退款正常”。

17.12 现实映射：急诊分诊助手

急诊助手可以查询患者登记信息和分诊手册，但不能因为文本模型说“可以回家”就完成出院。模型建议后，规则门禁检查生命体征和必填检查，医生承担最终状态迁移。

客服系统	急诊系统
只读工单/知识库工具	患者资料/指南查询
`max_turns`	问诊步骤上限
部分 checkpoint	未完成的问诊记录
resolution gate	必填检查与风险门禁
人工接管	医生决策

映射的核心不是“都用了 AI”，而是非确定性建议被包在确定性责任边界里。

17.13 轮到你：为一个 **Agent** 画权限与退出卡

选择你系统里一个候选 Agent：

1. 列出最多三个只读工具和零个默认写工具。
2. 给每个参数标注模型提供或宿主注入。
3. 定义 `max_turns`、单轮工具并发和真正的总预算 owner。
4. 写出 `stop`、tool failure、max turns 三条结果路径。
5. 设计一个只依赖结构化事实的 downstream gate。

产物是一张权限边界图和一张退出矩阵。停止条件是：任何自然语言输出都不能绕过门禁直接触发高风险 effect 或终态。

17.14 本章证明了什么，没有证明什么

RC1 聚焦测试证明：声明的 Agent DSL 可编译为嵌入式 operator；模型可调用白名单工具；并发上限被执行；工具失败可回注；耗尽轮次会抛出带 checkpoint 的异常。它没有证明模型回答正确、prompt 能构成权限边界、总 token 预算已受控，也没有证明工单可自动关闭。

参考字段、Builder、memory、streaming 和 listener 目录保留在附录 E。下一章先进入第 18 章——测试你的 Graph，建立测试层次；Arc 6 再把业务 verdict、trust 和 capability 分开。

17.15 实验验收卡

- 预期与观察：受限 Agent 以 finish reason、turn budget 和确定性门禁交付。
- 失败与恢复：把 max_turns 改成 2 或让工具报错；恢复为明确 partial result。
- 证明边界：证明 loop 和权限受控，不证明模型回答正确或获批。
- 练习合同：一张工单；只改 allowlist 或预算；交付退出卡；每种退出都有下游动作即停止。

17.16 精确事实入口

- `AgentLoopOperator.java`
- `AgentLoopOperatorTest.java`
- `AgentDslCompileTest.java`
- `AgentOutput.java`
- `MaxTurnsExceededException.java`
- 客服工单 Story Bible

第四阶段回顾——生命周期模型（第 14–17 章）

章节

—

第四阶段回顾——生命周期模型（第 14–17 章）

你现在已经可以走出“一次性 *graph*”的边界，用更清晰的方式建模长生命周期编排。

0.1 你学到了什么

第 **14** 章——多轮 **Session** 把原本只放在附录中的 session 内容提升到了主线阅读路径。你学会了如何用 `SessionGraph`、`ONCE` phase 和 `ROUND` phase 描述跨越多次外部信号的交互；持久化和重启恢复还需要 durable manager 与 store。

第 **15** 章——状态机 引入了显式生命周期建模。你学会了如何用状态、事件迁移、守卫条件和超时规则，让状态推进过程保持可见，而不是藏在一堆分支和循环里。

第 **16** 章——组合 **Session** 与状态机 展示了如何把两种运行时组合起来而不丢失信号所有权和生命周期边界。你学会了：嵌套组合是有价值的，但绝不是对称的，真正重要的是外层运行时拥有主要业务生命周期。

第 **17** 章——把非确定性 **Agent** 放进确定性生命周期 增加了受限工具调用循环。你学会了限制工具和轮数，并让显式 graph——而不是模型——拥有退出与失败策略。

0.2 你现在应该能做到

- 判断一段流程应该建模成 graph、`session`、状态机，还是它们的有意识组合
- 把一段多轮交互拆成 `ONCE` / `ROUND` phase，并为其设置明确的限制和超时策略
- 用事件迁移、自动迁移、守卫迁移和超时迁移建模显式生命周期
- 解释嵌套流程中究竟由哪个运行时拥有信号、恢复和身份
- 在嵌套输出路径里保留状态和 phase 的来源信息，而不是把它们压平

0.3 这个阶段的常见错误

错误	为什么会犯	如何纠正
把每个 `await` 都升级成 `session`	感觉 session 是“更高级”的工具	只有真正多轮的交互才该用 `session`；只有一次挂起 / 恢复的流程依然适合普通 graph
把状态机当成“高级版 branch 节点”	表面上看也还是条件分流	先问自己：命名状态和回退迁移是不是业务模型的核心
忘了配 phase 级 `max_rounds`	session 总上限看起来像是已经够了	每个真实 `ROUND` phase 都要显式设置自己的轮次上限
在嵌套流程里忽视所有权	两套运行时似乎都能挂起 / 恢复	永远先选外层所有者，所有信号都先进入那个边界
误以为内嵌 session 与内嵌状态机是对称结构	表面语法看起来很相似	记住当前约束：状态机里的内嵌 session 必须同步完成

0.4 新增核心词汇

术语	含义
`SessionGraph`	由一组命名 phase 构成的多轮交互定义
`ROUND` phase	会按信号 payload 恢复，并可持续重复直到 `until` 命中的 phase
`SessionExecutor`	用于启动、发信号、恢复和终止 session 的运行时入口
`StateMachineDef`	由命名状态和迁移构成的不可变生命周期定义
守卫迁移	只有当某个基于状态输出的谓词成立时才会触发的迁移
生命周期所有权	在嵌套编排中，由外层运行时拥有信号、恢复和业务身份的原则
受限智能体循环	明确工具、轮数上限、退出条件和 graph 所有失败行为的工具调用循环

0.5 试一试：说出生命周期 **Owner**

一个客服工单有 30 天生命周期，一段四轮客户对话，每轮里的 agent 最多调用两个工具。请画三个嵌套框，分别标出谁拥有工单状态、会话轮次和工具循环退出。如果两个框同时声称拥有同一个迁移，就重画边界，直到每个决策只有一个 owner。

0.6 下一步

第五阶段回顾 将把重心从生命周期模型转向技术信心：测试、可观测性、Spring 接线和规模化运行。先从第 18 章——测试你的 Graph开始。

学习阶段

第五阶段——技术信心

测试、可观测性、Spring 接线与容量边界

目标：让每个技术 *claim* 都匹配真正支持它的证据。

章节
18

第五阶段——技术信心

第 18 章——测试你的 Graph

测试、可观测性、Spring 接线与容量边界

目标：让每个技术 *claim* 都匹配真正支持它的证据。

承诺：读完本章后，你将知道如何在 *graph* 进入生产环境之前验证它声明的契约——通过 `GraphResult` 做断言、用测试替身隔离 *operator*、借助 `GraphTestRunner` 执行整张 *graph*、在运行前用 `DslTestHelper` 捕获 *DSL* 错误，并用确定性时间控制验证时间相关行为。

18.1 学习目标

1. 把 `GraphResult` 作为首要测试面：断言成功、以类型安全方式读取输出，并在行为变化时检查节点状态或耗时。
2. 使用 `MockOperator` 隔离 operator 行为，让测试聚焦在 graph 接线而不是外部 I/O。
3. 使用 `GraphTestRunner` 在专用测试夹具中执行 graph，并验证执行顺序、跳过行为和状态变化。
4. 使用 `DslTestHelper` 编译 DSL 片段，在运行前捕获 parse / compile 错误，并让测试层级与要阻止的失败相匹配。
5. 使用 `ManualTimeSource`、`DeterministicTimerService` 和 `TestGraphEngine` 对 retry、delay 和 timer 行为做确定性时间控制。

18.2 前置条件

- 第 2 章——你的第一个 Graph ——你需要知道 graph 如何加载与执行。
- 第 6 章——韧性设计 ——retry、timeout 和 fallback 会影响“正确行为”的断言方式。
- 第 7 章——设计良好的 Operator ——在隔离 operator 之前，你需要先理解 operator 契约。

18.3 源示例

文件	展示内容
GraphResultObservabilityTest.java	GraphResult 的执行标识、耗时和兼容性行为
GraphResultSafeAccessTest.java	测试中安全读取类型化输出的模式
MockOperator.java	返回、抛错、延迟、记录调用的测试替身
GraphTestRunner.java	带顺序、状态和执行日志断言的 graph 测试夹具
DslTestHelper.java	.bloge 片段的 parse / compile 断言
TestGraphEngine.java	内置 ManualTimeSource 与 timer 控制的确定性测试引擎
TestGraphEngineTest.java	用逻辑时间推进替代墙钟等待的最小示例
DeterministicRetryTest.java	用手动时间推进断言 retry/backoff 的测试

18.4 为什么这很重要

一张 graph 在代码评审里看起来没问题，仍然可能以三种昂贵的方式出错：

- 分支调整后跳过了原本应该执行的节点；
- fallback 掩盖了你本来想暴露的失败；
- DSL 重构在脑海里成立，但在真实编译器里并不合法。

好的 BLOGE 测试会在最便宜的层级把这些问题暴露出来。你不需要为了验证一个依赖边、一条 fallback 路径或一个类型化输出契约而启动 Spring 或调用真实服务。

BLOGE 的测试思路很简单：断言 **graph** 的契约，而不是偶然的实现细节。先从 GraphResult 开始；当你需要更聚焦时，再用 MockOperator、GraphTestRunner 或 DslTestHelper 放大问题。

18.5 一次测试全绿，不等于四层声明都成立

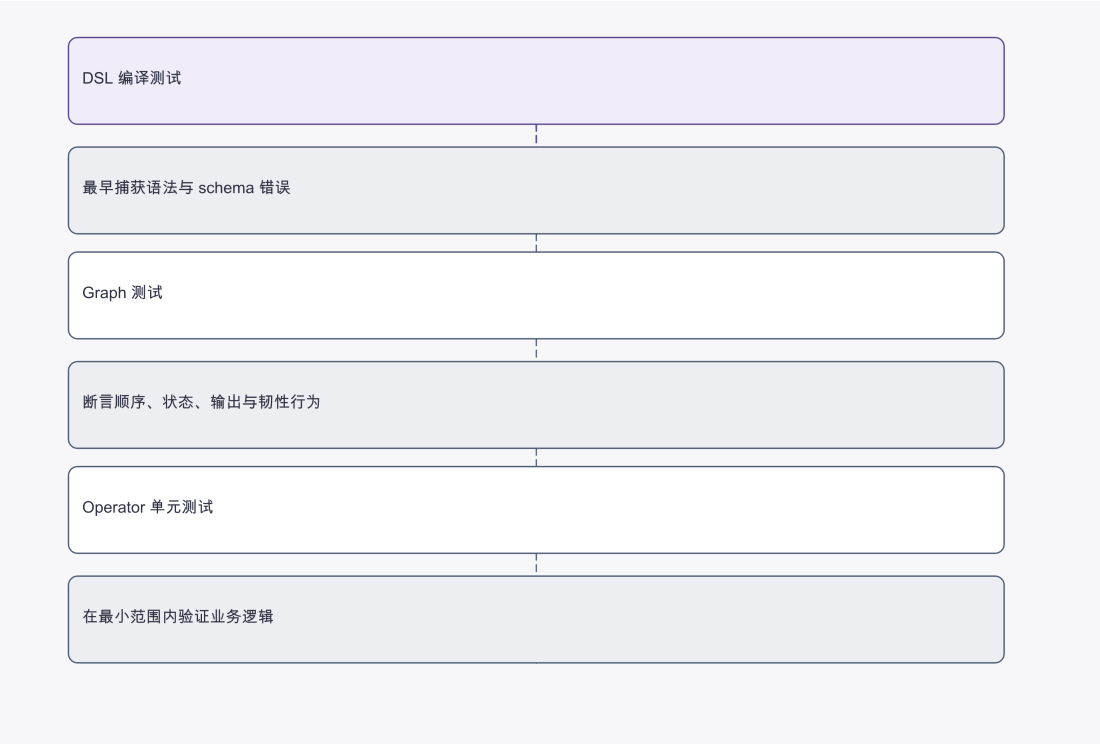
订单测试可以完全通过，但更宽的声明仍然未知。每次读测试结果时，都要同时读 它实际覆盖的边界：

声明	所需证据	本章能提供什么
DSL 合法	真实 parser/compiler 结果	`DslTestHelper`
graph 满足声明的契约	状态、顺序、输出、时间与失败断言	`GraphTestRunner` 与 `GraphResult`
业务故事已获批准并仍被满足	Owner 批准的 Scenario、Policy、Fixture 与 Oracle	本章不能建立
发布声明已绑定源码并受治理	密封证据、精确源码身份、Requirement 与 ClaimCapability	本章不能建立

因此，`OrderGraphTest` 通过只支持一句精确结论：“在这些测试输入和替身下，graph 产生了这些可观察节点事实。”它不支持“订单业务正确”或“该版本具备发布资格”。

18.6 心智模型

把测试看作三层同心安全网：



每一层回答的问题都不同：

- **DSL** 编译测试 问：”这个 graph 定义合法吗？”
- **Graph** 测试 问：”工作流满足这个测试声明的契约吗？”
- **Operator** 单元测试 问：”这个 operator 的输入输出转换对吗？”

一旦测试失败，所处层级就会告诉你下一步该往哪里看。

18.7 第一个可运行的示例

这个示例展示了最常见的生产需求：运行一个 graph，断言它成功了，检查输出，并验证逐节点的耗时。以下代码是纯 JUnit 5，除了 `bloge-core` 之外没有外部依赖。这 55 行 class 保留完整，目的是让 imports、graph 构造、执行和六类断言仍是一份 可复制测试，而不是彼此脱节的片段。

```
import com.leanowtech.bloge.core.dsl.GraphBuilder;
import com.leanowtech.bloge.core.engine.GraphEngine;
import com.leanowtech.bloge.core.engine.GraphResult;
import com.leanowtech.bloge.core.model.NodeStatus;
import com.leanowtech.bloge.core.operator.Operator;
import com.leanowtech.bloge.core.spi.DefaultOperatorRegistry;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;
```

```
class OrderGraphTest {

    @Test
    void orderGraph_completesSuccessfully() {
        Operator<Void, String>   validate = (in, ctx) -> "valid";
        Operator<String, String> price    = (in, ctx) -> "$42.00";
        Operator<String, String> confirm  = (in, ctx) -> "ORD-001";

        var gb = new GraphBuilder("order");
        var graph = gb
                .node("validate", validate)
                .node("price", price).dependsOn("validate")
                    .input((results, ctx) -> results.get("validate", String.class))
                .node("confirm", confirm).dependsOn("price")
                    .input((results, ctx) -> results.get("price", String.class))
                .build();

        var engine = GraphEngine.builder()
                .registry(new DefaultOperatorRegistry())
                .build();

        GraphResult result = engine.executeWithOperators(graph, null, gb.operators());

        // 1. 整体成功
        result.requireSuccess();                          // 如果有错误则抛异常

        // 2. 类型化输出提取
        assertEquals("ORD-001", result.getOutput("confirm", String.class));

        // 3. 当节点可能不存在时的安全访问
        assertTrue(result.findOutput("confirm", String.class).isPresent());
        assertTrue(result.findOutput("missing", String.class).isEmpty());

        // 4. 逐节点状态
        assertEquals(NodeStatus.COMPLETED, result.getStatus("validate"));
        assertTrue(result.nodeSucceeded("price"));

        // 5. 逐节点耗时（执行后始终非 null）
        assertNotNull(result.nodeTimings().get("validate"));
        assertFalse(result.nodeTimings().get("validate").isZero());

        // 6. 执行标识
```

```
        assertNotNull(result.executionId());
    }
}
```

源码：GraphResultObservabilityTest.java 和 GraphResultSafeAccessTest.java 演示了上述所有访问方法。

18.8 拆解分析

18.8.1 GraphResult ——你的首要断言目标

GraphResult 是一个包含九个字段的不可变 record。在测试中最常使用的访问器：

方法	返回类型	使用场景
isSuccess()	boolean	快速通过/失败检查
requireSuccess()	this 或抛出 GraphExecutionException	在测试中快速失败——异常消息列出所有错误
getOutput(nodeId, type)	T	你确定节点已完成
findOutput(nodeId, type)	Optional<T>	节点可能被跳过或失败
getOutputOrDefault(nodeId, type, default)	T	需要一个兜底值
getOutputIfSuccess(nodeId, type)	Optional<T>	只在整个 graph 成功时才关注输出
nodeSucceeded(nodeId)	boolean	检查单个节点，无需提取输出
statusMap()	Map<String, NodeStatus>	遍历所有状态
nodeTimings()	Map<String, Duration>	性能断言
errors()	List<NodeError>	查看具体的失败详情
suspendedNodes()	Map<String, String>	nodeId → suspendKey，表示挂起的节点
compensationResults()	List<CompensationResult>	graph 失败后的补偿结果
nodeSchemas()	Map<String, SchemaDescriptor>	逐节点输出 schema

18.8.2 bloge-test 模块

bloge-test 模块提供了三个专为 graph 测试设计的工具：

MockOperator<I, O> ——一个记录每次调用的测试替身。

```
var fetch = MockOperator.<String, String>returning("Alice");
// ... 执行 graph ...
assertEquals(1, fetch.callCount());
assertEquals("test-input", fetch.lastInput());
```

工厂方法：returning(value)、throwing(exception)、delaying(duration, value)、recording()、of(function)。

源码：MockOperator.java

GraphTestRunner ——包装 GraphEngine，内部使用一个 ExecutionListener 来捕获 START:、COMPLETE:、

`FAILED:` 和 `SKIPPED:` 日志条目。

```
var runner = new GraphTestRunner(Map.of(
    "validate", validate,
    "price",    price
));
runner.execute(graph);

runner.assertNodeExecuted("validate");
runner.assertNodeSkipped("unreachable");
runner.assertExecutionOrder("validate", "price");

List<String> log = runner.executionLog();
// ["START:validate", "COMPLETE:validate", "START:price", "COMPLETE:price"]
```

源码：`GraphTestRunner.java`

`DslTestHelper` ——编译 DSL 字符串并断言解析/编译错误。

```
var helper = new DslTestHelper(registry);
Graph graph = helper.compile("""
    graph test {
        node a : MyOperator { input { key = ctx.value } }
    }
    """);

ParseException err = helper.expectParseError("graph { bad }");
assertTrue(err.getMessage().contains("Expected"));
```

源码：`DslTestHelper.java`

18.8.3 对挂起节点、补偿和 schema 做断言

`GraphResult` 提供了三个额外字段，为挂起/恢复工作流、saga 风格的补偿和 schema 契约测试开辟了新的断言面。

`suspendedNodes()` ——一个 `Map<String, String>`，其中 key 是 nodeId，value 是该节点交回的 suspend key。用它来验证节点确实正确挂起了，且其 key 与你的恢复逻辑预期一致。

```
GraphResult result = engine.execute(graph, context);

// "approval" 节点应该已挂起
assertTrue(result.isSuspended());
assertEquals("pending-manager-approval",
             result.suspendedNodes().get("approval"));

// 其他节点不应挂起
assertFalse(result.suspendedNodes().containsKey("validate"));
```

`compensationResults()` ——一个 `List<CompensationResult>`，在配置了补偿 operator 的 graph 失败后收集。每个条目告诉你补偿是否运行了、是否成功。

```
GraphResult result = engine.execute(failingGraph, context);
assertFalse(result.isSuccess());

List<CompensationResult> compensations = result.compensationResults();
assertFalse(compensations.isEmpty());

// 验证支付节点已成功补偿
CompensationResult paymentComp = compensations.stream()
        .filter(c -> c.nodeId().equals("processPayment"))
        .findFirst().orElseThrow();
assertTrue(paymentComp.isSuccess());
assertEquals("RefundOperator", paymentComp.operatorRef());

// 验证失败的补偿被暴露出来，而不是被吞掉
CompensationResult inventoryComp = compensations.stream()
        .filter(c -> c.nodeId().equals("reserveInventory"))
        .findFirst().orElseThrow();
assertFalse(inventoryComp.isSuccess());
assertNotNull(inventoryComp.error());
```

`nodeSchemas()` ——一个 `Map<String, SchemaDescriptor>`，包含逐节点的输出 schema。用它做 **schema** 契约测试，在不运行完整 graph 的情况下验证每个节点的输出结构。

```
GraphResult result = engine.execute(graph, context);

// 验证 "fetchUser" 节点暴露了预期的 schema
SchemaDescriptor userSchema = result.nodeSchemas().get("fetchUser");
assertNotNull(userSchema);
assertInstanceOf(StructuredSchema.class, userSchema);

StructuredSchema structured = (StructuredSchema) userSchema;
assertTrue(structured.fieldNames().containsAll(
        List.of("id", "name", "email", "vipLevel")));
```

这三个字段在未填充时默认为空集合，因此已有测试无需修改即可继续通过。

18.8.4 测试时间相关的工作流

时间是最容易让测试套件变慢、变脆，甚至两者兼而有之的因素之一。

BLOGE 的测试模块专门提供了确定性时间控制：

- `ManualTimeSource` 只会在测试显式推进时前进
- `DeterministicTimerService` 会在时间推进时同步触发已调度的定时器

- TestGraphEngine 把这两者接成一个现成可用的 GraphEngine

```
var registry = new DefaultOperatorRegistry();
var testEngine = TestGraphEngine.create(registry);

var op = MockOperator.<Object, String>delaying(Duration.ofMinutes(10), "done");
var builder = new GraphBuilder("delay-test");
Graph graph = builder.node("slow", op).build();

var resultRef = new AtomicReference<GraphResult>();
var done = new CountDownLatch(1);

Thread.startVirtualThread(() -> {
    resultRef.set(testEngine.executeWithOperators(graph, new GraphContext(), builder.operators()));
    done.countDown();
});

testEngine.awaitPendingSleepers(1);      // 先等延迟真正注册完成
testEngine.advanceTime(Duration.ofMinutes(10));

assertTrue(done.await(10, TimeUnit.SECONDS));
assertEquals("done", resultRef.get().getOutput("slow", String.class));
```

这里最关键的细节是：只有在 **sleeper** 或 **timer** 已经注册完成之后，才推进逻辑时间。这就是 awaitPending

`Sleepers(...)` 存在的意义。

`ManualTimeSource.advance(...)` 还会同步触发 timer 回调，所以一次时间推进就可能同时释放 sleeper、触发持久 timer，并让整张 graph 在没有任何墙钟等待的情况下继续跑完。

18.8.5 当前最适合用确定性时间验证的场景

这套测试夹具尤其适合下面这些时间相关行为：

- 延迟 operator 和 retry backoff
- 普通 graph 中的 loop / wait timer
- durable lease 过期与定时恢复基础设施
- 执行器显式接入 timer service 时，状态机里的 timer 驱动路径

但也有一个值得明确写出来的现实约束：当前 session 空闲超时路径仍然使用它自己的调度运行机制，而不是直接挂在 `ManualTimeSource` 上。因此，确定性单元测试目前最适合 graph 与状态机级 timer；session 空闲超时依然应该测试，但更适合用聚焦 smoke test 和策略级断言，而不是假设它已经拥有完整的逻辑时钟控制。

18.9 限定租约围栏测试范围——`@WithExecutionLeaseFencing`

当两个引擎共享一个持久化 store 时，同一次执行的执行租约应该只被其中一个 持有。如果租约过期、第二个引擎接手了，原来的引擎就不能再被允许提交陈旧的 checkpoint。运行时通过给每次 store 写入加 **fencing token** 来强制这一点，但测试需要一种方式来证明这套强制是正确接上的。

`@WithExecutionLeaseFencing` 要与 `bloge-test` 的 `ExecutionLeaseFencingExtension` 配合使用。它只负责一件聚焦的事：

1. 在当前测试范围内临时修改默认 fencing policy；
2. 可选择在缺少 ambient lease context 时启用严格拒绝；
3. 测试结束后恢复此前的默认值。

它不会创建 engine、注入 store、让 lease 过期，也不会自动断言每次写入。这些准备与断言仍由测试本身负责。

```
@ExtendWith(ExecutionLeaseFencingExtension.class)
@WithExecutionLeaseFencing
class CheckoutDurabilityTest {

    @Test
    @WithExecutionLeaseFencing(enabled = true, strictMode = true)
    void missingLeaseContextRejectsDurableWrite() {
        InMemoryExecutionStore store = preparedStoreWithExecution("checkout-1");

        assertThrows(StaleFencingEpochException.class, () ->
            store.updateStatus("checkout-1", ExecutionStatus.COMPLETED, 1));
    }
}
```

这个精简示例里的 helper 只创建 execution record。真正的 takeover 测试还要：第一次 claim、release、由第二个 owner 再 claim，然后在第一次 claim 的 `ExecutionLeaseContext` 中尝试旧写入并断言被拒绝。证明旧 owner 被围栏挡住的 是这段时序，而不是 annotation 本身。

18.10 快照测试

只断言某一个最终输出值，你会丢掉整次执行的形状。快照测试把每一次节点 生命周期事件都捕获成一个稳定的 `GraphExecutionSnapshot`，然后整体断言。这 是写" 这张图还和以前一样跑" 这种回归测试的正确工具。

```
SnapshotCapturingListener listener = new SnapshotCapturingListener();

GraphEngine engine = GraphEngine.builder()
    .registry(registry)
    .listeners(List.of(listener))
    .build();

GraphResult result = engine.execute(checkoutGraph, ctx);

GraphExecutionSnapshot snapshot = GraphExecutionSnapshot.capture(result, listener);

GraphSnapshotAssert.assertMatchesBaseline(
    snapshot,
    "checkout-happy-path.snapshot.json"
);
```

`GraphSnapshotAssert` 用 JSON baseline 比较已捕获的 snapshot：

- `assertMatchesBaseline(...)` 对 completed/skipped 节点使用集合语义，避免合法并行完成顺序造成假失败。
- `assertMatchesBaselineStrict(...)` 还要求 completed 节点顺序一致。
- `assertEquals(...)` 与 `assertEqualsStrict(...)` 比较两个内存 snapshot。
- `updateBaseline(...)` 写出可审阅 baseline；它应由人明确触发，而不是测试自动刷新。

只在整段可观察执行形状确实是稳定合同时使用 snapshot。对窄规则和边界用例，单点断言通常更容易说明到底哪里错了。

18.11 常见陷阱

18.11.1 [X] 只断言最终输出

```
GraphResult result = runner.execute(graph);
assertEquals("ORD-001", result.getOutput("confirm", String.class));
```

这个断言只能证明一件事：终点节点返回了你期望的值。它不能证明 graph 确实干净地成功了、上游节点没有被跳过，或者某条你关心的失败没有被韧性策略掩盖。

更稳妥的模式是分层断言：

```
GraphResult result = runner.execute(graph);
result.requireSuccess();
assertTrue(result.nodeSucceeded("validate"));
assertTrue(result.nodeSucceeded("price"));
assertEquals("ORD-001", result.getOutput("confirm", String.class));
```

先断言整体成功，再检查这个场景真正重要的少数节点事实，最后再断言最终输出。

18.12 引导式重写

把第一个示例中的三节点 graph 按下面三步加强测试：

1. 用 `MockOperator.returning("$42.00")` 替换真实的 `price` operator，并断言它只被调用一次。
2. 使用 `GraphTestRunner` 执行 graph，并断言顺序为 `validate → price → confirm`。
3. 故意破坏 DSL ——删除一个节点的 operator 类型——然后用 `DslTestHelper.expectCompileError(...)` 断言这个失败，而不是等到运行时才发现。

完成后，你应该拥有针对同一张小 graph 的三层覆盖：编译、执行和 operator 行为。

18.13 思维检查

1. 在失败时提供清晰节点错误摘要、并让测试立即失败的最快方法是什么？
   （对 *`GraphResult`* 调用 *`requireSuccess()`*。）
2. 什么情况下应该使用 `findOutput(nodeId, type)` 而不是 `getOutput(nodeId, type)`？
   （当该节点在当前场景下可能被合法跳过、失败或不存在时。）
3. `MockOperator` 相比手写 lambda，更擅长解决什么问题？
   （它会记录调用信息，并提供可复用的返回 / 抛错 / 延迟行为，减少测试样板代码。）
4. 哪类问题最适合由 `DslTestHelper` 捕获？
   （*`.bloge`* 源中的 *parse / compile* 问题——比如缺失 *operator*、无效绑定、*schema* 错误或语法错误。）
5. 为什么 `GraphTestRunner` 比” 只看最终输出” 更有用？
   （因为它让你断言执行顺序、跳过行为和逐节点结果，而不仅是终点值。）

18.14 实验

目标：在你信任一张小型订单 graph 之前，先为它建立测试护栏。

1. 用 `GraphBuilder` 构建一张 `validate → price → confirm` 的 graph。
2. 三个 operator 都使用 `MockOperator`。

3. 使用 `GraphTestRunner` 执行 graph，并断言：
   - 三个节点都执行了；
   - 没有节点被跳过；
   - 执行顺序是 `validate`、`price`、`confirm`。
4. 再次执行同一张 graph，并对返回的 `GraphResult` 断言：
   - `requireSuccess()` 通过；
   - `confirm` 返回预期订单号；
   - `price` 具有非空耗时。
5. 写一个会失败的 DSL 片段，并断言 `DslTestHelper` 报告了你预期的编译错误。

进阶：给 `price` 加上 fallback，强制主逻辑抛异常，并断言 graph 仍然是按你想要的原因成功的。

18.15 桥接：从 **graph** 断言到业务验证

`GraphResult`、`GraphTestRunner` 和 `DslTestHelper` 能说明 graph 是否按测试预期执行，却不会绑定业务负责人批准的 Scenario、用 Policy 保护必需 Cases、用 Fixture 控制声明的 effect，也不会发布 source-bound evidence。当问题从“这张 graph 是否运行”变为“这个业务故事是否仍满足批准的合同”时，继续阅读第 23 章——业务正确性不等于测试通过。

18.16 实验验收卡

- 预期与观察：绿色 graph test 只支持它声明的 graph contract。
- 失败与恢复：删除 node status 断言制造假绿；恢复 output、status 和 evidence 断言。
- 证明边界：证明受控运行事实，不证明外部系统或业务答案。
- 练习合同：一个绿色测试；只删一个断言；交付前后 claim 卡；每条结论能指回断言即停止。

18.17 回顾

- **`GraphResult`** 是断言工作流行为的第一入口。它还提供了 `suspendedNodes()`、`compensationResults()` 和 `nodeSchemas()`，分别用于 挂起/恢复、saga 补偿和 schema 契约断言。
- **`MockOperator`** 通过替换无关 I/O，让测试保持聚焦。
- **`GraphTestRunner`** 提供 graph 级断言：顺序、执行与跳过行为。
- **`DslTestHelper`** 让你以编译器的真实视角测试 `.bloge` 源文件。
- **`TestGraphEngine`**、`ManualTimeSource` 和 `DeterministicTimerService` 让时间相关测试既快速又可重复。
- 强健的 BLOGE 测试不是堆一个巨大的集成测试，而是分层建立安全网。

18.18 下一步

在 第 19 章——生产环境中的可观测性 中，你将从“验证声明的 graph 契约”切换到“理解 graph 在真实运行中正在发生什么”。

18.19 参考链接

- `bloge-test` README ——测试辅助总览
- `GraphResult` ——不可变执行结果 record
- `MockOperator` ——可复用测试替身
- `GraphTestRunner` ——graph 执行夹具与断言
- `DslTestHelper` ——DSL parse / compile 断言
- `GraphResultSafeAccessTest.java` ——安全访问示例
- 核心架构 ——与 graph 执行断言相关的引擎内部机制

章节
19

第五阶段——技术信心

第 19 章——生产环境中的可观测性

测试、可观测性、Spring 接线与容量边界

目标：让每个技术 *claim* 都匹配真正支持它的证据。

承诺：读完本章后，你将知道如何在生产环境中看见一张 *BLOGE graph* 正在做什么——通过 *listener*、*interceptor*、*metrics*、*tracing*、结构化日志、上下文传播与审计日志——而不需要把这些逻辑写进 *workflow* 本身。

19.1 学习目标

1. 解释两条主要的可观测性扩展点：负责生命周期事件的 `ExecutionListener`，以及负责包裹调用过程的 `OperatorInterceptor`。
2. 使用自定义 listener 捕获 graph / node 生命周期信号，并知道什么时候应该实现更细粒度的子接口。
3. 正确地把 `MetricsExecutionListener`、`TracingOperatorInterceptor` 和 `LoggingExecutionListener` 接到 `GraphEngine` 上。
4. 使用 `CallerContextCarrier` 把 MDC 或 OpenTelemetry 上下文传播到引擎的虚拟线程中。
5. 知道什么时候聚合指标就足够，什么时候必须使用 `AuditJournalListener` 提供的结构化逐事件视角。

19.2 前置条件

- 第 6 章——韧性设计 ——只有先理解 retry、timeout 和 fallback，这些事件信号才有意义。
- 第 7 章——设计良好的 Operator ——interceptor 包裹的是 operator 执行，所以你需要先理解 operator 契约。
- 第 18 章——测试你的 Graph ——本章直接建立在你在测试中已经接触过的断言面和 listener 机制之上。

19.3 源示例

文件	展示内容
`GraphEngineListenerTest.java`	生命周期回调与 listener 异常隔离
`ResilientOperatorWrapperObservabilityTest.java`	retry、timeout 与 fallback 事件
`MetricsExecutionListenerTest.java`	Micrometer 定时器与计数器
`TracingOperatorInterceptorTest.java`	graph span 与 node span
`AuditJournalListenerTest.java`	结构化审计条目与事件覆盖
`BlogeObservabilityAutoConfiguration.java`	Spring Boot 可观测性自动配置

19.4 为什么这很重要

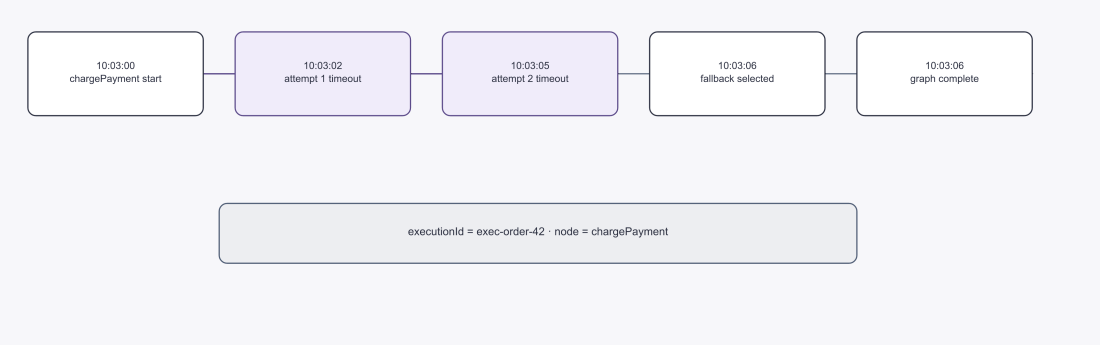
一张 graph 今天在你的机器上跑得正确，仍然可能在生产环境里悄悄出问题：

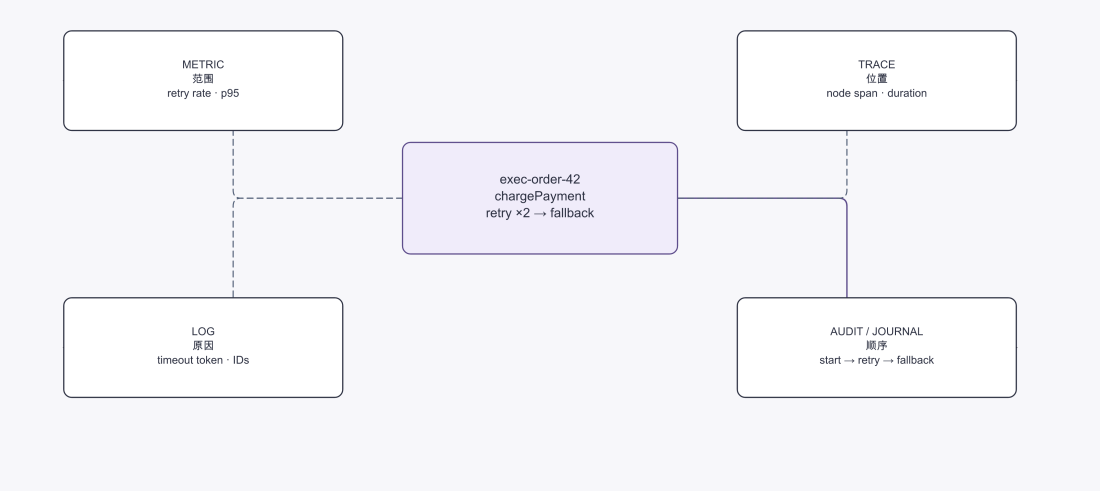
- 某个 **node** 在依赖升级后开始抛错。没有 metrics，你要等到用户投诉才知道。
- 某个 **retry** 回路每次请求都烧掉 **30** 秒。没有逐节点耗时数据，你根本定位不到瓶颈。
- 某条 分支改动让过去会执行的节点被跳过了。没有生命周期信号，系统看起来像是空闲，实际上却在做错误的选择。

BLOGE 将关注点明确分离：引擎 负责运行节点，扩展钩子 负责观察结果。正是这种分离，才让你能够在不把指标、trace 或日志代码塞进每个 operator 的前提下，获得生产可见性。

19.5 一次事故、一个 execution ID、四种投影

10:03，订单看板出现延迟尖峰。问题请求是 `executionId=exec-order-42`；它的 `chargePayment` 节点重试两次后进入 fallback。固定这一个身份，再切换问题：



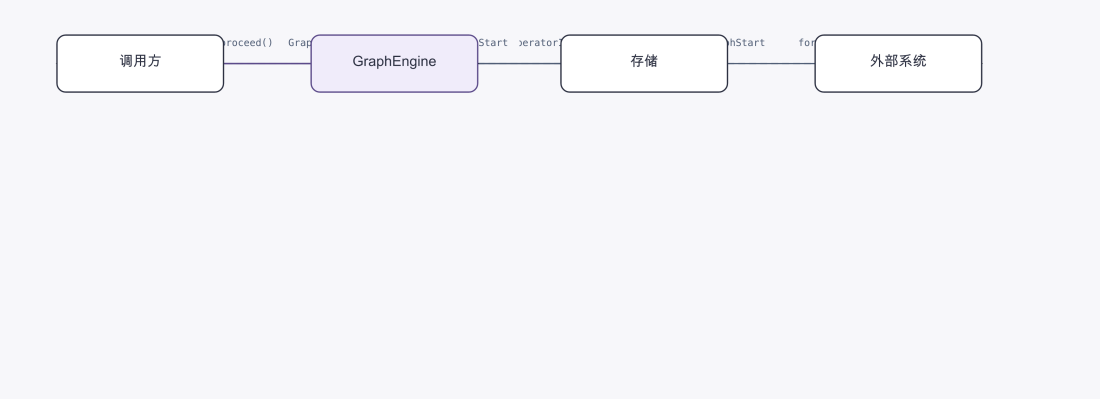


投影	最适合回答的问题	`exec-order-42` 的观察	单独不能证明什么
Metric	问题是否普遍?	retry 数和 p95 延迟上升	究竟哪次 attempt 失败
Trace	时间花在哪里?	`chargePayment` 子 span 占主要耗时	持久事件的完整顺序
结构化 log	runtime 报告了什么?	timeout token 及 execution/node ID	完整、有序的历史
Audit/event journal	事件按什么顺序发生?	start → retry → retry → fallback → complete	未聚合时的全局发生率

关联键是主脊，不是第五种可观测产品。Metric 触发调查；同一个 execution ID 缩小 trace 与 log 范围；journal 复原顺序。这个事故是根据源码测试覆盖的事件 类型构造的教学复盘，不是从生产事故中截取的记录。

19.6 心智模型

把引擎想象成一个有固定聚光灯位点的舞台。每一次节点执行都会经过同一条可观测链路：



一旦你知道自己需要哪个”聚光灯位点”，就知道应该把可观测性能力接到哪里。

19.7 第一个可运行的示例

一个最小但生产味道很足的接线方式，是在普通引擎外面挂上 listener 和 interceptor：

```
var meterRegistry = new SimpleMeterRegistry();
var metrics = new MetricsExecutionListener(meterRegistry, "bloge");
var tracing = new TracingOperatorInterceptor(tracer);

var engine = GraphEngine.builder()
        .registry(registry)
        .listeners(List.of(metrics, tracing))
        .interceptors(List.of(tracing))
        .build();
```

这一处 builder 接线，就给了你同一次执行的三种视角：

- graph / node 耗时指标，
- graph 与 node 的 tracing span，
- 驱动这两者的原始生命周期事件流。

关键不是具体用哪套库，而是契约：listener 观察事件，interceptor 包裹执行。

19.8 拆解分析

19.8.1 ExecutionListener ——事件骨干

`ExecutionListener` 是一个复合接口，扩展了七个细粒度子接口：

子接口	回调方法
`GraphLifecycleListener`	`onGraphStart`、`onGraphComplete`
`NodeLifecycleListener`	`onNodeStart`、`onNodeComplete`、`onNodeFailed`、`onNodeSkipped`、`onNodeSuspended`、`onNodeResumed`
`ResilienceListener`	`onNodeRetry`、`onNodeTimeout`、`onNodeFallback`
`StreamingListener`	`onNodeStreamStart`、`onNodeStreamChunk`、`onNodeStreamEnd`、`onNodeStreamError`
`IterationLifecycleListener`	循环/foreach 迭代事件
`TimerEventListener`	定时器触发事件
`EventCorrelationListener`	事件匹配事件

新的集成只需实现它所需的最窄子接口。已有代码可以继续实现聚合的 `ExecutionListener`。

> 安全保证：一个抛异常的 *listener* 不会破坏 *graph* 执行。引擎会捕获 *listener* 的异常，然后继续调用其余的 *listener*。这在 *`GraphEngineListenerTest.listener_exceptionInListener_doesNotBreakExecution`* 中有测试。

19.8.2 OperatorInterceptor ——AOP 层

`OperatorInterceptor` 包装每一次 operator 调用。拦截器接收一个 `OperatorInvocation`，并必须调用 `invocation.proceed()` 来继续链路（或通过直接返回值来短路）。

```
public interface OperatorInterceptor {
    Object intercept(OperatorInvocation invocation) throws Exception;
}
```

`TracingOperatorInterceptor` 就是这样为每个节点创建子 span 的——它在 `proceed()` 之前启动 span，在 `finally` 块中结束。

19.8.3 CallerContextCarrier ——跨虚拟线程的上下文传播

引擎在虚拟线程上运行每个节点。像 OpenTelemetry 上下文或 SLF4J MDC 这样的线程本地状态不会自动传播。`CallerContextCarrier` 通过三阶段契约弥补了这一空白：

1. **`capture()`** ——在调用者线程上调用，快照上下文。
2. **`Snapshot.attach()`** ——在虚拟线程上调用，恢复上下文。
3. **`Scope.close()`** ——在节点完成时调用，清理上下文。`Scope.NOOP` 可用于不需要清理的 carrier。

`bloge-metrics-otel` 中提供了两个实现：

- `OtelContextCarrier` ——传播 `io.opentelemetry.context.Context`
- `MdcContextCarrier` ——传播 SLF4J MDC map

19.8.4 生产可观测性组件

`bloge-metrics-otel` 模块提供了三个开箱即用的组件：

`MetricsExecutionListener` ——发射 Micrometer 定时器和计数器：

- `bloge.graph.duration` ——Timer，标签为 `graph` + `outcome`
- `bloge.node.duration` ——Timer，标签为 `graph` + `node` + `outcome`
- `bloge.node.errors` ——Counter，按异常类标记
- `bloge.node.retries`、`bloge.node.timeouts`、`bloge.node.fallbacks`、`bloge.node.skipped` ——Counter
- `bloge.stream.chunk.count`、`bloge.stream.duration`、`bloge.stream.errors` ——流式指标

所有指标名使用可配置的前缀（默认为 `bloge`）。

源码：`MetricsExecutionListener.java`

`TracingOperatorInterceptor` ——在 graph 启动时创建一个父 span（`bloge.graph.execute`），并在每次 operator 调用时创建一个子 span（`bloge.node.execute`）。它 同时 实现了 `OperatorInterceptor` 和 `ExecutionListener` ——将同一个实例注册到 `interceptors` 和 `listeners` 列表中。

源码：`TracingOperatorInterceptor.java`

`LoggingExecutionListener` ——结构化 SLF4J 日志，使用 `bloge.graph`、`bloge.node` 和 `bloge.executionId` MDC 键。两个可选标志——`includeInput` 和 `includeOutput` ——控制载荷是否出现在日志中。除非你已审查过载荷中没有敏感数据，否则保持两者禁用。

源码：`LoggingExecutionListener.java`

19.8.5 审计日志

对于合规级别的事件记录，bloge-runtime-spi 提供了 AuditJournalListener。它将 NODE_START、NODE_COMPLETE、NODE_FAILED、NODE_RETRY、NODE_SUSPEND 和 NODE_RESUME 事件捕获为结构化的 AuditEntry 记录，并写入 AuditJournalStore。异步刷新模式（虚拟线程 worker + 可配置批量大小）让热路径保持轻量。

AuditJournalListener 接受一个可选的 TimeSource 参数，使审计时间戳与引擎使用的时钟保持一致。在测试中你可以注入 ManualTimeSource 来获得确定性审计条目；在生产环境中默认使用 SystemTimeSource.INSTANCE。

```
// 生产环境 ——墙钟
var listener = new AuditJournalListener(store, config, jsonCodec);

// 测试 ——确定性时间
var listener = new AuditJournalListener(store, config, jsonCodec, manualTimeSource);
```

AuditConfig 可控制：captureInput、captureOutput、asyncFlush、flushInterval（默认 100ms）、batchSize（默认 64）。

19.8.6 ExecutionEventStore ——租户级查询与清理

ExecutionEventStore 接口暴露了几个对生产可观测性管线至关重要的方法：

方法	用途
loadEventsByTenant(tenantId, namespace, from, to, page, size)	按租户/命名空间和时间范围分页查询事件
countEventsByTenant(tenantId, namespace)	统计某个租户的事件总数——适用于仪表盘指标
purgeEventsBefore(cutoff)	删除早于截止时间点的事件
purgeEventsBefore(cutoff, limit)	批量感知的清理，避免长时间的删除事务
purgeExecutionEvents(executionId)	删除单次执行的所有事件（例如归档后）

所有新方法都提供了默认实现（抛出 UnsupportedOperationException），因此现有的 store 实现仍然可以编译。根据你的可观测性管线需求逐步升级即可。

19.9 按级别读取稳定日志标记

指标和 trace 只告诉你某个东西慢了。稳定 token 能定位具体 runtime 或 DSL fallback，又不会把周围散文变成合同。报警策略应依据实际级别与运行影响；FINE/FINER 只提供诊断线索，不应单独触发报警。

标记	级别	含义	第一步排查
[TIMER_RESTORE_FAILED]	SEVERE	TimerManager 无法重新加载 active durable timers；已持久化 timer 到下一次重启前不会触发。	检查附带 exception、durable timer context、store 连接与被加载记录。

标记	级别	含义	第一步排查
[STREAMING_ITEM_ERROR_SEND_FAILED]	WARNING	streaming foreach item 失败后，runtime 还因 output channel 拒绝发送而无法写出 error chunk。	检查原始 item failure、channel 关闭竞态和 item-failure policy。
[SCHEMA_NUMBER_PARSE_FALLBACK]	FINER	数字强制转换在 integer 或 long 解析失败后尝试更宽类型。	只用于 coercion 诊断；除非另有业务或错误率影响，不单独报警。
[ACCESSOR_METHOD_LOOKUP_MISS]	FINER	直接方法 accessor 不存在，DSL path access 正尝试 JavaBean getter fallback。	若后续 path resolution 失败，再核对 property/getter 合同。
[SCHEMA_ENRICH_SKIPPED]	FINE	DSL schema enrichment 查询 Operator metadata 失败；编译会在 enrichment 减少的情况下继续。	diagnostics 丢失 schema 细节时，检查附带 cause 与 registry metadata。
[FINGERPRINT_SKIPPED]	FINE	DSL compiler 无法计算 Operator fingerprint，并在缺少 fingerprint 时继续。	在依赖恢复期 mismatch 检测前，检查 Operator 注册和 fingerprint 支持。

这些标记是稳定的，周围的文本不是。匹配方括号里的 *token*，不要匹配 散文。

19.10 执行事件日志 (Execution Event Journal)

EventJournalListener 是一个内置的 ExecutionListener，把每一个引擎 级事件追加到一个只追加日志里。区别于审计日志 (记录用于持久化的状态变更) 和 operator 拦截器链 (只包单次节点调用)，事件日志是你这次执行的 可回放记录。

```
ExecutionEventJournal journal = new InMemoryExecutionEventJournal();

GraphEngine engine = GraphEngine.builder()
    .registry(registry)
    .listeners(List.of(new EventJournalListener(journal)))
    .build();
```

每条记录是一个 ExecutionEvent，带 execution id、单调序号、时间戳，以及 以下事件类型之一:

类型	触发时机
GRAPH_STARTED / GRAPH_COMPLETED / GRAPH_FAILED	引擎进入/离开一张图。
NODE_STARTED / NODE_COMPLETED / NODE_FAILED / NODE_SKIPPED	单节点生命周期。
FOREACH_BATCH_STARTED / FOREACH_BATCH_COMPLETED	foreach 的每一批。
STREAM_ITEM_EMITTED / STREAM_COMPLETED	流式 source。
WAIT_FOR_TIMER_ARMED / WAIT_FOR_TIMER_FIRED	持久化 timer 生命周期。
LEASE_ACQUIRED / LEASE_REFRESHED / LEASE_LOST	持久化租约生命周期。

19.10.1 Agent 事件——AgentEventJournalBridge

用上 bloge-agent-ext 模块时，AgentEventJournalBridge 把 agent 内部的 生命周期事件桥接到同一个日志里。这就给你一条统一的” 图轮次 + agent 轮次 + 工具调用” 回放时间线:

```
GraphEngine engine = GraphEngine.builder()
    .registry(registry)
    .listeners(List.of(
        new EventJournalListener(journal),
        new AgentEventJournalBridge(journal)))   // 把 agent 事件也接入
    .build();
```

bridge 增加的 agent 事件类型:

类型	含义
AGENT_TURN_STARTED / AGENT_TURN_COMPLETED	agent 推理循环的一次迭代。
AGENT_LLM_CALLED	发了一次 LLM 请求 (带 token 用量和模型 id)。
AGENT_TOOL_INVOKED / AGENT_TOOL_COMPLETED / AGENT_TOOL_FAILED	工具调用生命周期。
AGENT_MEMORY_TRIMMED	某个 memory 策略丢弃或总结了较早的轮次。
AGENT_STREAM_CHUNK_EMITTED	streaming agent 输出一个 chunk。

常见做法是: 短期调试用内存 journal，长期回放挂一个 JdbcExecutionEventJournal 落到数据仓库。

19.11 常见陷阱

19.11.1 [X] 只将 TracingOperatorInterceptor 注册为 listener

```
var tracing = new TracingOperatorInterceptor(tracer);

var engine = GraphEngine.builder()
    .registry(registry)
    .listeners(List.of(tracing))       // graph span [OK]
    // missing .interceptors(List.of(tracing))  — no node spans! [X]
    .build();
```

TracingOperatorInterceptor 有两个身份。ExecutionListener 这一侧负责在 onGraphStart / onGraphComplete 中创建 **graph** 级别的 span；OperatorInterceptor 这一侧负责在 intercept() 中创建 **node** 级别的子 span。若你只把它注册为 listener，你只会看到一条平坦的 graph span，而看不到逐节点的拆解。

修复方式：将同一个实例同时注册到两个列表中。

19.12 引导式重写

打开 `MetricsExecutionListenerTest.java`，按下面三种方式改编它：

1. 把指标前缀改为 `myapp.bloge`，并验证生成的 timer 名称确实跟着变化。
2. 增加一条 fallback 路径，并断言 fallback 计数器正好递增一次。
3. 注册一个内联 listener 来记录事件字符串，并把它看到的顺序与同一次执行发出的 metrics 对照起来。

这个练习的价值在于：graph 不变，但你不断切换观察它的镜头。

19.13 思维检查

1. `ExecutionListener` 和 `OperatorInterceptor` 的区别是什么？
   （*listener* 响应生命周期事件；*interceptor* 直接包裹 *operator* 调用，并可在调用前后增加行为。）
2. 为什么 `TracingOperatorInterceptor` 必须注册在两个地方？
   （它的 *listener* 一侧管理 *graph* 级 *span*；它的 *interceptor* 一侧创建 *node* 级 *span*。）
3. `CallerContextCarrier` 解决了什么问题？
   （把调用者线程中的 *MDC / OpenTelemetry* 等上下文传播到引擎的虚拟线程里。）
4. 什么情况下应该选择 `AuditJournalListener` 而不是普通 metrics？
   （当你需要结构化、逐事件、可审计的记录，而不是聚合计数器或定时器时。）
5. 如果 listener 抛出异常，引擎提供什么保证？
   （*listener* 失败会被隔离；执行继续，其它 *listener* 仍然会运行。）

19.14 实验

目标：用三种互补方式观察同一张 graph。

1. 构建一张包含一个成功节点和一个不稳定节点的小 graph。
2. 注册 `MetricsExecutionListener`，并验证出现了 retry 计数器。
3. 在两个位置都注册 `TracingOperatorInterceptor`，并验证 graph 与 node span 都被创建。
4. 使用内存版 store 注册 `AuditJournalListener`，并断言存储了预期的 `NODE_START`、`NODE_RETRY` 和 `NODE_COMPLETE` 事件。
5. 通过 `MdcContextCarrier` 打开 MDC 传播，并验证节点日志能看到调用方的关联 ID。

进阶：同一张 graph 分别在开启与关闭可观测性的情况下运行一次，然后比较每一种观察手段最擅长回答什么问题。

19.15 实验验收卡

- 预期与观察：同一 executionId 关联 metric、trace、log 和 audit。
- 失败与恢复：移除 correlation 字段；恢复统一 identity 后重建时间线。
- 证明边界：证明投影可关联，不证明指标解释业务原因。
- 练习合同：一次 retry→fallback 事故；只改 correlation；交付四投影时间线；全部回到同一执行即停止。

19.16 回顾

- **listener** 观察生命周期事件；**interceptor** 包裹 operator 调用。
- **metrics**、**tracing**、**logging** 与 **audit journaling** 都建立在同一组引擎扩展点之上。
- **`CallerContextCarrier`** 是调用方上下文与虚拟线程执行之间的关键桥梁。
- 最稳妥的可观测性集成是可复用的：它增加可见性，但不污染 graph 逻辑。

19.17 下一步

在第 20 章——Spring 与生产环境接线中，你会把运行时可见性接进应用：条件化 bean、配置边界与面向运维的 endpoint。

19.18 参考链接

- `ExecutionListener` ——聚合 listener 接口
- `OperatorInterceptor` ——AOP 拦截契约
- `CallerContextCarrier` ——capture / attach / close 契约
- `MetricsExecutionListener` ——Micrometer listener
- `TracingOperatorInterceptor` ——graph 与 node span 接线
- `LoggingExecutionListener` ——结构化日志 listener
- `AuditJournalListener` ——结构化审计捕获
- `ExecutionEventStore` ——事件日志存储 SPI，支持租户级查询
- `BlogeObservabilityAutoConfiguration` ——Spring Boot 自动配置
- 核心架构 ——引擎内部机制与 SPI 概览

章节

20

第五阶段——技术信心

第 20 章——Spring 与生产环境接线

测试、可观测性、Spring 接线与容量边界

目标：让每个技术 *claim* 都匹配真正支持它的证据。

承诺：读完本章后，你能够通过 *Spring Boot* 启动一张 *BLOGE graph*，不手写 *engine bean*；能够读懂 *health* 结果；也能只用一次 `GraphEngineCustomizer` 覆盖完成必要调整，而不接管整棵自动配置图。

20.1 学习目标

1. 添加 `bloge-spring` 依赖，让 `BlogeAutoConfiguration` 自动接线 `GraphEngine`、`OperatorRegistry`、`GraphLoader` 和编译后的 `List<Graph>`，无需手动定义任何 bean。
2. 使用 `@BlogeOperator` 标注实现 `Operator`、`SuspendableOperator` 或 `StreamingOperator` 的运行时对象，让 Spring 发现并注册它们。
3. 将 `GraphEngine` 和编译后的 graph 列表注入到一个 service 类中，执行 graph 并通过 HTTP 端点返回结果。
4. 把 `UP + graphCount=1` 读成接线观察，而不是业务证明。
5. 使用 `GraphEngineCustomizer` 调整轻量引擎，并判断何时替换整个 bean 会制造不必要的责任。

20.2 前置条件

- 第 7 章——设计良好的 Operator ——在将 operator 接入 Spring 之前，你需要掌握 operator 的设计模型。
- 第 13 章——持久执行 ——运行时存储、检查点和挂起/恢复，因为 starter 会有条件地接线所有这些组件。
- 熟悉 Spring Boot 自动配置、`@ConfigurationProperties` 和 Spring Boot Actuator 基础。

20.3 源示例

文件	展示内容
`BlogeAutoConfiguration.java`	导入 core slice 的入口自动配置
`BlogeProperties.java`	所有 `spring.bloge.*` 配置属性及嵌套分组
`BlogeOperator.java`	由 `@Component` 组合而成的 operator 扫描注解
`BlogeObservabilityAutoConfiguration.java`	指标、链路追踪、MDC、健康指示器接线
`BlogeEndpointAutoConfiguration.java`	`/actuator/bloge` 和 `/actuator/blogeStats` 端点
`BlogeAuditAutoConfiguration.java`	审计日志监听器激活
`SessionExecutorAutoConfiguration.java`	`SessionExecutor` 接线，包含访问守卫和监听器
`spring-ticket-triage.bloge`	starter 示例加载的最小 DSL
`SpringBootTicketTriageApplication.java`	端到端的 Spring Boot 应用，使用 starter
`SpringTicketTriageService.java`	注入 `GraphEngine` 和 `List<Graph>` 的 Service 层

20.4 为什么这很重要

在前面的章节中，你通过手动方式构建引擎：

```
var engine = GraphEngine.builder()
        .registry(registry)
        .executionCheckpointStore(checkpointStore)
        .listeners(listeners)
        .build();
```

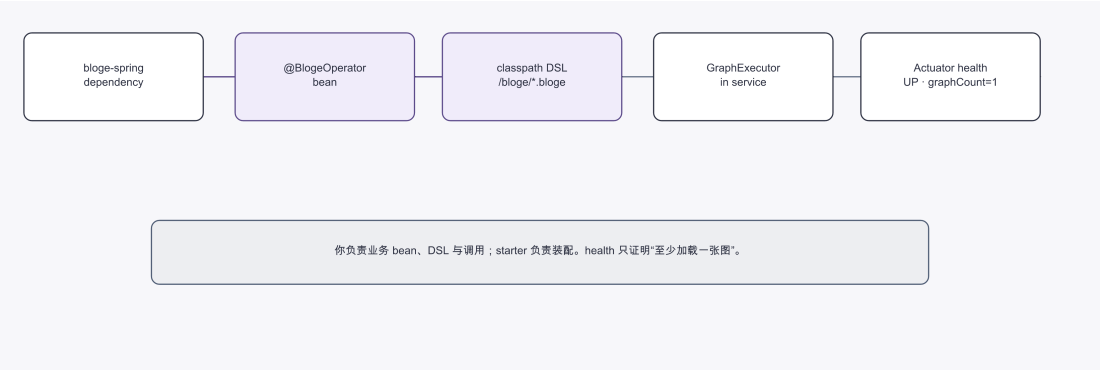
这在测试和学习时没问题。但在生产环境中，需要接线的列表会迅速膨胀：拦截器、监听器、上下文载体、持久化存储、检查点编解码器、恢复配置、分片解析器、定时器服务、事件去重——而且每个 bean 都有自己的条件和回退逻辑。

`bloge-spring` 模块处理这部分组装工作。它是一个 Spring Boot Starter，会自动配置 classpath 与属性条件已经满足的组件。把 operator 声明为 Spring bean，并将 `.bloge` 文件放到 classpath 后，starter 可以为这个应用切片组装一套可运行引擎。加入 `bloge-durable` 或 `bloge-metrics-otel` 后，starter 会检测并接线其中满足条件的 bean，不需要重复声明 `@Bean`。这只证明应用接线已经建立，不证明持久化 store、外部 effect 或生产容量已经就绪。

> 核心洞察：*starter* 并没有隐藏引擎——它只是组装引擎。多数可替换的 *singleton* 默认值使用 `@ConditionalOnMissingBean`；扩展集合和 *endpoint slice* 有自己的条件，因此覆盖前仍要确认目标 *bean* 的具体合同。

20.5 最短接线路径

不要从所有可选生产能力开始。先打通一个纵向切片：一个依赖、一个带注解的 operator、一份 classpath DSL、一个注入的 executor，以及一次 health 观察。

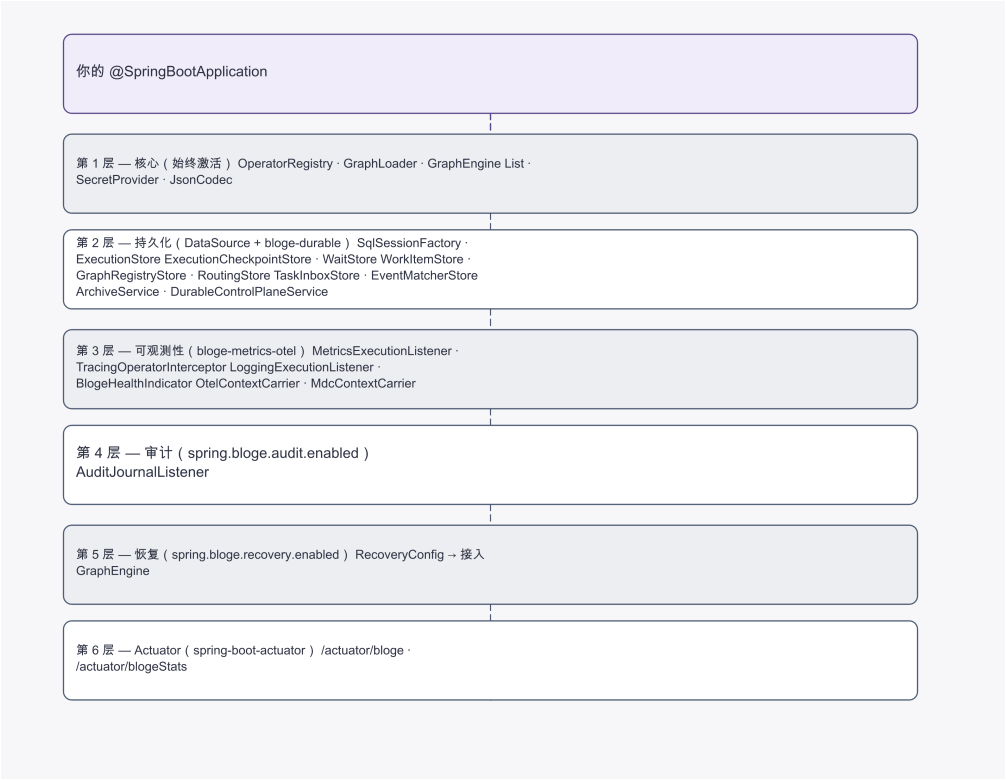


你负责	**Starter** 负责	可观察检查
实现运行时 operator 合同的 `@BlogeOperator` bean	`OperatorRegistry`	DSL 加载时能解析 operator 名
`classpath:/bloge/*.bloge`	`GraphLoader` 与编译后的 `List<Graph>`	目标 graph 存在
service 输入与输出	轻量 `GraphEngine` / `GraphExecutor`	一次执行返回预期节点事实
部署依赖	Actuator 存在时的 `BlogeHealthIndicator`	`UP`、`graphCount=1`

这里的 `UP` 只表示应用至少加载了一张 graph。它不表示订单 Scenario 通过、外部支付系统健康，也不表示该版本具备发布资格。

20.6 心智模型

把 `bloge-spring` 想象成一条 分层装配线。每一层只有在其 classpath 和属性前置条件都满足时才会激活：



第 2–6 层是叠加且独立的。你不需要一次全部启用，移除某个 classpath 依赖就能干净地移除对应层。

20.7 第一个可运行示例

这是来自 `bloge-examples/src/main/java/com/leanowtech/bloge/examples/integration/spring/` 的工单分流示例。

20.7.1 步骤 1 —添加依赖

```
<dependency>
    <groupId>com.leanowtech.bloge</groupId>
    <artifactId>bloge-spring</artifactId>
    <version>${bloge.version}</version>
</dependency>
```

20.7.2 步骤 2 —把 operator 编写为 Spring bean

下面的示例对 `SpringTicketClassifierOperator.java` 做了简化：

```
@BlogeOperator(
        value = "SpringTicketClassifierOperator",
        description = "Classifies a support ticket",
        owner = "examples",
        tags = {"spring", "starter", "triage"}
)
```

```
public class SpringTicketClassifierOperator implements
        Operator<Map<String, Object>, ClassifiedTicket> {

    public record ClassifiedTicket(String queue, int priorityScore, boolean vip) {}

    @Override
    public ClassifiedTicket execute(Map<String, Object> input, OperatorContext ctx) {
        boolean vip = "vip".equalsIgnoreCase(String.valueOf(input.get("customerTier")));
        String queue = vip ? "vip-escalation" : "general-support";
        int priorityScore = vip ? 70 : 30;
        return new ClassifiedTicket(queue, priorityScore, vip);
    }
}
```

`@BlogeOperator` 由 `@Component` 组合而成，因此 Spring 会发现这个 bean；registry 只会接收其中实现 `Operator`、`SuspendableOperator` 或 `StreamingOperator` 的对象。`value` 设置 DSL 中的 operator 名。仓库里的完整 示例刻意把 domain bean 保持为普通对象，再通过显式 `OperatorRegistry` 适配；上面的精简版选择的是零手写 registry 路径。

20.7.3 步骤 3 —把 DSL 放到 classpath 上

来自 `spring-ticket-triage.bloge`：

```
graph springTicketTriage {
  node classifyTicket : SpringTicketClassifierOperator {
    input {
      ticketId = ctx.ticketId
      message = ctx.message
      customerTier = ctx.customerTier
    }
    timeout = 2s
  }

  node draftReply : SpringReplyDraftOperator {
    depends_on = [classifyTicket]
    input {
      ticketId = ctx.ticketId
      queue = classifyTicket.output.queue
      priorityScore = classifyTicket.output.priorityScore
      vip = classifyTicket.output.vip
    }
    timeout = 2s
  }
}
```

starter 的默认 DSL 位置是 `classpath:bloge/`。将此文件放在 `src/main/resources/bloge/spring-ticket-`

triage.bloge，它就会被自动加载。

20.7.4 步骤 4 —注入并执行

来自 SpringTicketTriageService.java：

```
@Service
public class SpringTicketTriageService {

    private final GraphEngine engine;
    private final Map<String, Graph> graphsByName;

    public SpringTicketTriageService(GraphEngine engine, List<Graph> graphs) {
        this.engine = engine;
        this.graphsByName = graphs.stream()
            .collect(Collectors.toUnmodifiableMap(Graph::name, Function.identity()));
    }

    public TicketTriageResponse triage(String ticketId, String message, String customerTier) {
        Graph graph = graphsByName.get("springTicketTriage");
        var result = engine.execute(graph, new GraphContext(Map.of(
                "ticketId", ticketId,
                "message", message,
                "customerTier", customerTier
        )));
        return /* 将 result 映射为 response */;
    }
}
```

没有 GraphEngine.builder()。没有 DefaultOperatorRegistry。没有 GraphLoader 调用。starter 已经把这一切全部组装好了。

20.7.5 步骤 5 —通过 HTTP 暴露

来自 SpringTicketTriageController.java：

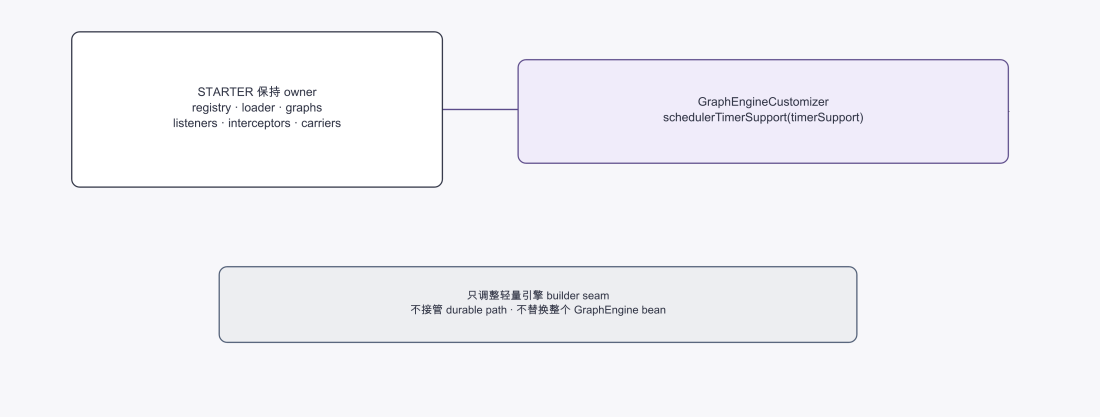
```
@RestController
@RequestMapping("/api/bloge/tickets")
public class SpringTicketTriageController {

    private final SpringTicketTriageService triageService;

    public SpringTicketTriageController(SpringTicketTriageService triageService) {
        this.triageService = triageService;
    }
```

```
    @GetMapping("/triage")
    public TicketTriageResponse triage(
            @RequestParam("ticketId") String ticketId,
            @RequestParam("message") String message,
            @RequestParam(name = "customerTier", defaultValue = "standard") String customerTier) {
        return triageService.triage(ticketId, message, customerTier);
    }
}
```

20.8 一次 override，只增加一个明确 owner

假设默认轻量引擎都合适，只有 scheduler 集成需要调整。保留 starter 管理的 其它 bean，只使用 builder 接缝：

```
@Bean
GraphEngineCustomizer schedulerOverride(MyTimerSupport timerSupport) {
    return builder -> builder.schedulerTimerSupport(timerSupport);
}
```



Customizer 在轻量 engine-mode preset 之后、核心 `GraphEngine` 构建之前运行；它不配置 durable runtime 路径。如果替换整个 `GraphEngine` bean，你也会 同时接管 listener、interceptor、context carrier 和未来 starter 默认值。

本章余下部分作为生产参考。首次阅读只需完成最短切片、health 边界和这一次 override，就已经形成完整闭环。

20.9 生产参考

上面的最短路径已经足以理解所有权。需要查询完整 bean 清单、属性分组、持久化 store 接线、健康检查或 HTTP 错误契约时，请转到附录 I——Spring 生产参考。

20.10 常见陷阱

20.10.1 [X] 在 `@BlogeOperator` 已存在时仍手动注册 operator

```
// 如果你同时也用 @BlogeOperator 标注了这个类，就不要这么做：
@Bean
public OperatorRegistry operatorRegistry() {
    var registry = new DefaultOperatorRegistry();
    registry.register("FetchUserOperator", new FetchUserOperator(userService));
    return registry;
}
```

因为 `BlogeAutoConfiguration` 提供了一个会扫描 `@BlogeOperator` 标注 bean 的 `OperatorRegistry` bean，而且它的 bean 带有 `@ConditionalOnMissingBean`，声明你自己的 `OperatorRegistry` bean 会完全替换掉自动配置的那个。你的自定义 registry 不会扫描 `@BlogeOperator` bean ——因此所有标注的 operator 会悄然消失。

修复：要么完全依赖 `@BlogeOperator` 扫描（标准路径），要么提供一个包含所有需要内容的完整 `OperatorRegistry` bean。不要混用两种方式。

20.11 常见故障

20.11.1 Operator 悄然消失

如果一个类已经标注 `@BlogeOperator("MyOperator")`，同时又声明了空的自定义 `OperatorRegistry` bean，`@ConditionalOnMissingBean` 会让自动扫描 registry 退出。DSL 随后会以 `unregistered operator 'MyOperator'` 停止加载。

修复边界只有两个：删除自定义 registry，完全交给自动扫描；或者让自定义 registry 显式注册应用需要的全部 operator。不能把两种 owner 混在一起。

20.12 每次只增加一个生产扩展

只有明确的运维需求出现时，才增加可观测性、审计、恢复或 interceptor。附录 I 提供配置地图与部署检查；本章只保留一条设计规则：每个 override 都必须只有一个理由和一个 owner。

20.13 脑力检查

1. 自动注册一个 operator 需要同时满足什么？（答案：*bean* 带有 *`@BlogeOperator`*，并实现 *`Operator`*、*`SuspendableOperator`* 或 *`StreamingOperator`*；单有注解只保证 *Spring* 发现。）
2. starter 默认在哪里查找 `.bloge` 文件？哪个属性可以覆盖它？（答案：*`classpath:bloge/`*，可通过 *`spring.bloge.dsl-locations`* 配置。）
3. 如果你声明了自己的 `OperatorRegistry` bean，自动配置的那个会怎样？（答案：它会被跳过——*`@ConditionalOnMissingBean`* 意味着你的 *bean* 优先，而自动 *`@BlogeOperator`* 扫描将不再执行。）
4. 列举两个仅通过添加 classpath 依赖就能激活、无需修改任何属性的生产特性。（答案：当 *`spring-boot-actuator`* 存在时，*Actuator* 端点自动激活；当 *`bloge-metrics-otel`* 存在时，可观测性指标和链路追踪自动激活。）

5. 当没有找到任何 `.bloge` 文件时，`BlogeHealthIndicator` 报告什么？（答案：*`DOWN`*，附带详情 `"reason": "No graphs loaded"`。）

20.14 实验

从一个全新的 Spring Boot 应用开始：

1. 添加 **`bloge-spring`** 到你的 POM，创建一个用 `@BlogeOperator` 标注、并实现 `Operator` 的对象，返回一个问候字符串。
2. 编写一个单节点 **`.bloge graph`**，放在 `src/main/resources/bloge/` 下，引用你的 operator。
3. 创建一个 **`@Service`**，注入 `GraphEngine` 和 `List<Graph>`，按名称查找你的 graph 并执行它。
4. 验证健康指示器：启动应用并访问 `/actuator/health`。确认 `bloge` 组件显示 `UP` 以及你的 graph 名称。
5. 故意破坏它：重命名 `.bloge` 文件使其无法加载任何 graph。重启并验证健康指示器报告 `DOWN`。
6. 添加一个自定义 **`ExecutionListener`** bean，记录 graph 级别的完成耗时。再次执行你的 graph，确认该监听器被触发，且无需对引擎 bean 做任何修改。

20.15 实验验收卡

- 预期与观察： starter 装配 engine、loader 和 health，一个 customizer 只有一个 owner。
- 失败与恢复：移除 bean 或重复 override；恢复最短自动配置。
- 证明边界：证明 Spring 接线和 health，不证明生产容量或业务正确性。
- 练习合同：最小应用；只改一个 bean/customizer；交付 context 和 health；启动成功且 owner 唯一即停止。

20.16 回顾

- `bloge-spring` 是 Spring Boot Starter；只有 classpath、property 与 bean 条件满足时，它才会自动配置 `GraphEngine`、`OperatorRegistry`、`GraphLoader` 和编译后的 graph。
- `@BlogeOperator` 是由 `@Component` 组合而成的注解。标注的 bean 只有实现 BLOGE 运行时 operator 合同后才会自动注册；普通 domain bean 需要 adapter/customizer 路径。其 `promptHint`、`usageExample`、`constraints` `Description` 属性支持 agent/LLM 集成。
- starter 使用 分层的条件化自动配置：核心 bean 只在 starter 条件生效的 context 中存在；持久化存储在 `bloge-durable` + `DataSource` 存在时激活；可观测性在 `bloge-metrics-otel` 存在时激活；审计和恢复在各自的属性设置为 `true` 时激活。
- 自动配置被拆分为聚焦的切片（`BlogeCoreAutoConfiguration`、`BlogeGraphEngineAutoConfiguration`、`BlogeDurableAutoConfiguration` 等），实现了更清晰的关注点分离。
- 新增 `spring.bloge.engine-mode` 属性用于选择引擎预设（`AUTO` 或 `REQUEST_RESPONSE`）。
- 新增 `spring.bloge.tenant.*` 属性启用自动配置的多租户支持——租户解析、命名空间解析、并发上限和速率限制——无需编写自定义 `TenantContextResolver` bean。
- 新增 `spring.bloge.version-routing.*` 属性在 starter 层面支持 `latest`、`pinned` 和 `canary` graph 版本路由。
- `GraphEngineCustomizer` 让你无需替换整个 bean 即可微调引擎 builder。

- 多数可替换的 singleton 默认值带有 `@ConditionalOnMissingBean`；覆盖前应检查具体 slice。
- 自定义的 `OperatorInterceptor`、`ExecutionListener` 和 `CallerContextCarrier` bean 通过 `ObjectProvider` 自动收集并注入引擎。
- Actuator 端点（`/actuator/bloge`、`/actuator/blogeStats`）提供已加载 graph 和执行指标的运行时可见性。
- `BlogeHealthIndicator` 开箱即用地提供 `/actuator/health` 集成。

20.17 下一步

在第 21 章——单 JVM 里的调度与复杂度中，你会先隔离本地调度机制：ready set、完成事件与 graph 复杂度护栏。分布式所有权和容量留到第 22 章。

20.18 参考链接

- `bloge-spring` 模块 README ——完整属性表和快速入门指南
- `BlogeAutoConfiguration.java` ——核心自动配置源码
- `BlogeObservabilityAutoConfiguration.java` ——指标、链路追踪、健康指示器
- `BlogeEndpointAutoConfiguration.java` ——Actuator 端点
- `BlogeAuditAutoConfiguration.java` ——审计日志
- `SpringBootTicketTriageApplication.java` ——完整的 Spring Boot 示例
- `AutoConfiguration.imports` ——starter 注册文件
- 入门指南 ——环境搭建指南
- 核心架构 ——引擎内部机制参考

章节
21

第五阶段——技术信心

第 21 章——单 JVM 里的调度与复杂度

测试、可观测性、Spring 接线与容量边界

目标：让每个技术 *claim* 都匹配真正支持它的证据。

本章承诺：你会用一笔订单的四节点 *graph* 看清 *BLOGE* 如何找到就绪节点、并发执行、等待汇合，并能解释复杂度护栏保护的究竟是什么。

21.1 学习目标

学完本章，你能够：

1. 画出一轮完成驱动调度中 ready set 的变化。
2. 区分 graph 间并发、graph 内并发和共享资源容量。
3. 解释 `DependencyScheduler` 为什么按出边更新下游，而不是反复扫描整个 graph。
4. 用节点数、深度、扇出和分支嵌套判断复杂度风险。
5. 写出一次能复核的单机测量说明，而不是只给一个 QPS 数字。

21.2 这一章只增加一个变量

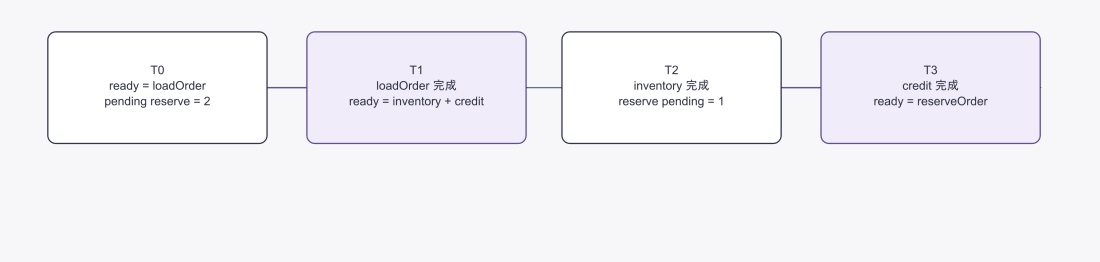
前面已经知道订单 graph 会按依赖关系执行。本章把问题缩小到一台机器、一个 JVM、一个 `GraphEngine`：当多个节点同时具备运行条件时，引擎如何推进？

租约、分片、远端 worker 和数据库容量全部留到下一章。先把本地机制看清，否则分布式问题会把调度问题淹没。

21.3 开场：四个订单步骤，谁能先跑

订单处理有四个节点：

在 `loadOrder` 完成后，`checkInventory` 与 `checkCredit` 谁先结束并不确定；但 `reserveOrder` 必须等两者都结束。先写下你的预测：每一个时刻，ready set 中有哪些节点？



21.4 用 BLOGE 自己的测试观察调度合同

运行 RC1 的调度与复杂度聚焦测试：

```
cd submodule/bloge
mvn -pl bloge-core \
  -Dtest=SchedulerSequenceTest,GraphComplexityValidatorTest test
```

2026-09-14 在 commit `cc38fbe5` 上观察到：

```
GraphComplexityValidatorTest  Tests run: 18, Failures: 0
SchedulerSequenceTest         Tests run: 12, Failures: 0
Tests run: 30, Failures: 0, Errors: 0, Skipped: 0
```

其中 `fanInFromMany_waitsForAll` 让四个前驱通过 latch 同时到达，再断言 sink 才能运行；`multiDiamond_orderingConstraints` 则只断言必须成立的先后关系，不要求并发节点有固定顺序。

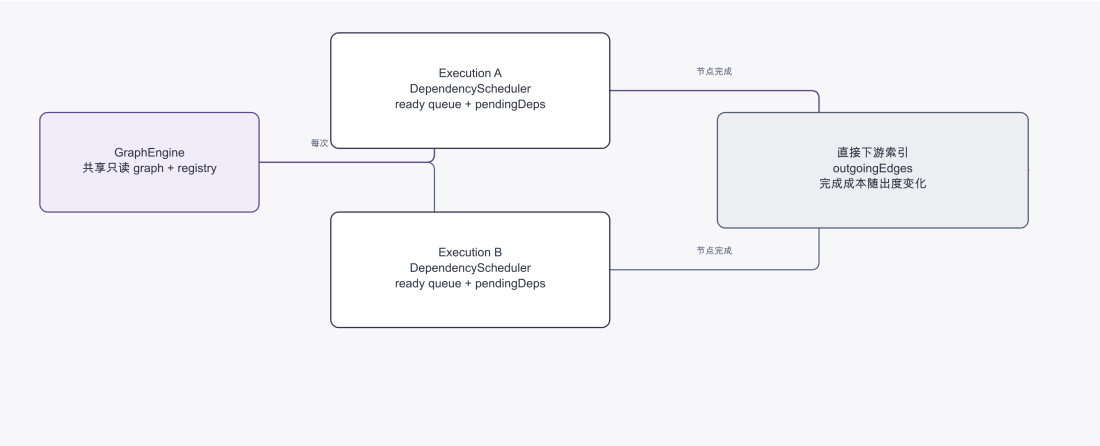
这正是调度合同：依赖顺序确定，同层完成顺序可以变化。

21.5 机制：不是定时扫描，而是完成事件推进

每次执行拥有自己的 `DependencyScheduler`。初始化时，它为每个节点记录未完成依赖数，并建立 `outgoingEdges` 索引：

```
pendingDeps(loadOrder)      = 0
pendingDeps(checkInventory) = 1
pendingDeps(checkCredit)    = 1
pendingDeps(reserveOrder)   = 2
```

第一轮只有 `loadOrder` 进入 ready queue。它完成后，调度器沿两条出边把两个计数减到零，于是两个检查节点进入 ready queue。任一检查完成只会把 `reserveOrder` 从 2 减到 1；第二个完成才把它减到零。



这种设计的关键不是“使用了虚拟线程”，而是推进成本与刚完成节点的出度相关。若每次完成都扫描全部边，大 graph 会反复支付无关检查成本。

21.6 三种并发不要混在一起

问题	所有者	本章能证明什么
多笔订单同时执行	宿主调用方 + engine	每次执行状态彼此隔离
一笔订单内并行检查	dependency scheduler	无依赖或依赖已满足的节点可并行
数据库/API 能否承受	外部资源与容量配置	本章不能证明

`GraphEngine` 可以被多个调用共享，不等于下游连接池无限。1000 个虚拟线程等待一个 20 连接的数据库，瓶颈仍是 20 个连接。

21.7 复杂度护栏：保护反馈速度，而不是限制想象力

RC1 在 graph 构建时检查四个维度。默认推荐/硬上限由 `ComplexityLimits.DEFAULT` 给出：

维度	推荐上限	硬上限	风险直觉
operator 节点数	15	30	状态和失败面扩大
DAG 深度	6	10	关键路径变长
单节点扇出	5	8	瞬时并发放大
分支嵌套	2	3	路径组合难以理解

超过推荐值产生 warning；超过硬上限拒绝 graph。聚焦测试实际覆盖了 31 节点、11 层深度和 9 路扇出的拒绝路径。

21.7.1 单因素破坏：把扇出从 **8** 改成 **9**

保持 operator、输入和其他边不变，只给 root 增加第九个 child。`GraphComplexityValidatorTest` 断言构建失败，消息包含 graph、root、hard limit 8 和拆分建议。

恢复方式有三种，按优先级判断：

1. 检查九个任务是否真属于一个 graph，能否抽成子图或批处理。
2. 检查是否把纯数据投影错误建成 operator 节点。
3. 业务确实需要时，显式调整 limits，并为该拓扑建立测量基线。

直接提高上限只能让 graph 合法，不能让它更快、更容易调试。

21.8 如何写一份可信的单机测量记录

一个 QPS 数字至少要带上：commit、JDK、CPU 配额、heap、graph 拓扑、operator 行为、并发度、warmup、measurement、fork 数和原始结果文件。缺少这些字段，数字只能是当时电脑上的传闻。

测量时逐层增加变量：

空 `operator` 调度成本
→ 固定拓扑并发
→ 真实 `operator`（仍用受控依赖）
→ `metrics / tracing` 开销

先找到哪一层改变了结果，再谈优化。不要把一次功能测试的 elapsed time 当 benchmark，也不要把 JMH 的平均值当生产容量承诺。

21.9 现实映射：餐厅出餐板

后厨不会每秒扫描所有订单，问每道菜能否开始。某道前置菜完成时，它只通知直接依赖它的工位。冷菜与热菜可能并行，摆盘必须等两边到齐。

- ready queue 是“现在可以做”的工位卡。
- `pendingDeps` 是还缺几道前置工序。
- `outgoingEdges` 是完成后要通知谁。
- 复杂度护栏是在一张单据过大时要求拆桌或拆批。

餐厅有很多厨师不代表烤箱容量无限，对应到系统里就是线程并发不等于外部资源容量。

21.10 轮到你：画出并测量自己的 **ready set**

选择一条 4—8 节点 graph：

1. 只画依赖边，标出初始 ready set。
2. 任选一种合法完成顺序，逐步更新 `pendingDeps`。
3. 只增加一条边或一个 child，预测并运行复杂度检查。
4. 建立最小 JMH 或重复测量记录，写全环境与拓扑。

产物是一张四帧 ready-set 图和一份测量卡。停止条件是别人能从卡片复现实验，并明确这个结果只代表固定单机环境。

21.11 本章证明了什么，没有证明什么

本章的 RC1 测试证明：被覆盖拓扑满足依赖顺序，fan-in 等待全部前驱，默认复杂度边界按预期拒绝过大 graph。它没有证明你的 operator 线程安全、外部 API 有容量、数据库不会成为瓶颈，也没有给出可复用的生产 QPS。

下一章只增加一个变量：第 22 章——分布式运行与容量验证会检查多 JVM 共享 durable 状态后的所有权、路由和容量瓶颈。

21.12 实验验收卡

- 预期与观察： ready set 与完成事件驱动单 JVM 调度，fan-out 护栏只提示反馈成本。
- 失败与恢复：把 fan-out 从 8 改成 9；恢复结构或记录接受理由。
- 证明边界：证明调度合同和复杂度信号，不证明通用 QPS。
- 练习合同：四节点图；只加一条 fan-out 边；交付时间线和测量卡；结论仅覆盖本机单轴实验即停止。

21.13 精确事实入口

- `CompletionDrivenScheduler.java`
- `DependencyScheduler.java`
- `SchedulerSequenceTest.java`
- `GraphComplexityValidator.java`
- `GraphComplexityValidatorTest.java`
- Durable 与 benchmark 参数参考

章节

22

第五阶段——技术信心

第 22 章——分布式运行与容量验证

测试、可观测性、Spring 接线与容量边界

目标：让每个技术 *claim* 都匹配真正支持它的证据。

本章承诺：你会把同一条订单 *graph* 放到两个共享 *durable store* 的 *worker* 上，解释租约为何需要 *fencing*、路由为何必须稳定，并产出一份只对明确负载和环境成立的容量判断。

22.1 学习目标

学完本章，你能够：

1. 解释 lease owner、timeout 和 epoch 如何完成一次安全接管。
2. 区分 execution 路由稳定性与当前 shard resolver 的新选择。
3. 找出扩容前真正饱和的资源，而不是默认增加 worker。
4. 设计 arrival rate、service time、并发和 backlog 的最小容量实验。
5. 写清一次测试证明的分布式合同及其外部边界。

22.2 这一章只增加一个变量

上一章把世界固定在一个 JVM。现在订单仍是同一条 graph，但 Worker A 与 Worker B 共享 durable store。新增的问题只有一个：执行的所有权不再由进程内存天然保证。

这会带来两种危险：A 已失联却再次写入；同一业务键因新路由决定漂到另一个 shard。

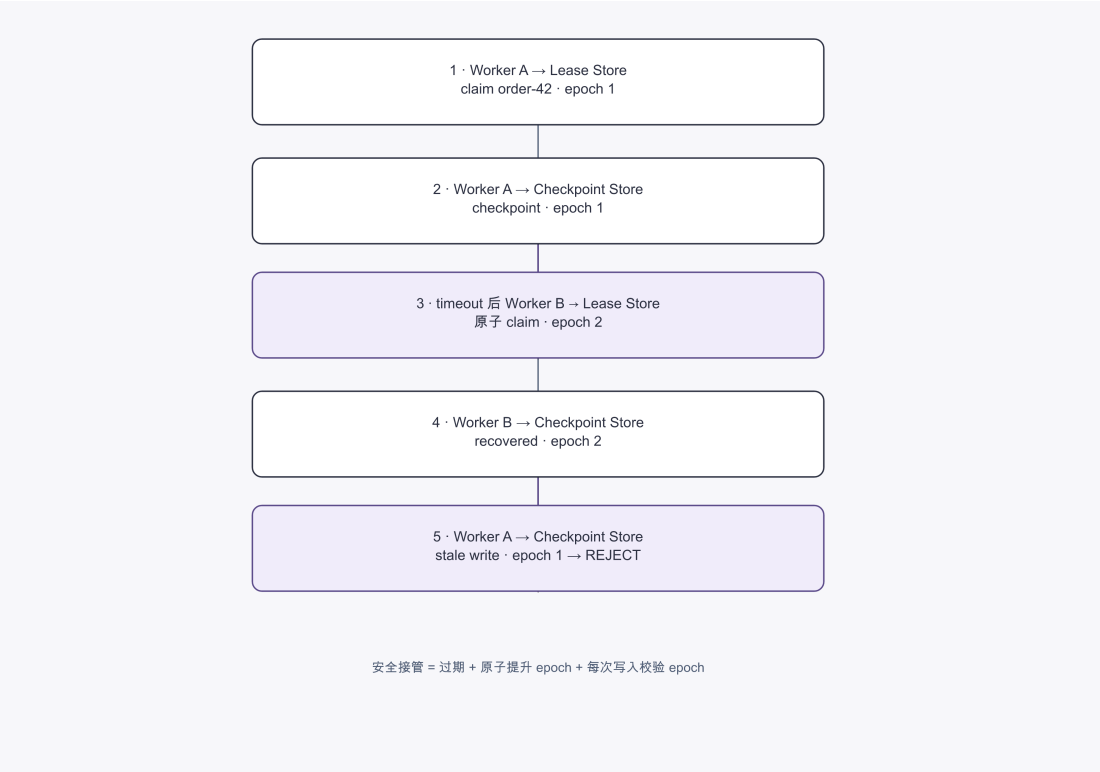
22.3 开场：A 超时后，B 能直接继续吗

先预测这段时间线是否安全：

```
10:00:00  Worker A claim order-42, epoch=1
10:00:20  A 写入 checkpoint
10:01:01  lease 过期；Worker B claim, epoch=2
```

`10:01:03` A 网络恢复，再写一次

如果 store 只检查“owner 名字”，A 可能覆盖 B 的新状态。安全接管不仅需要 timeout，还需要单调递增的 fencing epoch。



22.4 运行两项 RC1 分布式合同测试

```
cd submodule/bloge
mvn -pl bloge-durable \
  -Dtest=ClusterFailoverSimulationTest,RuntimeRoutingShardingTest test
```

2026-09-14 在 commit `cc38fbe5` 上观察到：

```
RuntimeRoutingShardingTest    Tests run: 2, Failures: 0
ClusterFailoverSimulationTest Tests run: 1, Failures: 0
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

测试使用逻辑时钟和内存 store，不依赖真实等待。它建立的是 deterministic contract，不是数据库压测。

22.5 机制一：接管是一条三条件链

`ClusterFailoverSimulationTest` 做了四件事：

1. A 认领 execution，得到 epoch 1，并在该上下文写 checkpoint。
2. 逻辑时钟越过一分钟 timeout。
3. B 的 recovery loop 重新认领，epoch 变为 2，写恢复标记并完成 execution。
4. A 携带旧 epoch 1 再写，store 抛出 `StaleFencingEpochException`。

安全接管需要三个条件同时成立：

`lease` 已过期
+ 新 `owner` 原子认领并提升 `epoch`
+ 每次受保护写入校验当前 `epoch`

只有前两个条件时，僵尸 worker 仍能污染状态；只有第三个条件而没有可接管扫描，执行会永远停住。

22.5.1 单因素破坏：取消 **stale epoch** 校验

不改变 timeout、checkpoint 或恢复逻辑，只让 store 接受 epoch 1 的最后一次写入。最终状态可能仍显示 `COMPLETED`，但恢复标记已被旧 owner 覆盖。这说明“最终状态一样”不足以证明接管正确。

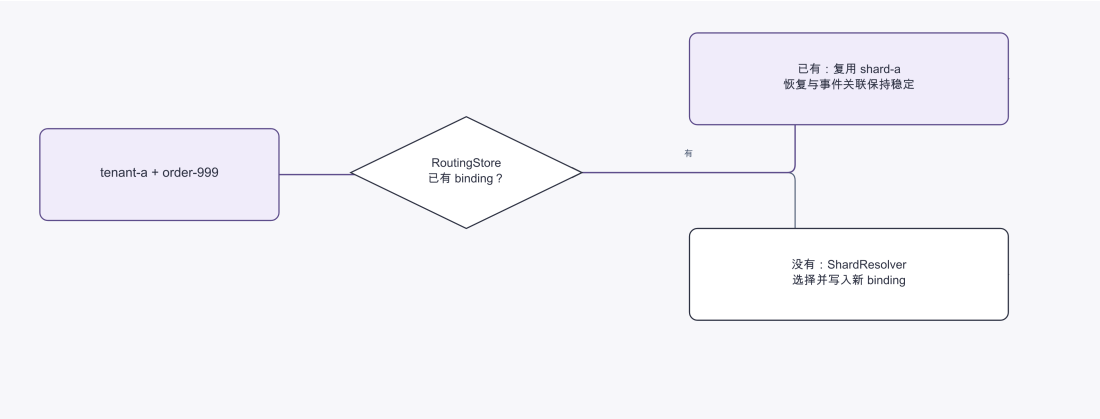
恢复动作是让所有改变受保护 execution 状态的 store 写入携带 lease context，并由持久化层 fail closed；不能靠应用日志事后猜冲突。

22.6 机制二：路由是业务键的稳定绑定

`RuntimeRoutingShardingTest` 首次执行时把：

`tenant-a + order-999 → shard-a`

写入 `RoutingStore`。第二个 engine 的 resolver 即使返回 `shard-b`，已有绑定仍胜出，第二次执行继续使用 `shard-a`。



resolver 是“没有绑定时去哪”的策略；routing store 是“这个业务对象已经在哪”的事实。若每次都重新 hash，扩容改变 shard 数量后，恢复和事件关联可能找不到旧状态。

迁移必须是一项显式控制面动作：冻结业务键、搬运状态、更新 binding、验证读写，再恢复流量。它不是普通 resolver 重新计算。

22.7 容量判断：先找瓶颈，再决定扩什么

把吞吐想成一条窄管道：

入口 `arrival rate`
→ `worker` 可运行并发
→ `external API` 配额
→ `connection pool`
→ `durable store` 写入与锁竞争

增加 worker 只扩大第二段。若 database checkpoint latency 已饱和，更多 worker 会增加排队和锁竞争。

用 Little's Law 建立第一个估算：稳定系统中 `in-flight ≈ arrival rate × average service time`。它是测量设计的起点，不是 BLOGE 的容量保证。

要观察的量	说明
arrival rate	每秒新 execution / work item
service time	从可运行到完成的分布，不只平均值
in-flight	当前占用 worker 或外部资源的数量
backlog age	最老待处理项等待多久
store latency	checkpoint、claim、poll 的 p50/p95/p99

22.8 一个可归因的扩容实验

固定 graph、输入分布、数据库、连接池和外部 stub，只把 worker 从 1 增加到 2、4、8。每一步记录吞吐、p95、backlog age 和 store latency。

- 若吞吐近似增加且 store latency 稳定，worker 可能是当前瓶颈。
- 若吞吐不变而 store latency 上升，store/锁竞争更可疑。
- 若 backlog 下降但外部 429 增加，你只是把压力推给下游。

一次只改变一个轴，否则不能把变化归因给 worker 数量。

22.9 远端 worker 是派发边界，不是免费容量

`WorkerDispatcher` 把特定 work item 发到远端执行池；Kafka 模块提供一种 transport。派发后新增序列化、投递、重复消息、heartbeat、结果关联和取消语义。远端 GPU worker 可以解决硬件亲和性，却不自动解决 exactly-once effect。

正文只保留这个边界。topic、heartbeat、batch size、schema migration 状态和 benchmark 参数集中在运行参考，避免把架构判断埋在配置目录里。

22.10 现实映射：连锁仓库的订单接管

仓库 A 扫描了包裹却断网，仓库 B 不能只因为“一分钟没消息”就同时发货。总部要发出新一代作业凭证，并拒绝 A 的旧凭证。订单原先绑定华东仓，也不能因为今天新增华南仓就自动漂移。

这分别对应 fencing epoch 和 routing binding。增加仓库数量只有在拣货工位是瓶颈时才提升吞吐；如果瓶颈是中央库存锁，仓库越多争用越严重。

22.11 轮到你：提交一份容量结论，而不是一张曲线

选择一条 durable graph，定义：

1. 一个固定输入分布和一个目标 arrival rate。
2. 1/2/4 worker 三个档位，其他条件不变。
3. throughput、p95、backlog age、store p95 四个指标。

4. 一次 worker 中断和一次旧 epoch 写入尝试。
5. 一个停止规则，例如 backlog age 连续三窗上升即停止加压。

产物包含原始结果、环境卡、单轴比较和一句容量结论：“在这些条件下，哪项资源先饱和”。不要写“系统可承载生产流量”，除非生产流量模型和外部依赖也在证据范围内。

22.12 本章证明了什么，没有证明什么

聚焦测试证明：内存 store 模拟中，过期执行能由新 owner 以更高 epoch 接管，旧 epoch 写入被拒绝；既有业务键 binding 胜过新 resolver 选择。它没有证明真实数据库的隔离级别、Kafka 投递语义、跨地域故障恢复或你的生产容量。

下一阶段从第 23 章——业务正确性不等于测试通过开始：从有边界的运行证据转向“声明的业务承诺是否满足”。

22.13 实验验收卡

- 预期与观察：接管需要 lease、epoch 和 route 一致，业务键保持稳定绑定。
- 失败与恢复：用陈旧 epoch 接管；恢复新 lease/epoch 并拒绝旧 owner。
- 证明边界：证明内存测试中的接管与路由，不证明数据库隔离或容量。
- 练习合同：一次 failover；只改 epoch；交付接管 receipt 和路由；旧写入失败且新 owner 唯一即停止。

22.14 精确事实入口

- `ClusterFailoverSimulationTest.java`
- `RuntimeRoutingShardingTest.java`
- `ExecutionLeaseContext.java`
- `StoreRouter.java`
- `bloge-dispatch-kafka`
- Durable 与规模化运行参考

第五阶段回顾——技术信心（第 18–22 章）

章节

第五阶段回顾——技术信心（第 18–22 章）

你现在可以准确说明测试、观测、接线和容量证据证明了什么，以及证据到哪里为止。

0.1 信心阶梯

章节	回答的问题	产出的证据
18 ——测试你的 Graph	选定行为是否发生?	Operator、graph、DSL 与受控时间检查
19 ——生产环境中的可观测性	能否解释一次真实执行?	用四个投影重建一个事故
20 ——Spring 与生产环境接线	谁创建、谁覆盖运行时组件?	最短自动配置加一次有 owner 的 override
21——单 JVM 里的调度与复杂度	ready set 与 graph 复杂度如何塑造本地工作?	调度合同测试与本地测量协议
22——分布式运行与容量验证	集群能力声明何时可信?	Routing、lease、store、recovery 与 migration 边界

这五章是一架梯子。Graph 测试变绿不等于生产可解释；trace 可查询不等于业务正确；单机吞吐也不等于水平扩展。

0.2 你现在应该能做到

- 按 claim 拆分 operator、graph、DSL 和时间相关测试。
- 从 lifecycle、business、trace 与 capacity 四个视角重建一个事故。
- 自动配置 Spring 应用，并让一个组件 override 只有一个 owner。
- 调整复杂度限制，并报告来自所记录环境、可复现的单 JVM 测量。
- 在 routing、lease、共享 store、recovery 与 migration 证据齐备前，拒绝分布式容量 claim。

0.3 常见声明错误

声明	隐藏缺口	更准确的说法
“工作流正确。”	只检查了终点输出	“该场景产生了预期的节点状态与输出。”
“生产问题可调试。”	日志无法连接到同一次执行	“四个投影共享同一 execution identity。”
“Spring 管理所有组件。”	手工 bean 悄然替换自动配置	“这个命名 override 只有一个明确 owner。”
“BLOGE 可以水平扩展。”	基准只跑在一台主机上	“单 JVM 在所记录负载下持续达到 X。”

0.4 试一试：拒绝一个过度声明

某基准在一台 32 核主机上达到每秒 20,000 次 graph 运行，团队据此写下“BLOGE 可以水平扩展”。请列出缺失证据，再把结论改写到证据恰好能支持的强度。答案应区分 scheduler 吞吐与 routing、lease ownership、共享 store recovery、migration。

0.5 下一步

第六阶段回顾把问题从“运行时是否执行”改成“业务承诺是否成立，以及哪些证据可以进入发布决策”。继续阅读第 23 章——业务正确性不等于测试通过。生产配置细节请查附录 I。

学习阶段

第六阶段——业务承诺

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

章节

23

第六阶段——业务承诺

第 23 章——业务正确性不等于测试通过

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

本章承诺：你会把“代码能跑”“*Graph* 成功”“业务结果满足”和“证据可以用于发布”拆成四个问题，并知道 `PASS` 为什么不能直接写成“可以发布”。

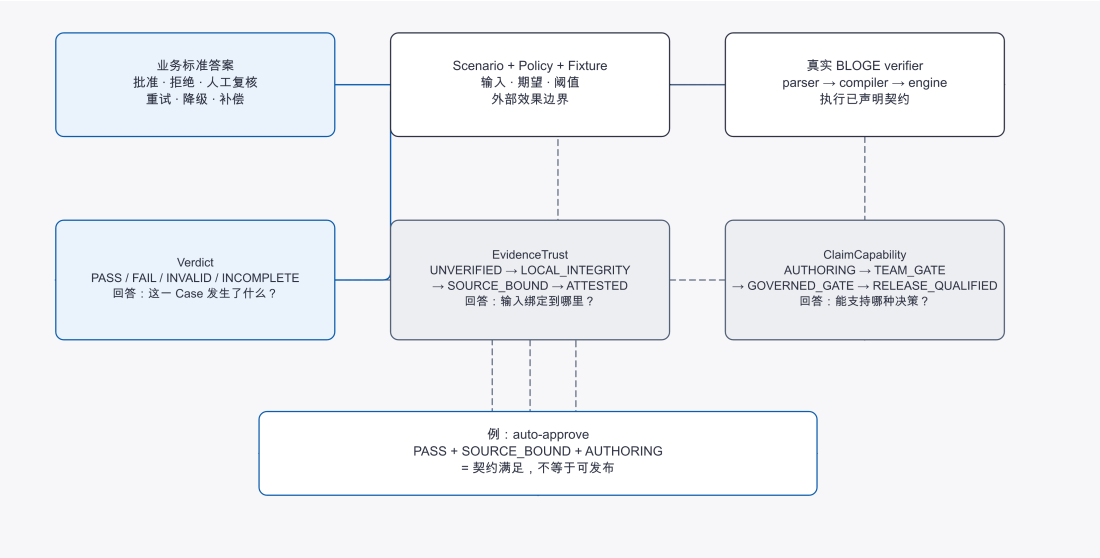
23.1 学习目标

1. 区分语法、执行、业务和证据四种正确性。
2. 建立 `VerificationStatus`、`EvidenceTrust`、`ClaimCapability` 三条独立轴。
3. 找到业务负责人、BLOGE 开发者、verifier 和发布负责人的责任边界。
4. 用一个最小 Scenario 运行真实 BLOGE，而不是只比较最终 `GraphResult`。

23.2 先看业务问题

贷款运营负责人给出了一组可验收的结果：低风险申请自动批准，高风险申请自动拒绝，边界申请进入人工复核；放款失败时必须按政策重试、回退或补偿。

这不是一句“测试通过”可以概括的要求。先看本章的关系图：



图上最重要的不是箭头数量，而是三条轴不能互相代替：

`VerificationStatus` ——回答这次 Suite / Case 发生了什么；不能证明证据是否绑定到声明的源码与运行输入。

`EvidenceTrust` ——回答结果与哪些源码、Scenario、Policy、Fixture 和 runtime 绑定；不能证明业务政策本身是否正确。

`ClaimCapability` ——回答证据最多支持本地开发、团队门禁、治理门禁还是发布；不能证明业务 verdict 是否为 `PASS`。

23.3 先把四种“正确”分开

从第 18 章的 Graph 测试开始，读者已经能验证节点状态、执行顺序和 `GraphResult`。这些能力仍然重要，但它们只覆盖了完整问题的一部分。

语法正确——`.bloge` 能否解析、编译？使用 parser、compiler 和 DSL test；不代表业务分支选对。

执行正确——节点是否按依赖执行、错误是否传播？使用 `GraphTestRunner`、`GraphResult`；不代表业务标准答案满足。

业务正确——`riskDecision` 是否为 `auto_approve`？使用 `Scenario` + `BUSINESS_CONTRACT`；不代表源码和运行身份可复核。

证据可用——这次结果能否支持团队或发布决策？使用 source-bound reader、`VerificationClaims`；不替业务负责人批准政策。

一个 `GraphResult.isSuccess()` 只说明图执行没有按引擎语义失败。它不能自动说明一个拒绝分支没有被错误地绕过，也不能说明测试使用的外部 effect 与生产合同一致。

23.4 第一个 verifier 入口

`ScenarioVerifier.verify(Path)` 是单 Suite 的最短入口。实际 starter 使用 `BlogeScenarioTests.fromDirectory(...)` 把同一目录下的 Scenario 变成 JUnit dynamic tests；两者都经过真实 parser、compiler 和 engine。

下面的片段只展示入口形状，`bootstrap`、`projectRoot` 和 Policy 必须替换成自己的项目事实：

```
ScenarioVerifier verifier = ScenarioVerifier.builder()
        .bootstrap(caseContext -> VerificationEnvironment.controlled(operatorRegistry))
        .projectRoot(projectRoot)
        .policy(projectRoot.resolve("src/test/bloge/verification-policy.yaml"))
        .build();

VerificationReport report = verifier.verify(
        projectRoot.resolve("src/test/bloge/scenarios/loan-approval.scenario.yaml"));

assertEquals(VerificationStatus.PASS, report.status());
```

这里的 `PASS` 只回答“声明的 Suite 在这次运行中满足了它的 Expectation”。如果没有 source-bound artifact、receipt 和独立 reader，不能顺手把它升级为 `SOURCE_BOUND` 或 `RELEASE_QUALIFIED`。

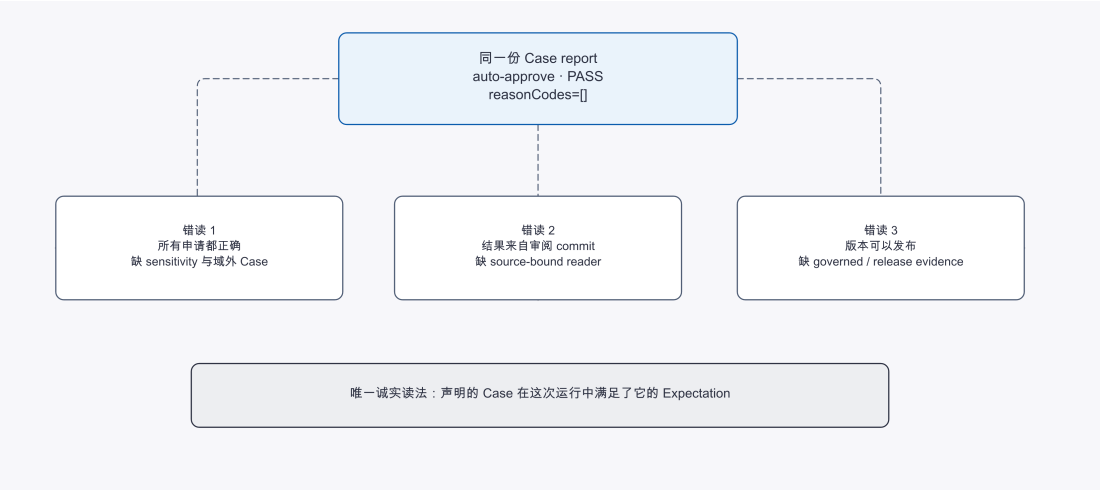
23.5 同一份 PASS，读出三种错结论

假设贷款报告只有这些事实：

```
suiteId=loan-approval-decisions
caseId=auto-approve
status=PASS
reasonCodes=[]
```

三个读者可能把一份诚实报告升级成三句不诚实的话：

错误解读	缺了什么	正确读法
“所有申请人的决策规则都正确。”	Suite 之外的 Case 和 sensitivity 挑战	这个 Case 满足了声明的 Expectation。
“结果一定来自审阅过的 commit。”	seal、源码摘要、worktree identity 和 reader 结果	单独的 report object 仍是 `UNVERIFIED`。
“这个版本可以发布。”	Policy continuity、治理 receipt、comparison 和 release attestation	`PASS` 本身推不出发布能力。



这不是抽象的谨慎。RC1 有意让三条轴能够彼此不同。`VerificationReportWriterTest.rechecksArtifactBytesAfterThePublicationGuard` 先构造一份 `PASS` report，再让 publication guard 把 summary 扩大到 4096 字节以上；writer 最终得到 `INCOMPLETE / RESOURCE_LIMIT_EXCEEDED`，并保持输出目录为空。业务 verdict 不能覆盖出版边界失败。

`VerificationContractEvidenceTest` 从另一侧封口：`PASS` Case 不能同时携带失败原因，`FAIL` Case 不能声明 satisfied；聚合后的 Suite、Case、Requirement 和 assurance identity 必须一致。2026-09-14 在 BLOGE commit `cc38fbe5` 上，这两个聚焦测试类共运行 5 tests，0 failures，0 errors，0 skips。

所以阅读顺序必须固定：

1. 先读 `VerificationStatus`：声明的合同在这次运行里是否满足？
2. 再读 `EvidenceTrust`：证据绑定了哪些不可变输入？
3. 最后读 `ClaimCapability`：完整证据最多能支持哪类决策？

开发阶段停在第一步没有问题；把第一步改名成第三步，才是问题。

23.6 三个容易出现的“假绿”

跳过 **`manual_review`** ——普通测试可能只看到最终对象仍然是 `approved`。业务验证必须分别断言业务输出和人工路径；跳过节点属于结构事实。

隐藏 **fallback** ——普通测试可能只看到 Graph 仍然 `SUCCESS`。业务验证必须记录 retry disposition、fallback 和 effect 处置，再确认这是政策允许的降级。

直接调用 **Operator** ——普通测试可能只看到 Operator 单测 `PASS`。业务验证还要让 Scenario 进入真实 parser/compiler/engine，并限制未声明 effect。

这不是把 JUnit 判定为无用。正确的做法是让每一层回答自己擅长的问题，再把结果按证据边界组合起来。

23.7 四类角色，各自提供什么

业务负责人——提供标准答案、边界 Case、Requirement、可接受失败和 Oracle 来源；不把绿色测试直接批准为发布。

BLOGE 开发者——提供 DSL、Operator、Scenario、Policy、Fixture 和 bootstrap；不用实现代码复制标准答

案。

verifier / 平台——提供受控执行、观察、reason code、artifact、seal 和 reader；不修改业务政策来消除失败。

发布负责人——提供 commit、worktree、receipt、ClaimCapability 和缺口；不把 `AUTHORING` 改写为 `RELEASE_QUALIFIED`。

23.8 本章实验：把一个“绿测试”拆开

1. 选一个只断言 `GraphResult` 的贷款 Case，补一个 `BUSINESS_OUTPUT` 断言。
2. 再补一个分支跳过或调用事实的 `STRUCTURAL_CONTRACT` 断言。
3. 列出这两个断言仍然没有证明的两件事，例如源码未绑定、外部服务未被独立复核。
4. 将报告状态、证据可信度和 claim 能力写成三列，不允许用一个词覆盖三列。

实验停止条件：如果输入、Policy 或 bootstrap 不完整，报告应保持 `INVALID` 或 `INCOMPLETE`；不要通过重试把合同问题改写成业务 `FAIL` 以外的绿灯。

23.9 现实迁移：药品放行流程

在医院药房里，“流程执行完成”只说明处方走完了声明步骤，不代表剂量满足已批准 规则、结果来自已审阅药品目录，也不代表药师可以放行。把 graph status、剂量 Expectation、与药品目录绑定的 evidence、放行权限分别映射到本章四层。如果一个 绿色徽章同时承担两个以上责任，就先停止发布判断。

23.10 实验验收卡

- 预期与观察：同一 PASS 分别读取 verdict、evidence trust 和 claim capability。
- 失败与恢复：删除 guard 或污染 evidence；恢复证据后重读三轴。
- 证明边界：证明受控 Case 的声明，不证明发布已批准。
- 练习合同：一个绿色 Case；只攻击一条证据边界；交付三轴读数；结论不越过 capability 即停止。

23.11 本章小结

可靠的 `FAIL` 能指出一个业务合同确实没有满足；没有来源绑定的 `PASS` 只说明一次运行看起来成功。下一章把这条原则落到 starter 的两个 Suite、七个 Case 上。

下一章：第 24 章——第一条贷款业务正确性验证

章节
24

第六阶段——业务承诺

第 24 章——第一条贷款业务正确性验证

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

本章承诺：你会把一条由业务负责人确认的贷款决策写成可执行场景，沿真实 *graph* 路径解释每个断言，并读懂 *BLOGE* `0.9.8-RC1` 生成的验证报告。你也会知道：没有业务标准答案时，验证必须停在哪里。

24.1 学习目标

学完本章，你能够：

1. 区分“程序执行成功”和“业务结果正确”。
2. 把业务规则拆成 `given / when / then`，并让 `then` 指向可观察的业务输出或结构事实。
3. 从场景、graph、输出映射、验证策略和报告之间追踪一条完整证据链。
4. 用少量代表性案例覆盖正常决策、人工复核、重试、降级和补偿。
5. 在业务答案缺失时停止自动化，不让技术团队替业务拍板。

24.2 这一章只增加一个变量

前面的章节主要回答：“graph 能不能按设计运行？”本章只增加一个问题：它跑出的业务答案对不对？

这两个问题不能合并。一个贷款申请可能没有抛异常、每个节点都已完成，却把本该人工复核的客户自动批准了。运行时会说“执行成功”，业务负责人却会说“决策错误”。

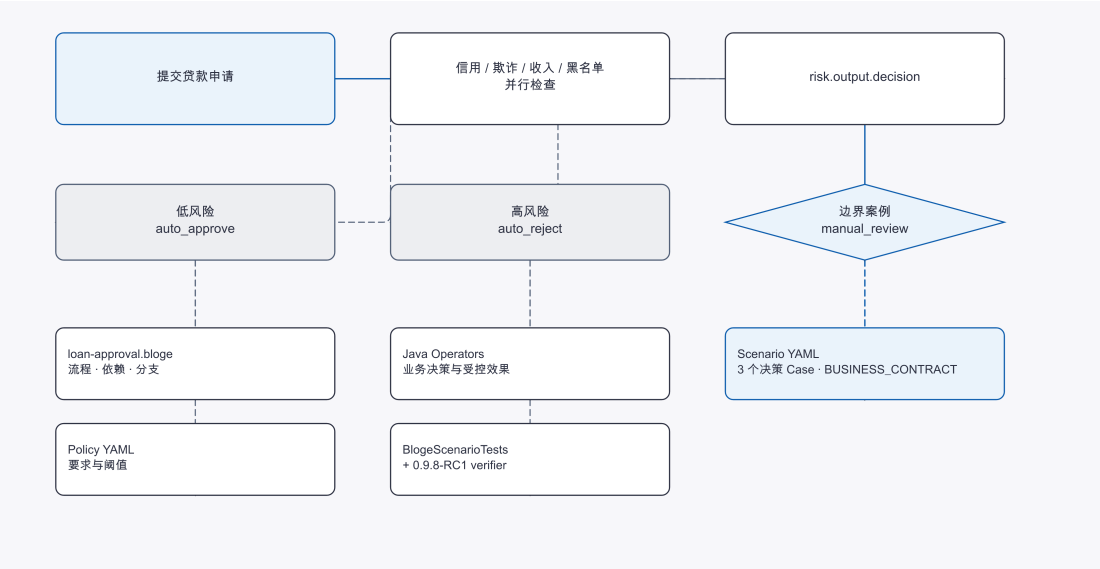
因此，业务正确性验证不是更多单元测试的别名。它把业务认可的答案、真实 graph 路径和可复核报告连成一条链。

24.3 先预测：低风险申请应该走哪条路

贷款策略负责人给出一个已批准样例：申请 `LA-VERIFY-1` 风险较低，应该自动通过。先不要看测试代码，预测 graph 最终应留下哪些可观察事实：

观察点	你的预测	为什么不能只看最终 **HTTP 200**
风险决策	`auto_approve`	证明路由依据正确
批准状态	`approved`	证明批准分支真的产生业务结果
拒绝节点	`SKIPPED`	证明互斥分支没有被误执行

第三个观察点很重要。只检查 `approved`，可能漏掉“批准和拒绝都执行了”这种结构性错误。



24.4 五份资产，各自回答一个问题

第一条可审计证据链不是一个大文件，而是五份职责不同、可以互相追踪的资产：

资产	回答的问题	本章中的实例
业务标准答案	什么结果才算对	低风险申请应自动批准
场景	用什么输入、执行什么动作、检查什么事实	`loan-approval.scenario.yaml`
graph	系统实际经过哪些节点和分支	`loan-approval.bloge`
验证策略	哪些案例必须存在、证据至少多深	`verification-policy.yaml`
报告	这次运行究竟观察到了什么	`target/bloge-verification/`

它们的关系是：业务负责人给答案，技术团队把答案变成可执行观察，**BLOGE** 负责运行并留下证据。

24.5 把一句业务规则写成 Given / When / Then

仓库中的真实场景先声明责任人、graph 来源和 case 身份，然后写三段式合同。下面只保留 `auto-approve` 的关键部分：

```
- caseId: auto-approve
  title: 低风险申请自动通过
```

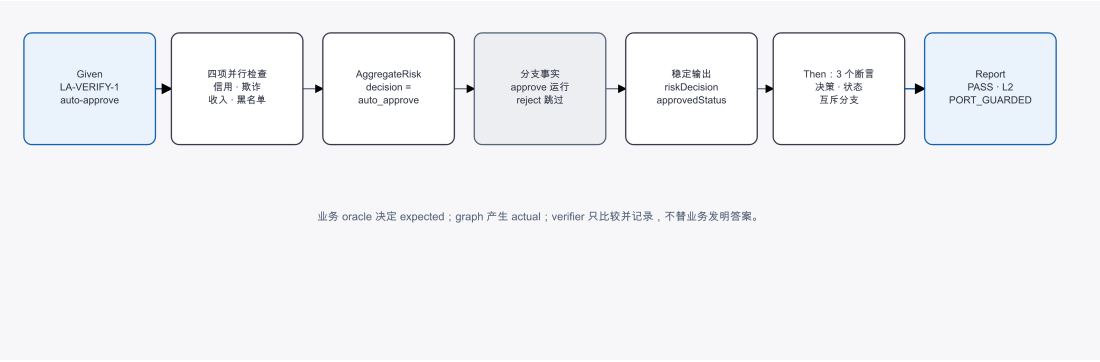
```yaml
  requirementRefs: [LOAN-DECISION-001]
  oracle:
    type: APPROVED_EXAMPLE
    sourceRef: starter-policy:auto-approve
    owner: loan-policy-owner
  given:
    context:
      applicationId: LA-VERIFY-1
      caseName: auto-approve
  when:
    action: EXECUTE
  then:
    outcome: SUCCESS
    expectations:
      - actual: {kind: BUSINESS_OUTPUT, outputId: riskDecision}
        matcher: EQUALS
        expected: auto_approve
      - actual: {kind: BUSINESS_OUTPUT, outputId: approvedStatus}
        matcher: EQUALS
        expected: approved
      - actual: {kind: NODE_STATUS, nodeId: reject}
        matcher: EQUALS
        expected: SKIPPED
```

读它时不要先记 YAML 字段，先问三个业务问题：

- `given`：当前 Case 描述哪一类现实申请？
- `when`：要让系统做什么？这里是执行真实 graph。
- `then`：哪些事实足以让业务负责人说“这个答案对”？

`oracle` 不是一个神秘算法。它说明标准答案来自哪里、属于哪一类证据、由谁负责确认。`APPROVED_EXAMPLE` 的含义是“业务方认可的代表性样例”，不是 BLOGE 自己推导出的政策。

24.6 沿 graph 追一遍：断言从哪里拿到事实



自动批准案例实际经过以下因果链：

1. `FetchLoanApplication` 根据 `caseName=auto-approve` 构造确定性的申请资料。
2. `credit`、`fraud`、`income`、`blacklist` 四项检查读取同一份申请，各自产生风险事实。
3. `AggregateRisk` 汇总这些事实，输出 `decision=auto_approve`。
4. graph 只进入 `approve` 分支；`reject` 和人工复核分支不应执行。
5. `outputs` 把节点内部数据映射成稳定的 `riskDecision`、`approvedStatus` 等业务输出，验证器再比较预期和实际值。

这解释了两类断言为什么要同时存在：

- `BUSINESS_OUTPUT` 检查业务结果，例如“批准状态是什么”。
- `NODE_STATUS` 检查执行结构，例如“拒绝分支确实跳过”。

如果业务输出命名稳定，即使 graph 内部重构了若干节点，大部分业务合同仍可保持不变；只有真正关心互斥、重试或补偿行为时，才下探到结构事实。

24.7 跑一次真实验证

在书稿根目录执行聚焦测试：

```
cd submodule/bloge
mvn -pl bloge-starter-loan-approval -am \
  -Dtest='LoanApprovalApplicationTest,LoanApprovalBusinessScenarioTest,LoanApprovalOperatorFlowTest' \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

2026-09-14 在 BLOGE commit `cc38fbe5`、版本 `0.9.8-RC1` 上得到：

```
LoanApprovalApplicationTest       1 passed
LoanApprovalOperatorFlowTest      3 passed
LoanApprovalBusinessScenarioTest  7 passed
Total                             11 passed, 0 failed
```

这里的七个动态场景不是七个“方法能否返回”的单元测试，而是三条决策规则和四条放款韧性规则。报告摘要进一步给出：

```
Suite: loan-approval-decisions
Status: PASS
Cases: 3/3 passed
Executed nodes: 10/10
auto-approve depth=L2 effectSafety=PORT_GUARDED replay=CONTRACT_CONTROLLED
Requirement LOAN-DECISION-001 satisfied=true cases=1/1
```

`auto-approve` 的机器可读结果为：

```
{
  "caseId": "auto-approve",
  "effectSafety": "PORT_GUARDED",
  "realEffectPortIds": [],
  "reasonCodes": [],
```

```
  "replayAssurance": "CONTRACT_CONTROLLED",
  "status": "PASS",
  "unverifiedOperatorRefs": [],
  "verificationDepth": "L2"
}
```

不要把这些标签读成宣传语：

- `L2` 表示证据达到了 graph 场景执行层，不代表生产环境已验收。
- `PORT_GUARDED` 表示副作用端口受到验证边界约束；空的 `realEffectPortIds` 说明这次没有触达真实外部副作用端口。
- `CONTRACT_CONTROLLED` 表示重放条件由合同控制，不等于任意生产现场都可完全复现。
- `reasonCodes=[]` 和 `unverifiedOperatorRefs=[]` 表示本次没有额外降级理由或未验证 Operator 引用。

24.8 从一条案例扩到决策覆盖图

一条自动批准 Case 只支持一个政策区域内已经声明的断言。这个决策 suite 抽样三个互斥业务区域：

Case	业务形状	必须命中的结果	必须排除的结果
`auto-approve`	低风险	自动批准	拒绝分支跳过
`auto-reject`	高风险	自动拒绝	批准分支跳过
`manual-review`	中风险或信息需复核	进入人工审核并得到最终状态	自动批准分支跳过

这不是为了追求“Case 数量”。三个 Case 对应三个互斥的业务决策区域；少一个，就有一块已声明政策空间没有标准答案，但三条样例本身也不等于政策全集。

24.9 再看放款：正确性也包括失败后的世界

放款 suite 把问题从“选哪条分支”推进到“失败后系统应留下什么业务状态”：

Case	现实映射	关键证据
`funding-success`	正常放款并通知借款人	`fundingStatus=notified`
`transient-booking-retried`	账务服务短暂抖动	第一次安排重试，第二次成功
`booking-fallback-used`	重试耗尽，采用已批准降级	第三次调用留下 `FALLBACK_APPLIED`
`notification-failure-compensated`	钱已预订，但通知发生不可重试错误	graph 失败，账务预订撤销，资金释放

最后一条尤其容易写错：业务正确的结果不是“整条 graph 成功”，而是明确失败，同时把已经发生的可逆副作用补偿掉。所以 `then.outcome` 可以是 `FAILURE`，而案例本身仍然验证通过。

24.10 没有业务答案时，必须停下来

假设产品经理只说：“高净值客户的人工复核可以更宽松。”这句话还不能直接写成可执行 oracle。至少缺少：

- “高净值”的可判断定义；

- 哪些风险维度可以放宽、放宽到什么值；
- 最终应该批准、拒绝，还是仍由人工决定；
- 谁对这个标准答案负责。

此时正确动作不是让开发者猜一个 `expected`，也不是从当前代码反推出“代码现在怎么做，什么就是正确”。应把场景标成待业务澄清，记录缺失的规则和 owner，然后停止发布门禁的通过判断。

停止规则：没有可追溯的业务 *oracle*，就可以验证“系统做了什么”，但不能宣称“业务结果正确”。

24.11 Maven 插件失败时，也不要猜

`0.9.8-RC1` 的 Maven 验证入口需要测试 classpath。如果插件无法构造它，会稳定报告：

```
TEST_CLASSPATH_UNAVAILABLE: cannot execute BLOGE verification
```

这不是业务案例失败，而是验证环境没有建立起来。应先修复依赖或 reactor 范围，再重新执行；不要把“验证没有运行”写成 PASS，也不要仅凭已有报告目录判断这次构建成功。

24.12 当前证据支持什么，又没有证明什么

本章证据支持以下结论：

- 固定 commit 上的三条贷款决策和四条放款场景全部通过。
- `auto-approve` 的业务输出、互斥分支和需求映射都留下了机器可读证据。
- 场景没有触达真实外部副作用端口，重放条件受合同控制。

它没有证明：

- 所有可能输入组合都正确；
- 业务政策本身合理或符合法规；
- 生产数据库、消息系统和外部服务已经验收；
- 未来版本仍会得到相同结果。

业务正确性验证的价值不在于给系统盖一个永久的“正确”印章，而在于让一次结论能够回答：谁定义了答案、用什么输入、跑过哪条路径、观察到什么、在哪个版本上成立。

24.13 换一个行业，方法仍然成立

把贷款换成电商退款：

- `given`：订单已签收 3 天，商品未拆封，金额 199 元；
- `when`：执行退款资格 graph；
- `then`：`refundDecision=auto_approve`，原路退款分支执行，人工审核分支跳过；
- `oracle`：售后负责人批准的退款政策样例。

变化的是业务词汇和标准答案，不变的是证据链：业务 oracle → 场景 → graph → 可观察事实 → 报告。

24.14 轮到你：只改一个变量

复制 `auto-approve` 场景，只把 `caseName` 改成 `manual-review`，先写下你预测的三个事实，再与仓库中的正式 case 比较：

1. `riskDecision` 应是什么?
2. 哪个最终状态应出现?
3. 哪条自动分支必须是 `SKIPPED`?

如果你的答案与业务 owner 的批准样例冲突，不要修改 expected 让测试变绿；先确认是实现错误、样例过期，还是政策已经改变。

24.15 实验验收卡

- 预期与观察：低风险贷款满足 Given/When/Then，报告 3/3 passed。
- 失败与恢复：删除标准答案或让插件失败；恢复已确认答案或构建，缺事实时停下。
- 证明边界：证明固定 Case 和 graph 合同，不证明政策完整或已签署发布。
- 练习合同： auto-approve Case；只改一个字段；交付 path、Expectation、Report；结果可归因且无猜测即停止。

24.16 继续阅读源码

- 贷款决策 graph
- 贷款决策场景
- 放款韧性场景
- 验证策略
- 动态业务场景测试

下一章第 25 章——Policy、Fixture 与 Assurance会继续追问：通过的 Suite 是否在预期动作、依赖和 assurance 边界内执行。

章节

25

第六阶段——业务承诺

第 25 章——Policy、Fixture 与 Assurance

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

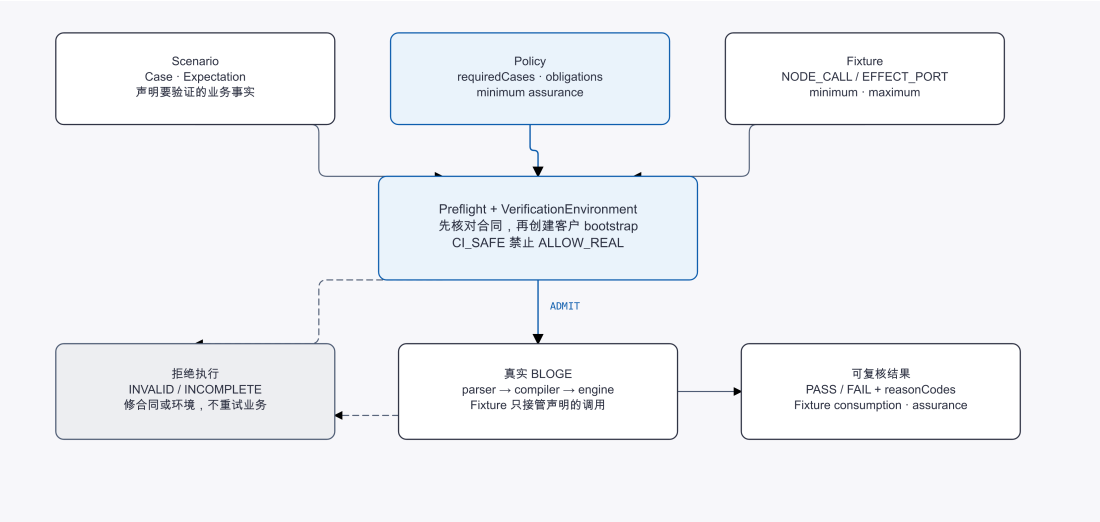
本章承诺：你会把「必须验证什么」「外部行为怎样受控」「证据最低达到什么程度」拆成三份声明。删除受保护 *Case*、*Fixture* 消费越界或放行真实 *effect* 时，验证在执行边界内停止，而不是制造一个假绿结果。

25.1 学习目标

1. 区分 Scenario、Policy、Fixture 和 `VerificationExecutionProfile` 的责任。
2. 看懂 `FAIL`、`INVALID`、`INCOMPLETE` 对应的下一步动作。
3. 用 `verifyGoverned(Path)` 检查 Requirement，用 `verifyFixtureFidelity(Path)` 检查 Fixture 合同。
4. 复现三个单因素失败，并从最早 reason code 开始排查。

25.2 三份声明守住一个执行入口

团队为了让贷款测试变绿，可能删除 `manual-review`，把 Fixture 改成「任何请求都返回成功」，或者让未声明的 effect 访问真实账务系统。三种修改都没有改变断言，却缩小或污染了证明范围。



图中的橙色入口是 preflight。Scenario、Policy 和 Fixture 先完成一致性检查，验证器才创建客户 bootstrap 并进入真实 BLOGE parser、compiler 和 engine。被拒绝的配置不会先执行客户代码再补一个警告。

Scenario 声明一个 Case 的输入、动作和 Expectation。它回答「这次要观察什么」。

Policy 声明不能静默删除的 Suite、Case、Requirement 和最低 assurance。它回答「最少必须保留什么」。

Fixture 声明哪一个 node call 或 effect port 被接管、返回什么以及允许消费几次。它回答「外部行为怎样受控」。

`VerificationExecutionProfile` 固定逻辑时钟、随机种子、ID 种子、harness timeout 和 persistence mode。host capability 只是被绑定的输入，不是隔离证明。

25.3 先保护人工复核 Case

starter Policy 已把三个决策 Case 写进 `requiredCases`：

```
requiredSuites:
  - suiteId: loan-approval-decisions
    requiredCases: [auto-approve, auto-reject, manual-review]
    minimumDepth: L2
    minimumBusinessExpectations: 6
    minimumReplayAssurance: CONTRACT_CONTROLLED
maximumUnverifiedEffects: 0
```

运行治理入口：

```
VerificationGovernedReport report = ScenarioVerifier.builder()
        .bootstrap(caseContext -> loanVerificationEnvironment(caseContext))
        .projectRoot(projectRoot)
        .policy(projectRoot.resolve("src/test/bloge/verification-policy.yaml"))
        .build()
        .verifyGoverned(projectRoot.resolve(
                "src/test/bloge/scenarios/loan-approval.scenario.yaml"));
```

requiredCases 只防止 Case 消失。Requirement 是否满足还取决于 requirementObligations 和 requirementBindings：只有通过的 BUSINESS_CONTRACT Expectation 能贡献 Requirement evidence。Case 仅引用 requirementRefs 不等于 Requirement 已满足。

25.4 Fixture 不是全局 mock

替换整个 Operator 调用时使用 NODE_CALL；保留 Operator 计算、只接管外部访问时使用 EFFECT_PORT。两类 target 不能混用。

```yaml
schemaVersion: 1
fixtureSetId: funding-effect
rules:
  - ruleId: book-once
    target:
      type: EFFECT_PORT
      portId: ledger.book
      parentNodeId: bookDisbursement
      inputEquals: {applicationId: LA-FUND-1}
    behavior:
      type: RETURN
      value: {status: booked}
    consumption: {minimum: 1, maximum: 1}
```

规则零命中、歧义、欠消费、超额消费和未绑定 effect 都 fail closed。CI_SAFE 是当前唯一 profile，并明确拒绝 ALLOW_REAL。Effect DELAY 使用 bootstrap 提供的 TimeSource；Fixture 本身不会用 Thread.sleep 等待墙钟。

25.5 Assurance 与 fidelity 不混写

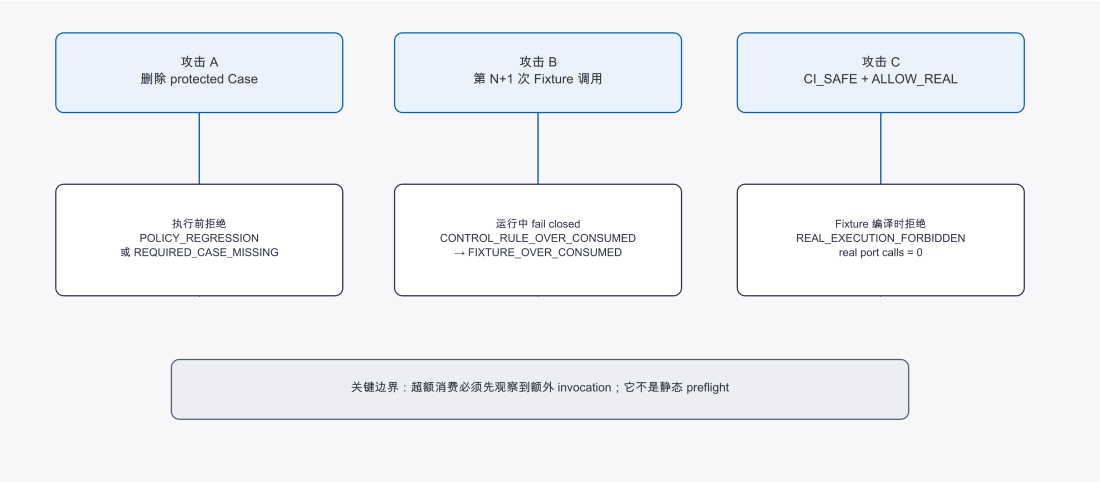
一个 Case 可以业务 PASS，但 Fixture fidelity 仍然只达到 DECLARED_SYNTHETIC。实现 VerificationFixtureFidelityProvider 后，provider 提供按 fixtureSetId/ruleId 定位的合同；验证器自行检查 Schema 和可选 adapter observation，provider 无权直接返回 PASS 或指定等级。

```
VerificationFidelityReport fidelity = verifier.verifyFixtureFidelity(scenarioPath);
```

本地 provider 的首版上限是 PROVIDER_ATTESTED_ADAPTER。只有合同来源、adapter 实现与配置、治理主体都进入 source-bound evidence，才可能得到 ADAPTER_CONTRACT_VERIFIED。未配置 provider 时返回较低的 DECLARED_SYNTHETIC，不能把「没有 fidelity 证据」写成「验证失败」。

25.6 做三次单因素攻击

三次都使用同一个 manual-review Case，每次攻击后先恢复输入。否则第二次结果无法归因到一个变化。



只改一个事实	最早可靠观察	客户代码是否运行	恢复动作
删除 Policy 保护的 Case	候选 Policy 变弱：`POLICY_REGRESSION`；Scenario 缺 Case：`REQUIRED_CASE_MISSING`	否	恢复 Case，不降低 baseline。
Fixture rule 调用超过 `maximum`	下一次 resolve fail closed：`CONTROL_RULE_OVER_CONSUMED`；verifier diagnostic 映射为 `FIXTURE_OVER_CONSUMED:<set>/<rule>`	已开始执行，但额外的真实调用不会放行	修调用次数，或由业务确认新上限。
在 `CI_SAFE` 下把 behavior 改成 `ALLOW_REAL`	Fixture 编译阶段得到 `REAL_EXECUTION_FORBIDDEN`	不调用 real port	保留 double；只有明确隔离 profile 存在时才迁移测试。

这里的时机区别很重要。Case 是否存在、`ALLOW_REAL` 是否出现，都能在执行前知道；超额消费只有看到多出来的那次 invocation 才能判断。把三者都称为“preflight”会掩盖真正的安全边界。

2026-09-14 在 BLOGE commit `cc38fbe5` 上运行 `PolicyContinuityValidatorTest`（11）、`FixtureBehaviorCompilerTest`（2）和 `VerificationExecutionProfileTest`（2），共 15 tests 全部通过。`BehaviorActionTest` 另有 7 个 execution-control tests，其中两次返回序列的第三次调用抛出 `CONTROL_RULE_OVER_CONSUMED`。`ScenarioVerifier.controlReasons` 再把结束诊断翻译为 `FIXTURE_OVER_CONSUMED:<ruleReference>`。

25.7 状态决定下一步动作

`FAIL`——业务答案错了。合同有效且执行完成，但业务或 temporal Expectation 不满足。先检查业务实现、输入或批准的标准答案，不要先改 Policy。

`INVALID`——验证声明错了。 Scenario、Policy、Fixture、DSL 或能力组合无效。修复声明；用同一份输入直接重试没有意义。

`INCOMPLETE`——运行没有完成。超时、资源、adapter 或 evidence capture 中断了运行。保留诊断，修复环境，再从一次干净运行开始重试。

从最早 reason code 开始，而不是从最后一条异常文本猜原因。代表性边界如下：

- `REQUIRED_CASE_MISSING`：Policy 要求的 Case 已被删除。

- `FIXTURE_UNDER_CONSUMED:<fixtureSetId>/<ruleId>`：声明至少消费一次，但实际没有命中。
- `FIXTURE_OVER_CONSUMED:<fixtureSetId>/<ruleId>`：实际调用次数超过 `maximum`。
- `REAL_EXECUTION_FORBIDDEN`：`CI_SAFE` 遇到 `ALLOW_REAL`。
- `FIXTURE_TARGET_ZERO_MATCH` / `FIXTURE_TARGET_AMBIGUOUS`：target 不能唯一绑定受控调用点。

25.8 实验：依次制造三个失败

每轮只改变一个因素，并在下一轮前恢复该因素。

1. 从 `loan-approval.scenario.yaml` 删除 `manual-review`。运行治理入口，预期在客户 bootstrap 前得到 `INVALID` 和 `REQUIRED_CASE_MISSING`。停止并恢复 Case，不要降低 Policy。
2. 把一个会调用两次的 Fixture rule 的 `maximum` 改为 `1`。预期得到 `FIXTURE_OVER_CONSUMED:<fixtureSetId>/<ruleId>`。检查 attempt 和 rule target，不要扩大为无界消费。
3. 把 effect behavior 改为 `ALLOW_REAL`。在 `CI_SAFE` 下预期得到 `INVALID` 和 `REAL_EXECUTION_FORBIDDEN`。停止运行；若真实 adapter 必须参与，应在隔离环境另建明确能力边界，不能在本门禁内放行。

记录表只保留可复核事实：改动、状态、第一条 reason code、bootstrap 是否被调用、恢复动作。不要只保存终端截图。

25.9 现实迁移：高额退款控制

在电商退款流程中，让 Policy 保护“高额退款必须人工复核”的 Case，让 Fixture 只接管 `payment.refund`，并让 `CI_SAFE` 禁止真实支付 adapter。分三次运行删除 Case、重复消费 Fixture、放行真实 port。真正有用的结果是每次停在哪一层，而不是 控制台最终能否变绿。

25.10 本章边界

- `BUSINESS_CONTRACT` 回答业务结果；`STRUCTURAL_CONTRACT` 回答 node、invocation、Fixture 消费和补偿事实。
- `INVOCATION` 与 `EFFECT_INPUT` 是观察选择器，不是生产 adapter 认证。
- Fixture 通过不证明生产 HTTP、数据库或消息协议兼容。
- `VerificationEffectResolverFingerprintProvider` 绑定 resolver 配置；它不把运行时自然变化的状态写进配置摘要。

25.11 实验验收卡

- 预期与观察： Policy、Fixture、Assurance 分别约束动作、模拟行为和证据强度。
- 失败与恢复：删除 Case、超额消费或切到 ALLOW_REAL；按 reason 修复。
- 证明边界：证明入口受控，不证明模拟服务等同生产。
- 练习合同：人工复核 Case；每次攻击一个控制；交付三份失败 receipt；每个 reason 对应唯一动作即停止。

25.12 本章小结

Policy 防止验证范围被静默缩小，Fixture 控制外部行为，execution profile 固定可重复运行的基础输入。只有 policy/profile preflight 成功且 Fixture 消费检查完成后，业务 `PASS` 或 `FAIL` 才值得解释。下一章把同样的纪律扩展到暂停、信号、恢复和逻辑时间。

下一章：第 26 章——Durable 与 Temporal 验证

章节

26

第六阶段——业务承诺

第 26 章——Durable 与 Temporal 验证

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

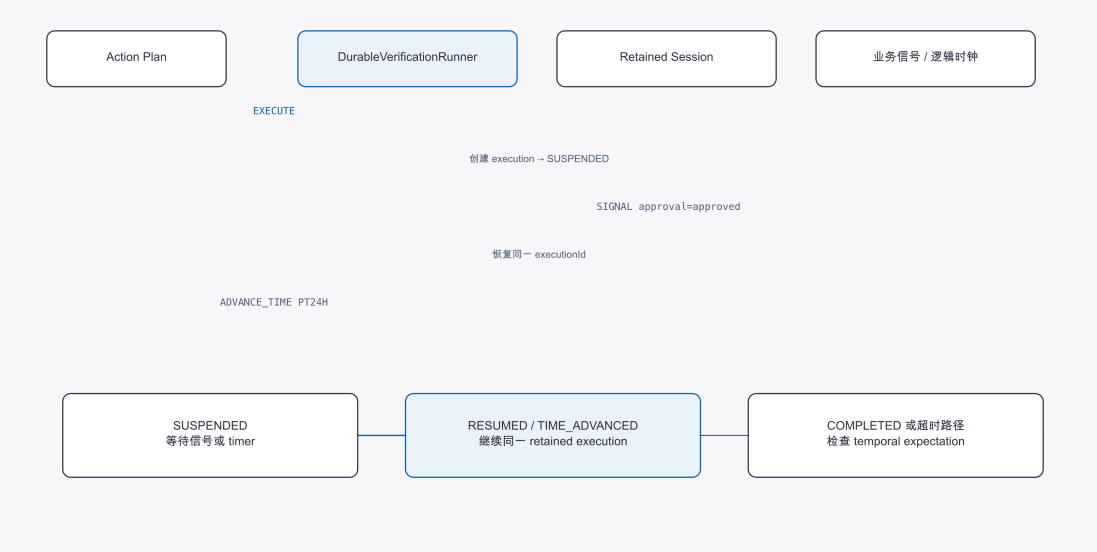
本章承诺：你会用有限 *action plan* 验证同一条 *retained execution* 的暂停、信号、恢复和逻辑时间，并把「最终结果」「不能提前发生」「限定时间内必须发生」写成不同的可观察合同。

26.1 学习目标

1. 使用 `EXECUTE`、`SIGNAL`、`ADVANCE_TIME` 和 `RESUME`，不把长流程压成一次方法调用。
2. 区分 `DurableVerificationRunner` 的本地执行报告和 `DurableSourceBoundVerifier` 的源码绑定证据。
3. 用 `DurableTemporalContract` 表达 step、trace 和 effect 的有限事实。
4. 识别 streaming、timer、checkpoint 和非协作超时的停止边界。

26.2 最终 **output** 看不到等待过程

人工复核的最终结果可能都是 `approved`，但两个流程仍然不同：一个在收到人工决定后才通知，另一个在等待期间提前通知。只比较最终 output 会把第二个错误漏掉。



`DurableVerificationRunner` 为一个 Case 创建真实 `DurableGraphEngine` 和 retained session。默认使用隔离的 in-memory durable stores。action 按列表串行执行，并绑定同一个 execution identity；逻辑时间由 verifier 控制，不依赖测试运行时经过了多少墙钟时间。

26.3 Action plan 是有限脚本，不是工作流语言

下面是「开始后暂停，24 小时内收到人工批准」路径：

```yaml
schemaVersion: 1
actions:
  - actionId: start
    type: EXECUTE
  - actionId: wait-before-decision
    type: ADVANCE_TIME
    duration: PT23H
  - actionId: approve
    type: SIGNAL
    nodeId: approval
    signalData: approved
```

约束比语法更重要：

- action 数量必须在 1 到 64 之间。
- 第一条且只有第一条 action 必须是 `EXECUTE`。
- `ADVANCE_TIME` 的 ISO-8601 duration 必须为正。
- `SIGNAL` 必须精确命中当前 suspended node。
- `RESUME` 只用于已经失败且已写入 checkpoint 的 execution；仍在 suspended 的 execution 应先 `SIGNAL`。
- `CANCEL` 没有稳定 durable seam，sidecar 会返回 `DURABLE_ACTION_UNSUPPORTED`。

超时路径应使用另一份 action plan，把逻辑时间推进到 `PT24H` 或业务确认的边界。不要把「收到信号」和「没有收到信号」硬塞进一条线性 plan；它们是两个业务 Case。

26.4 同时检查 step 与事件顺序

```yaml
schemaVersion: 1
stepExpectations:
  - expectationId: initially-suspended
    afterActionId: start
    outcome: SUSPENDED
    suspendedNodeIds: [approval]
traceExpectations:
  - expectationId: notify-after-approval
    matcher: MUST_NOT_OCCUR_BEFORE
    event: {type: NODE_COMPLETED, graphName: approval-flow, nodeId: notification}
    afterActionId: approve
  - expectationId: notify-within-five-minutes
    matcher: MUST_OCCUR_WITHIN_LOGICAL_TIME
    event: {type: NODE_COMPLETED, graphName: approval-flow, nodeId: notification}
    within: PT5M
```

`stepExpectations` 检查某个 action 后的 outcome 和 suspended set。`traceExpectations` 使用 BLOGE listener 记录的 node identity、action 和逻辑时间偏移；它不执行脚本，也不匹配异常文本或原始业务 payload。

需要验证业务 effect 的重复发送时，使用 temporal v2：

```yaml
schemaVersion: 2
effectExpectations:
  - expectationId: ledger-booked-once
    matcher: EXACTLY_ONCE_EFFECT_KEY
    effect: {portId: ledger.book, key: LA-42}
```

所有参与 Operator 必须提供完整 dependency manifest，目标 `EffectPort` 必须声明稳定 `idempotencyKey(input)`。成功、失败和被拒绝的调用都计数，因为调用方看到失败之前，外部系统可能已经接受 effect。报告只保存类型化摘要，不保存原始 key。

26.5 运行本地 durable 验证

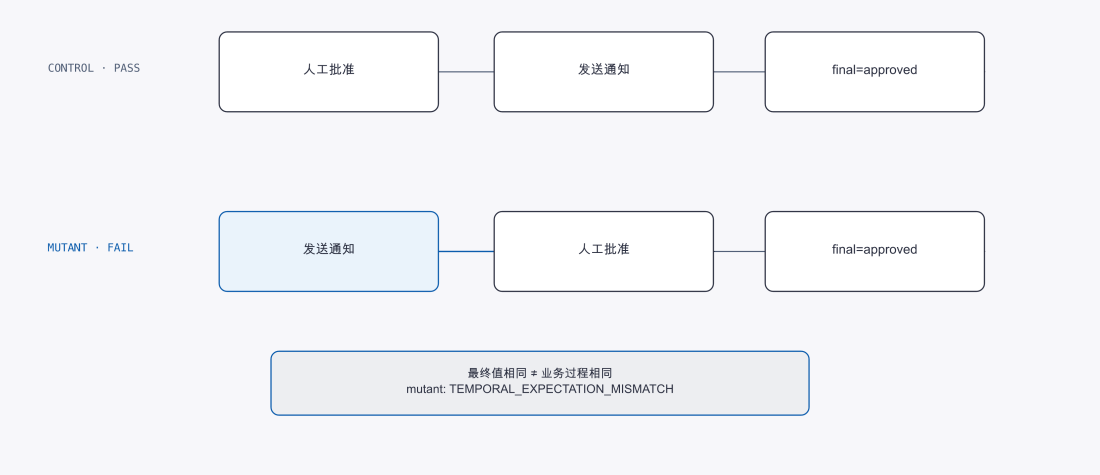
```
DurableVerificationPlan plan = DurableVerificationPlan.load(actionPath);
DurableTemporalContract temporal = DurableTemporalContract.load(temporalPath);

DurableVerificationReport report = DurableVerificationRunner.builder()
        .harnessTimeout(Duration.ofSeconds(30))
        .build()
        .verify(
                new DurableVerificationInput(
                        new VerificationCaseContext("loan-review", "approved-in-time"),
                        compiledGraph,
```

```
                initialContext,
                compiledOperators),
        plan,
        temporal);
```

这份 report 是本地 authoring evidence。若要发布 source-bound durable evidence，使用 `DurableSourceBoundVerifier` 从 tracked graph 和 sidecar 编译，并提供干净 Git worktree、可冻结的 registry/resolver、`DURABLE` execution profile，以及 ignored 且不在 classpath 中的输出目录。reader 不调用 bootstrap 重新编译 candidate。

26.6 最终 **output** 相同，事件顺序仍会改判决

运行同一条人工复核 graph 的两个版本。两者最后都是 `decision=approved`，唯一变化是 notification 的位置。




```
control: approve → notification → final approved
mutant:  notification → approve → final approved
```

`DurableTemporalContractTest.detectsAnEventThatOccurredBeforeItsRequiredActionDespiteFinalSuccess` 执行的是 mutant。最后一条 action 仍记录为 `COMPLETED`，但 report 为 `FAIL`，唯一 reason code 是 `TEMPORAL_EXPECTATION_MISMATCH`，并且 `notification-after-approval.matched=false`。

这一个测试把最终 JSON 会压扁的三件事重新分开：

- durable execution 已经完成；
- 最终业务值确实存在；
- 业务要求的事件顺序仍然被破坏。

BLOGE `cc38fbe5` 的聚焦运行包含 19 个 temporal-contract tests 和 11 个 durable-scenario tests，共 30 个全部通过，无 failure、error 或 skip。这里的证据只覆盖声明的有限关系，不代表探索过所有可能 interleaving。

26.7 **Temporal mismatch** 是业务 **FAIL**

合同有效、运行完成，但 `notification` 在 `approve` 之前发生时，状态为 `FAIL`，reason code 为 `TEMPORAL_EXPECTATION_MISMATCH`。这和 sidecar 无效不同：未知 action、错误 node identity、缺失 dependency manifest 或不支持的能力组合通常是 `INVALID`。

- **DURABLE_SIGNAL_TARGET_INVALID**——signal 没有精确命中 suspended node。核对 suspended set；不要把目标放宽成任意节点。
- **DURABLE_RESUME_STATE_INVALID**——对 suspended execution 使用了 RESUME。改用 SIGNAL；只对失败且有 checkpoint 的 execution 恢复。
- **DURABLE_CHECKPOINT_MISSING**——checkpoint 不存在。停止并检查 persistence；不要创建伪 checkpoint。
- **DURABLE_TIMER_SET_AMBIGUOUS**——一次推进同时到期多个 timer。当前 Preview fail closed；拆分计划或调整业务模型。
- **STREAM_DURABLE_UNSUPPORTED**——streaming graph 进入了 durable wrapper。使用 streaming 自身的测试边界；不要声明不存在的 durable 能力。
- **TEMPORAL_EXPECTATION_MISMATCH**——temporal 合同有效但事实不满足。把它作为业务 FAIL，分析最早偏离事件。

非协作客户代码超过 harness timeout 后，runner 会尝试中断并等待 settlement grace。若任务仍然不退出，报告 DURABLE_HARNESS_ORPHANED，该 runner 随后 poisoned；不要在同一进程继续接纳新 plan。

26.8 实验：信号路径与超时路径

1. 建立 approved-in-time plan：EXECUTE → ADVANCE_TIME PT23H → SIGNAL approved。
2. 声明 notification 不得早于 approve，且批准后在业务确认的逻辑时限内完成。
3. 建立独立 review-timeout plan：EXECUTE → ADVANCE_TIME PT24H，预期进入超时路径。
4. 先把 notification 移到 signal 之前，确认得到 FAIL/TEMPORAL_EXPECTATION_MISMATCH；恢复后再验证两条路径。
5. 保存 action trace、temporal expectation result 和第一条 reason code；不保存原始信号 payload 到报告。

实验停止条件：如果图包含未分类 streaming 节点、custom nested provider 无法静态检查，或 effect inventory 不完整，不创建 session。先修复能力声明，再讨论业务结果。

26.9 现实迁移：保险理赔等待补件

保险理赔可能都得到最终 APPROVED，但缺失材料是在复核期限前到达，还是在系统 已经升级人工后才到达，业务含义不同。把材料到达建模为 signal，把服务时限建模为 logical time，再声明“升级不得早于批准”的 temporal relation。必须保留事件轨迹；最终理赔对象无法区分这两段历史。

26.10 本章边界

- ADVANCE_TIME 只推进 verifier 持有的逻辑时钟，不模拟真实调度延迟。
- step/trace 合同证明声明的有限事件关系，不证明所有可能 interleaving。
- persistent provider 只提供每 Case store 和唯一 mutable-resource namespace；engine、action 顺序和关闭仍由 verifier 持有。
- source-bound manifest 保存初始 context 的 canonical SHA-256，不保存业务 payload。

26.11 实验验收卡

- 预期与观察：最终 output 相同，错误事件顺序仍使 temporal expectation FAIL。
- 失败与恢复：让 notification 早于 approval；恢复顺序后重放。
- 证明边界：证明固定 action plan 的时序，不证明全部并发交错。
- 练习合同：审批和通知事件；只交换顺序；交付 matched 前后结果；FAIL 只由顺序解释即停止。

26.12 本章小结

长流程的正确性由结果、状态边界和事件时间关系共同构成。有限 action plan 让这些事实可重复，temporal sidecar 让它们可诊断。下一章继续追问：这些 Case 即使全通过，是否真的有能力发现一个错误规则?

下一章：第 27 章——发现验证盲区

章节

27

第六阶段——业务承诺

第 27 章——合同敏感性：Suite 能拒绝错误答案吗?

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

本章承诺：把一个绿色贷款 *Case* 变成受控负向实验，分清 `KILLED` 见证与 `SURVIVED` 盲区，同时不把两者误写成穷尽式 *mutation coverage*。

27.1 学习目标

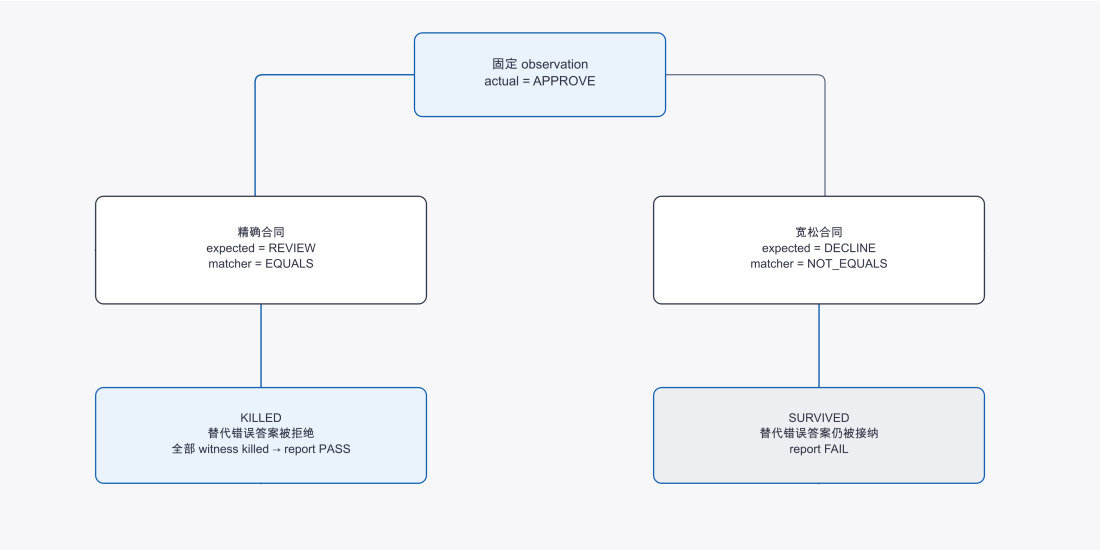
1. 解释“candidate 等于 expected”为何还不能证明合同会拒绝相邻错误答案。
2. 使用 `verifyContractSensitivity(Path)` 检查一个受治理的 `BUSINESS_CONTRACT` Expectation。
3. 分别读取 `KILLED`、`SURVIVED`、report status 和 source-bound evidence。
4. 设计业务 Owner 能审阅的单变量 witness。

27.2 同一个绿色 Case 可能藏着两种完全不同的合同

低风险贷款 Case 期望 `APPROVE`，graph 也返回 `APPROVE`。现在设想两个 matcher：

- 精确合同：actual 必须等于 `APPROVE`；
- 宽松合同：actual 只要不等于 `DECLINE`。

两者今天都会通过，但只有前者会拒绝 `REVIEW`。正向例子只能说明一个点匹配；sensitivity 问的是：选定错误答案是否越过合同边界。



27.3 改 **expected**，不改运行中的系统

sidecar 精确指向受治理的 Expectation，只替换它的 expected value：

```
schemaVersion: 1
sensitivityId: loan-decision-witnesses
scenario: loan-approval.scenario.yaml
mutants:
  - mutantId: low-risk-must-not-review
    requirementRef: LOAN-DECISION-001
    caseId: auto-approve
    expectationId: decision-approved
    kind: EXPECTATION_VALUE_REPLACED
    replacementExpected: REVIEW
```

```
VerificationContractSensitivityReport result =
        verifier.verifyContractSensitivity(sensitivityPath);
```

BLOGE 只执行一次 baseline Case，再把真实 observation 投影到替代 expected value 上。它不修改生产代码、不换一个 Operator 重跑，也不发明业务规则。这个隔离很重要：如果实现和 Oracle 同时改变，失败就无法归因给合同敏感性。

27.4 观察状态转移，不只看最后徽章

RC1 测试 `ContractSensitivityTest.passesOnlyWhenEveryRequiredWitnessIsKilled` 使用精确 `EQUALS` Expectation。graph 仍观察到 `APPROVE`；把 expected 换成 `REVIEW` 后会被拒绝，因此 witness 为 `KILLED`、report 为 `PASS`，`reasonCodes` 为空。

配对测试 `sealsASurvivingWitnessAsARecheckableSensitivityFailure` 从更宽的“not review”式合同开始，再把 expected 换成 `DECLINE`。同一个 observation 仍被接纳；witness 变成 `SURVIVED`，report 变成 `FAIL`，发布后的 artifact 仍可作为 `SOURCE_BOUND` evidence 被独立复核。

Observation	Witness outcome	Report	含义
替代 expected 被拒绝	`KILLED`	只有全部必要 witness 都被杀死才 `PASS`	合同能识别这个选定错误答案
替代 expected 仍被接纳	`SURVIVED`	`FAIL`	matcher 或 Oracle 边界对这个 witness 过宽
plan 指向缺失或不支持的目标	`NOT_EVALUATED`	`INVALID`	实验本身无效

这里故意不对称：存活 witness 是有价值的失败 evidence，不能因为原业务 Case 是绿色就被隐藏。

27.5 从业务错误设计 witness

不要枚举随机字符串，先找业务能命名的错误：

1. 选择一个受保护决策，例如“低风险申请不得进入人工复核”。
2. 固定 Scenario、Policy、Fixture、graph、execution profile 和 source commit。
3. 只把一个 expected value 换成一个可信的错误答案。
4. 请业务 Owner 判断该 witness 是否代表真实决策边界。
5. 通过评审后再把 witness 纳入版本控制。

贷款审批中，auto-approve Case 的 `REVIEW`、blacklist Case 的 `APPROVE` 都是有意义的 witness。胡乱填写的值可能只测到序列化，教不了读者业务合同。

27.6 实验：让一个 witness 存活

1. 运行低风险 Case，记录 baseline `PASS` 和 actual `APPROVE`。
2. 加入上面的 `REVIEW` replacement，确认精确 matcher 得到 `KILLED`。
3. 只把 matcher 改成也能接受 `REVIEW` 的宽松条件。
4. 再运行同一 sidecar，记录 `SURVIVED` 和 `CONTRACT_MUTANT_SURVIVED:<mutantId>`。
5. 恢复精确 matcher 并重跑，witness 必须回到 `KILLED`。
6. 把 artifact receipt 保存到运行目录之外，再交给另一个 reader 复验。

这是两次运行、一个变量的实验：witness 固定，只改变合同严格程度，因此结果可解释。

27.7 现实迁移：折扣规则敏感性

在零售定价中，保持购物车和实现不变，只把期望的会员折扣从 10% 替换为 30%。精确金额 matcher 应该杀死这个 witness；只检查 `discount > 0` 的宽松 matcher 会让它存活。这个结果只说明合同能否拒绝所选定价错误，不代表覆盖了所有促销组合。

27.8 本章边界

- Sensitivity 只覆盖作者声明的 mutation，不是全部规则和输出的 mutation coverage。
- `KILLED` 只证明拒绝一个选定错误答案，不证明输入域上的正确性。

- `SURVIVED` 定位歧义，但不替业务决定应该改 matcher 还是 Oracle。
- Source-bound 发布让结果可复核，不会让工具成为业务 Owner。

27.9 实验验收卡

- 预期与观察：同一 observation 在精确与宽松合同下得到 KILLED 与 SURVIVED。
- 失败与恢复：放宽一条 expectation 让 witness 存活；恢复精确答案。
- 证明边界：证明已声明 witness 的敏感性，不证明发现全部盲点。
- 练习合同：绿色 Case 和一个 witness；只改一条 expectation；交付状态转移；错误 witness 被 KILLED 即停止。

27.10 小结

绿色例子只说“这个答案匹配”；被杀死的 witness 增加“这个选定错误答案不会匹配”；存活 witness 暴露合同过宽的位置。第 28 章会再次换问题：不再只试一个人工挑选的错误答案，而是探索有限输入域并缩小真实失败。

下一章：第 28 章——有限属性验证与有用反例

章节

28

第六阶段——业务承诺

第 28 章——有限属性验证与有用反例

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

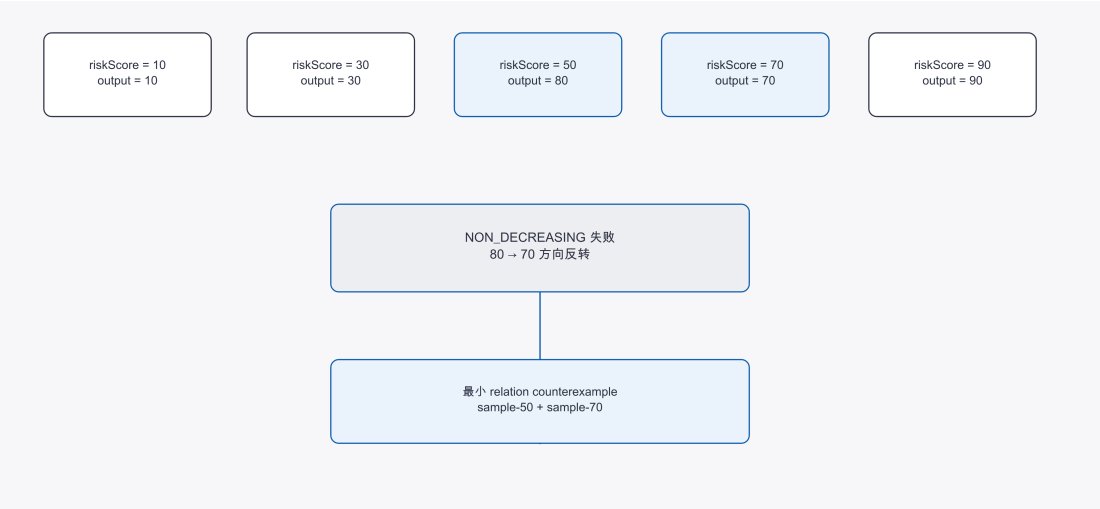
本章承诺：探索声明过的有限贷款风险域，分清 *sample failure*、*relation counterexample* 和 *value-shrink*，重放 *BLOGE* 的确定性收缩轨迹，并判断更小反例何时仍需业务评审。

28.1 学习目标

1. 定义附着在既有 Case 与 Expectations 上的 bounded generator。
2. 区分失败 sample、relation counterexample 与 value-shrink 结果。
3. 把预算耗尽和 shrink 不完整读成非通过结果。
4. 让 Oracle 保持足够独立，真正能挑战 candidate。

28.2 五个 **sample** 比一个例子回答得多，却仍不是证明

贷款团队已经信任三个命名 Case，但担心 risk-score 边界。他们保留 `auto-approve` 作为锚点，只把 `given.context.riskScore` 替换为五个声明值。




```yaml
schemaVersion: 1
propertySetId: loan-risk-boundaries
scenario: loan-approval.scenario.yaml
rootSeed: 11
sampleBudget: 5
generator:
  generatorId: risk-scores
  requirementRef: LOAN-DECISION-001
  caseId: auto-approve
  contextKey: riskScore
  integerRange: {minimum: 10, maximum: 90, step: 20}
  expectationIds: [decision-valid, risk-score-recorded]
relations:
  - relationId: risk-never-decreases
    kind: NON_DECREASING
    actual: {kind: BUSINESS_OUTPUT, outputId: computedRiskScore}
```

```
VerificationPropertyReport property = verifier.verifyProperties(propertyPath);
```

每个 sample 仍执行锚点 Case 的全部非诊断 Expectation。`rootSeed` 让受支持的样本选择和顺序可重复，但不控制客户 Operator 内的时钟、随机数、网络状态或环境读取。结论只能覆盖五个声明点，不能写成“全部整数风险分”。

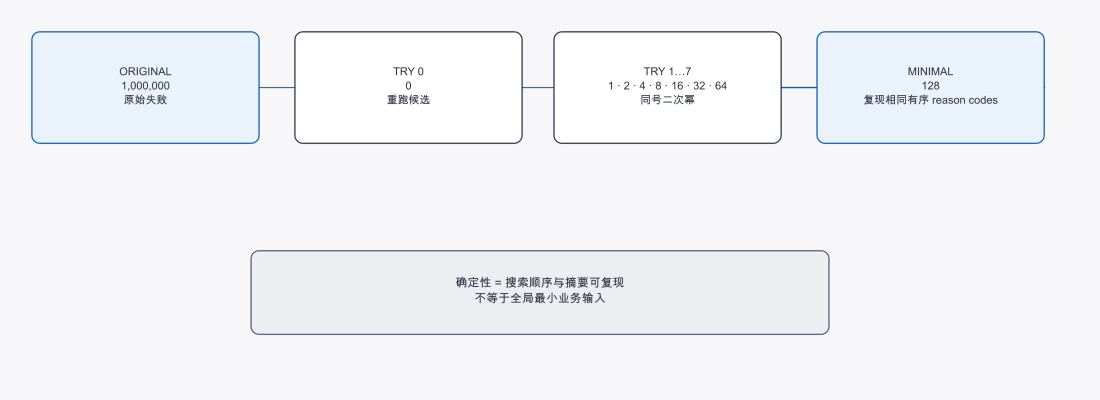
28.3 三种失败回答三个不同问题

Evidence object	失败对象	如何变小
sample	一个生成输入下的普通业务 Expectations	value shrink 可重跑候选
relation counterexample	已执行 samples 之间的 relation	选择更小的 sample ID 子集
property report	任一必要 sample、relation 或 shrink protocol	聚合 status 和稳定 reason codes

relation 的 `shrinkTrace` 不会发明输入。monotonicity 失败可以把执行集合缩到第一对能证明倒置的 sample。value shrink 不同：它在一个无 relation 的 sample 已失败后执行新的候选。

28.4 重放真实整数 shrink

对 `maximumAttempts: 64` 的无 relation 整数 plan，RC1 测试 `deterministicallyShrinksAnIntegerFailureByReexecutingCandidates` 连续两次观察到完全相同的序列：




```
1,000,000 → 0 → 1 → 2 → 4 → 8 → 16 → 32 → 64 → 128
```

第一个值是原始失败 sample。候选从零开始，再尝试同号二次幂。只有复现原始有序 reason codes 的候选才算有效。测试重跑整段实验，同时比较观察顺序和候选 context digest；两轮完全一致。

最后接纳的候选并不是“数学意义上最小的坏贷款”，只是当前候选顺序与失败等价规则产生的 minimal witness。

28.5 缩小 counterexample 不等于升级 Oracle

假设 `[10, 30, 50, 70, 90]` 得到 `[10, 30, 80, 70, 90]`。每个 sample 都可能通过本地 shape 检查，但 `NON_DECREASING` 会因 `80 → 70` 反向而失败。relation counterexample 可以只留下这两个 sample ID。

这对 sample 是诊断资产，不会自动成为新的 Golden Case。业务 Owner 仍要判断 relation 是否正确、生成 context 是否合规，以及约简输入是否值得获得一个批准过的 expected answer。

28.6 Shrink 可能结束却没有可用最小值

- 原始 sample 通过时，BLOGE 不尝试 shrink。
- 尝试预算耗尽且仍有候选时，结果是 `INCOMPLETE / RESOURCE_LIMIT_EXCEEDED`。
- 没有候选复现原始有序 reason codes 时，结果是 `INCOMPLETE / PROPERTY_SHRINK_INCOMPLETE`。
- 候选因另一原因失败时，它不是同一 bug 的 evidence。

这些结果都不能改写为业务 `PASS`。“诊断没有完成”和“已经验证正确”方向相反。

28.7 让裁判留在 candidate 之外

`verifyOracleIndependence(Path)` 分别报告三个可检查事实：

- `CONTENT_BOUND`：批准 Oracle 的字节匹配 catalog digest。

- `OWNER_SEPARATED`：catalog 的 owner 标签不同；这是声明，不是身份认证。
- `REFERENCE_MODEL`：绑定实现可在 child JVM 中运行，并检查其内容闭包与 candidate runtime、Operator 字节的关系。

evidence reader 只可重跑该 reference model，不运行 candidate Operators 或客户 bootstrap。字节不相交识别不了复制过的决策逻辑，所以概念独立性和 owner 身份仍是外部治理问题。

28.8 实验：把一次倒置变成评审卡

1. 运行五分值 plan，保存 sample IDs、outputs、root seed 和 receipt。
2. 只引入一处阈值倒置，保持其他输入固定。
3. 记录失败 relation 及其最小 sample-ID 子集。
4. 用独立的无 relation plan 触发 value shrink，不要混用两种机制。
5. 把完整候选顺序复制到评审卡。
6. 请业务 Owner 判断约简 witness 是否合法、有意义、值得进入 governed Case。
7. 恢复实现，确认 property 与原 Case 都重新通过。

28.9 现实迁移：缩小物流事件顺序错误

在物流追踪中，对有限事件序列声明 `deliveredAt >= shippedAt`。一旦某个序列违反关系，就在保留原 seed 和候选顺序的前提下，把它缩成最小可重放事件对。缩短后的轨迹只帮助诊断；它是否是合法、值得治理的 Case，仍由物流业务负责人判断。

28.10 本章边界

- Bounded property verification 只覆盖声明的有限 plan，不代表无限域或统计置信度。
- Relation reduction 在已执行 samples 中选择；value shrink 会创建并重跑候选。
- 确定性 shrink 只表示搜索顺序可复现，不表示全局最小业务输入。
- Oracle 字节隔离是有用 evidence，不证明人员独立或概念原创。

28.11 实验验收卡

- 预期与观察：有限域找到反例并把整数失败缩小到可评审候选。
- 失败与恢复：把 Oracle 放进 candidate 或让 shrink 停滞；恢复外部裁判和停止 reason。
- 证明边界：证明给定生成器与边界内性质，不是全域证明。
- 练习合同：一个性质和有限域；只改边界；交付 seed、反例、shrink 轨迹；候选稳定可重放即停止。

28.12 小结

Property verification 把一个命名 Case 扩成受控有限实验。counterexample 找到小而明确的已观察关系失败，value shrink 重跑候选并留下更小的可复现 witness。下一章会检查：文件、receipt、source 或 runtime context 改变后，这些结果还剩多少信任。

下一章：第 29 章——从报告到 Source-bound Evidence

章节
29

第六阶段——业务承诺

第 29 章——从报告到 Source-bound Evidence

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

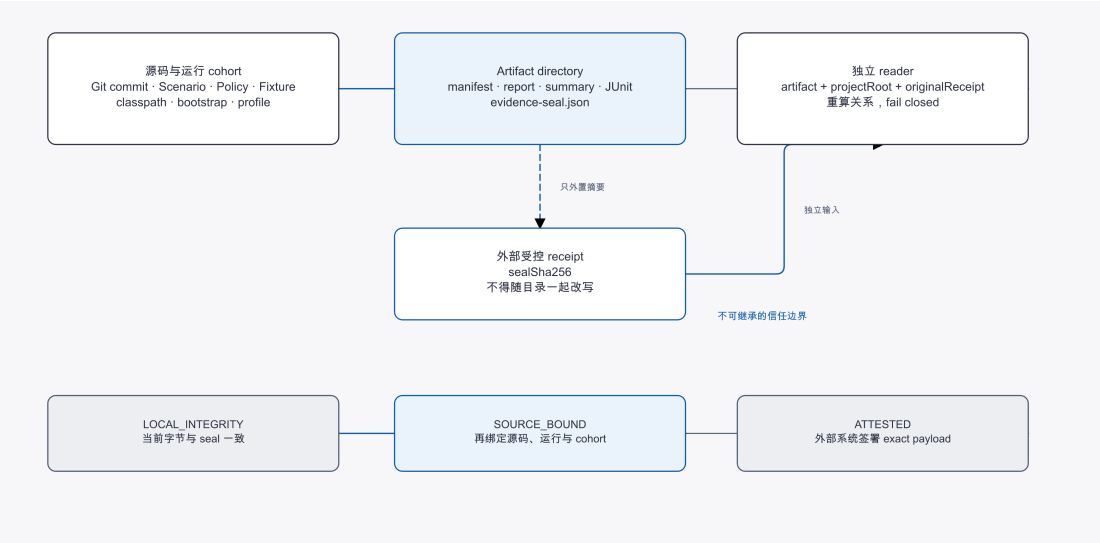
本章承诺：区分业务结果和密封目录，把 *receipt* 保存在目录之外，并用独立 *reader* 判断 *evidence* 只是字节完整、已经绑定源码，还是得到外部 *attestation*。

29.1 学习目标

1. 解释 report、manifest、seal、receipt 和 reader 各自的职责。
2. 区分 `LOCAL_INTEGRITY`、`SOURCE_BOUND` 和 `ATTESTED`。
3. 判断字节、目录位置、Git commit 或 runtime cohort 改变后会丢失什么 claim。
4. 通过 typed reader 读取 evidence，不把 JSON 字段直接解释成发布结论。

29.2 复制的绿色报告仍只是一份复制品

团队把开发机上的 `verification-report.json` 复制到发布机。文件仍写着 `PASS`，但它只保留了某人做过的声明；单独看这份文本，无法识别生成它的源码、toolchain、输入和运行。



橙色断点是刻意设计的：receipt 不能和可修改的 artifact 目录一起搬运。若能改写目录的人也能提供一份新计算的 receipt，这条边界就已经消失。

29.3 五个文件形成一份密封投影

发布后的 Suite 目录可以包含：

- `verification-manifest.json`：绑定输入、runtime identity、Git 和 run cohort 事实。
- `verification-report.json`：payload-free Suite、Case、Requirement、assurance、coverage 和专用结果。
- `verification-summary.md`：供人工审阅的投影。
- JUnit XML：相同 verdict 的构建系统投影。
- `evidence-seal.json`：所有发布文件的摘要和关系。

`VerificationArtifacts` 返回 run directory、run ID 和 `sealSha256()`。把这份 SHA-256 放进独立受控的 CI 记录；后续 reader 必须从目录之外取得它，作为 original receipt。

29.4 完整性与来源是两次不同升级

`LOCAL_INTEGRITY` 表示目录中的文件与 seal 一致。能改写目录的人仍能创建一份新的内部一致 seal。

`SOURCE_BOUND` 进一步绑定精确 tracked inputs、干净 Git commit 和 worktree、execution profile、classpath/runtime、bootstrap 和 Operator identity，以及该 evidence version 支持的 run cohort。

`ATTESTED` 要求外部受控 provenance system 用明确受信 key 和 domain 对精确 source-bound payload 签名。签名不能补齐缺失的 source binding。

业务 status 始终是另一条轴。可信 `FAIL` 可以是 source-bound；未绑定的 `PASS` 仍不适合作为团队门禁。

29.5 发布一次，从外部复验

运行前配置发布：

```
ScenarioVerifier verifier = ScenarioVerifier.builder()
        .bootstrap(new LoanVerificationBootstrap())
```

```
        .projectRoot(projectRoot)
        .policy(policyPath)
        .outputDirectory(projectRoot.resolve("target/bloge-verification"))
        .executionProfile(VerificationExecutionProfile.builder("ci-safe").build())
        .sourceBoundEvidence()
        .build();

VerificationReport report = verifier.verify(scenarioPath);
VerificationArtifacts artifacts = report.artifacts();
String externalReceipt = artifacts.sealSha256();
```

之后把三个独立输入交给 reader：

```
VerificationEvidenceTrustResult trust = VerificationSourceBoundEvidence.verify(
        artifacts.directory(), projectRoot, externalReceipt);
```

reader 重算 report、summary、JUnit、manifest、source 和 receipt 之间的关系，不运行 candidate Operator 或客户 bootstrap。Oracle v10/v11 是狭窄例外：reader 可以重跑显式绑定的 reference-model child JVM。

29.6 攻击五条边界，观察 **trust** 如何移动

每一行都从同一份未修改 artifact 开始：business status 为 `PASS`，receipt 从目录外提供，reader trust 为 `SOURCE_BOUND`。每次只改变一个变量。

单变量攻击	独立 **reader observation**	最高可辩护 **trust**	丢失的关系
替换外部 receipt	`EVIDENCE_RECEIPT_MISMATCH`	`UNVERIFIED`	密封字节快照身份
修改 summary、更新摘要并重封	`DURABLE_EVIDENCE_PROJECTION_MISMATCH`	低于有效本地证据	人类投影与机器投影的一致性
修改 report 语义、重建投影并重封	`DURABLE_EVIDENCE_INVALID`	低于有效本地证据	verifier 是否可能产生这种状态

单变量攻击	独立 **reader observation**	最高可辩护 **trust**	丢失的关系
修改 action digest、重建投影并重封	`SOURCE_BINDING_CASE_EVIDENCE_MISMATCH`	`LOCAL_INTEGRITY`	与 tracked action plan 的关系
发布后修改 tracked source	`SOURCE_BINDING_DRIFT`	`LOCAL_INTEGRITY`	与当前 source cohort 的关系

这些是实际观察到的 RC1 合同，不是假设标签。`DurableVerificationEvidenceTest.rejectsAProjectionMutationEvenWhenTheArtifactDigestWasUpdated` 证明更新 hash 不能让矛盾 summary 合法；`rejectsAContradictoryReportEvenAfterAllFilesAreReprojectedAndResealed` 证明完全自洽的伪造仍可能描述 verifier 不可能产生的状态；`DurableSourceBoundVerifierTest.rejectsReplacementReceiptBeforeTrustingSourceBoundDocuments` 会在信任 source-bound documents 之前先拒绝借来的 receipt。

source-drift 测试还给出一个重要运行观察：artifact 发布后修改 plan，reader 返回 `LOCAL_INTEGRITY / SOURCE_BINDING_DRIFT`，客户调用计数仍为 `1`。reader 在没有再次执行客户代码的情况下完成了降级。

29.7 为什么重封恢复不了 **provenance**

artifact owner 本来就能重算 hash，因此新 seal 有时能恢复字节一致性，却恢复不了外置 receipt、原始 run identity 或与 tracked source 的关系。reader 计算当前仍有证据支持的最高等级，不接受目录的自我描述。

移动或复制目录遵循同一原则。文件仍可阅读，但 source-bound 发布要求 bound worktree 内 ignored、non-classpath 的原位置。文件名和 JSON 字段保留下来，不代表副本继承决策权。

reader 返回稳定 diagnostics 并 fail closed。不要用文件名、绿色 `status` 字段或手工解析的 `evidenceTrust` 覆盖它的结论。

29.8 实验：一次攻击一条边界

1. 发布一份 source-bound 贷款 Suite，把 `sealSha256()` 保存在 run directory 之外。
2. 用原 project root 和 receipt 复验未修改 artifact，记录 `SOURCE_BOUND`。
3. 按表格完成五次攻击，每次攻击前都恢复未修改 artifact。
4. 记录 business status、reader trust、完整 reason codes 和客户调用计数。
5. 对重封攻击同时保留 original receipt 与 replacement receipt；分别复验并解释它们回答的问题为何不同。
6. 把目录复制到原 worktree 之外，确认可阅读文件不会继承 source-bound authority。
7. 最后填写一张卡：`status`、`trust`、receipt custodian、project commit、第一条失败关系和 rerun policy。

29.9 现实迁移：薪资证据保管

从另一个 worktree 复制来的薪资报告仍可能是合法 JSON，并显示每位员工计算都为 `PASS`，但它不能继承原运行的 authority。绑定薪资规则、员工输入摘要、runtime inventory、commit 和外置 receipt；再逐次移动或重封一个组件，记录第一条失效的 trust 关系。

29.10 本章边界

- Seal 证明当前字节间的关系，不识别生成者身份。
- `SOURCE_BOUND` 不证明外部账号身份、host 隔离、生产拓扑或 Policy 审批真实性。

- 复制 report 便于阅读，但不能继承其 source-bound 决策能力。
- 使用匹配的 reader 和 schema catalog；不要从类名或 JSON shape 猜 evidence version。

29.11 实验验收卡

- 预期与观察：五个文件形成 source-bound evidence，完整性与来源独立升级。
- 失败与恢复：替换 receipt、修改 source 或重封 identity；恢复原链或重新执行。
- 证明边界：证明固定 reader 可验证证据链，不证明客户代码从未被篡改。
- 练习合同：一份密封投影；每次攻击一个文件；交付 trust/reason；五种攻击均可归因即停止。

29.12 小结

artifact 文件让结果可检查，外部 receipt 让改写可检测，reader 再把两者接回当前源码事实。下一章先判断一条可信 Case 事实能否贡献给一条精确 Requirement；Project 与 Reactor 聚合留到第 31 章。

下一章：第 30 章——从 Case 事实到精确 Requirement Contribution

章节

30

第六阶段——业务承诺

第 30 章——从 Case 事实到精确 Requirement Contribution

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

本章承诺：为业务 *Requirement* 建立精确身份，把它绑定到批准的 *Oracle* 字节和 *Expectation* 语义，并解释为什么同样引用该规则的 *passing Case* 仍可能没有任何贡献。

30.1 学习目标

1. 区分 Requirement association 与 governed contribution。
2. 用显式 evaluation context 选择唯一 versioned partition。
3. 绑定精确 Expectation 语义和 Oracle revision，而不只绑定名称。
4. 正确读取 bootstrap 前的 governance failure，不把它叫作业务失败。

30.2 “提到规则”不等于“满足规则”

三个贷款 Case 都提到 `LOAN-DECISION-001`。但本次发布 obligation 是 version 3、`cn-retail` partition、release `2026.09`。Case 即使通过，也可能因为属于 version 2、EU partition、retired Oracle、不同 Expectation digest 或 assurance depth 不足而完全不贡献。

这个区别阻断了一个简单但危险的捷径：在 report 中统计字符串，再宣布 Requirement 已满足。

30.3 建立不会碰撞的身份

```yaml
schemaVersion: 1
catalogId: loan-governance
requirements:
  - requirementRef: LOAN-DECISION-001
    version: 3
    owner: risk-policy-team
    status: ACTIVE
    effectiveFrom: 2026-09-01T00:00:00Z
    partitions:
      - partitionId: cn-retail
        dimensions: {jurisdiction: CN, product: RETAIL_LOAN}
        targetReleaseIds: ['2026.09']
        obligations:
          minimumPassingCases: 2
          minimumDepth: L2
          allowedOracleTypes: [APPROVED_EXAMPLE]
```

有效身份是：

```
catalogId + requirementRef + version + partitionId
```

`GovernanceEvaluationContext` 提供 `asOf`、target release 和有限 dimensions。必须且只能有一个 partition 适用。裸 `LOAN-DECISION-001` 不能借用另一个版本或市场的 evidence。

RC1 测试 `producesOnlyTheExactApplicableVersionedPartitionContribution` 只观察到一个 key：`loan-governance / LOAN-R-017 / v3 / cn-retail`，`approve-low-risk` 同时是 associated 与 contributing Case。同 ref 的 version 2 或 `eu-retail` 是不同 key。选择 release `2026.10` 会返回 `REQUIREMENT_NOT_APPLICABLE`，且不会再次调用客户 bootstrap。

30.4 绑定答案，而不只绑定标签

binding 固定：

- `suiteId + caseId`;
- 精确 `expectationIds` 与规范化 `expectationSha256`;
- `oracleId + oracleRevision` 与批准 artifact digest;
- minimum assurance 与 allowed Oracle type。

这关闭了“名称相同、含义已变”的漏洞。在 `bindsExactExpectationSemanticsAndReportsInsufficientAssurance` 中，只把 expected `APPROVE` 改成 `REVIEW`，所有名称都不变，仍会在 bootstrap 前得到 `INVALID / GOVERNANCE_BINDING_INVALID`。把 required depth 从 `L0` 提高到 `L4` 后允许执行，但 evidence 会记录 `DEPTH_INSUFFICIENT` 和 `contributes=false`。

Case 状态	Associated?	Contributes?	原因
ref、version、partition、binding 精确匹配且 assurance 足够	是	是	governed business evidence 匹配
同 ref、错误 partition	文字上是	否	`GOVERNANCE_BINDING_MISSING`
名称相同、expected 语义改变	是	否	`GOVERNANCE_BINDING_INVALID`
binding 正确、depth 不足	是	否	`DEPTH_INSUFFICIENT`
Oracle 过期或 digest 不匹配	没有可用贡献	否	`ORACLE_GOVERNANCE_INVALID`

30.5 跟踪一个贷款 Case 通过 contribution filter

1. Scenario 把 `auto-approve` 关联到 `LOAN-DECISION-001`。
2. Evaluation context 选择 version 3、`cn-retail`、release `2026.09`。
3. Binding 找到精确 Suite、Case 和 business Expectation digest。
4. Oracle catalog 匹配批准内容与 revision。
5. Verifier 执行 Case，并从观察到的控制推导 assurance。
6. 到这里，`verifyVersionedGoverned(Path)` 才输出 contribution。

Contribution 故意只属于一个 Suite。它描述该 Case 提供了什么，还不裁决跨 Project 或 module 所需的两个 Case 是否齐全。

30.6 实验：保留 ref，破坏含义

1. 运行有效的 version-3 `cn-retail` Case，记录 Requirement key 与 contribution。
2. 只把 partition context 改成 EU，确认 `GOVERNANCE_BINDING_MISSING` 和零次 bootstrap。
3. 恢复 context，只把 expected `APPROVE` 改成 `REVIEW`，ID 全部保留；确认 `GOVERNANCE_BINDING_INVALID`。
4. 恢复语义，把 `minimumDepth` 提高到 observation 之上；确认发生执行，但 contribution 因 `DEPTH_INSUFFICIENT` 被拒绝。
5. 填写四列评审卡：associated、evaluated、contributes、first reason。

30.7 映射到你的业务

税务规则、医疗资格、欺诈冻结或账户注销都适用同一模型。为你的规则写一个精确身份，再指出一个同 ref Case：它必须因为市场、版本、Oracle 或 assurance 不同而不能贡献。

30.8 本章边界

- Requirement ref 只是 association hint，不是 satisfaction evidence。
- 一个 Suite contribution 无法独自满足跨 Suite 的 minimum。
- Owner label 和 approved digest 是治理事实，不是人员身份认证。
- Governance preflight failure 表示实验无效，不是客户决策 verdict。

30.9 实验验收卡

- 预期与观察： Requirement 的 identity 与 semantics 同时匹配才算 contribution。
- 失败与恢复：保留 ref 但改变含义；恢复精确版本和语义绑定。
- 证明边界：证明 contribution filter，不证明 Owner 已批准 Requirement。
- 练习合同：一个 Requirement 和 Case；只改 semantics；交付 accepted/rejected；同名异义被拒绝即停止。

30.10 小结

Governance 把熟悉的 Requirement 名称变成精确 obligation，再从 passing Cases 中筛出有效 contribution。第 31 章补上完整性事实源：Project 与 Reactor child 必须来自同一冻结 cohort，之后才能推导 capability 和 gap。

下一章：第 31 章——Project、Reactor 与 Claim Capability

章节

31

第六阶段——业务承诺

第 31 章——Project、Reactor 与 Claim Capability

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

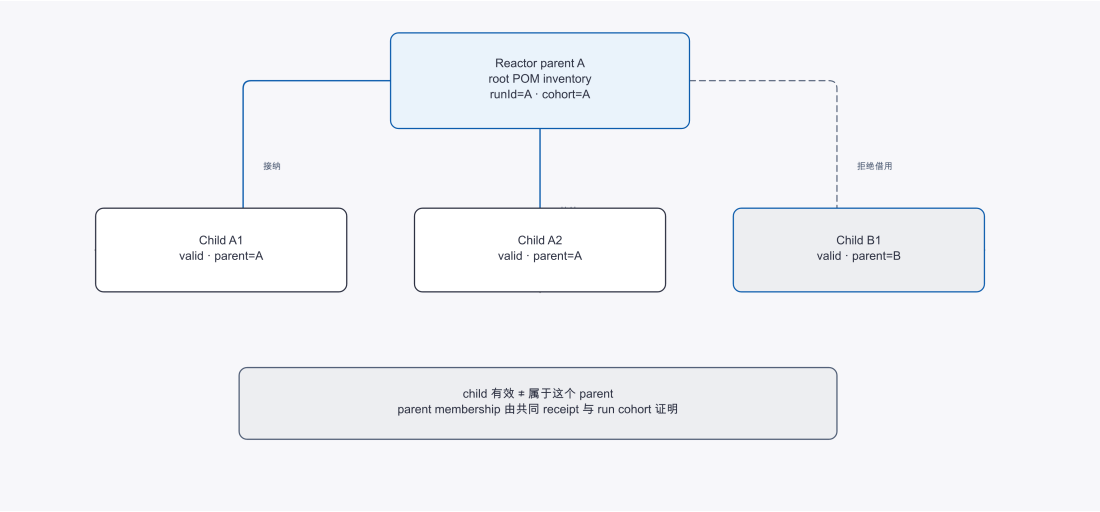
本章承诺：只从完整、冻结的 *cohort* 聚合精确 *Requirement contributions*，拒绝借来的 *child*，最后输出 *evidence* 支持的最高决策能力和明确 *gap*。

31.1 学习目标

1. 解释 aggregation 为什么需要权威 inventory 与单一 run cohort。
2. 区分 Project completeness 与 direct-module Reactor completeness。
3. 在相信总数前观察 cohort mismatch 与 borrowed-child rejection。
4. 把 `ClaimCapability` 和 `missingEvidence` 作为 reader 推导结果。

31.2 两个有效 **child** 仍可能组成无效 **parent**

decision module 贡献一个有效贷款 Case，funding module 贡献另一个；两者合起来满足 `minimumPassingCases: 2`，但前提是它们属于同一个 governance snapshot、selection、source commit 和 parent run。



这是 aggregation 版本的复制粘贴 report 问题。child 有效是必要条件，parent membership 是另一项事实。

31.3 Project aggregation 在执行前冻结 discovery

```
VerificationGovernedProjectEvidence evidence =
        VerificationProjectRunner.runGovernedEvidence(
                scenarioDirectory,
                verifier,
                VerificationSelection.all(),
                projectIdentity,
                outputDirectory);
```

runner 在计算 Requirement 前冻结 Scenario discovery、effective selection、governance context、project POM identity 和 source-bound child receipts。required binding 缺失或被 filter 排除时，它会报告 `GOVERNANCE_BINDING_MISSING` 或 `GOVERNANCE_BINDING_FILTERED`，不会为了适配现有 child 而降低 minimum。

RC1 测试 `aggregatesDistinctSuiteCasesAndRejectsAMissingGovernedChildBeforeBootstrap` 保护这个区别：不同 Suite contribution 只能通过拥有完整 inventory 的 Project aggregate 合并。

31.4 Reactor 增加另一种 completeness authority

`VerificationReactorRunner.runGovernedEvidence(...)` 读取 root `pom.xml` 的 direct `<modules>` 列表。交给 runner 的每个 module 都必须匹配该 inventory、literal GAV、Suite/Case inventory、共同 Git commit、selection 与 governance cohort。

在 `recomputesOneGlobalRequirementFromTwoIndependentModules` 中，任一 module 单独运行都会得到 `INVALID / GOVERNANCE_BINDING_MISSING`，且调用数为零。Reactor run 各执行一次，列出两个 qualified Suite identity，再重算一个拥有两个 satisfied Cases 的 Requirement。

当前 Preview 只覆盖同一 Git worktree 中的 direct modules。Nested reactor、任意 Maven profile、external-parent effective model 和 deployment topology 不在该 completeness claim 中。

31.5 先攻击 cohort，再读取总数

测试 `rejectsAReactorGovernanceCohortMismatchBeforeCustomerBootstrap` 只改变第二个 module 的 `catalogId`，结果是：

```
status       = INCOMPLETE
reasonCodes  = [REACTOR_GOVERNANCE_COHORT_MISMATCH]
module calls = 0 + 0
```

runner 在客户 bootstrap 前拒绝 parent。这比两次昂贵或带 effect 的执行结束后才发现不匹配更强。

另一项攻击先运行两个有效 Reactor parent，再把第一个内存 parent 的一个 Suite 换成第二个 parent 的有效 Suite。`reactorClaimRejectsAValidChildBorrowedFromAnotherParentRun` 把 capability 降到 `AUTHORING`，并重新把 `SOURCE_BOUND_PROJECT_AGGREGATE` 列为 `TEAM_GATE` gap。有效 child 加错误 parent 仍是无效 aggregate。

31.6 让 Claims 明确剩余决策边界

```
VerificationClaimAssessment assessment =
        VerificationClaims.assess(evidence, projectRoot);

ClaimCapability current = assessment.capability();
Map<ClaimCapability, List<VerificationEvidenceGap>> gaps =
        assessment.missingEvidence();
```

Capability	Evidence 支持的决策语境	常见剩余 gap
`AUTHORING`	本地诊断与 witness discovery	source-bound Suite/Project evidence
`TEAM_GATE`	带 continuity、replay 和 CI-safe control 的团队集成	governed aggregate 或 Oracle provenance
`GOVERNED_GATE`	versioned business obligation 与可信 Oracle provenance	attributed comparison 与 release attestation
`RELEASE_QUALIFIED`	为发布签署的精确 governed evidence 与 comparison	声明 release contract 内没有缺项

Gap 是累计的。写进 JSON 的字符串不能提升 evidence，Policy 也不能给自己颁发 capability。可信 `FAIL` 可以有很强 provenance；capability 仍不替代 business status。

31.7 实验：借用一个 **child**

1. 运行一个双 module parent，并外置保留 receipt。
2. 把同样两个 modules 再运行到第二个 parent directory。
3. 确认两个 parent 都能独立复验。
4. 用 parent B 的有效 child 替换 parent A 的一个 child projection。
5. assess 混合 parent，记录 `AUTHORING` 与 `SOURCE_BOUND_PROJECT_AGGREGATE`。
6. 恢复 parent A，画出 ownership links：root POM → parent run ID → child receipt → Suite contribution。
7. 用 gap 卡结束：current capability、desired capability、missing evidence、owner、next experiment。

31.8 现实迁移：多模块计费发布

假设计税、开票、账本三个 module 都发布了有效 child evidence。把昨天 parent 的 计税 child 与今天的开票、账本 child 拼在一起，文件仍可读，却已经混合 cohort。只有 root build identity、parent run 和 child receipt 的 ownership 同时成立，才能声明项目完整；否则停在 `AUTHORING`，并写明缺失证据的 owner。

31.9 本章边界

- 有效 child evidence 不证明它属于某个特定 parent run。
- Project completeness 不代表 direct-module Reactor completeness 或 deployment completeness。
- 只有通过 inventory 与 cohort 检查，Reactor total 才有意义。
- `ClaimCapability` 说明支持的决策语境，既不改变 verdict，也不批准发布。

31.10 实验验收卡

- 预期与观察：两个有效 child 仍可能让 parent capability 留下聚合缺口。
- 失败与恢复：借用另一 cohort 的 child；恢复冻结 discovery 集。
- 证明边界：证明 Project/Reactor completeness，不证明组织接受剩余风险。
- 练习合同：一个 parent 和两个 child；只替换一个 child；交付 capability gap；borrowed child 被识别即停止。

31.11 小结

Aggregation 不是简单相加，而是在权威 inventory 和单一冻结 cohort 下相加。Claims reader 再说明 evidence 能走多远，以及什么仍阻挡下一决策。第 32 章会执行最后两次受控实验：一个 DSL change 和一个 BLOGE toolchain change。

下一章：第 32 章——变化归因、Maven 与发布决策

章节

32

第六阶段——业务承诺

第 32 章——变化归因、Maven 与发布决策

受控 Scenario、Source-bound Evidence、治理与归因

目标：证明经批准的业务承诺，同时不夸大证据能力。

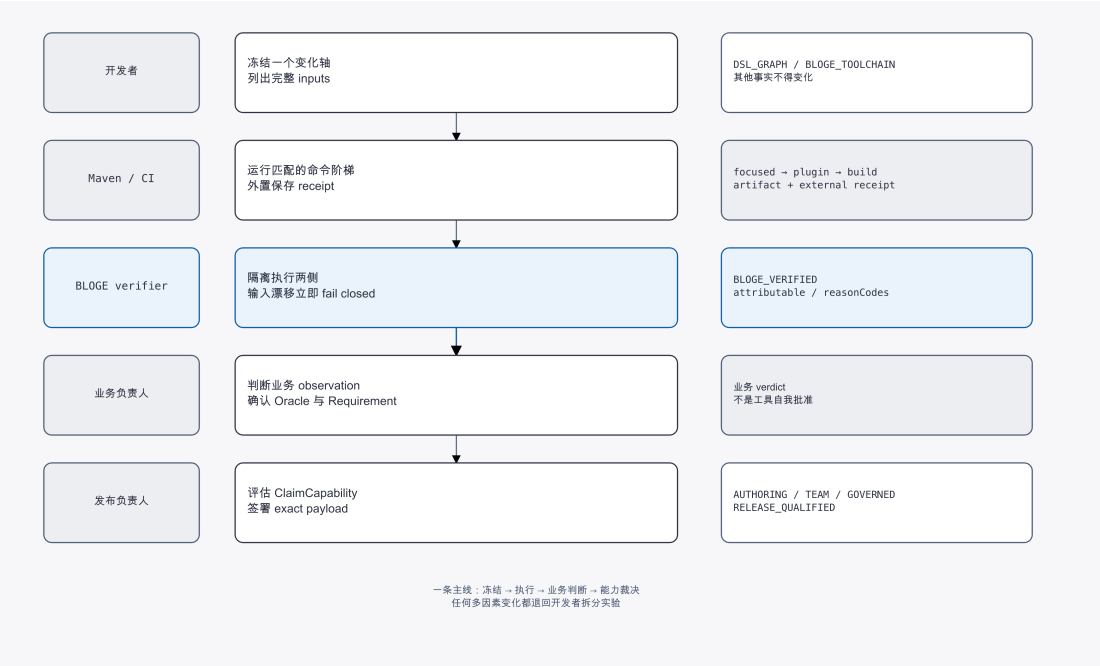
本章承诺：固定一个变化轴，运行与问题相称的最小 *Maven* 阶梯，并判断现有 *evidence* 只支持本地诊断、团队门禁、治理审批，还是发布决策。

32.1 学习目标

1. 分开 DSL graph 变更、BLOGE toolchain 升级和业务代码变更。
2. 区分“观察到差异”和“BLOGE 能归因差异”。
3. 按需要支持的 claim 选择 Maven 命令，而不是按控制台有多绿来选择。
4. 保存发布决策所需的 receipt 和 external attestation。

32.2 升级改变了结果，但原因是什么？

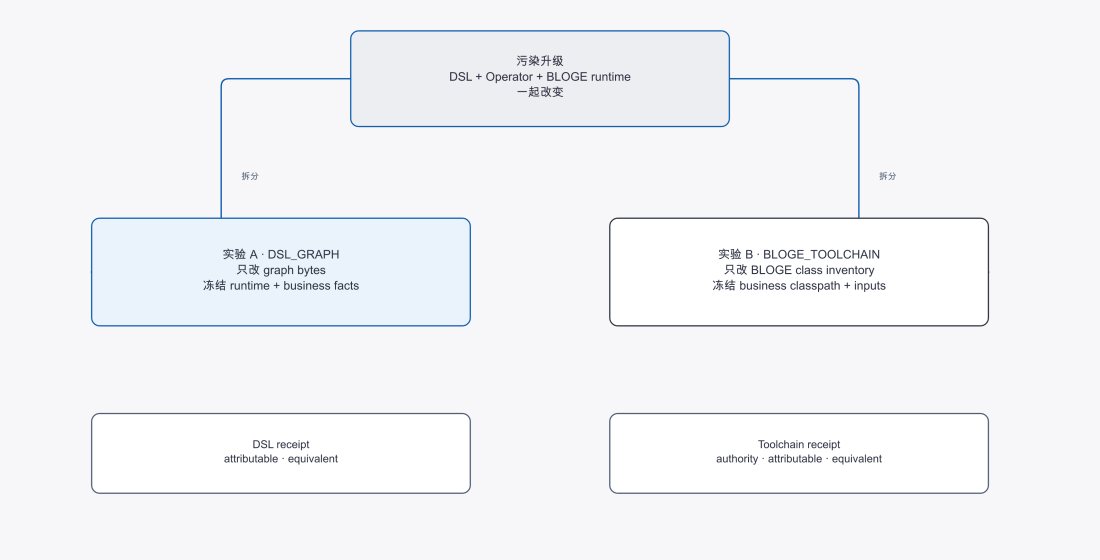
贷款团队升级到 BLOGE `0.9.8-RC1` 后，边界申请从自动批准变为人工复核。同一分支里还有人修改了 risk Operator。前后测试能显示差异，却不能把差异归给其中一项变更。



归因需要受控实验：只改变一个声明轴，冻结所有重要的非目标事实。Operator 和 runtime 同时变化时，先停在诊断层，把实验拆成两次 comparison。

32.3 把一次升级拆成两张 **receipt**

“升级分支”不是有效 change axis，它混合 source edit、graph edit、dependency resolution 和 runtime bytes。贷款团队从同一 common baseline 建立两个实验：



实验	唯一变化事实	必须冻结的事实	诚实结果
A —`DSL_GRAPH`	baseline graph → candidate graph	BLOGE runtime、Scenario、Policy、Fixture、bootstrap、registry、profile	graph 差异可归因，或 `COMPARISON_INPUT_DRIFT`

实验	唯一变化事实	必须冻结的事实	诚实结果
B —BLOGE_TOOLCHAIN	baseline BLOGE class inventory → RC1 inventory	business classpath、graph、声明输入、Java launcher/protocol	只有 BLOGE_VERIFIED observation 才能归因

实验 A 对应 attributesAnObservedBusinessRegressionToTheDslGraphAxis：只改变 candidate graph 的 decision input。两侧各执行一次；candidate report 为 FAIL，change code 包含 OUTPUT_DIGEST_CHANGED:approve-low-risk/decision，comparison 可归因但不 equivalent。

控制实验 attributesOnlyDslGraphChangeWhenEveryFrozenExecutionFactMatches 改变 graph bytes，却保持 observed contract 不变；结果同时 attributable 和 equivalent。因此 attributable 与 equivalent 是两个独立问题。

实验 B 对应 blogeOwnedChildExecutesTheFrozenScenarioBeforeAttributingTheToolchainAxis：固定 business classes 与声明输入，改变 runtime inventory，由两个终止式 child JVM 返回 BLOGE_VERIFIED observations。结果 attributable、observationally equivalent、equivalent，并发布为 SOURCE_BOUND。把 artifact 复制到 bound project location 之外，会得到 COMPARISON_SOURCE_BINDING_DRIFT。

再攻击一个冻结事实：businessClasspathMutationIsReportedAsInputDrift 在 toolchain 实验中改变 business classpath marker。结果变成 COMPARISON_INPUT_DRIFT 与 attributable=false。runtime upgrade 可能仍值得研究，但这次运行已经不能声称它导致了 observation。

32.4 只比较一个 DSL graph 变化轴

只变 graph 时，两次执行复用同一个 canonical Scenario 和 verifier 的 source-bound 环境：

```
ComparisonPlan plan = ComparisonPlan.dslGraph(
        "loan-risk-threshold-v3",
        scenarioPath,
        baselineGraphPath,
        candidateGraphPath);

VerificationAttributedComparisonEvidence evidence =
        verifier.compareEvidence(plan);
```

compareEvidence(...) 只在 DSL_GRAPH 可归因时发布。它会冻结 Scenario、Policy、Fixture、selected Cases、execution profile、bootstrap、Operator registry、service identity、governance inputs 和 Git source binding。非目标事实漂移会产生 COMPARISON_INPUT_DRIFT，只保留内存诊断。

attributable() 回答观察差异是否属于声明轴。equivalent() 更严格：必须可以归因、两侧都通过，且 payload-free observations 一致。输出相同不证明 candidate 更正确，只说明当前合同范围内没有观察到回归。

32.5 用冻结的业务代码比较两套 BLOGE toolchain

runtime 升级需要两个执行后终止的 child JVM，并共享一份 business runtime：

```
ToolchainBusinessRuntime business = ToolchainBusinessRuntime.of(
        businessClasspath,
```

```
        "com.acme.loan.LoanApprovalVerificationBootstrap");

ToolchainComparisonPlan plan = ToolchainComparisonPlan.blogeVerification(
        "bloge-0.9.7-to-0.9.8-rc1",
        inputs,
        business,
        ToolchainEndpoint.blogeVerifier("baseline", baselineToolchain),
        ToolchainEndpoint.blogeVerifier("candidate", candidateToolchain));

VerificationToolchainComparisonEvidence evidence =
        PairedExecutionCoordinator.builder()
                .projectRoot(projectRoot)
                .build()
                .compareEvidence(plan, outputRoot);
```

`inputs` 必须包含 `scenario`、`policy` 和所有被引用的 DSL、Fixture、Oracle 文件。两个 child 使用 coordinator 的 Java launcher、同一 ordered business classpath，以及不同的 BLOGE class inventory。固定 BLOGE child 按 project-relative path 重建文件，再运行真实 `ScenarioVerifier`。

自定义 `javaMain(...)` adapter 只有 `PROVIDER_ATTESTED` 权威。它可以报告 `observationallyEquivalent()`，但不能让 `attributable()` 或 `equivalent()` 为 true。只有 `blogeVerifier(...)` 能提供足以归因的 `BLOGE_VERIFIED` observation。

32.6 运行与问题匹配的 Maven 阶梯

1. 聚焦故事检查——快速反馈。运行贷款 starter 的三个测试。11 个 passing tests 覆盖选中的故事，不代表整个 Reactor。

   ```
   mvn -pl bloge-starter-loan-approval -am \
     -Dtest='LoanApprovalApplicationTest,LoanApprovalBusinessScenarioTest,LoanApprovalOperatorFlowTest' \
     -Dsurefire.failIfNoSpecifiedTests=false test
   ```

2. 源码检出时先安装 **Preview plugin**。如果 `0.9.8-RC1` 尚未发布到你使用的 Maven 仓库，先从同一源码快照安装。`invoker.skip` 只跳过插件自己的集成 fixture；它不能替代后面的完整 Reactor 检查。

   ```
   mvn -pl bloge-maven-plugin -am install \
     -DskipTests -Dinvoker.skip=true
   ```

3. 发布 **customer module verification artifacts**——**Preview**。在客户 module 中，把编译与 goal 放在同一次 Maven 调用里。goal 读取 `.blogeverify.yaml`，写入配置的 report directory。

   ```
   mvn test-compile \
     com.leanowtech.bloge:bloge-maven-plugin:0.9.8-RC1:verify
   ```

   这里的 `PASS` 仍受 config、source binding、receipt custody 和 evidence capability 限制。receipt 应保存在可变 artifact directory 之外。

   > ***RC1*** 已知边界：本书贷款 *starter* 的 *bootstrap* 包名是 `com.leanowtech.bloge.starter.*`。`0.9.8-RC1` 的 *plugin classloader* 会把该前缀误判为 *plugin-owned class*，因此对这个 *starter* 执行 *goal* 会 *fail closed*：`TEST_CLASSPATH_UNAVAILABLE`。这不等于 *Scenario* 失败；当前可运行入口是第 *1* 步的 *JUnit* 测试。通

用客户包名的 *Plain Java* 和 *Spring bootstrap* 由 *plugin Invoker fixture* 覆盖。修复类加载边界前，不把 *starter* 写成 *artifact* 发布成功案例。

4. 检查完整 **Java build**。
   `mvn clean install`
   这会检查 Reactor build，不会创建缺失的业务 Requirement，也不会批准 Oracle。`verify-reactor` 是另一条 Preview 路径，需要 schema-v2 per-module governance 配置，以及同一 Git worktree 内的 direct modules。本书的贷款 starter 配置不声称已有这份 Reactor evidence。

32.7 把 evidence 转成决策

读取现有 evidence 支持的最高 `ClaimCapability`，同时保留它的边界：

- 本地诊断——**`AUTHORING`**：comparison 没有 source binding、没有固定单轴，或缺少 continuity evidence。先修实验，再讨论因果。
- 团队门禁——**`TEAM_GATE`**：具备 source binding、Policy continuity、replay assurance 和 CI-safe execution。它支持团队集成，不代表业务治理审批。
- 治理审批——**`GOVERNED_GATE`**：可信 Project evidence 还满足 versioned Requirements 和所需 Oracle provenance。
- 发布决策——**`RELEASE_QUALIFIED`**：governed Project evidence、exact attributable comparison receipt 和可信 external release attestation 绑定同一 change axis 与 Oracle closure。

external signature 只能证明 signer 认可 exact digests，不能证明未声明行为、生产兼容性，也不能证明业务政策本身合理。

32.8 实验：拆开一次被污染的升级

1. 保持同一 Scenario、Policy、Fixture 集、bootstrap 和 source-bound verifier，只比较 `baseline-risk.bloge` 与 `candidate-risk.bloge`。
2. 记录 `changeAxis()`、`attributable()`、`equivalent()`、reason codes 和 external receipt digest。
3. 恢复 DSL。冻结 business classpath，再用 `blogeVerification(...)` 比较两套不同的 BLOGE toolchain inventory。
4. 同时修改 risk Operator 和 BLOGE runtime。确认这不是合法的 single-axis plan，不要把观察差异称为 runtime regression。
5. 把被污染的变更拆成 business-code experiment 和 toolchain experiment，请业务负责人判断变化后的业务 observation。
6. assess 当前 governed 和 comparison evidence。在发布负责人签署前，写出最高支持能力和全部剩余 gap。

最后交付两张独立 receipt 和一张 gap card，而不是一片绿色 console：

```
DSL receipt       axis · baseline digest · candidate digest · attributable · equivalent
Toolchain receipt axis · runtime digests · business-classpath digest · authority · attributable
Release gap card  current capability · missing evidence · owner · next action
```

32.9 现实迁移：支付网关升级归因

同一发布分支如果同时升级支付 SDK 并修改 retry policy，成功率上升不能归因给其中 任何一项。先冻结业务代码，只比较 SDK inventory；再恢复 SDK，只比较 retry policy。保留两张独立 receipt，由发布负责人根据两个有边界的结果判断，不读取一张 被多变量污染的 dashboard。

32.10 本章边界

- semantic difference 不会自动成为 attributable difference。
- `PROVIDER_ATTESTED` protocol correctness 不是 BLOGE-owned execution authority。
- `mvn clean install` 是 build evidence，不是 Requirement satisfaction。
- attributable comparison 单独仍为 `AUTHORING`；release qualification 还需要 governed Project evidence 和可信 external attestation。
- 本章任何 comparison 都不能证明声明的 Scenario、Oracle、inputs 和 runtime 边界以外的正确性。

32.11 实验验收卡

- 预期与观察：升级拆成 DSL_GRAPH 与 BLOGE_TOOLCHAIN 两张单轴 receipt。
- 失败与恢复：同时改 classpath 和 toolchain；冻结一轴后分别比较。
- 证明边界：证明固定输入的变化归因，不证明新版本可发布。
- 练习合同：前后两个版本；每次只变一轴；交付两张 receipt；每个差异只有一个候选原因即停止。

32.12 小结

冻结除一个声明轴以外的全部事实。DSL graph 使用 `ScenarioVerifier.compareEvidence(...)`，BLOGE-owned toolchain comparison 使用 `PairedExecutionCoordinator.compareEvidence(...)`。再用 Maven 阶梯和 `VerificationClaims` 只报告当前 evidence 真正支持的决策能力。第六阶段在负责任的发布判断开始处结束：留下可归因、有边界、可独立复核的 evidence。

下一步：第六阶段回顾——从业务承诺到发布证据。

章节

—

第六阶段回顾——业务证据

第 23–32 章始终追踪一个承诺：graph、runtime 或环境变化后，贷款决策仍应符合业务负责人批准的业务合同。

0.1 十章证据路径

1. 限定 **claim**——第 **23** 章。分开 verdict、evidence trust 和 decision capability。
2. 运行第一个故事——第 **24** 章。把批准、拒绝、人工复核映射为 Suite、Case 和 Expectation。
3. 控制输入——第 **25** 章。用 Policy、Fixture、execution profile 和 assurance 拒绝假绿。
4. 让时间可信——第 **26** 章。观察 suspend、signal、resume、logical time 和 temporal contract。
5. 杀死弱 **mutant**——第 **27** 章。测量 suite 能否察觉选定的受控业务规则变化。
6. 搜索有限领域——第 **28** 章。把 property 展开成有限 cases，保留 counterexample，并明确 Oracle boundary。
7. 绑定源码——第 **29** 章。把 artifact 连接到 source、run cohort、seal、external receipt 和 independent reader。
8. 精确贡献——第 **30** 章。判断可信 Case 事实能否贡献给一条 versioned Requirement。
9. 聚合同一 **cohort**——第 **31** 章。从权威 inventory 推导 Project/Reactor completeness 与 capability，再写出剩余 gap。
10. 归因变化——第 **32** 章。冻结一个变化轴，完成 comparison，只报告现有 evidence 支持的最高发布能力。

0.2 用三个轴读取每个结果

- `VerificationStatus`：本次 Case 发生了什么——`PASS`、`FAIL`、`INVALID` 还是 `INCOMPLETE`？
- `EvidenceTrust`：独立 reader 能否把结果绑定到声明的 source 和 run inputs？
- `ClaimCapability`：这份 evidence 只支持 `AUTHORING`，还是也支持 `TEAM_GATE`、`GOVERNED_GATE` 或 `RELEASE_QUALIFIED`？

不要把三个轴压成一个绿色徽标。

0.3 发布负责人的快速检查

做发布决策前，逐项指向具体事实：

1. active Requirement version、partition、binding 和 approved Oracle closure。
2. 完整、同 cohort 的 governed Project aggregate。
3. 保存在可变 artifact directory 外的 source-bound child receipts。

4. 一份可归因的 `DSL_GRAPH` 或 `BLOGE_TOOLCHAIN` comparison。
5. 可信 external attestation 绑定 exact governed receipt、comparison receipt、change axis 和 Oracle closure。
6. 明确的证明边界：这份 evidence 仍不能证明哪些生产事实。

缺少任何一项，就保留较低 capability，并写出 gap。不要用协商把 enum 抬高。

0.4 最终挑战：先分类，再声称

一个 candidate build 的贷款 Cases 全部通过，并有有效 local seal；但 business Operator 和 BLOGE runtime 同时变化。report directory 里同时放着 receipt，没有 governed Project aggregate，也没有发布负责人签署的 attestation。

请这样分类：

- **Verdict**：声明的 Cases 可以是 `PASS`。
- **Trust**： local seal 可以建立字节完整性，但同目录 custody 和缺失的 source/governance 事实无法建立发布信任链。
- **Attribution**：两因素变化不能识别单一原因。
- **Capability**： `AUTHORING`，并明确 source、governance、comparison 和 attestation gap。

接着拆开两个变化轴，重新生成 source-bound evidence，满足 governed Requirement，并把 receipts 保存到外部。完成这些后，才能请发布负责人 assess 并签署 exact payload。

0.5 保留这张地图

claim 混乱时回到第 23 章，需要可运行 starter 时看第 24 章，需要 counterexample 时看第 28 章，需要 trust boundary 时看第 29 章，需要精确 Requirement contribution 时看第 30 章，需要 aggregation 与 capability limit 时看第 31 章，需要 comparison 和 release decision 时看第 32 章。最后用第七阶段回顾判断这个问题是否根本需要 BLOGE。

学习阶段

第七阶段——选型与毕业项目

约束优先的选型与可运维毕业作品

目标：只在理由充分时使用 *BLOGE*，并交付别人能质疑的结果。

章节
33

第七阶段——选型与毕业项目

第 33 章——迁移与对比

约束优先的选型与可运维毕业作品

目标：只在理由充分时使用 *BLOGE*，并交付别人能质疑的结果。

承诺：读完本章后，你将能先写出约束，再决定是否选择 *BLOGE*；你会明确至少三种“不应选择 *BLOGE*”的情况，完成一份包含依据、反事实和第一步迁移实验的短 *ADR*。

33.1 学习目标

1. 把选型问题改写成部署、所有权、恢复语义、验证治理和生态五类约束。
2. 用硬停止条件淘汰不合适的候选，而不是给功能勾选表加总分。
3. 区分“功能存在”“团队能运营”“证据足以发布”三种完全不同的结论。
4. 为合适的候选设计可撤销的迁移第一步，不以全量改写验证选型。
5. 写出一份短 ADR：选择、依据、反事实条件、迁移探针和退出标准。

33.2 前置条件

- 第 20 章——Spring 与生产环境接线 ——你需要 `bloge-spring` 的接线模型来理解 BLOGE 如何嵌入运行中的应用。
- 第 13 章——持久执行 ——检查点存储、挂起/恢复和恢复。
- 第 21 章——单 JVM 里的调度与复杂度 ——引擎无状态性、虚拟线程调度和分片。

33.3 源示例

文件	展示内容
`BpmnToDslExample.java`	端到端：翻译 BPMN XML → `.bloge` DSL → 编译 → 执行

文件	展示内容
`bloge-bpmn-transformer/README.md`	翻译管道、支持范围、诊断信息和已知限制
`order-process.bloge`	标准订单处理 graph ——并行拉取、分支、韧性
`OrderProcessingDslExample.java`	DSL 编译、注册表接线和从 Java 执行
`order-saga.bloge`	Saga 风格的补偿：每个节点上的 `compensate`
`payment-wait.bloge`	长时间运行的 await，带事件关联和超时
`batch-order-processing.bloge`	`foreach` 并行扇出处理集合
`SpringTicketTriageService.java`	在 Spring 服务中注入 `GraphEngine` 和 `List<Graph>`
ADR-001	核心零依赖决策
ADR-002	虚拟线程优于 CompletableFuture
ADR-004	受 HCL 启发的 DSL 优于 YAML/JSON

33.4 为什么这很重要

你已经花了十六章学习如何构建、测试和运行 BLOGE 工作流。现在一个更难的问题来了：这个具体项目应该用 **BLOGE** 吗?

团队很少从零开始。他们已经有:

- 业务方法中嵌入了编排逻辑的现有服务。
- 使用 Spring Batch 处理步骤导向批处理的任务。
- 在可视化建模器中编写的 BPMN 流程图。
- 或者正在评估需要独立服务器集群的 workflow-as-code 平台。

本章给你一个结构化的方式，将 BLOGE 实际做的事与那些替代方案实际做的事进行对比——基于你可以在仓库中指出的架构属性，而不是营销宣传。

33.5 心智模型

把编排工具想象为坐落在两个轴上:

纵轴——工作流形状可见性。上半部分在模型（XML、DSL 或可视化图）中捕获工作流结构。下半部分将结构完全嵌入代码中。

横轴——部署拓扑。左侧需要独立的服务器集群。右侧嵌入到你现有的应用进程中。

BLOGE 位于右上象限：一个声明式的 DAG 模型，作为库运行在你的 JVM 内部，默认不需要外部基础设施。

这并非普遍最优——而是一组特定的取舍：

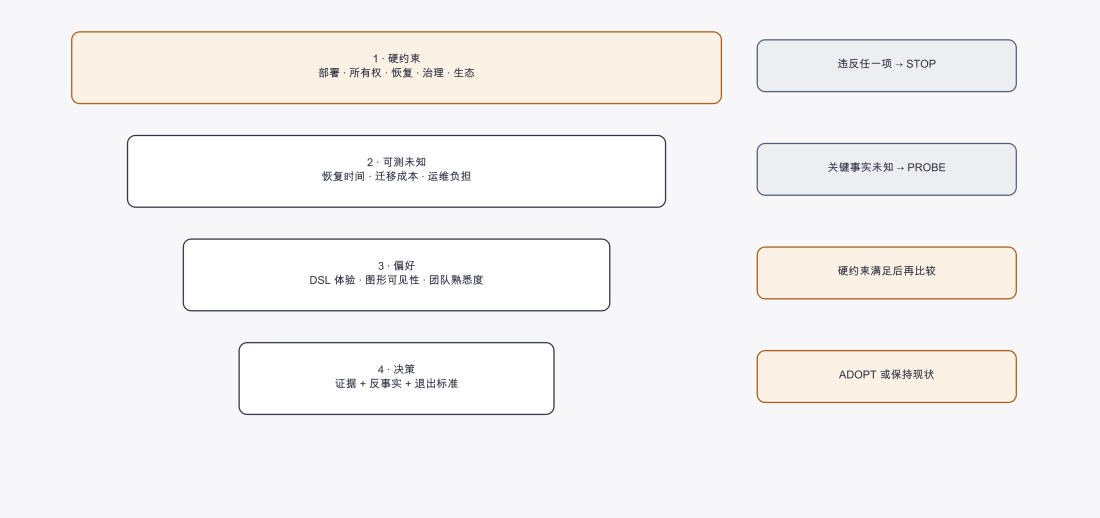
属性	**BLOGE** 的选择	文档位置
核心零外部依赖	`bloge-core` 仅依赖 `java.base`	ADR-001
虚拟线程执行	每个节点运行在自己的虚拟线程中	ADR-002
受 HCL 启发的 DSL	块导向、支持表达式、方便注释	ADR-004
基于检查点的持久化	通过 `bloge-durable` 可选启用；存储可插拔	第 13 章、README
可嵌入引擎	`GraphEngine` 是一个无状态的库对象	第 21 章
节点级韧性	Retry → Timeout → Fallback 链，按节点配置	ADR-007

三个编排模块将平台从纯图编排扩展到更多模式：

模块	新增能力
`bloge-agent-ext`	LLM 驱动的 agent 循环：`agent` DSL、工具调度、记忆策略和流式输出
`bloge-session-durable`	`DurableSessionManager`——可崩溃恢复的多轮 session，带 checkpoint 持久化
`bloge-state-durable`	`DurableStateMachineManager`——可崩溃恢复的状态机，带 checkpoint 持久化

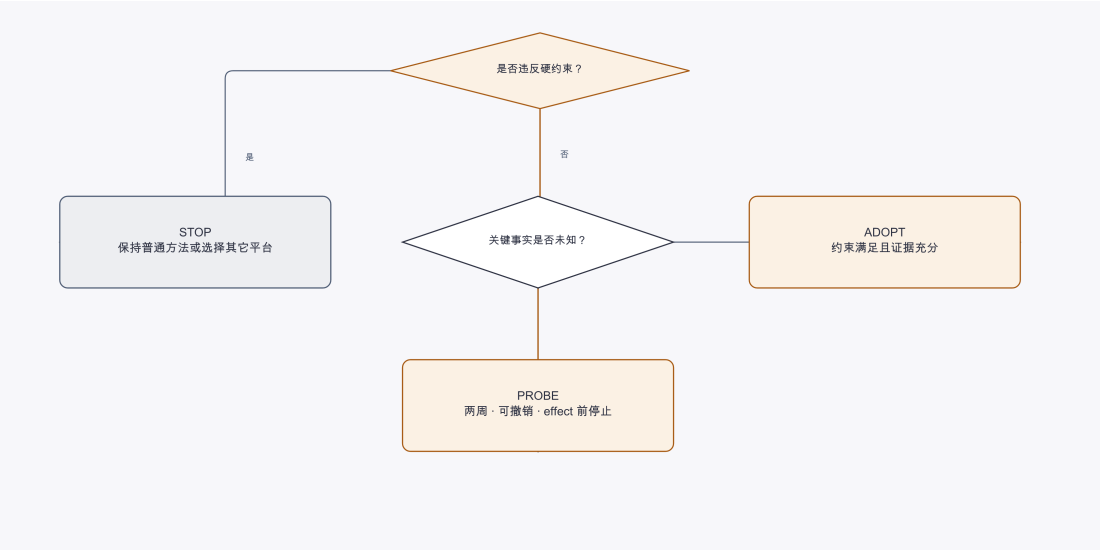
33.6 先约束，再比较

贷款平台刚完成业务正确性验证。架构委员会问的不是“BLOGE 有多少功能”，而是：在项目的非协商约束下，**BLOGE** 能否成为一个可运营、可验证、可退出的选择？



先写硬约束，再评估偏好，最后才设计迁移实验。任何候选违反硬约束，就停止；不要用其它功能分数把它“平均回来”。

约束	贷款平台的问题	**BLOGE** 事实	决策含义
部署拓扑	能否新增独立 workflow 集群?	引擎嵌入 JVM；durable store 可选	禁止新集群时仍可候选
模型所有权	谁审阅和修改流程?	`.bloge` 是代码友好的显式 DAG	业务只接受 BPMN 时是冲突
恢复语义	需要 checkpoint 还是完整历史重放?	BLOGE 保存 checkpoint，不提供通用历史重放平台	强依赖 replay 时停止
验证治理	发布是否必须绑定 Scenario、证据与源码身份?	RC1 提供 Scenario/Policy/Fixture、source-bound evidence 与 claims	能支持门禁，但仍需组织 Owner
生态边界	是否必须原生运行多语言 worker?	当前主运行时是 JVM	硬性多语言要求时停止



这张表不产生“BLOGE 得 82 分”的伪精度。它产生三个结果：`STOP`、`PROBE`、`ADOPT`。`PROBE` 表示关键事实仍未知，必须用小实验取得证据；它不是委婉的通过。

33.7 三种明确不应选择 **BLOGE** 的情况

1. 流程模型由非研发团队在 **BPMN** 平台中直接拥有。如果可视化协作、人工任务 收件箱和既有流程服务器是组织合同，迁移会破坏所有权，而不是只替换语法。
2. 恢复合同要求完整历史重放。如果审计或灾备必须从事件历史重建每一步，checkpoint 模型不能凭“也能恢复”替代 replay 语义。
3. 多语言 **worker** 是硬约束。如果同一编排必须原生调度 Go、Python 与 TypeScript worker，JVM 内嵌模型会把适配成本转嫁给团队。

还有一个更朴素的停止条件：只有三个顺序调用、没有等待、分支、独立恢复或共享 治理需求时，普通方法更便宜。架构能力不是免费的；额外模型必须换来真实复杂度下降。

33.8 一个完成的贷款选型 ADR

贷款平台的硬约束是“不新增独立集群”“JVM 内运行”“每次发布绑定批准的 Scenario 与源码证据”；完整历史重放和多语言 worker 不是当前要求。因此结论是 `PROBE BLOGE`，而不是直接全量采用。

ADR 字段	本例内容
选择	用 `auto-approve` 单路径做两周可撤销探针
依据	JVM 内嵌、显式 DAG、RC1 验证与 source-bound evidence 符合硬约束
反事实	若必须引入历史重放或非 JVM worker，转向 server-based workflow 平台
第一迁移步	只把读取型风险检查迁成 operator；放款 effect 留在旧路径
退出标准	graph contract、业务 Scenario、恢复演练任一失败，停止扩面

这个 ADR 的价值不是“批准 BLOGE”，而是把下一步变成可证伪的实验。一次绿色 graph test 不能替代业务 Scenario；一次 `PASS` 也不能替代 source binding 和发布治理。

33.9 第一个可运行示例

最常见的迁移场景是将手写的 Java 编排方法替换为 BLOGE graph。下面按步骤完成这次迁移。

33.9.1 迁移前：手写胶水代码

```
// 典型的过程式编排，埋在一个服务方法里
public OrderResult processOrder(String userId, List<String> productIds) {
    // 1. 顺序执行——但这两个调用实际上互不依赖
    User user = userService.fetchUser(userId);
    List<Product> products = productService.fetchProducts(productIds);

    // 2. 重试逻辑和业务代码纠缠在一起
    BigDecimal total = null;
    for (int attempt = 0; attempt < 3; attempt++) {
        try {
            total = pricingService.calculate(user, products);
            break;
        } catch (Exception e) {
            if (attempt == 2) throw e;
            Thread.sleep(100 * (attempt + 1));
        }
    }

    // 3. 分支逻辑藏在一个 if 语句里
    CreditResult credit = creditService.check(user.id(), total);
    if (credit.approved()) {
```

```
        return orderService.create(user, total);
    } else {
        return orderService.reject(user.id(), credit.reason());
    }
}
```

问题：

- 步骤 **1** 和步骤 **2** 互不依赖，却被顺序执行。
- 重试逻辑是手写的，混在业务方法里。
- 没人能看到工作流的形状，除非逐行读代码。

33.9.2 迁移后：**BLOGE graph**

同样的工作流，表达为 `order-process.bloge`：

这 46 行 graph 保留完整，因为迁移对比需要一份可复制单元，同时展示并行读取、fan-in、韧性和两条分支目标。

```
graph orderProcess {

  node fetchUser : FetchUserOperator {
    input { userId = ctx.userId }
    timeout = 3s
    retry = { attempts: 2, backoff: 200ms, strategy: exponential }
  }

  node fetchProducts : FetchProductsOperator {
    input { productIds = ctx.productIds }
    timeout = 5s
  }

  node calcPrice : CalcPriceOperator {
    depends_on = [fetchUser, fetchProducts]
    input {
      user     = fetchUser.output
      products = fetchProducts.output
    }
  }

  node checkCredit : CreditCheckOperator {
    depends_on = [fetchUser, calcPrice]
    input {
      userId = fetchUser.output.id
      amount = calcPrice.output.total
    }
    retry = { attempts: 3, backoff: 100ms, strategy: jitter }
    fallback = { approved: false, reason: "credit service unavailable" }
```

```
  }

  branch on checkCredit.output.approved {
    true  -> createOrder
    false -> rejectOrder
  }

  node createOrder : CreateOrderOperator {
    depends_on = [calcPrice]
    input { user = fetchUser.output, price = calcPrice.output }
  }

  node rejectOrder : RejectOrderOperator {
    depends_on = [checkCredit]
    input { userId = fetchUser.output.id, reason = checkCredit.output.reason }
  }
}
```

变化之处：

- `fetchUser` 和 `fetchProducts` 并行运行——引擎看到它们之间没有依赖。
- 重试和超时声明在节点上，而不是写在循环里。
- 分支是一个一等路由决策，而不是隐藏的 `if`。
- 工作流的形状可见、可审查、可工具化。

当 DSL 保存在文件里时，同样的 registry / engine 接线可以写成这样：

```
var registry = new DefaultOperatorRegistry();
registry.register("FetchUserOperator", fetchUserOp);
registry.register("FetchProductsOperator", fetchProductsOp);
// …注册其余 operator …

Graph graph = new GraphLoader(registry).load(
        Path.of("bloge-examples/src/main/resources/bloge/order-process.bloge"));
GraphEngine engine = GraphEngine.builder().registry(registry).build();
GraphResult result = engine.execute(graph, new GraphContext(Map.of(
        "userId", "user-42",
        "productIds", List.of("prod-1", "prod-2", "prod-3")
)));
```

33.10 拆开来看

33.10.1 BLOGE vs 手写编排

维度	手写胶水代码	BLOGE graph
并发	你自己管理线程、线程池、Future	引擎根据 DAG 形状调度虚拟线程（ADR-002）
韧性	手写重试循环、try/catch 降级	按节点声明：`retry`、`timeout`、`fallback`（ADR-007）
可见性	读代码；寄希望于注释是准确的	`.bloge` 文件 本身就是 文档；Studio 可以可视化渲染
可测试性	在每个测试中 mock 每个服务调用	`bloge-test` 提供 `MockOperator`、`GraphTestRunner`、`DslTestHelper`
持久化	自己构建检查点/恢复机制	通过 `bloge-durable` 存储可选启用（第 13 章）

手写代码更合适的场景：如果工作流只是三个顺序调用，没有重试、没有分支、也不需要持久化，那么一个普通方法更简单。当编排的形状足够复杂，需要并发、韧性或可见性时，BLOGE 才能体现出价值。

33.10.2 BLOGE vs 步骤导向的批处理框架

步骤导向的批处理框架（如 Spring Batch）将工作建模为线性的步骤管道，采用块导向的读取、处理和写入模式。

维度	步骤导向的批处理	BLOGE graph
执行模型	Step 1 → Step 2 → Step 3，线性执行	DAG：独立节点自动并行运行
迭代	块导向的 reader/processor/writer	`foreach`，支持并行或顺序模式（`batch-order-processing.bloge`）
韧性	步骤级或块级的跳过/重试策略	节点级的 retry → timeout → fallback 链
长时间等待	不是主要关注点	`await`，带事件关联和超时（`payment-wait.bloge`）
依赖	通常需要框架运行时	`bloge-core` 零外部依赖（ADR-001）

步骤导向批处理更适合的场景：如果你的工作负载是纯 ETL——读取块、转换、写入——没有扇出、没有分支、也不需要等待外部事件，那么批处理框架是为这种模式量身定制的。当批处理步骤需要并行扇出、等待外部信号，或者与非批处理工作流共享编排模式时，BLOGE 才能体现出价值。

33.10.3 BLOGE vs BPMN 导向的工具

BPMN 导向的工具使用基于 XML 的可视化标记来建模工作流，通常由一个独立的执行服务器支撑。

维度	BPMN 导向的工具	BLOGE graph
工作流标记	BPMN 2.0 XML（冗长、面向机器）	`.bloge` DSL（支持表达式、方便注释；ADR-004）
网关	显式的 fork/join 网关元素	从 DAG 形状隐式推断：不需要网关节点
部署	独立的流程引擎服务器	嵌入式库——`GraphEngine` 运行在你的 JVM 中
迁移路径	—	`bloge-bpmn-transformer` 将 BPMN XML 翻译为 `.bloge` DSL（README）

`bloge-bpmn-transformer` 模块提供了一条具体的迁移路径。它读取 BPMN 2.0 XML，通过中间表示进行降级转换，然后输出 `.bloge` DSL 文本或可执行的 `Graph` 对象。

支持的 BPMN 元素包括：开始/结束事件、服务/脚本/用户任务、排他网关、并行网关和包容网关、定时器/消息/信号事件、调用活动和子流程结构。不支持的元素会生成诊断信息，而不是默默发明语义。

BPMN 包容网关支持。 BPMN 翻译器支持包容网关（翻译为 `branch mode=inclusive`）、用户任务元数据保留（`assignee`、`candidateGroups`、`formKey`）、扩展元素解析（`<inputOutput>` 参数和 `<properties>`），以及改进的 **UEL** 到 **BLOGE** 表达式翻译（`toString()`、`isEmpty()` 等安全方法不再触发 `COMPLEX_EXPRESSION` 诊断）。

翻译示例来自 `BpmnToDslExample.java`：

```
BpmnTranslator translator = new BpmnTranslator(OperatorMappingConfig.EMPTY);
try (InputStream bpmn = getClass().getResourceAsStream("/bpmn/order-process.bpmn")) {
    TranslationResult<String> dsl = translator.translateToDsl(bpmn);
    if (dsl.hasErrors()) {
        throw new IllegalStateException(dsl.diagnostics().toString());
    }
    // dsl.result() 是合法的 .bloge DSL 文本
    Graph graph = new GraphLoader(registry).load(dsl.result());
}
```

诊断代码	含义
`UNSUPPORTED_ELEMENT`	翻译器尚不支持的 BPMN 构造
`UNMAPPED_OPERATOR`	没有 operator 映射匹配到该任务
`COMPLEX_EXPRESSION`	UEL 表达式无法一对一翻译
`POTENTIAL_DATA_LOSS`	翻译可行，但某些 BPMN 意图可能无法完全保留
`AMBIGUOUS_TIMER`	定时器定义使用了无法直接映射到 BLOGE `wait` 语法的形式

BPMN 工具更适合的场景：如果你所在组织的业务分析师使用 BPMN 编写并拥有流程模型，且执行服务器已经部署和运维，那么迁移到 BLOGE 可能不值得带来的中断。当开发团队拥有工作流并且需要代码友好的编写体验和可嵌入的运行时时，BLOGE 才能体现出价值。

33.10.4 BLOGE vs workflow-as-code 平台

Workflow-as-code 平台允许你使用通用编程语言（Go、Java、TypeScript）编写工作流，持久化由一个独立的服务器集群通过重放函数历史来处理。

维度	**Workflow-as-code 平台**	**BLOGE**
部署	需要专用服务器集群	嵌入式库；无需外部服务器
持久化模型	历史重放 / 事件溯源	基于检查点：包括节点输出、loop / foreach 进度、挂起上下文、部分结果、事件关联和 session 快照（第 14 章，底层机制见第 13 章）
工作流形状	隐含在代码的控制流中	在 `.bloge` 文件和 Studio 中可见的显式 DAG 模型
依赖	客户端 SDK + 服务器基础设施	`bloge-core` 仅依赖 `java.base`（ADR-001）
并发	服务器管理 worker 任务分发	引擎在本地使用虚拟线程（ADR-002）

维度	Workflow-as-code 平台	BLOGE
补偿	框架特定的 saga 模式	声明式的节点级 `compensate`（`order-saga.bloge`）

Workflow-as-code 平台更适合的场景：如果你需要跨 Go、TypeScript 和 Python 的内建多语言支持，或者你已经运维着服务器集群并依赖其历史重放语义，那么该平台的生态系统优势是实实在在的。当你希望避免运维独立集群、团队是 JVM 原生的、或者显式 DAG 模型和可嵌入引擎比跨语言 SDK 支持更重要时，BLOGE 才能体现出价值。

33.11 常见陷阱

33.11.1 [X] 用功能清单对比替代架构适配对比

一个常见错误是构建一张功能对照表——"它有重试吗？有分支吗？"——然后选打勾最多的工具。

本章提到的每个工具都支持某种形式的重试、分支和持久化。真正重要的差异在于架构层面：

- 引擎在哪里运行？（嵌入式库 vs. 独立集群）
- 工作流形状如何表达？（显式模型 vs. 隐含在代码中）
- 持久化如何工作？（检查点 vs. 重放）
- 依赖和 **JDK** 要求是什么？（零依赖核心 vs. SDK + 服务器）

如果你在错误的层面上对比，你会选到一个功能齐全但部署模型完全不适合你团队的工具。

33.12 引导式重写

33.12.1 从手写代码迁移到 BLOGE 的四个步骤

拿你代码库中任意一个编排方法，按以下步骤操作：

步骤 **1** ——识别独立的调用。

标记每一对没有数据依赖的调用。在上面的手写示例中，`fetchUser` 和 `fetchProducts` 是独立的。

步骤 **2** ——提取 **operator**。

每个调用变成一个 `Operator<I, O>` 实现。将它们注册到 `DefaultOperatorRegistry`：

```
registry.register("FetchUserOperator", (input, ctx) -> {
    String userId = ((Map<String, Object>) input).get("userId").toString();
    return Map.of("id", userId, "name", userService.fetchUser(userId).name());
});
```

步骤 **3** ——声明 **graph**。

编写 `.bloge` 文件。使用 `depends_on` 实现显式汇合。使用 `branch on` 实现条件路由。在调用外部服务的节点上添加 `retry`、`timeout` 和 `fallback`。

步骤 **4** ——验证行为。

使用 bloge-test 中的 GraphTestRunner 来断言节点执行顺序、状态和输出：

```
var runner = new GraphTestRunner(
        registry,
        Map.of(
                "FetchUserOperator", fetchUserOp,
                "FetchProductsOperator", fetchProductsOp
                // …注册其余 operator …
        )
);

var result = runner.execute(graph, new GraphContext(Map.of("userId", "u1")));
runner.assertNodeExecuted("fetchUser");
runner.assertNodeExecuted("fetchProducts");
runner.assertNodeSkipped("rejectOrder");   // 授信已通过
```

33.12.2 从 BPMN 迁移到 BLOGE

步骤 **1** ——翻译。使用 BpmnTranslator.translateToDsl() 并传入 operator 映射配置。

步骤 **2** ——阅读诊断信息。修复所有 UNMAPPED_OPERATOR 或 COMPLEX_EXPRESSION 诊断。在 OperatorMappingConfig 中将 BPMN 服务任务 ID 映射到 BLOGE 的 operator 名称。

步骤 **3** ——审查生成的 **DSL**。翻译器会生成 // Source: bpmn:serviceTask id="xxx" 注释以便追溯（需启用 generateSourceComments）。

步骤 **4** ——编译并运行。

```
Graph graph = new GraphLoader(registry).load(dsl.result());
GraphEngine engine = GraphEngine.builder().registry(registry).build();
engine.execute(graph, context);
```

33.13 脑力检查

1. 心智模型图中的两个轴是什么？BLOGE 位于哪里？（工作流形状可见性 *vs.* 部署拓扑；*BLOGE* 位于右上象限：声明式模型 + 可嵌入的库。）
2. 说出 BLOGE 的三个在 ADR 中记录的架构属性，并解释每个属性的取舍。（零依赖核心 → 一些便利功能需要在适配器模块中重新实现。虚拟线程 → 需要现代 *JDK*。受 *HCL* 启发的 *DSL* → 需要专用解析器，不兼容通用 *YAML/JSON* 工具。）
3. 如果翻译 BPMN 文件时收到 COMPLEX_EXPRESSION 诊断，应该怎么做？（手动将 *UEL* 表达式改写为 *BLOGE* 路径表达式，或者将逻辑移入 *operator*。）
4. 什么时候应该选择 workflow-as-code 平台而不是 BLOGE？（当你需要跨 *Go/TypeScript/Python* 的多语言 *SDK* 支持，或者你已经运维着服务器集群并依赖其重放语义。）

33.14 实验

33.14.1 实验——写一份可证伪的短 **ADR**

从你自己的系统选择一个包含至少五个步骤的流程，交付不超过一页的 ADR：

1. 写出三条硬约束和两条偏好；硬约束必须可判定为满足或不满足。
2. 至少比较普通方法、BLOGE 和一个 server-based 候选。
3. 明确一个“不选 BLOGE”的反事实条件，不允许写“视情况而定”。
4. 设计一个两周内可撤销的迁移探针，限定修改范围和真实 effect 边界。
5. 写出 `STOP / PROBE / ADOPT` 结论，以及使结论失效的新证据。

完成标准：另一个读者只看 ADR，就能复述选择、证据、未知项、第一步和停止条件；不能只看到一张功能勾选表。参考答案与评分量表在 `solutions/ch33/adr-review-guide.md`，完成前不要打开。

33.15 桥接：从变化归因到架构选择

本章消费变化归因的结果来作选择，却不能把设计判断包装成受控因果实验。只有当两次执行共享 Scenario、Policy、Fixture、business runtime 和 source cohort，并且只改变 `DSL_GRAPH` 或 `BLOGE_TOOLCHAIN` 一个轴时，comparison receipt 才能说明结果变化来自哪里。ADR 还必须加入组织约束、运营成本和退出条件；这些不由 `compareEvidence(...)` 自动决定。

33.16 实验验收卡

- 预期与观察：约束漏斗得到 STOP、PROBE 或 ADOPT，不预设 BLOGE 胜出。
- 失败与恢复：加入硬约束使方案失效；恢复为停止或可撤销探针。
- 证明边界：证明选择可证伪，不证明 BLOGE 适合所有组织。
- 练习合同：贷款系统约束；只改一个约束；交付短 ADR；反事实能改变决定且退出条件明确即停止。

33.17 回顾

- BLOGE 占据一个特定的架构位置：声明式 **DAG** 模型 + 可嵌入的库引擎，核心零外部依赖。
- 在架构适配（部署模型、持久化语义、工作流可见性、依赖足迹）层面对比工具，而不是功能清单。
- 手写代码适合简单的顺序调用；当编排的形状需要并发、韧性或可见性时，BLOGE 才能体现出价值。
- 步骤导向的批处理是为 ETL 量身定制的；当批处理步骤需要扇出、分支或等待外部事件时，BLOGE 才能体现出价值。
- **BPMN** 工具在业务分析师拥有模型时表现出色；当开发者拥有工作流并需要代码友好的编写体验时，BLOGE 才能体现出价值。`bloge-bpmn-transformer` 提供了一条具体的迁移路径。
- **Workflow-as-code** 平台在多语言、服务器托管的持久化方面具有优势；当你需要可嵌入的、JVM 原生的引擎且不需要独立集群时，BLOGE 才能体现出价值。
- 从手写代码迁移遵循四个步骤：识别独立调用、提取 operator、声明 graph、验证行为。
- 决策只输出 `STOP / PROBE / ADOPT`；违反硬约束时停止，未知关键事实时做可撤销探针。

- 短 ADR 必须包含选择、依据、反事实、第一迁移步和退出标准。

33.18 下一步

在第 34 章——实验中，你将完成端到端的练习，将整个系列中的所有内容——graph、韧性、子图、持久化、可观测性，以及本章的迁移模式——组合成完整的、可运行的项目。

33.19 参考链接

- 项目 README ——模块概览与快速开始
- ADR-001 ——核心零依赖
- ADR-002 ——虚拟线程优于 CompletableFuture
- ADR-004 ——受 HCL 启发的 DSL
- ADR-007 ——韧性顺序
- DSL 规范 ——完整的语言语法与编译语义
- Operator 设计规范 ——operator 边界与粒度规则
- `bloge-bpmn-transformer` README ——BPMN 翻译管道与诊断
- `order-process.bloge` ——标准订单处理示例
- `order-saga.bloge` ——saga 补偿示例
- `payment-wait.bloge` ——长时间运行的 await 示例
- `BpmnToDslExample.java` ——BPMN → DSL → 执行示例
- `OrderProcessingDslExample.java` ——DSL 编译与执行
- 入门指南 ——完整安装指南

章节

34

第七阶段——选型与毕业项目

第 34 章——实验

约束优先的选型与可运维毕业作品

目标：只在理由充分时使用 *BLOGE*，并交付别人能质疑的结果。

承诺：读完本章后，你将完成基础、进阶、毕业三级 *capstone*。最终作品不只包含 *graph*，还包含技术测试、业务 *Scenario/Policy*、证据解读、架构图和“尚未证明事项”。

34.1 学习目标

1. 在限定范围内从 starter 演进 graph，而不是从空文件猜语法。
2. 分别交付 graph contract、business contract 和 release boundary 的证据。
3. 对 effect、等待、恢复和 fallback 做单因素失败实验。
4. 用一张架构图解释运行边界，用一张 receipt 解读图说明证据边界。
5. 在完成声明中同时列出已证明、未证明和必须由人批准的事项。

34.2 前置条件

前面全部三十三章。这些实验假设你已经熟练掌握：

- Graph 结构、依赖和数据流（第 1 章–第 4 章）
- 分支和韧性（第 5 章–第 6 章）
- Operator 设计和 DSL 编写（第 7 章–第 8 章）
- 子图、迭代、等待和持久化（第 10 章–第 13 章）
- 编排模式——多轮 session、状态机和嵌套生命周期所有权（第 14 章–第 16 章）
- 工具链与评审反馈（第 9 章）
- 测试、生产环境接线和规模化（第 18 章–第 22 章）
- 业务正确性、证据和发布声明（第 23 章–第 32 章）
- 迁移思维（第 33 章）

34.3 源示例

文件	展示内容
`order-process.bloge`	基础 capstone 起点——扇出、汇合、分支、韧性
`ticket-routing.bloge`	进阶 capstone 起点——情感分析与多路分支
`loan-approval.bloge`	毕业 capstone 起点——贷款业务与验证资产
`batch-order-parallel.bloge`	可选题——`foreach` 并行处理每一项
`status-polling.bloge`	可选题——`loop` 加 `until` 和 `loopIteration`
`payment-wait.bloge`	可选题参考——`await` 事件关联、超时和分支
`order-saga.bloge`	毕业题参考——saga 补偿模式
`GraphTestRunner.java`	每个实验验证步骤使用的测试工具
`MockOperator.java`	测试替身——`returning`、`throwing`、`delaying`、`recording`
`DslTestHelper.java`	DSL 编译 + `expectParseError` / `expectCompileError`
`missing-timeout.bloge`	反模式参考——节点没有超时预算
`over-broad-fallback.bloge`	反模式参考——降级范围过宽，掩盖了真实的故障模式

34.4 为什么这很重要

阅读 graph 的知识和亲手设计 graph 是两回事。前面每一章都只引入一个概念。而真实的工作流会同时组合所有概念：

- 一个订单管道需要扇出 加 韧性 加 分支。
- 一个工单路由器需要情感分析 加 三路分支 加 对不稳定 ML 服务的降级。
- 一个批处理器需要 `foreach` 加 下游聚合 加 对每一项结果的测试断言。

这些实验用于检验前述概念能否在实践中组合。每个实验都从仓库中已有的一致性测试 fixture 开始，因此不需要凭空发明语法；需要完成的动作是阅读、扩展和测试。

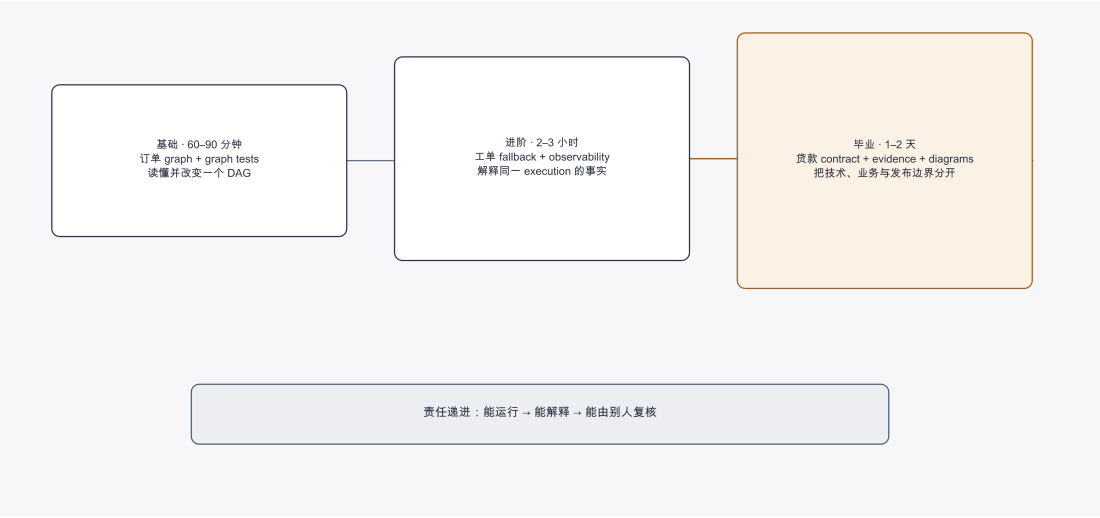
34.5 心智模型

把三级 capstone 想象成一段逐层扩大的责任：

阅读——打开一致性测试 fixture。追踪每一个依赖。识别哪些节点并行运行、哪些节点需要等待。

扩展——添加实验要求的功能。将上游输出接入下游输入。有意识地选择韧性策略。

验证——使用 `GraphTestRunner` 和 `MockOperator` 执行 graph，断言 `NodeStatus` 值，并检查执行顺序。



级别	预计投入	起始文件	工作合同	完成标准
基础	60—90 分钟	`order-process.bloge`	一份 DSL 副本 + 一组 graph tests；给出明确步骤	成功/拒绝两路状态和输出可复核
进阶	2—3 小时	`ticket-routing.bloge`	DSL、mock、listener 测试；只给边界提示	fallback 与观测事实可区分

级别	预计投入	起始文件	工作合同	完成标准
毕业	1—2 天	loan starter + 验证资产	graph、tests、Scenario/Policy/Fixture、图与报告；只给验收合同	可运行、可解释、可复核，并写明未证明事项

每一级都先复制只读 starter 到自己的工作目录。不要修改 `submodule/examples` 或 `submodule/bloge`；它们分别是远端示例基线和 RC1 事实源。

34.6 第一个可运行示例

在开始实验之前，确保你能运行最简单的测试驱动 graph 循环。这是你的热身。

```
// 1. 编译 DSL
var helper = new DslTestHelper(registry);
Graph graph = helper.compile("""
    graph warmup {
      node greet : GreetOperator {
        input { name = ctx.name }
      }
    }
    """);

// 2. 接入 mock operator
var runner = new GraphTestRunner(Map.of(
    "GreetOperator", MockOperator.returning(Map.of("message", "Hello"))
));

// 3. 执行
var ctx = new GraphContext(Map.of("name", "Lab Runner"));
GraphResult result = runner.execute(graph, ctx);

// 4. 断言
assert result.isSuccess();
runner.assertNodeExecuted("greet");
```

如果编译通过并且断言成功，你就准备好了。

34.7 拆开来看

每个实验都遵循相同的结构。以下是每个阶段练习的技能：

阶段	练习的技能	对应章节
阅读 fixture	依赖追踪、并行识别	第 3 章
添加节点	Operator 绑定、输入接线、`depends_on`	第 2 章、第 4 章
添加韧性	`retry`、`timeout`、`fallback`、`compensate`	第 6 章
添加分支	`branch on`、`otherwise`、跳过节点推理	第 5 章
添加迭代	`foreach`、`loop`、`carry`、`until`	第 11 章
添加挂起	`await`、事件关联、`on_timeout`	第 12 章
编写测试	`GraphTestRunner`、`MockOperator`、`DslTestHelper`	第 18 章

34.8 常见陷阱

34.8.1 [X] 给每个节点都添加韧性策略

一个常见的冲动是在 graph 中的每个节点上都加上 `retry + timeout + fallback`。别这样做。

看看 `over-broad-fallback.bloge` 中的反模式：在 `authorizePayment` 上使用全覆盖的 fallback，把校验错误和瞬时超时都藏在同一个"人工审核"结果后面。下游节点无法区分合理的降级和永久性 bug。

再看看 `missing-timeout.bloge`：一个调用外部服务却没有任何 timeout 的节点，可能让整个 graph 无限期停滞。

实验中的经验法则：

- 外部调用 → 必须加 `timeout`。
- 瞬时不稳定的调用 → 加 `retry` 并且每次尝试带 `timeout`。
- 只有在下游每个消费者都能真正接受部分数据时，才添加 `fallback`。
- 纯本地计算（转换、聚合）→ 不需要韧性策略。

34.9 引导式重写

在进入正式实验之前，先用一个小改动练习扩展-验证的循环。

打开 `order-process.bloge`。当前 `createOrder` 仅依赖 `calcPrice`：

```
node createOrder : CreateOrderOperator {
    depends_on = [calcPrice]
    input {
      user  = fetchUser.output
      price = calcPrice.output
    }
  }
```

改写任务：业务现在要求 `createOrder` 也记录授信检查的结果以便审计——但只在订单被批准时。添加一个新的输入绑定：

```
node createOrder : CreateOrderOperator {
    depends_on = [calcPrice, checkCredit]
    input {
      user        = fetchUser.output
      price       = calcPrice.output
      creditCheck = checkCredit.output
    }
  }
```

现在验证这个修改是安全的：

```
var runner = new GraphTestRunner(Map.of(
    "FetchUserOperator",     MockOperator.returning(Map.of("id", "u1")),
    "FetchProductsOperator", MockOperator.returning(Map.of("items", List.of())),
    "CalcPriceOperator",     MockOperator.returning(Map.of("total", 99.0)),
    "CreditCheckOperator",   MockOperator.returning(Map.of("approved", true)),
    "CreateOrderOperator",   MockOperator.recording(),
    "RejectOrderOperator",   MockOperator.recording()
));

var ctx = new GraphContext(Map.of(
    "userId", "u1",
    "productIds", List.of("p1")
));

GraphResult result = runner.execute(graph, ctx);
assert result.isSuccess();
runner.assertNodeExecuted("createOrder");
runner.assertNodeSkipped("rejectOrder");
```

这个模式——扩展 graph，然后编写测试证明新接线正确——就是这些实验要培养的核心能力。

34.10 脑力检查

在开始实验之前，不看回顾地回答以下问题：

1. 在 `ticket-routing.bloge` 中，哪些节点并行运行？（答案：*`fetchCustomer`* 和 *`fetchTicketHistory`*——两者之间没有依赖。）
2. 在 `loan-approval.bloge` 中，`fetchApplication` 完成后的最大并行度是多少？（答案：四——*`checkCredit`*、*`detectFraud`*、*`verifyIncome`* 和 *`checkBlacklist`* 都只依赖 *`fetchApplication`*。）
3. 在 `batch-order-parallel.bloge` 中，`summarize` 能否在所有 foreach 迭代完成之前启动？（答案：不能——*`summarize`* 依赖 *`processOrders`*，而 *`processOrders`* 只有在所有迭代完成后才算完成。）
4. 你会使用哪个 `MockOperator` 工厂方法来模拟一个耗时 500 毫秒然后成功的节点？（答案：*`MockOperator.`*

delaying(Duration.ofMillis(500), output)。）

5. 哪个 GraphTestRunner 方法可以证明一个分支目标没有被选中？（答案：*assertNodeSkipped("nodeId")*。）

34.11 实验

34.11.1 基础 capstone ——读懂并扩展订单 graph

预计投入： 60—90 分钟 起始文件： order-process.bloge，复制后修改 允许范围：一份 DSL 副本、一组 GraphTestRunner 测试；不得修改 operator 实现 提示层级：明确步骤

需求：

1. 添加一个 fetchInventory 节点，使其与 fetchUser 和 fetchProducts 并行运行。输入：ctx.productIds。超时：2s。
2. 将 fetchInventory.output 作为新的 inventory 输入接入 calcPrice。相应更新 depends_on。
3. 给 createOrder 添加一个 compensate 块，调用 ReleaseInventoryOperator——模式参考 order-saga.bloge。
4. 验证：编写一个测试，断言 fetchInventory、fetchUser 和 fetchProducts 全部完成；当授信通过时 createOrder 执行；rejectOrder 被跳过。

完成标准： DSL 真实编译；批准与拒绝两路分别断言 EXECUTED/SKIPPED；库存超时只改变预期节点；写出一句“本实验没有证明业务规则正确”。

34.11.2 进阶 capstone ——让工单降级可观察

预计投入： 2—3 小时 起始文件： ticket-routing.bloge，复制后修改 允许范围： DSL、mock operator、一个 listener 测试；不得接真实工单系统 提示层级：只给边界，不给完整 DSL

需求：

1. analyzeSentiment 已经有了 fallback。给 fetchTicketHistory 添加 retry = { attempts: 2, backoff: 300ms, strategy: jitter }，使其能够承受历史服务的瞬时故障。
2. 添加一个 escalateToManager 节点，作为 classifyPriority.output.priority == "critical" 时的第四个分支目标。输入：来自 fetchCustomer.output.id 的 customerId 和来自 analyzeSentiment.output 的 sentiment。
3. 验证：编写一个测试，其中 classifyPriority 返回 { priority: "critical" }，断言 escalateToManager 执行，而 assignVipAgent、assignNormalAgent 和 autoResolve 全部被跳过。

完成标准： critical、normal、fallback 三条路径可复核；同一 executionId 下能区分 metric、log 与 node status；不能从 fallback 成功推出业务成功。

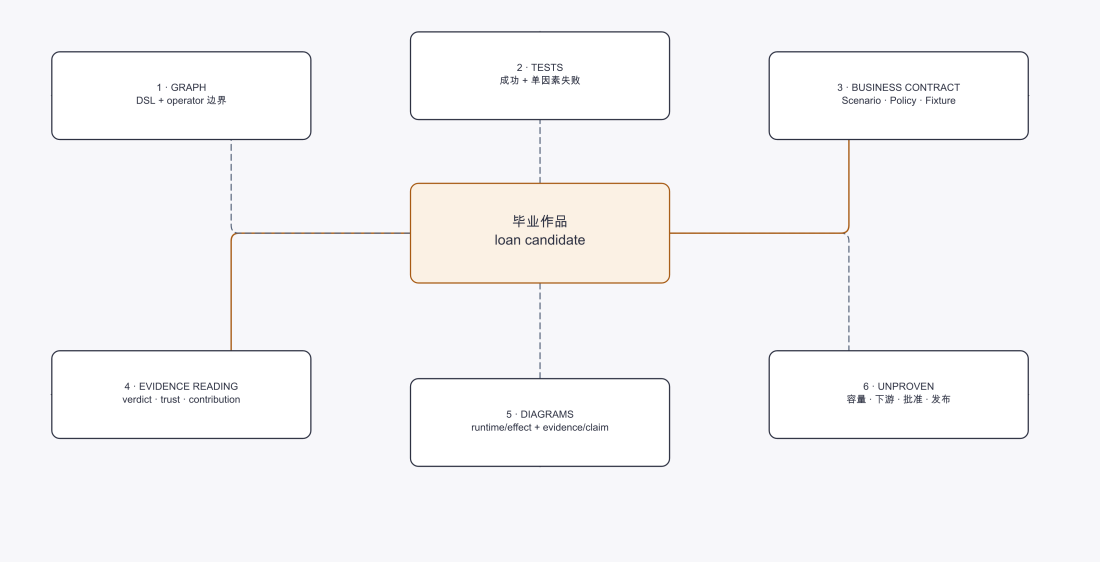
34.11.3 毕业 capstone ——交付可复核的贷款变更

预计投入： 1—2 天 起始文件： loan-approval.bloge 与同模块 src/test 中的 Scenario/Policy/Fixture 测试资产，复制后修改 允许范围： graph、受控 operator test doubles、Scenario、Policy、Fixture、证据解读与图；不

得替业务 Owner 批准 GOLDEN，不得调用真实放款 effect 提示层级：只给验收合同

需求：

1. 在 `aggregateRisk` 和 `makeDecision` 之间添加一个 `reserveFunds` 节点。`reserveFunds` 依赖 `aggregateRisk`，输入为 `application = fetchApplication.output` 和 `risk = aggregateRisk.output`。
2. 给 `reserveFunds` 声明 `output { reservationId: String }` 并添加一个调用 `ReleaseFundsOperator` 的 `compensate` 块——与 `order-saga.bloge` 中的 saga 模式相同。
3. 更新 `makeDecision`，使其也 `depends_on reserveFunds`。
4. 给 `reserveFunds` 添加 `timeout = 3s` 和 `fallback = { reservationId: "NONE", status: "degraded" }`，使慢速的预留操作不会阻塞管道。
5. 技术验证：编写一个测试，让 `ReserveFundsOperator` 使用 `MockOperator.throwing(...)`，证明 fallback 被触发且 graph 仍能到达 `makeDecision`。
6. 业务验证：至少覆盖 `auto-approve`、`manual-review`、`reject` 三个 Case；Policy 中禁止真实 effect，Fixture 固定授信、反欺诈和收入事实。
7. 证据解读：对报告写出 verdict、evidence trust/source binding、Requirement contribution 和 release claim 四层结论，不得互相替代。
8. 架构表达：提交一张 graph/runtime/effect 边界图和一张证据到 claim 的 解释图，标出业务 Owner 与发布负责人。
9. 未证明事项：至少列出生产容量、真实下游健康、GOLDEN 批准和发布授权。



完成标准：六类资产齐全；单因素失败可以复现；报告中的每句话都能指向 Scenario、源码测试或人工职责之一；任何 `PASS` 都没有被扩写成发布批准。

34.11.4 可选扩展题库

以下三题不计入三级 capstone 完成标准。它们用于工作坊加练，答案和评分说明已 移到教师指南，避免与主任务并列造成“必须做完六题才毕业”的误读。

34.11.4.1 批量订单：添加逐项韧性和汇总分支 **Fixture**：`batch-order-parallel.bloge`

需求：

1. 给 foreach 主体内的 deductStock 节点添加 retry = { attempts: 1, backoff: 100ms, strategy: exponential } 和 timeout = 3s。
2. 给 deductStock 添加 fallback = { deducted: false, reason: "stock service unavailable" }，使单个库存扣减失败不会破坏整个批次。
3. 在 summarize 之后，添加一个 branch on summarize.output.allSucceeded：
   - true → notifySuccess（一个新的 NotifySuccessOperator 节点）
   - false → notifyPartialFailure（一个新的 NotifyPartialFailureOperator 节点）
4. 验证：让一个 item 的 deductStock 使用 MockOperator.throwing(...)，确认 fallback 被触发，然后断言 notifyPartialFailure 执行而 notifySuccess 被跳过。

练习的概念： foreach、逐项韧性、迭代内的 fallback、基于聚合输出的下游分支（第 5、6、10 章）。

进阶任务——全局限流与逐项失败模式。在上面需求的基础上，把 foreach 的 batch_size = 10，这样同时最多 10 个 item 在跑; 再把 on_item_failure = continue，这样单个坏 item 不会让整批中止。用 MockOperator.throwing(...) 让 20 个 item 里有 3 个失败，然后断言：

1. deductStock 被调用了正好 20 次 (每个 item 都尝试过)。
2. summarize.output.failedCount == 3，且 allSucceeded == false。
3. notifyPartialFailure 跑一次;notifySuccess 被 skip。
4. 在 batch_size = 10 下，测试观察到的并发峰值不超过 10 （在 MockOperator.of(...) 外层用 AtomicInteger active 记录当前并发，再用第二个 peak 计数器在每次进入时更新峰值）。

这套组合——batch_size + on_item_failure = continue + 基于聚合的下游 分支——就是生产环境做韧性批处理的标准模式。

34.11.4.2 状态轮询到事件驱动：用 Await 替代 Loop

Fixture：status-polling.bloge（起点）和 payment-wait.bloge（参考）

需求：

本实验分为两个部分。

Part A ——扩展轮询循环：

1. 在 fetchResult 之后添加一个 notifyReady 节点，当结果可用时发送通知。输入：result = fetchResult.output。
2. 给循环主体内的 checkStatus 添加 timeout = 5s。
3. 验证：断言 notifyReady 在 fetchResult 之后执行。

Part B ——用 **await** 替代循环：

1. 改写 graph，使用 await 代替轮询循环。在 submitJob 之后，声明：

```
await awaitJobReady {
  event "job.ready" where jobId = submitJob.output.jobId
  timeout = 60s
```

```
    on_timeout {
      status = "timeout"
      reason = "Job did not become ready within 60 seconds"
    }
  }
```

2. 添加一个 branch on awaitJobReady.output.status：
   - "ready" → fetchResult
   - otherwise → handleTimeout（一个新节点，记录失败信息）
3. 验证：编写两个测试——一个模拟事件到达（断言 fetchResult 执行、handleTimeout 被跳过），一个模拟超时（断言 handleTimeout 执行、fetchResult 被跳过）。

练习的概念： loop、await、事件关联、on_timeout、挂起分支、轮询 vs 事件驱动设计的对比（第 10、11、12 章）。

34.11.4.3 信贷审批 decision_table 目标：将一个基于分支的信用等级分类器重构为 decision_table 节点，然后探索多策略决策表和运行时错误路径。

第 1 步——基线：分支堆栈。打开（或创建）ch22/credit-approval-baseline.bloge。该图使用 branch on applicant.output.score，有三个显式 case 和一个 otherwise。确认它通过所有现有测试。

第 2 步——重构为 **hit=unique** + 命名字段输出。用以下代码替换分支：

```
decision_table credit_tier(
  score = applicant.output.score
) hit=unique -> { tier: String, rate: Decimal } {
  rule (score: score >= 750) -> { tier: "platinum", rate: 3.25 }
  rule (score: 680 <= score < 750) -> { tier: "gold", rate: 4.50 }
  otherwise -> { tier: "rejected", rate: 0.0 }
}
```

更新下游接线，使用 credit_tier.output.tier 和 credit_tier.output.rate。运行测试——所有测试均应通过。

第 3 步——添加 **hit=collect** 进行折扣累积。添加第二张表，收集所有适用的折扣标签：

```
decision_table applicable_discounts(
  score = applicant.output.score,
  amount = loan.output.amount
) hit=collect -> String {
  rule (score: score >= 700) -> "loyalty"
  rule (score: score >= 760) -> "premium"
  rule (amount: amount <= 100000) -> "small-loan"
}
```

将 applicable_discounts.output.items 接线到通知节点。编写一个测试，断言 score=780, amount=80000 返回三个折扣标签。

第 **4** 步——触发错误路径。编写两个负面测试用例：

1. 从 hit=unique 表中删除 otherwise，然后传入 score=500。断言图抛出 DecisionTableViolationException，code 为 RUNTIME_DECISION_TABLE_NO_MATCH。
2. 在 score >= 750 规则旁边添加一条重叠规则 score >= 700（两者在 score=800 时都会命中）。断言 hit=unique 时引擎抛出 RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH。验证后恢复表内容。

第 **5** 步——**bloge lint** 演示。从 hit=unique 表中删除 otherwise 子句，运行 bloge-lint（或观察编辑器警告）。确认 decision-table/missing-otherwise WARNING 出现。重新添加 otherwise 使 lint 通过。

退出标准：

- hit=unique + 命名输出：所有等级与利率断言通过。
- hit=collect：折扣累积测试通过（score=780, amount=80000 返回 3 个标签）。
- 两个错误路径测试均断言了正确的 code 字符串。
- 恢复 otherwise 后，bloge-lint 报告零 WARNING。

练习的概念：decision_table、命中策略（unique、collect）、命名字段输出、RUNTIME_DECISION_TABLE_NO_MATCH、RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH、decision-table/missing-otherwise lint 规则、DecisionTableViolationException 断言（第 5 章，附录 F）。

34.12 毕业运行与证据检查

毕业 capstone 最后必须回到一个由业务负责人定义的故事。在固定的 BLOGE 0.9.8-RC1 源码树运行基线测试：

```
mvn -pl bloge-starter-loan-approval -am \
  -Dtest='LoanApprovalApplicationTest,LoanApprovalBusinessScenarioTest,LoanApprovalOperatorFlowTest' \
  -Dsurefire.failIfNoSpecifiedTests=false test
```

预期范围是三个 test classes、11 个 tests：三个决策路径，以及放款成功、重试、fallback 和 compensation 行为。只把它当作 focused starter evidence。再运行你的 毕业变体，并明确记录测试数、Scenario verdict、evidence trust 和未证明事项。用第 24 章检查业务映射，再用第 23 章 写出 green run 仍不能证明什么。

完整参考解、常见错误和 20 分量表位于 solutions/ch34/teacher-guide.md。教师应在学员 提交六类资产后再开放；正文不紧邻泄露答案。

34.13 实验验收卡

- 预期与观察：三级 capstone 从 parser 到贷款 Case 再到证据治理，交付六类资产。
- 失败与恢复：让基线、Case 或证据绑定失败；只修该层，未绿不升级。
- 证明边界：证明学员变体满足毕业合同，不证明生产发布获批。
- 练习合同：前 33 章资产和固定 RC1；每级只增一类能力；交付六类资产；91 个基线测试及本级条件通过即停止。

34.14 回顾

- 基础 **capstone** 证明你能读懂、扩展并用 graph contract 验证订单 DAG。
- 进阶 **capstone** 证明你能让 fallback、分支与可观测事实保持可解释。
- 毕业 **capstone** 把 graph、测试、Scenario/Policy/Fixture、证据解读、架构图和 未证明事项组合成一个可复核作品。
- 三个可选扩展题覆盖 batch、await 与 decision table，但不改变毕业完成标准。
- 测试、业务验证和发布批准是三层责任；任何一层通过都不能代替后一层。

34.15 下一步

你已经完成了 Head First BLOGE。最后一次产出不是“又一份 DSL”，而是一套别人 能够运行、解释和质疑的毕业作品。从这里开始：

- 发布一个真实的 **graph** ——在你的系统中挑选一个工作流，编写 DSL，接入 operator，并使用 `GraphTestRunner` 测试它。
- 探索高级特性 ——尝试在测试夹具中接入 `DurableSessionManager` 和 `DurableStateMachineManager` 来练习可崩溃恢复的编排模式。如果你的场景涉及 LLM 工具调用，可以探索 `bloge-agent-ext` 中的 `agent` DSL（在第 17 章中讲解）。
- 当你需要精确的语法或边界行为时，查阅参考文档层：
  - DSL 规范
  - Operator 设计规范
  - 核心架构
  - 入门指南
  - 附录 F ——决策表
- 探索高级示例，位于 `bloge-examples/`——尤其是 `food-order.bloge`、`claim-processing.bloge` 和 `shipment-planning.bloge`，这些 graph 综合运用了你所学的一切。
- 配置工具 ——安装 VS Code 扩展 或 IntelliJ 插件，在编写时获得实时诊断。

34.16 参考链接

- 项目 README ——模块概览与快速开始
- 入门指南 ——完整安装指南

- DSL 规范——正式语法参考
- Operator 设计规范——operator 契约与生命周期
- 核心架构——引擎内部结构
- `GraphTestRunner.java`——测试工具源码
- `MockOperator.java`——测试替身源码
- `DslTestHelper.java`——DSL 编译辅助源码
- `order-process.bloge`——基础起点
- `ticket-routing.bloge`——进阶起点
- `loan-approval.bloge`——毕业起点
- `batch-order-parallel.bloge`——可选 batch 题
- `status-polling.bloge`——可选 loop 题
- `payment-wait.bloge`——可选 await 参考
- `order-saga.bloge`——saga 补偿参考

章节

—

第七阶段回顾——毕业

第 *33–34* 章用一个标准结束全书：毕业不等于“用遍 *BLOGE* 的所有能力”，而是选择能让关键行为可见的最小模型，并知道何时停止。

0.1 最后的决策闭环

步骤	问题	产出
STOP	这只是顺序代码、一次远程调用或 CRUD 吗?	一份“不使用 BLOGE”的明确理由
PROBE	编排风险真实存在，但仍不确定吗?	一条细纵切与一个对照基线
ADOPT	依赖、等待、恢复或证据是否值得引入 graph?	写明约束、成本与退出条件的 ADR
GRADUATE	另一个团队能否运行、运维并质疑结果?	可运行系统、测试、架构图、证据与限制说明

第 33 章从约束而不是功能出发。第 34 章提供基础、进阶、毕业三个项目层级。毕业层必须交付六类 artifact：业务故事、可执行流程、行为测试、架构视图、运行证据、限制说明。可选功能只有在解决已声明约束时才有价值。

0.2 试一试：拿掉一个功能

从毕业项目里拿掉最“炫”的 BLOGE 功能。如果业务故事、恢复边界和证据仍然清楚，就不要把它加回来；如果清晰度因此坍塌，请写出必须恢复该功能的精确约束。那句话就是最终 ADR 的核心。

0.3 本书最后一个结论

你得到的不是功能目录，而是一套方法：建模依赖、说清生命周期 owner、控制时间与输入、把 claim 绑定到 evidence，并停在证据真正支持的最强结论上。

附录 A ——内置函数

章节

—

附录 A ——内置函数

bloge DSL 中所有的 *guard*、*constraint* 与表达式共享同一组内置函数。本附录是这套工 具箱的唯一参考：每个函数名、每个签名、每个陷阱都在这里。当你开始写真实的 *DSL*，就把它收藏起来。

0.1 什么时候需要这份参考

问题	章节
`coalesce` / `ifNull` / `isNull` 各自做什么?	第 5 节：Null 与默认值
`when:` 守卫里能用正则吗?	第 1 节：字符串函数
怎么对列表字段求和或求平均?	第 2 节：数学与聚合
怎么读嵌套 map 里的值?	第 4 节：Map 与对象
怎么格式化时间戳或计算时长?	第 7 节：日期/时间
怎么签名或哈希一段载荷?	可选模块：`bloge-functions-crypto`
怎么判断某个函数能否在编译期使用?	纯函数与编译期求值

0.1.1 按任务查函数

我要做什么	先查哪里
清洗、拆分、匹配或格式化文本	字符串函数
汇总、比较、取整或聚合数值	数学与聚合
过滤、映射、展平或检查列表	集合函数
安全读取嵌套对象或 map	Map/对象与 null 处理
在 schema 校验前转换类型	类型转换
计算 deadline 或 duration	日期/时间函数
创建摘要或签名	可选 crypto 模块

第 5 章 (会分支的图) 在 `when:` guard 里引入了这些函数。第 7 章 (设计良好的算子) 在 算子的 `constraints` 与 `usageExample` 中使用它们。第 15 章 (状态机) 在转换 guard 中使用它们。在 bloge DSL 中只要见到表达式，能调用的就是本附录列出的这套函数。

0.2 函数调用语法

所有函数都使用标准调用语法:

```
when: concat("Hello, ", ctx.name) == "Hello, Alice"
when: contains(split(ctx.tags, ","), "premium")
when: padLeft(toString(ctx.id), 10, "0")
```

bloge 没有 Unix 风格的管道操作符 (|)。| 在词法分析中保留给逻辑或 (||)。要组合多个函数，就把调用嵌套起来。

0.2.1 Null 处理约定

核心内置函数默认 **null** 安全。传 null 时要么:

- 返回 null (safe-nav 语义)；要么
- 返回一个合理的默认值——size(null) → 0、isEmpty(null) → true、concat(null, "x") → "x" 等等。

你可以依赖这个约定，少写很多防御性判断。可选的加密模块是唯一的例外，参见下文。

0.2.2 纯函数与编译期求值

绝大多数函数是纯函数——相同输入总是产生相同输出，没有副作用。当参数是常量 时，DSL 编译器允许在编译期折叠这些纯函数调用。

少数函数是非纯函数，不可折叠:

- now() ——当前时间戳，来自引擎的 time source
- today() ——当前日期，来自引擎的 time source
- uuid() ——随机 UUID
- secret(name) ——密钥查询 (加密模块)

在严格的" 编译期专属" 位置 (常量、schema constraint)，编译器会拒绝非纯函数调用。把 它们留到运行期的 guard 里使用。

0.3 1. 字符串函数 (17)

函数	签名	返回	说明
concat	concat(arg1, arg2, …)	String	把所有参数按字符串拼接。null 视为空串。
substring	substring(str, start [, end])	String	从 start (含) 到 end (不含)；自动钳制越界。
uppercase / upper	uppercase(str)	String	转大写。
lowercase / lower	lowercase(str)	String	转小写。
trim	trim(str)	String	去掉首尾空白。
replace	replace(str, oldStr, newStr)	String	把所有字面量 oldStr 替换为 newStr。

函数	签名	返回	说明
startsWith	startsWith(str, prefix)	Boolean	前缀判断；遇 null 返回 false。
endsWith	endsWith(str, suffix)	Boolean	后缀判断；遇 null 返回 false。
indexOf	indexOf(str, target)	Number	第一次出现的下标，找不到返回 -1。
length / len	length(str)	Number	字符串长度；列表、map 同样适用；null 返回 0。
matches	matches(str, regex)	Boolean	正则匹配；null 或正则错误返回 false (不抛异常)。
replaceAll	replaceAll(str, regex, replacement)	String	正则全量替换；null 或正则错误返回原值。
split	split(str, delimiter)	List	按字面分隔符 (非正则) 切分；null 返回空列表。
join	join(list, delimiter)	String	用分隔符拼接列表元素；null 元素变空串。
padLeft	padLeft(str, length, padChar)	String	用单字符左填充到指定长度。
padRight	padRight(str, length, padChar)	String	用单字符右填充到指定长度。

```
when: startsWith(ctx.email, "admin@") and contains(ctx.email, "@")
when: matches(ctx.phone, "^\\+?[0-9 ]{6,}$")
when: padLeft(toString(ctx.orderId), 8, "0") == "00012345"
```

0.4 2. 数学与聚合 (10)

函数	签名	返回	说明
abs	abs(n)	Number	绝对值。
min	min(a, b)	Number	两数最小值。
max	max(a, b)	Number	两数最大值。
round	round(n)	Number	四舍五入到整数 (Math.round)。
ceil	ceil(n)	Number	向上取整。
floor	floor(n)	Number	向下取整。
clamp	clamp(n, min, max)	Number	把 n 钳到 [min, max]。
pow	pow(base, exponent)	Number	base ^ exponent。
sum	sum(list)	Number	数值元素求和；null 跳过；空列表返回 0。
avg	avg(list)	Number	数值元素求平均；空或 null 返回 null。

```
when: sum(ctx.items.*.price) > 1000
when: clamp(ctx.retries, 0, 5) == ctx.retries
```

0.5 3. 集合函数 (13)

函数	签名	返回	说明
size	size(collection)	Number	列表 / map / 字符串的长度；null 返回 0。
contains	contains(collection, item)	Boolean	成员判断。列表 → 精确匹配；字符串 → 子串；map→ 是否含该键。
first	first(list)	any	第一个元素；空或 null 返回 null。
last	last(list)	any	最后一个元素；空或 null 返回 null。
isEmpty	isEmpty(collection)	Boolean	空或 null 返回 true。
distinct	distinct(list)	List	去重并保持原顺序。
flatten	flatten(listOfLists)	List	展平一层；非列表元素透传。
sort	sort(list)	List	自然序排序，落到 toString 作为后备；null 排在前面。
take	take(list, n)	List	前 n 个元素。
drop	drop(list, n)	List	跳过前 n 个元素。
reverse	reverse(list)	List	反转顺序。
any	any(list, value)	Boolean	是否存在等于 value 的元素。
all	all(list, value)	Boolean	是否所有元素都等于 value；空列表返回 true。

```
when: contains(ctx.tags, "premium") and size(ctx.items) > 0
when: any(ctx.errors, "TIMEOUT")
when: all(distinct(ctx.statuses), "OK")
```

0.6 4. Map 与对象 (5)

函数	签名	返回	说明
keys	keys(map)	List	所有键作为列表返回。
values	values(map)	List	所有值作为列表返回。
entries	entries(map)	List	返回 {key, value} map 列表。
merge	merge(map1, map2)	Map	浅合并；map2 覆盖 map1。
has	has(map, key)	Boolean	map 是否含该键。

```
when: has(ctx.headers, "X-Trace-Id")
when: size(keys(ctx.attributes)) > 3
```

0.7 5. Null 与默认值 (3)

函数	签名	返回	说明
coalesce	coalesce(v1, v2, …)	any	第一个非 null 的参数；全都是 null 则返回 null。
isNull	isNull(value)	Boolean	是否为 null。
isNotNull	isNotNull(value)	Boolean	是否非 null。

coalesce 是默认值的主力。如果只需要一个回退值，第 9 节“高级工具”中的 ifNull 读起来更自然。

```
when: isNotNull(ctx.userId)
let region = coalesce(ctx.region, ctx.country, "GLOBAL")
```

0.8 6. 类型转换 (5)

函数	签名	返回	说明
toString	toString(value)	String	任意值转字符串。
toNumber	toNumber(value)	Number	从字符串解析数值，或直接返回数值；无法解析的字符串归约为 0。
toBoolean	toBoolean(value)	Boolean	解析 "true" / "false" 字符串或透传布尔值。
toInt	toInt(value)	Number	转整数；截断小数。
typeOf	typeOf(value)	String	返回 "String"、"Number"、"Boolean"、"List"、"Map"、"Object" 或 "Null"。

```
when: toNumber(ctx.amountStr) > 100
when: typeOf(ctx.payload) == "Map"
```

0.9 7. 日期/时间 (6)

所有日期都以 **ISO-8601** 字符串 (yyyy-MM-ddTHH:mm:ssZ) 流转。用以下函数格式化、解析、加减和计算时长。

函数	签名	返回	纯?	说明
now	now()	String	[X]	来自 GraphEngine.currentTimeSource() 的当前时间戳。
today	today()	String	[X]	当前日期 yyyy-MM-dd (UTC)。
formatDate	formatDate(isoStr, pattern)	String	[OK]	用 DateTimeFormatter 模式格式化 ISO 时间 (如 "yyyy-MM-dd HH:mm")。
parseDate	parseDate(str, pattern)	String	[OK]	按模式解析；返回 ISO-8601 时间。
addDuration	addDuration(isoStr, durationStr)	String	[OK]	加一段时长。支持 "2h"、"30m"、"1d"、"500ms" 或 ISO-8601 形式 "PT2H"。
diffDuration	diffDuration(iso1, iso2, unit)	Number	[OK]	两个时间点之间的距离。unit: "s"/"seconds"、"m"/"minutes"、"h"/"hours"、"d"/"days"。

```
when: diffDuration(ctx.createdAt, now(), "h") > 24
let expiresAt = addDuration(now(), "30m")
```

确定性小贴士。在对重放敏感的场景 *(durable session、确定性测试)* 中，优先用 基于存储时间戳的 *parseDate/addDuration*，而不要直接调用 *now() / today()*。

0.10 8. 工具 / ID (2)

函数	签名	返回	纯?	说明
uuid	uuid()	String	[X]	随机 UUID v4。
format	format(template, arg1, arg2, ⋯)	String	[OK]	使用 Java String.format 风格的占位符 (%s、%d 等)；格式错误时返回模板原文。

```
let traceId = uuid()
let message = format("Order %s failed after %d retries", ctx.orderId, ctx.retries)
```

0.11 9. 高级工具 (8)

函数	签名	返回	说明
regexExtract	regexExtract(str, pattern [, group])	String	取正则匹配的分组 (默认 1)；未匹配返回 null。
regexExtractAll	regexExtractAll(str, pattern)	List	所有不重叠的匹配。
getField	getField(obj, fieldName)	any	通过反射读 map 或 bean 的字段；null 安全。

函数	签名	返回	说明
setField	setField(obj, fieldName, value)	Map	返回更新字段后的浅拷贝 (不可变语义)。
template	template(str, bindings)	String	用 map 替换 ${key} 占位符。
range	range(start, end)	List	整数序列 [start, end)。
mapOf	mapOf(k1, v1, k2, v2, …)	Map	用交替的键值参数构造 map。
listOf	listOf(e1, e2, …)	List	用参数构造列表。
ifNull	ifNull(value, default)	any	非 null 返回 value，否则返回 default。

```
let host = regexExtract(ctx.url, "https?://([^/]+)/")
let updated = setField(ctx.user, "lastSeenAt", now())
let greeting = template("Hello, ${name}!", mapOf("name", ctx.name))
```

0.12 10. JSON 函数 (2)

当 DSL 编译器配置了 JsonCodec，JSON 工具就会注册到函数表里——BuiltInFunctions.registerAll(registry, jsonCodec)。

函数	签名	返回	说明
toJson	toJson(obj)	String	通过配置的 codec 序列化为 JSON。
fromJson	fromJson(str)	any	反序列化为 Map / List / 基础类型。

```
let payload = toJson(mapOf("orderId", ctx.orderId, "total", ctx.total))
let parsed = fromJson(ctx.responseBody)
```

若未配置 JsonCodec，DSL 编译时将以” 未知函数” 报错。

0.13 可选模块 (bloge-functions-crypto)

当 bloge-functions-crypto JAR 出现在类路径上，Java ServiceLoader 会自动加载该模 块。它提供三组函数。涉及密钥解析的加密函数还需要在引擎上配置 SecretProvider。

0.13.1 加密 (5)

函数	签名	返回	纯?	说明
hmacSha256	hmacSha256(data, keyRef)	String	[OK]	HMAC-SHA256 hex 摘要；keyRef 通过 SecretProvider 解析。
hmacSha512	hmacSha512(data, keyRef)	String	[OK]	HMAC-SHA512 hex 摘要。
aesEncrypt	aesEncrypt(plaintext, keyRef [, ivRef])	String	[OK]	AES-256-CBC；base64 输出；不传 IV 时随机生成。
aesDecrypt	aesDecrypt(ciphertext, keyRef [, ivRef])	String	[OK]	AES-256-CBC；输入 base64。
secret	secret(name)	String	[X]	从 SecretProvider 按名字解析密钥。

0.13.2 哈希 (4)

函数	签名	返回	说明
md5	md5(str)	String	MD5 hex。
sha1	sha1(str)	String	SHA-1 hex。
sha256	sha256(str)	String	SHA-256 hex。
sha512	sha512(str)	String	SHA-512 hex。

0.13.3 编码 (6)

函数	签名	返回	说明
base64Encode	base64Encode(str)	String	UTF-8 字符串 base64 编码。
base64Decode	base64Decode(str)	String	base64 解码；输入非法返回 null。
hexEncode	hexEncode(str)	String	UTF-8 字符串 hex 编码。
hexDecode	hexDecode(str)	String	hex 解码；输入非法返回 null。
urlEncode	urlEncode(str)	String	URL 编码 (UTF-8)。
urlDecode	urlDecode(str)	String	URL 解码 (UTF-8)；输入非法返回 null。

```
let signature = hmacSha256(ctx.requestBody, "webhook-secret")
let fingerprint = sha256(ctx.userId)
let safeQuery = urlEncode(ctx.search)
```

加密模块的例外。与核心函数不同，加密 / 哈希函数遇到非法输入 *(*错误密钥、损坏 的密文*)* 可能抛异常。请在边界做校验，或把调用放进能处理对应失败模式的算子内部。

0.14 常见错误

- 管道操作符。 `value | uppercase` 在 bloge DSL 不合法。要写 `uppercase(value)`。词法器把 `|` 当作 `||` 的开头来读。
- 正则切分。 `split(s, ",")` 是按字面字符串 `","` 切分，不是正则。如果需要按 模式切分，先用 `replaceAll` 预处理。
- 不传 **ISO** 字符串做日期运算。`addDuration("2024-01-01", "1d")` 会失败——第一个参数必须是 ISO-8601 时间。如果你拿到的是裸日期，先用 `addDuration(parseDate("2024-01-01", "yyyy-MM-dd"), "1d")`。
- 就地修改 **map**。 `setField` 不会原地修改，它返回一个副本。要绑定结果: `let updated = setField(ctx.user, "name", "Alice")`。
- 常量位置调用非纯函数。 `now()`、`uuid()` 在编译期专属位置 (常量折叠、schema 常量) 会被拒绝。把它们移到运行期 guard 或算子输出里。
- 没配 **`SecretProvider`** 就用加密。 `hmacSha256` / `aesEncrypt` 在缺少 `SecretProvider` 时会失败。在 Spring 或手工 bootstrap 里注入它 (见第 20 章)。
- **`toBoolean`** 不接受数字。 `toBoolean(1)` 返回 `null`，不是 `true`；只接受 `"true"` / `"false"` 字符串或原生布尔值。

0.15 与正文的关联

- 第 **5** 章——会分支的图。 `when:` guard 的引入章节，也是组合这些函数最主要的 场所。
- 第 **7** 章——设计良好的算子。在算子的 `constraints` 与 `usageExample` 表达式 中使用同一套函数。
- 第 **14** 章——多轮 **session** 与 第 **15** 章——状态机。 session 阶段转换与状 态机转换的 guard 都使用 guard 表达式，两者通过同一个 `ExpressionEvaluator` 编 译。
- 第 **9** 章——工具链工作流。 `operator-metadata.json` 导出器会把这些函数名暴 露给 IDE，让补全和校验与编译器的能力保持一致。
- 第 **20** 章——**Spring** 与生产接线。 `JsonCodec` 与 `SecretProvider` 在此被装配，从而启用上文的 JSON / 加密扩展。

校验。 72 个核心函数对应 `bloge-core` 中的 `BuiltInFunctionsTest.registerAll_populatesExpectedNumberOfEntries()`。再加上 2 个 JSON 工具与 11 个 crypto/hash/encoding 工具，总数即本附录列出的 **85** 个。

附录 B ——Session 速查表

章节

—

附录 B ——Session 速查表

请先读第 *14* 章。编写或排查多轮 *session* 时再回到这里。

0.1 选择边界

需求	使用	不要使用
一次外部挂起与恢复	Graph `await`	只有一轮的伪 session
外部反复交互，每轮处理相似	`session`	手工包住 HTTP 请求的循环
命名业务状态与回退迁移	`state_machine`	用 phase 名伪装 state

0.2 Phase 卡片

项目	必须回答的问题
`ONCE` 或 `ROUND`	该 phase 只运行一次，还是每个 signal 都运行一次?
`max_rounds`	什么有限预算能终止失控对话?
`until`	哪个可观察结果结束 phase?
signal schema	这个边界允许什么 payload?
timeout / idle timeout	没有 signal 时，等待如何结束?
owner	哪个外层运行时拥有身份、恢复与终止?

0.3 运行时动作

动作	含义	应保留的证据
start	创建一个稳定 session identity	Definition version 与初始 context
signal	交付一次经过校验的 round payload	Signal type、correlation identity、接受/拒绝结果
restore	从最后一个持久快照恢复	Snapshot version 与原始 deadline

动作	含义	应保留的证据
terminate	因策略而非正常 `until` 结束	稳定 reason 与最终 status

0.4 失败扫描

- 无效 signal 必须在修改 round state 前被拒绝。
- Phase 与 session 轮数上限必须有限。
- 测试迟到、重复与并发 signal。
- 声称崩溃恢复时使用 `bloge-session-durable`；内存 session 不构成持久性证据。
- 嵌套编排中的外部 signal 必须先进入声明的外层 owner。

主线请看第 14 章与第 16 章。精确契约请查 BLOGE 源码树中的 `docs/session-phase-round-specification.md`、`bloge-session-ext` 与 `bloge-session-durable`。

章节

附录 C ——Streaming

普通 *graph* 会在上游节点完全结束后才把输出交给下游。*Streaming* 改变了这个契约：数据在上游仍在产出的过程中，通过 *channel* 逐条流向下游。

0.1 什么场景需要 Streaming

大多数 BLOGE graph 采用”批量完成”模式：每个节点执行完毕、输出被存储，之后下游节点才能看到该输出。当数据集有界且可以舒适地放进内存时，这种模式完全够用。

以下场景需要 streaming：

- 上游增量产出数据 ——音频块、数据库游标分页、实时传感器读数——你希望一边收到一边处理。
- 数据集大到一次性物化很浪费 ——你更希望通过有界缓冲区做流水线。
- 延迟比吞吐量更重要 ——在剩余数据还在计算时就向用户展示部分结果。

如果以上都不适用，普通节点和 `foreach` 更简单，也更容易测试。如果你真正困难的不是数据量，而是生命周期——多轮外部交互、命名状态或信号所有权——那么应该继续阅读第 14 章到第 16 章，而不是优先选择 streaming 原语。

0.2 三种 Streaming 原语

BLOGE 提供三种产出流式输出的构造，各服务于不同模式。

0.2.1 1. `stream node` ——流式源或变换

`stream node` 声明一个 operator，其输出通过内部 `NodeChannel` 逐条发射，而非作为单个值返回。

```
stream node audioCapture : AudioCapture {
  buffer = 64
}
```

Operator 按自身节奏向 channel 推送条目。下游节点可以实时消费 channel，也可以等待 channel 关闭后接收物化后的完整列表。

适用场景：单个 operator 产出无界或长时运行的条目序列（音频采集、事件流、分页 API）。

0.2.2 2. stream foreach ——遍历集合并流式输出

stream foreach 是 foreach 的流式变体。它遍历集合，每个元素处理完成后立即将结果发射到下游，而非等所有元素都处理完。

```
stream foreach processOrders : item in loadOrders.output.orders {
  buffer = 16
  node processItem : OrderProcessor {
    input {
      order = item
    }
  }
}
```

适用场景：集合是有界的，但你希望下游节点在最后一个元素处理完之前就开始消费结果。

0.2.3 3. stream loop ——轮询并流式输出

stream loop 是 loop 的流式变体。每次迭代的输出会立即发射到下游，循环持续到 until 满足或达到 max_iterations。

```
stream loop checkStatus {
  max_iterations = 100
  delay = 2s
  depends_on = [initMonitor]
  buffer = 8
  node pollStatus : StatusPoller {
    input {
      serviceId = initMonitor.output.serviceId
      iteration = loopIteration
    }
  }
  until pollStatus.output.status == "ready"
}
```

适用场景：你在轮询或持续产出条目，希望立即转发每个结果，而不是收集所有迭代后再一次性交付。

0.3 .stream 与 .output ——两种边类型

当下游节点引用流式节点的结果时，边的类型决定了数据何时以及如何到达。

语法	边类型	行为
upstream.stream	**StreamEdge**	下游实时接收条目，逐条到达。下游 operator 必须能处理增量输入。

`upstream.output`	**DirectEdge**	下游等待流关闭后，接收完整的物化 `List<T>`。适用于下游 operator 需要全部数据才能开始工作的场景。

0.3.1 如何选择

问自己：下游 **operator** 是否需要所有数据才能开始?

- 是 → 使用 `.output`。例如：报告生成器需要所有已处理的订单来计算汇总。
- 否 → 使用 `.stream`。例如：语音转文字节点在音频块到达时即开始转录。

voice pipeline 示例在同一个 graph 中展示了两种边：

```
/// 通过 StreamEdge 实时接收音频块
stream node speechToText : SpeechToText {
  input {
    audio = audioCapture.stream       // StreamEdge —— 实时转发
  }
  buffer = 16
}

/// 通过 DirectEdge 等待完整转录
node textAnalysis : TextAnalyzer {
  depends_on = [speechToText]
  input {
    transcript = speechToText.output   // DirectEdge —— 物化 List<T>
  }
}
```

0.4 缓冲区大小

每个流式构造都有 `buffer` 参数，用于设置内部 `NodeChannel` 环形缓冲区的容量。

```
stream node audioCapture : AudioCapture {
  buffer = 64    // 环形缓冲区最多持有 64 个条目
}
```

默认值： 16 个条目。

0.4.1 如何思考缓冲区大小

缓冲区解耦了生产者速率和消费者速率。当缓冲区满时，生产者会阻塞（背压），直到消费者追上来。

场景	建议
生产者远快于消费者	增大缓冲区以吸收突发流量；差距过大时背压仍然会启动。
生产者和消费者速率接近	默认值 16 通常够用。

场景	建议
内存受限或条目较大	减小缓冲区以限制内存使用；接受更频繁的背压暂停。
延迟敏感的流水线	保持缓冲区较小，避免条目排队时间过长。

没有万能公式——从默认值开始，测量，再调整。当可观测性监听器激活时，引擎会记录背压事件，调优变得很直观。

0.5 Back-pressure、取消与持久化边界

事件	契约	要测试什么
Buffer 已满	Producer 等待，不能静默丢 item	放慢 consumer，观察内存有界且最终继续
下游取消	取消向上游传播并关闭 channel	断言 producer 停止，取消后不再发射
Producer 失败	Channel 以失败关闭，物化 `.output` 不能伪装完整	断言下游失败与部分输出策略
进程崩溃	内存 channel 内容不是 durable checkpoint	重启并证明哪些 item 来自 source 或 store 重放

Streaming 与 durable execution 解决不同问题。有界 channel 控制运行进程里的实时流动，不会让途中 item 自动跨崩溃保存。需要 replay 时，必须说出 durable source offset 或 checkpoint，并定义重复处理。不要从 buffer 大小推导 exactly-once delivery。

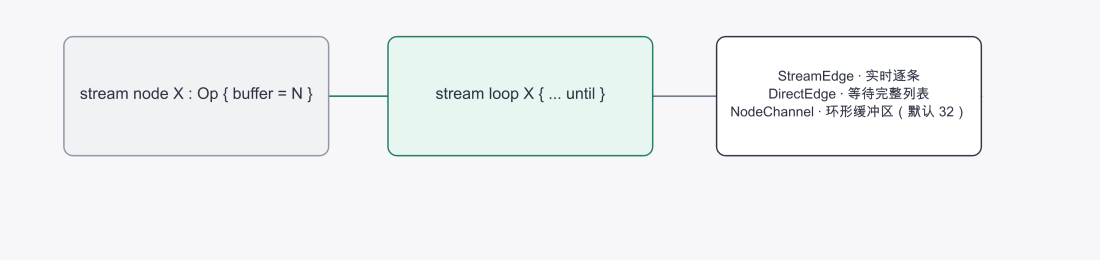
0.6 完整示例

仓库中以下文件用真实场景演示了三种 streaming 原语。

文件	原语	展示内容
`streaming-batch.bloge`	`stream foreach`	流式处理客户订单；下游报告使用 `.output`（DirectEdge）物化全部结果。
`streaming-status-monitor.bloge`	`stream loop`	每 2 秒轮询服务状态，将每次状态更新流式转发到下游；服务就绪时提前退出循环。
`voice-pipeline.bloge`	`stream node`	音频采集 → 语音转文字 → 文本分析。在同一个 graph 中展示 `.stream`（实时）和 `.output`（物化）两种边。

打开这些文件，阅读文档注释，追踪数据如何从流式源经过缓冲区流向下游消费者。修改 `buffer` 值，观察背压行为的变化。

0.7 速查表



0.8 常见错误

错误	为什么会犯	如何纠正
在 operator 需要全部数据时使用 `.stream`	习惯性地选择”更快的路径”	如果 operator 在看到所有条目之前无法产出输出，就使用 `.output`。
把缓冲区设得很大”以防万一”	想完全避免背压	过大的缓冲区会延迟背压信号并浪费内存。从默认值开始。
忘了 `stream foreach` 同时也在收集 `.output`	以为 streaming 意味着什么都不会物化	条目被流式转发的同时也在收集；`.output` 会在流关闭后提供完整列表。
不测试背压路径	只用小数据集测试，缓冲区从未被填满	在测试中用 `MockOperator.delaying(...)` 模拟慢消费者，验证 graph 在背压下的行为。

0.9 与主线章节的关系

Streaming 没有被安排在单独一章，因为它依赖多个概念：

- 节点和依赖 ——第 2 章和第 3 章
- 批处理和迭代 ——第 11 章引入了 `foreach` 和 `loop`；streaming 变体在此基础上增加了 channel。
- 测试 ——第 18 章展示了如何对流式 graph 结果做断言。
- 可观测性 ——第 19 章涵盖了记录背压事件的监听器。

本附录将 streaming 的心智模型集中在一处，方便你在需要增量数据流时随时查阅。

章节

—

附录 D ——状态机

第 *15* 章用叙事方式讲状态机，第 *16* 章讲它与 *session* 的组合。本附录是 查阅手册 ——当你编写一台稍微复杂的状态机时，会反复回到这里 查看操作符表、配置项、监听器回调和 *checkpoint* 字段。

0.1 什么时候需要本附录

问题	去这里
” 怎么用具名状态建模生命周期?”	第 15 章
” 状态机和 session 怎么嵌套?”	第 16 章
”when guard 里允许哪些运算符?”	本附录——Guard 运算符目录
”checkpoint 写入时引擎持久化了什么?”	本附录——Checkpoint 结构
” 哪个回调能给我 state-enter / state-exit 的时机?”	本附录——监听器 SPI 速查
” 这个流程到底该用 SM、session 还是普通 graph?”	本附录——选型表

如果还没读过第 15 章，先从那里开始。本附录默认你已经知道 `[initial]`、`on EVENT -> target` 和 `[terminal]` 是什么意思。

0.2 DSL 关键字一览

第 15 章用到的每一个关键字，一张表说清。关键字都写在 `state machine` 块里，大小写敏感。

关键字	位置	作用
`[initial]`	状态头标记	一台机器恰好有一个初始状态，执行从这里开始。
`[terminal]`	状态头标记	终态可以有多个，进入任意一个就把状态置为 `COMPLETED`。
`on EVENT -> target`	状态内部	等待一个 `event` 名匹配的信号，收到后迁移到 `target`。

关键字	位置	作用
`on EVENT when GUARD -> target`	状态内部	同上, 但只在 guard 真值时触发。多个 when 按声明顺序匹配。
`on *`	状态内部	自动迁移——进入状态后立即触发, 在执行器挂起之前。常用于必跑的子图扇出。
`timeout = <duration>`	状态内部	单状态超时, 需要配合 on_timeout 声明落点。
`on_timeout -> target`	状态内部	单状态超时的迁移目标。
`global_timeout = <duration>`	机器顶部	整机截止时间。如果触发时没有匹配的 global_transition, 状态机进入 FAILED。
`global_transitions { …}`	机器顶部	对所有非终态生效的迁移。适合可以在任何状态到达的 CANCEL / ERROR 事件。
`max_transitions = N`	机器顶部	迁移次数上限, 默认 100。超过就 FAILED。
`max_state_visits = N`	机器顶部	单个状态最多可进入的次数, 默认 10。

Duration 语法 与 *BLOGE* 其他部分一致:*30s*、*5m*、*2h*、*24h*、*PT15M (ISO-8601)*。按 *lint* 允许的写法即可。

0.3 Guard 运算符目录

Guard 表达式由 bloge-dsl 的 ExpressionEvaluator 解析, 跟 DSL 里其他 when 子句、条件边用的是同一个求值器。

0.3.1 运算符

类别	运算符	说明
算术	+ - * / %	都是数值运算;+ 不会 拼接字符串, 用字符串插值。
比较	== != < <= > >=	== / != 支持任意可比类型; 数值比较遵循 Java widening。
逻辑	&& || !	短路求值。! 是唯一的逻辑一元。
一元	-(取负)	仅数值。
空合并	??	a ?? b:a 非 null 时返回 a, 否则返回 b。
三元	cond ? a : b	右结合。
路径	. 与 […]	点号路径在上下文 map 里查; 索引也适用于 list。

0.3.2 示例

```
# 与某节点输出做数值比较
on submit when ctx.scoreNode.output.score >= 80 -> approved

# 字符串比较
on classify when ctx.input.tier == "platinum" -> fastTrack
```

```
# 空合并 + 三元
on review when (ctx.audit.output.flags ?? []) == [] ? true : false -> noManualReview

# 复合逻辑
on dispatch when ctx.geo.region == "EU" && ctx.user.consent.marketing -> sendEmail
```

0.3.3 求值规则

- 路径找不到——没有命中的路径求值为 `null`。顶层为 `null` 的 guard 视为 假, 迁移被跳过。
- 类型不匹配——数字和字符串比较结果是 `false`, 不会 抛错。Lint 在编译期能识别明显的不匹配。
- 布尔化——`null`、`false`、`0`、`""` 和空集合都是假; 其余皆真。
- 求值顺序——同一个事件下有多个 `on EVENT when …`时, 从上到下 尝试, 第一个真值的赢。最特化的 guard 写在前面。

0.4 配置项

配置项	默认	调高的场景	调低的场景
`max_transitions`	`100`	自动迁移多, 或者有受控的重试回路。	业务本应该很短; 失控循环说明缺终态。
`max_state_visits`	`10`	单状态合理地重入 (例如重试预算), 外层另有计数。	状态最多进入一次, 调低能更快失败。
`global_timeout`	未设置	业务没有自然 deadline, 但你想要跨重启的强制过期。	每个状态已有自己的 `timeout`, 机器级再设反而会盖住状态级 bug。
状态 `timeout`	未设置	状态会无限等待外部事件, 需要兜底。	状态本就靠 `on *` 自动迁出, 设置反而徒增噪音。

触发上限会发生什么: 执行器把状态置为 *FAILED*, 触发 *onStateMachineComplete* 并带 *FAILED* 标记, 然后写最后一次 *checkpoint*。之后再来的信号会被拒绝。

0.5 失败与拒绝结果

状态机 API 使用类型化 exception，而不是一个通用错误码 enum。跨服务边界时请保留类型，不要解析 message。

结果	类型化信号	含义
Event 没有合法迁移	`UnhandledEventException`	当前 state 无法消费该 event
完成后或非法时点收到 signal	`StateMachineSignalRejectedException`	实例状态拒绝 signal
State graph 失败	`StateGraphExecutionException`	所选 state 内部工作失败

结果	类型化信号	含义
迁移或访问预算耗尽	`StateMachineTransitionLimitExceeded Exception` / `StateVisitLimitExceededException`	安全上限终止失控机器
Deadline 到期	`StateMachineTimeoutExceededException`	State 或全局 timeout 结束执行
持久定义变化	`StateMachineVersionMismatchException`	Checkpoint hash 与当前定义不一致
Migration 非法	`StateMachineMigrationException`	声明的 state 映射无法应用

如果状态机使用通用 durable store，存储失败还会携带 `DurableErrorCode`；该 enum 描述 store 语义，不描述状态机业务迁移。

0.6 监听器 SPI 速查

`StateMachineListener` 是审计、指标、调试的统一钩子。所有方法都是 `default` 空实现——只覆盖你需要的那几个。

回调	触发时机	典型用途
`onStateMachineStart(StateMachine StartEvent)`	`execute(def, ctx)` 调用, 初始状态进入前。	审计开行、span 开启。
`onStateEnter(StateEnterEvent)`	迁移落地后, 该状态的图运行前。	启动状态耗时计时器、MDC 注入。
`onStateExit(StateExitEvent)`	该状态的图跑完后, 选定的迁移触发前。	关闭状态耗时计时器。
`onTransition(TransitionEvent)`	选中了一条迁移——涵盖 `on EVENT`、`on *`、`on_timeout`、global。	迁移计数器、结构化日志。
`onWaitingForEvent(WaitingForEvent Event)`	没有自动迁移命中, 执行器挂起前。	”等待信号” 计量、看板标识。
`onSignalReceived(SignalReceived Event)`	机器挂起期间收到 `signal(event, pa yload)`。	信号速率计数、payload 审计。
`onStateTimeout(StateTimeoutEvent)`	状态级 `timeout` 在事件到来之前触发。	单状态超时指标。
`onGlobalTimeout(GlobalTimeoutEvent)`	整机 `global_timeout` 触发。	告警 / on-call 通知。
`onCheckpointSaved(CheckpointSaved Event)`	一次 checkpoint 已写入存储。	复制位点、滞后探测。
`onCheckpointRestored(Checkpoint RestoredEvent)`	`resumeFromCheckpoint()` 把实例恢复出来。	”崩溃后恢复” 日志。
`onStateMachineComplete(StateMachine CompleteEvent)`	到达终态 或 机器进入 `FAILED`。	关闭审计、关闭 span。

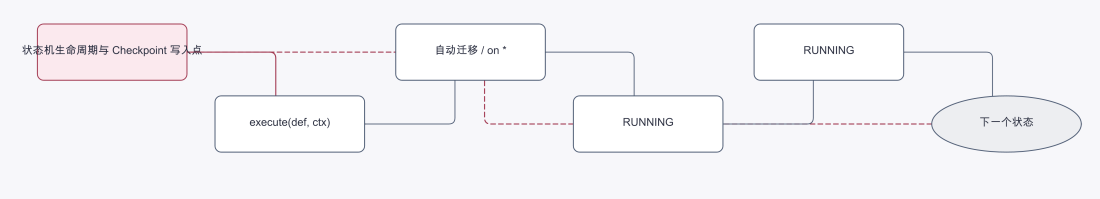
通过 `StateMachineExecutor.Builder.listeners(...)` 注册, 或在持久化路径上用 `DurableStateMachine Manager.Builder.listeners(...)`。

0.7 Checkpoint 结构

`StateMachineCheckpoint` 就是持久化 SPI 存的那条记录。理解它的结构, 在 排查部分恢复或者自己实现 `ExecutionCheckpointStateMachineStore` 时帮助 很大。

字段	类型	说明
`instanceId`	`String`	稳定标识, 由 `execute(def, ctx, instanceId)` 给出或自动生成。
`stateMachineName`	`String`	`state machine` 块上声明的 `name`。
`currentStateId`	`String`	机器所在状态; 已完成时是最后的终态。
`status`	`StateMachineStatus`	`RUNNING` / `WAITING_EVENT` / `COMPLETED` / `FAILED`。
`totalTransitions`	`int`	每次迁移加 1, 受 `max_transitions` 约束。
`stateVisitCount`	`Map<String, Integer>`	每个状态被进入的次数, 受 `max_state_visits` 约束。
`stateOutputs`	`Map<String, Map<String, Object>>`	`ctx.stateName.output.nodeId` 的命名空间树。
`sharedContext`	`Map<String, Object>`	跨状态共享、由算子写入的上下文。
`history`	`List<StateExecutionRecord>`	每个进入并退出的状态都有一条记录, 旧的在前。
`startedAt`	`Instant`	`execute(...)` 时的 wall clock。
`lastTransitionAt`	`Instant`	最近一次迁移的 wall clock。
`checkpointedAt`	`Instant`	最近一次写入的 wall clock。
`stateTimeoutDeadline`	`Instant?`	当前状态有 `timeout` 时设置, 否则 `null`。
`globalTimeoutDeadline`	`Instant?`	机器有 `global_timeout` 时设置。

两个 deadline 字段都会持久化, 这样崩溃后恢复 loop 可以按 原始 wall clock 重新挂定时器, 而不是”now + timeout”。这是持久化状态机能跨重启 保持正确性的关键。



0.8 SM / Session / 普通 graph

写 DSL 之前用这张表挑选合适的原语。

问题	是 → 用
工作负载是否会停留在一组有限的具名生命周期阶段、停留时长不固定、可能回路?	状态机 (第 15 章)
工作负载是否是与外部反复来回的对话形态, 每一轮处理逻辑相似?	**Session** (第 14 章)

问题	是 → 用
工作负载是否 两者兼有——阶段里包含对话?	组合 (第 16 章), 按信号来源选外层。
依赖关系是否是一遍跑完的纯 DAG?	普通 **graph** (从 第 2 章 开始)
是否有一个长时间运行的算子在增量产出?	流式 (附录 C)

当 变化的是阶段 而不是数据形态时, 状态机就是合适的工具。如果分支 是被工作项的 *schema* 驱动的,*DAG* 更简单。

0.9 真实示例

仓库里有验证过的 fixture 可以端到端阅读。

文件	演示什么
`order-lifecycle-state-machine.bloge`	4 个状态、有回退迁移、24h 全局超时。第 15 章的主线案例。
`ticket-state-machine.bloge`	6 个状态、升级路径、多事件流。
`order-session-with-state-machine.bloge`	模式 A——session 在外层。
`review-state-machine-with-session.bloge`	模式 B——状态机在外层 (注意信号死锁约束)。

0.10 常见错误

错误	为什么发生	修复
一台机器里有两个 `[initial]`。	拷贝草稿时忘了清掉旧标记。	编译器会拒绝。提交前先 `bloge lint`。
重试回路触发了 `max_transitions`。	重试状态在自旋, 没有退出条件。	要么调低这个状态的 `max_state_visits`, 要么加一条 `on * when ctx.retry.output.count > 3 -> giveUp`。
`signal(event, payload)` 静默返回"no transition matched"。	事件名匹配了, 但所有 `when` guard 求值为假。	在所有带 guard 的变体后面加一条 `on EVENT -> fallback`, 或在测试里用 `Mock Operator.invocations()` 审计 guard 的路径。
模式 B 的内层 session 想等待外部信号。	把内层 session 当成了外层。	执行器抛 `SM_NESTED_SESSION_AWAITS_SIGNAL`。把 session 的外部触点移到外层机器 (模式 A)——见第 16 章。
状态 `timeout` 触发但缺 `on_timeout`。	加状态时忘了兜底。	机器直接 `FAILED`。除非你就想要这个结果, 否则 `timeout` 总要配 `on_timeout`。

错误	为什么发生	修复
Checkpoint 恢复落到错的状态。	自定义 `ExecutionCheckpointStateMachineStore` 返回了陈旧的行。	该 SPI 的契约要求 `instanceId` 写后立即可读。先用 `DurableStateMachineManager` 默认的 MyBatis 存储作为基线对比。

0.11 与正文的对应

状态机的知识分散在多章, 这份附录把它收拢:

- 编写——第 15 章 讲 DSL。
- 组合——第 16 章 讲两种嵌套模式和信号死锁规则。
- 持久化——第 13 章 讲存储 SPI; 本附录 列出 checkpoint 字段。
- 测试——第 18 章 讲确定性时间控制 和快照断言。
- 运维——第 19 章 讲监听器 驱动的指标; 本附录提供回调目录。

章节

附录 E ——智能体扩展

第 *17* 章讲的是怎么在 *bloge* 上”思考”智能体。这份附录是你写真实智能体时一直 摆在旁边的查找表：每一个 *DSL* 关键字、每一条记录字段、每一个 *SPI* 回调、每 一项默认值、每一种运行时可能写出的 *finish-reason* 字符串。

0.1 什么时候翻这份附录

问题	章节
`agent { …}` 里能用哪些 DSL 关键字?	DSL 关键字索引
一个 `tool { …}` 块怎么变成 LLM 的工具 schema?	工具块参考
有哪些记忆策略，开销分别是多少?	记忆策略
`exit_condition` 表达式里可以引用哪些上下文变量?	退出条件
`AgentOutput.finishReason` 实际上会是什么值?	Finish Reason 与异常
监听器回调按什么顺序触发?	AgentExecutionListener 速查
不用 DSL，怎么用 Java 构建智能体?	Java Builder API
我得满足哪些聊天模型 SPI?	LLM Provider 表面
这个模块进入 classpath 后 Spring 会自动装哪些 Bean?	Spring 装配
什么时候该用 agent，什么时候用 state machine 或 session?	选型表

这是 第 **17** 章——智能体编排 的参考层；那一章给的是概念图与走通流程。

0.2 模块表面

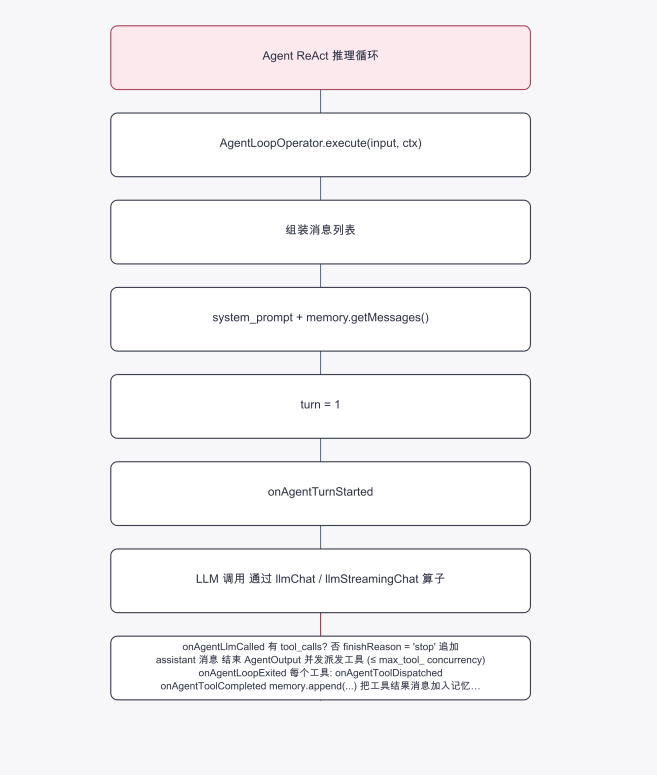
属性	取值
Maven 坐标	`com.leanowtech.bloge:bloge-agent-ext:${version}`
DSL 扩展 kind	`"agent"`
注册方式	`AgentExtensionProvider`，通过 `DslExtensionProvider` SPI 发现
硬依赖	`bloge-core`、`bloge-dsl`

属性	取值
运行依赖	`bloge-common-operators`（提供 `LlmProvider` / `llmChat` / `llmStreamingChat` / `TokenEstimator`）；可选 `JsonCodec` 以反序列化工具参数

当 JAR 进入 classpath，DSL 编译器会自动识别 `agent` 块语法，编译期无需手工 接线。运行时则要求 `Operator Registry` 已经注册了 LLM 聊天算子（同步：`"llmChat"`；流式：`"llmStreamingChat"`）。

0.3 ReAct 推理循环

下面这张图把” 循环 + 监听器回调点 + 流式 chunk 类型” 画在一张图里：



0.4 DSL 关键字索引

关键字	作用域	类型 / 取值	默认值	用途
`agent`	root、graph、phase、state 体内	块前缀	—	声明一个智能体扩展块。
`stream agent`	同上	块前缀	—	`agent` 的简写，等价于 `streaming = true`。
`model`	agent 体	`String`	必填	传给 `llmChat` 的模型标识。
`system_prompt`	agent 体	`String`	`null`	每次 LLM 调用都会前置的 system 消息。

关键字	作用域	类型 / 取值	默认值	用途
max_turns	agent 体	int	10	think–act–observe 的硬上限。超过抛 MaxTurnsExceededException。
max_tool_concurrency	agent 体	Integer	null（不限）	单轮工具并发派发上限。
temperature	agent 体	Double	null（用 provider 默认）	传给 llmChat 的采样温度。
memory	agent 体	策略表达式	full()	对话记忆策略——见下表。
exit_condition	agent 体	DSL 表达式	null（永不提前退出）	每轮工具后求值，真值则退出循环。
streaming	agent 体	boolean	false	物化成 StreamingAgentLoop Operator 而不是 AgentLoop Operator。
streaming_buffer_ capacity	agent 体	Integer	16	流式变体内部 LLM token 缓冲区大小。
tool <name> : <O peratorRef>	agent 体，可重复	块	—	声明一个工具，引用的算子就是其实现。
description	tool 体	String	""	传给 LLM 的工具描述。
input { …}	tool 体	赋值块	—	把工具参数映射到算子输入。
tool_args	input { …} 内	上下文变量	—	LLM 在本次工具调用中给出的参数对象。

0.4.1 记忆策略

memory = …接受以下几种写法（都对应 AgentMemoryStrategy）：

DSL 写法	行为	备注
full()	保留出现过的所有消息。	默认；CPU 最便宜，token 最贵。
sliding_window(n)	保留所有 system 消息加上最近 n 条非 system 消息。	n 必须 ≥ 1。
token_budget(n)	从最老的非 system 消息开始裁剪，直到 TokenEstimator 报告总 token 数 ≤ n。	需要 TokenEstimator（已作为 SPI 注册；模块也带一个启发式实现）。
summary(n)	完整保留最近 n 条非 system 消息，更早的用一段 LLM 生成的摘要替代。	摘要本身要再发一次 LLM 调用，记得评估代价。
AgentMemoryStrategy. Custom(factory)	仅 Java：传自己的 ConversationMemory. Factory。	DSL 不可表达。

0.4.2 退出条件

exit_condition 是一个 DSL 表达式，每轮工具结束后基于以下上下文求值：

变量	类型	含义
finish_reason	String	最近一次 LLM 响应的 finish reason。
content	String	最近一次 LLM 响应的 assistant 内容字符串。
turn	int	刚结束这一轮的 1 起始编号。

变量	类型	含义
`tool_calls`	`List<Map>`	本轮 LLM 产生的工具调用，每项包含 `id`、`name`、`arguments`。
`tool_results`	`Map<String,Object>`	当前工具批次的结果，按工具名为 key。
`messages`	`List<LlmMessage>`	完整对话状态（含刚刚追加的工具消息）。

求值结果为真时，listener 的 `LoopExitedPayload.exitReason` 为 `"exit_condition"`，循环不再调用模型；正常返回的 `AgentOutput.finishReason` 仍保留最近一次 provider reason。

0.5 工具块参考

一个 tool 块会编译成一份 `AgentToolRef`。从中流出的有三样东西：

1. 面向 LLM 的 **`ToolDefinition`** ——`name`（DSL 标签）、`description`（你写的 `description = …`）、由 `input { …}` 推导出的 JSON-schema `parameters` 对象。
2. 一个子图，把被引用的算子包起来，让 agent 循环可以像普通节点一样 调用它。
3. **`resultNodeId`**，循环用它从子图里取出该工具的输出。

0.5.1 工具 schema 推导

编译器会遍历 `input { …}` 中的每一条赋值。只有 形如 `tool_args.<field>` 的路径会被推到 JSON-schema 里，作为必填参数；其它 赋值（常量、引用外层 ctx 等）不出现在 schema 中，会在派发时填入。

```
tool searchKnowledgeBase : KBSearchOperator {
  description = "Search the knowledge base"
  input {
    query  = tool_args.query
    locale = ctx.user.locale     // 不出现在 LLM 看到的 schema 里
  }
}
```

LLM 看到的是一个名为 `searchKnowledgeBase` 的工具，要求一个必填字符串参数 `query`；`locale` 是从外层图上下文填进去的。如果你的工具参数包含嵌套对象，要给 DSL 编译器配置一个 `JsonCodec`，让 arguments JSON 可以反序列化成结构 化值。

0.6 一个完整示例

最短的端到端智能体——DSL 加运行所需的 Java 引导代码。

```
agent customerSupport {
  model = "gpt-4o"
  system_prompt = "You are a helpful support agent."
  max_turns = 15
  temperature = 0.7
  memory = sliding_window(20)
```

```
  tool searchKnowledgeBase : KBSearchOperator {
    description = "Search the knowledge base"
    input { query = tool_args.query }
  }

  tool escalateToHuman : EscalateOperator {
    description = "Escalate to a human agent"
    input { reason = tool_args.reason }
  }

  exit_condition = finish_reason == "stop" || tool_call("escalateToHuman")
}
```

```
OperatorRegistry registry = OperatorRegistry.builder()
    .register("KBSearchOperator", new KBSearchOperator())
    .register("EscalateOperator", new EscalateOperator())
    .register("llmChat", new LlmChatOperator(myLlmProvider))
    .build();

AgentDef def = new AgentDslCompiler(registry).compile(dslSource);
AgentLoopOperator agent = new AgentLoopOperator("supportAgent", def, registry);

AgentOutput result = agent.execute("My account is locked", OperatorContext.root());
System.out.println(result.content());
System.out.println(result.finishReason());   // provider reason，例如 "stop" / "tool_calls"
```

流式变体在 DSL 里写 `streaming = true`，或在 `AgentBuilder` 上调 `toStreamingOperator(...)`；运行时通过 `NodeChannel` 推出 `AgentStreamChunk`。

0.7 Java Builder API

`AgentBuilder` 与 DSL 关键字一一对应：

方法	用途
`AgentBuilder.create(id)`	开一个全新的 builder。
`AgentBuilder.from(def)`	从已有的 `AgentDef` 起手。
`.model(String)`	设置 LLM 模型 id。
`.systemPrompt(String)`	设置 system prompt。
`.maxTurns(int)`	设置轮次上限。
`.maxToolConcurrency(int)`	设置单轮工具并发上限。
`.temperature(Double)`	设置采样温度。
`.memory(AgentMemoryStrategy)`	选择记忆策略。
`.tool(AgentToolRef)`	追加一个工具。

方法	用途
`.exitCondition(Expression | String)`	编译并附加退出条件表达式。
`.streaming(boolean)`	切换到流式变体。
`.streamingBufferCapacity(int)`	设置流式 buffer 大小。
`.build()`	产出不可变的 `AgentDef`。
`.toOperator(ref [, registry])`	物化成 `AgentLoopOperator`。
`.toStreamingOperator(ref [, registry])`	物化成 `StreamingAgentLoopOperator`。

0.8 记录结构

所有 record 不可变；字段按声明顺序列出。

0.8.1 AgentDef

字段	类型	备注
`id`	`String`	DSL 中的标识符。
`model`	`String`	LLM 模型 id；必填。
`systemPrompt`	`String`	可以为 `null`。
`maxTurns`	`int`	默认 `10`。
`maxToolConcurrency`	`Integer`	`null` => 不限并发。
`temperature`	`Double`	`null` => 使用 provider 默认。
`memoryStrategy`	`AgentMemoryStrategy`	默认 `Full`。
`tools`	`List<AgentToolRef>`	保持 DSL 顺序。
`exitCondition`	`Expression`	可以为 `null`。
`streaming`	`boolean`	`true` => 使用流式算子。
`streamingBufferCapacity`	`Integer`	流式 buffer 大小，默认 `16`。

0.8.2 AgentInput

字段	类型	备注
`messages`	`List<LlmMessage>`	已规范化的形式；`AgentInput.from(Object)` 会把字符串、单条消息、列表都转过来。

0.8.3 AgentOutput

字段	类型	备注
`content`	`String`	最终 assistant 内容。

字段	类型	备注
finishReason	String	见 Finish Reason 与异常。
turnsUsed	int	循环实际跑了多少轮。
messages	List<LlmMessage>	完整对话，包括 system / assistant / tool 消息。
toolResults	Map<String,Object>	最近一轮工具结果，按 tool-call id 为 key。
checkpoint	AgentCheckpoint	用于恢复的快照；不会为 null。

0.8.4 AgentCheckpoint

字段	类型	备注
conversation	List<LlmMessage>	截至快照点的消息。
turnCount	int	快照时的轮次。
toolResults	Map<String,Object>	快照时的工具结果。

把它与你自己的 session / state-machine checkpoint 一起持久化，就可以在 重启后恢复一次智能体运行。

0.8.5 AgentToolRef

字段	类型	备注
name	String	暴露给 LLM 的工具名。
operatorRef	String	OperatorRegistry 中注册的算子名。
description	String	LLM 看到的同一份描述。
graph	Graph	包装该算子的编译期子图。
resultNodeId	String	输出作为工具结果的节点。
toolDefinition	LlmProvider.ToolDefinition	LLM 侧 schema（name + description + parameters）。

0.8.6 AgentStreamChunk（sealed）

变体	字段	触发时机
Token	text: String、turnNumber: int	每个 LLM token（或一组）。
ToolStart	toolName: String、toolCallId: String、turnNumber: int	工具派发之前。
ToolEnd	toolName: String、toolCallId: String、success: boolean、turnNumber: int	工具完成之后。
Done	finalOutput: AgentOutput	流的最后一个 chunk，循环已退出。

0.9 Finish Reason 与异常

表面	取值与出现时机
`AgentOutput.finishReason`	最近一次 provider reason，例如 `"stop"`、`"length"` 或 `"tool_calls"`；原样保留。
`LoopExitedPayload.exitReason`	`"stop"`、`"exit_condition"` 或 `"max_turns"`，说明 loop 为什么结束。
`MaxTurnsExceededException`	达到 `max_turns` 时直接抛出，携带 checkpoint；不会返回 finishReason 为 `"max_turns"` 的正常 output。

`MaxTurnsExceededException` 是从 **`AgentLoopOperator.execute`** 抛出来的；调用方代码负责捕获它并向上层报出有意义的错误。

0.10 `AgentExecutionListener` 速查

回调	载荷 **record**	字段
`onAgentTurnStarted`	`TurnStartedPayload`	`turnNumber`、`messageCount`
`onAgentLlmCalled`	`LlmCalledPayload`	`turnNumber`、`model`、`promptTokens`、`completionTokens`、`latencyMs`、`finishReason`
`onAgentToolDispatched`	`ToolDispatchedPayload`	`turnNumber`、`toolCallId`、`toolName`、`argumentsJson`
`onAgentToolCompleted`	`ToolCompletedPayload`	`turnNumber`、`toolCallId`、`toolName`、`success`、`latencyMs`、`resultSummary`
`onAgentLoopExited`	`LoopExitedPayload`	`totalTurns`、`totalLlmCalls`、`totalLlmTokens`、`totalLatencyMs`、`exitReason`

每个回调都会拿到一个 `AgentCallbackContext`，里面有 `executionId`、`graphName`、`nodeId`、`operatorRef`。所有回调都有默认空实现，只覆写你关心 的那几个就行。

典型一轮的顺序是：`onAgentTurnStarted` → `onAgentLlmCalled` →（每次工具调用）`onAgentToolDispatched` →（每次工具调用）`onAgentToolCompleted` → 进入下一轮 或者 `onAgentLoopExited`。

0.11 `ConversationMemory` SPI

`ConversationMemory` 是 sealed 接口（实现有 `FullConversationMemory`、`SlidingWindowConversationMemory`、`TokenBudgetConversationMemory`、`SummaryConversationMemory`）。用静态工厂方法构造实例：

工厂	备注
`ConversationMemory.full(initialMessages)`	不裁剪。
`ConversationMemory.slidingWindow(initialMessages, windowSize)`	保留所有 system 消息加上最近 `windowSize` 条非 system 消息。

工厂	备注
`ConversationMemory.tokenBudget(initialMessages, maxTokens, tokenEstimator)`	从最老的非 system 消息开始裁剪直到 token 总数符合预算。
`ConversationMemory.summary(initialMessages, summarizer, summaryThreshold)`	当非 system 消息条数超过 `summaryThreshold` 时调用 `summarizer`。

两个函数式 SPI 让你接入自定义行为：

- `ConversationMemory.Factory` ——`create(List<LlmMessage> initial)`，可以返回任何 `ConversationMemory` 实例，是 `AgentMemoryStrategy.Custom(factory)` 的钩子。
- `ConversationMemory.Summarizer` ——`summarize(List<LlmMessage>)`，是 `Summary` 策略所调用的函数。bloge 运行时不内置默认 summarizer；自己接一个 即可，通常就是再发一次 `llmChat`，prompt 写” 请总结对话”。

0.12 LLM Provider 表面（bloge-common-operators）

agent 循环从来不直接和 OpenAI / Anthropic / Azure 通信。它只调用 `OperatorRegistry` 里按标准名注册的算子。这套契约住在 `bloge-common-operators` 模块：

类型	FQN	用途
`LlmProvider`	`com.leanowtech.bloge.operators.spi.LlmProvider`	底层聊天模型客户端的 SPI。
`LlmProvider.LlmMessage`	嵌套 record	`role`、`content`、`parts`、`toolCalls`、`toolCallId`。
`LlmProvider.ToolDefinition`	嵌套 record	`name`、`description`、`parameters`（JSON-schema map）。
`LlmChatOperator`	`com.leanowtech.bloge.operators.ai.LlmChatOperator`	同步聊天算子；默认注册名 `"llmChat"`。
流式聊天算子	同模块	注册名为 `"llmStreamingChat"`，被流式 agent 使用。
`TokenEstimator`	`com.leanowtech.bloge.operators.spi.TokenEstimator`	估算消息 token 数，给 `token_budget` 记忆用。模块自带启发式默认实现。

新写一个 provider 就实现 `LlmProvider`，注册算子时传给 `LlmChatOperator`。agent 循环对此完全透明。

0.13 Spring 装配

`bloge-agent-ext` 自己不带任何 Spring 自动配置。在 `bloge-spring` 里有两个 条件钩子覆盖了 Spring 使用者需要的一切：

Bean	定义位置	触发条件
`AgentEventJournalBridge`	`BlogeEventJournalAutoConfiguration`	当 `bloge-agent-ext`、`bloge-event-journal` 同时在 classpath，且 `spring.bloge.event-journal.enabled=true` 时注册；把所有 `AgentExecutionListener` 回调注入引擎的事件日志。
可选 listener 装配	`OptionalAgentListenerSupport`	软加载 `AgentExecutionListener`，没有 agent-ext JAR 时 `bloge-spring` 其它部分仍能工作。容器里任何 `AgentExecutionListener` Bean 都会被自动拾取。

只有一条属性：

```
spring.bloge:
  event-journal:
    enabled: true
```

除此之外，把你的 provider、算子、监听器作为普通 Spring Bean 暴露即可——都会沿第 20 章描述的 `OperatorRegistryAutoConfiguration` 路径被拾取。

0.14 选型表

agent 循环和其它 bloge 概念在工程里怎么分工？

问题	选什么
是否有一条按固定顺序跑的算子流水线？	普通 bloge 图，不用 agent。
是否需要 LLM 来决定调哪些算子、按什么顺序、参数由它合成？	**`agent { …}`**。
是否要按 phase / 守卫驱动一个确定性工作流？	状态机（附录 D）。
是否要崩溃安全的多轮用户会话（不是 LLM 自己选工具）？	**Session**（第 14 章）。
是否要把 token 流向前端 UI？	流式变体——`streaming = true` 或 `stream agent`；流式整体模型见附录 C。
是否要细粒度追踪 LLM 调用、工具派发与耗时？	`AgentExecutionListener` + `AgentEventJournalBridge`。

agent 与状态机可以叠在一起：agent 可以当某一个 state 的 operator，状态机 也可以把若干” 专家 agent” 串起来。

0.15 常见错误

- 忘写 **`model = "…"`**。编译会失败。model id 必填，会原样传给聊天算子。
- 没注册 **`llmChat`** 算子。 DSL 编译通过，但第一轮就报”unknown operator”。搭 registry 时记得接 `LlmChatOperator`（或流式 `llmStreamingChat`）。

- 在 **`input { …}`** 之外用 **`tool_args.x`**。`tool_args` 只在某个 tool 的 `input` 块里有效；在 `exit_condition` 或别处用都是编译错。
- 长对话上用了 **`full()`**。每轮都把整段历史发回去——上生产前换成 `sliding_window(n)` 或 `token_budget(n)`。
- **`summary()`** 没接 **summarizer**。默认 `Summary` 策略需要外部摘要回调（通常是一次 LLM 调用）；没接的话会退化成" 只保留阈值窗口"。
- 永远为假的 **`exit_condition`**。如果引用了根本没人产生的 tool 结果 key，循环每次都会跑到 `max_turns`。跑测试时要同时验证 happy path 与封顶路径。
- 嵌套对象工具参数 **schema** 不匹配。工具参数是结构化值时，给 DSL 编译器 传一个 `JsonCodec`，这样 `tool_args.foo.bar` 才能解析到反序列化后的对象；否则只能干净地走顶层字符串字段。
- **`MaxTurnsExceededException`** 在边界没被捕获。异常会从 `execute()` 往上 传；调用 agent 的算子代码必须 catch 它并报出有意义的失败，否则外层图会 把节点标成 crashed。

0.16 与正文章节的关系

- 第 **7** 章——设计良好的算子。算子元数据字段（`promptHint`、`usageExample`、`constraintsDescription`）正是 agent 循环用来填充工具 `description` 与 JSON-schema 提示的来源。
- 第 **11** 章——批处理与迭代 与 附录 **C** ——流式。流式变体通过附录 C 说明的同一套 `NodeChannel` 推出 `AgentStreamChunk`。
- 第 **19** 章——生产可观测性。 `AgentEventJournalBridge` 把每一个回调 接入第 19 章描述的事件日志。
- 第 **9** 章——工具与工作流。 `operator-metadata.json` 导出器输出的结构 与 agent 循环发给模型的 `ToolDefinition` 同形。
- 第 **20** 章——**Spring** 与生产装配。上面说的条件 Bean 注册；`OperatorRegistryAutoConfiguration` 是暴露算子与监听器的标准路径。
- 第 **17** 章——智能体编排。本附录就是它的参考层。

章节

—

附录 F ——决策表

本附录是 decision_table 节点类型的权威速查手册。此处列出的每条语法规则、命中策略保证、输出形状和错误码均直接来源于 bloge 源码与合规测试装置。

当你需要查询精确语义时，请直接使用本附录，无需重读第 5 章。若想从概念引导出发，请从 第 5 章——做决策的分支开始。若需动手练习，请参见 第 34 章 Lab 6。

0.1 语法速查

```
decision_table <id>(
  <param1> = <expr1>,
  <param2> = <expr2>,
  ...
) [hit=<first|unique|any|collect>] [-> <OutputType>] {
  rule (<param>: <predicate>, ...) -> <output>
  ...
  [otherwise -> <output>]
}
```

- hit= 默认为 first。
- -> <OutputType> 可省略；省略时编译器从第一条规则的输出推断类型。
- 每条 rule 在 (...) 中声明要测试的参数。
- otherwise 仅在所有显式 rule 都不匹配时才执行（或对 collect 策略而言，总是追加为一个条目——见 §otherwise 语义）。

最小示例——**hit=first** + 标量输出：

```
graph creditScreening {
  decision_table credit_tier(score = applicant.output.score) hit=first -> String {
    rule (score: score >= 750) -> "platinum"
    rule (score: 680 <= score < 750) -> "gold"
    otherwise -> "rejected"
  }
}
```

0.2 命中策略矩阵

策略	多条规则命中	无匹配且无 `otherwise`	抛出异常
`first`	取声明顺序的第一个匹配	`RUNTIME_DECISION_TABLE_NO_MATCH`	`CODE_NO_MATCH`
`unique`	抛出 `RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH`	`RUNTIME_DECISION_TABLE_NO_MATCH`	`CODE_AMBIGUOUS_MATCH` / `CODE_NO_MATCH`
`any`	若所有匹配输出相等（`Objects.equals`）则允许；否则抛出 `RUNTIME_DECISION_TABLE_CONFLICTING_MATCH`	`RUNTIME_DECISION_TABLE_NO_MATCH`	`CODE_CONFLICTING_MATCH` / `CODE_NO_MATCH`
`collect`	收集所有匹配到 `output.items`	返回 `{ items: [] }`，不抛异常	—

`first` 示例——第一条命中的规则获胜：

```
graph decisionBasicFirst {
  decision_table credit_tier(score = applicant.output.score) hit=first -> String {
    rule (score: score >= 750) -> "platinum"
    rule (score: 680 <= score < 750) -> "gold"
    otherwise -> "rejected"
  }
}
```

`unique` 示例——至多一条规则命中，否则报错：

```
graph decisionBasicUnique {
  decision_table credit_band(score = applicant.output.score) hit=unique -> String {
    rule (score: score >= 750) -> "prime"
    rule (score: 650 <= score < 750) -> "standard"
    otherwise -> "manual"
  }
}
```

`any` 示例——多条规则可以同时命中，前提是输出值相同：

```
graph decisionBasicAny {
  decision_table approval(score = ctx.score) hit=any -> String {
    rule (score: score >= 700) -> "approved"
    rule (score: score >= 750) -> "approved"
    otherwise -> "manual"
  }
}
```

collect 示例——所有命中规则都贡献到 output.items：

```
graph decisionBasicCollect {
  decision_table applicable_discounts(score = applicant.output.score) hit=collect -> String {
    rule (score: score >= 700) -> "loyalty"
    rule (score: score >= 760) -> "premium"
  }
}
```

0.3 otherwise 语义

策略	存在 **otherwise**	不存在 **otherwise**
first	无显式规则命中时触发	无匹配 → RUNTIME_DECISION_TABLE_NO_MATCH
unique	无显式规则命中时触发	无匹配 → RUNTIME_DECISION_TABLE_NO_MATCH
any	无显式规则命中时触发	无匹配 → RUNTIME_DECISION_TABLE_NO_MATCH
collect	无论显式规则是否命中，总是追加一个条目	空匹配 → 返回 { items: [] }，不抛异常

> ***collect*** 的特殊之处：对于 *first*、*unique*、*any*、*otherwise* 是”兜底”——只在其他规则都不匹配时才触发。对于 *collect*，它是一条无条件规则，总是把自己的输出追加到 *items* 中。这一差异由 *decision-table/collect-otherwise lint* 规则（*INFO* 严重度）标记，提示你确认这是否是有意为之。

0.4 输出形状参考

共三种形状。同一张表中的所有 rule（包括 otherwise）必须使用相同的形状——混用形状会导致编译错误（COMPILER_DECISION_TABLE_OUTPUT_SHAPE_MISMATCH）。

形状	声明语法	下游访问方式
标量（scalar）	-> "value" / -> 3.5	<node>.output.value
命名字段（named fields）	-> { rate: 3.5, maxTerm: 30 }	<node>.output.rate、<node>.output.maxTerm
收集条目（collect items）	hit=collect + 任意输出	<node>.output.items（List 类型）

多输出（命名字段）示例：

```
graph decisionMultiOutput {
  decision_table loan_terms(
    score = applicant.output.score,
    amount = loan.output.amount
  ) hit=unique -> { rate: Decimal, maxTerm: Int } {
```

```
    rule (score: score >= 750, amount: amount <= 500000) -> { rate: 3.5, maxTerm: 30 }
    otherwise -> { rate: 6.0, maxTerm: 5 }
  }
}
```

0.5 in 运算符

in 在规则谓词中测试集合成员关系，共两种形式。

0.5.1 形式 A ——字面量数组

```
rule (type: type in ["vip", "enterprise", "gov"]) -> "priority"
```

- 右侧是字面量数组：[lit1, lit2, ...]。
- 每个元素必须是 **String**、**Number** 或 **Boolean** 字面量——不允许表达式、变量或节点路径。
- 违反此规则会触发编译错误：COMPILER_DECISION_TABLE_IN_LITERAL_ONLY。
- 空数组（type in []）永远不匹配任何输入。

示例：

```
graph decisionTableInStatic {
  decision_table customer_tier(type = ctx.type) hit=first -> String {
    rule (type: type in ["vip", "enterprise", "gov"]) -> "priority"
    otherwise -> "standard"
  }
}
```

0.5.2 形式 B ——动态参数引用

```
rule (type: type in allowed) -> "priority"
```

- 右侧是单个已声明参数名，在运行时必须解析为 Collection 类型。
- 不允许深路径如 ctx.allowed、obj.list、a[0]——请通过 decision_table(...) 头部的参数绑定把集合传入。
- 若参数在运行时解析为非 Collection 类型（如 String 或 Integer），bloge 抛出带有 RUNTIME_DECISION_TABLE_INVALID_COLLECTION_PARAM 代码的 DecisionTableViolationException。异常消息包含 in 表达式的行列号和实际运行时类型。
- 左值为 **null** 时直接短路返回 false（不抛异常）。

示例：

```
graph decisionTableInDynamic {
  decision_table customer_tier(type = ctx.type, allowed = ctx.allowed) hit=first -> String {
    rule (type: type in allowed) -> "priority"
    otherwise -> "standard"
```

```
  }
}
```

0.5.3 等价语义——两种形式一致

形式 A 和形式 B 内部均使用数值等价比较（`evaluateBinaryOp(EQ_EQ)`）。这意味着 `Integer(700)` 与 `BigDecimal(700.0)` 被视为相等。你不会因为不同图节点产生的数值类型不同而遭遇无声的不匹配。

0.6 常见陷阱

0.6.1 1. 作用域越界——在规则中引用非参数路径

```
// [X] 编译错误：COMPILER_DECISION_TABLE_RULE_SCOPE_VIOLATION
rule (score: applicant.output.score >= 750) -> "prime"

// [OK] 先通过参数绑定传入值
decision_table credit_band(score = applicant.output.score) hit=unique -> String {
  rule (score: score >= 750) -> "prime"
  ...
}
```

0.6.2 2. 输出形状不一致

```
// [X] 编译错误：COMPILER_DECISION_TABLE_OUTPUT_SHAPE_MISMATCH
decision_table tier(score = ctx.score) hit=first {
  rule (score: score >= 750) -> "prime"           // 标量
  otherwise -> { label: "manual", flag: true }    // 命名字段——形状不一致！
}
```

0.6.3 3. `in` 右侧不是简单参数名

```
// [X] 编译错误：COMPILER_DECISION_TABLE_IN_INVALID_RIGHT
rule (type: type in ctx.allowed) -> "priority"   // 深路径不允许

// [OK] 在参数列表中绑定
decision_table tier(type = ctx.type, allowed = ctx.allowed) hit=first -> String {
  rule (type: type in allowed) -> "priority"
  ...
}
```

0.6.4 4. `in` 右侧参数在运行时不是 `Collection`

若绑定给 `in` 右侧的参数在运行时解析为非 `Collection` 值（如 `String` 或 `Integer`），bloge 抛出：

`RUNTIME_DECISION_TABLE_INVALID_COLLECTION_PARAM`

异常消息会指出具体行列号和实际运行时类型。请修复产生错误类型的上游节点或绑定表达式。

0.6.5 5. collect + otherwise ——总是追加，而不只是兜底

```
// [INFO] decision-table/collect-otherwise
// 这里的 otherwise 子句会在每次运行时都将 "baseline" 追加到 items，
// 即使其他规则也命中了。请确认这是否是有意为之。
decision_table discounts(score = ctx.score) hit=collect -> String {
  rule (score: score >= 700) -> "loyalty"
  otherwise -> "baseline"   // 总是被追加
}
```

0.7 运行时错误码

所有运行时违规都以 `DecisionTableViolationException` 抛出。`code` 字段携带以下冻结的 wire 字符串之一：

Wire 字符串	触发时机
`RUNTIME_DECISION_TABLE_NO_MATCH`	`first`、`unique` 或 `any`——没有规则命中且未声明 `otherwise`
`RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH`	`unique`——超过一条规则命中
`RUNTIME_DECISION_TABLE_CONFLICTING_MATCH`	`any`——多条规则命中但输出值不相等
`RUNTIME_DECISION_TABLE_INVALID_COLLECTION_PARAM`	`in <param>`——参数未解析为 `Collection`

> 这些字符串是冻结契约。在应用程序错误处理器或测试断言中匹配时，请原文复制。

0.8 编译期错误码

bloge 编译器在编译期拒绝无效的决策表定义。主要错误标识符（用于诊断信息和 CI lint 输出）：

标识符	触发原因
`COMPILER_DECISION_TABLE_IN_LITERAL_ONLY`	`in [...]` 数组包含非字面量元素
`COMPILER_DECISION_TABLE_IN_INVALID_RIGHT`	`in <expr>` 右侧不是单个参数名
`COMPILER_DECISION_TABLE_RULE_SCOPE_VIOLATION`	规则谓词引用了已声明参数之外的内容
`COMPILER_DECISION_TABLE_OUTPUT_SHAPE_MISMATCH`	规则使用了不一致的输出形状（标量 vs. 命名字段）
`COMPILER_DECISION_TABLE_OUTPUT_TYPE_INVALID`	命名输出类型注解中的字段不符合必需的 `name: Type` 形式
`COMPILER_DECISION_TABLE_UNKNOWN_PARAMETER`	规则条件引用了表头中未声明的参数
`COMPILER_DECISION_TABLE_DUPLICATE_CONDITION`	两条规则的条件完全相同
`COMPILER_DECISION_TABLE_PARAM_RESERVED`	参数名与 DSL 关键字冲突
`COMPILER_DECISION_TABLE_PARAM_DUPLICATE`	表头中存在重复参数名
`COMPILER_DECISION_TABLE_PARAM_NODE_CONFLICT`	参数名与图节点名冲突

0.9 最小可运行示例

0.9.1 示例 1 ——hit=first + otherwise

```
graph ex1First {
  decision_table credit_tier(score = applicant.output.score) hit=first -> String {
    rule (score: score >= 750) -> "platinum"
    rule (score: 680 <= score < 750) -> "gold"
    otherwise -> "rejected"
  }
}
```

预期：score=720 → credit_tier.output.value == "gold"。

0.9.2 示例 2 ——hit=unique ——歧义匹配错误路径

```
graph ex2Unique {
  decision_table risk_band(score = ctx.score) hit=unique -> String {
    rule (score: score >= 650) -> "eligible"
    rule (score: score >= 700) -> "preferred"
    // 无 otherwise —— 若 score >= 700 则两条规则都命中 → RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH
  }
}
```

预期：score=720 → 抛出 DecisionTableViolationException，code 为 RUNTIME_DECISION_TABLE_AMBIGUOUS_MATCH。

0.9.3 示例 3 ——hit=collect + 命名字段输出

```
graph ex3Collect {
  decision_table loan_badges(
    score = applicant.output.score,
    amount = loan.output.amount
  ) hit=collect -> { label: String, discount: Decimal } {
    rule (score: score >= 700) -> { label: "loyalty", discount: 0.02 }
    rule (score: score >= 760) -> { label: "premium", discount: 0.05 }
    rule (amount: amount <= 100000) -> { label: "small-loan", discount: 0.01 }
  }
}
```

预期：score=780, amount=80000 → loan_badges.output.items 包含三个条目：loyalty、premium、small-loan。

0.10 另请参阅

- 第 5 章——做决策的分支 ——概念引导、分支 vs. decision_table 选型指南

- 第 9 章——工具链工作流——`decision-table/missing-otherwise` 和 `decision-table/collect-otherwise` 的 lint 实战
- 第 34 章——Lab，Lab 6——覆盖 `unique`、`collect` 与异常路径的动手练习
- 第 17 章——把非确定性 Agent 放进确定性生命周期——将智能体信号与决策表组合，实现可审计的策略执行

附录 G ——DSL 工程参考

章节

—

附录 G ——DSL 工程参考

第 8 章只讲“个人草稿如何成为团队资产”。本附录保存工程查阅细节，不要求第一次阅读时记住。

0.1 import 与 alias

推荐显式写出 alias：

```
import "./payment-flow" as paymentFlow
```

没有显式 alias 时，编译器可从路径最后一段推导名称；团队文件仍应显式声明，避免文件重命名同时改变引用名。`FileSystemGraphResolver` 以当前文件目录解析相对路径；找不到文件、循环导入和重复 alias 都属于建图前错误。

0.2 schema 选择速查

需要表达的事实	形式
必填标量	`field: String`
可缺失标量	`field: String?`
同文件复用结构	命名 `schema`
只出现一次的局部结构	内联 `{ ... }`
同类型集合	`List<T>`
键值结构	`Map<K,V>`

RC1 的路径验证对内联嵌套结构能给出逐段字段诊断。本书 ch08 probe 因此使用内联 `customer`；不要把这一观察外推为未来版本的永久语义。

0.3 `ValidatedSchema` 不是 DSL 语法

Java runtime 可用 `ValidatedSchema` 把 expected schema 与 `STRICT`、`COERCE` 或 `LENIENT` 策略组合。DSL 编译器做结构检查时读取 expected schema；运行时策略决定值如何被验证或转换。两者不能混写成一个结论：编译期路径正确，不等于运行期值一定满足策略。

0.4 resolver 扩展边界

自定义 `GraphResolver` 适合从受控仓库、classpath 或其他版本化来源读取 graph。它必须给出确定身份、明确失败和循环保护。远程读取、缓存失效和签名验证属于宿主系统职责，不由 import 语法自动解决。

0.5 查阅入口

- `DslCompiler.java`
- `FileSystemGraphResolver.java`
- `ValidatedSchema.java`
- `DslFeedbackLoopProbeTest.java`

附录 H ——Durable 与规模化运行参考

章节

—

附录 H ——Durable 与规模化运行参考

第 *21*、*22* 章保留机制与判断；本附录保存部署核对项和测量参数。

0.1 租约与恢复配置

记录 owner identity、claim timeout、scan interval、batch size、maximum recovery attempts 和逻辑/墙上时钟来源。timeout 越短不等于恢复越好：长暂停或存储抖动会增加误接管，必须配合 fencing epoch。

0.2 路由与迁移

新业务键由 `ShardResolver` 选择，既有键从 `RoutingStore` 读取。迁移顺序固定为：停止写入、复制完整状态、校验、原子更新 binding、恢复流量。不要用扩容后的新 hash 代替迁移。

0.3 远端派发参数

参数	必须同时记录的事实
topic / queue	worker 能力与版本
heartbeat	误判窗口与重派策略
batch size	全局 in-flight 上限
retry	effect 幂等与重复投递语义
result timeout	取消、迟到结果和 DLQ 处理

0.4 Store migration 核对

迁移版本以所用 `bloge-durable-mybatis` 包内 Flyway 脚本为准，不把旧章的 V25 名称当永久契约。每个环境记录 schema history、表/索引存在性、应用 artifact checksum、迁移时间和回退决策。

0.5 Benchmark 记录模板

```
commit / artifact:
JDK / CPU quota / heap:
topology / operator behavior:
```

```
arrival model / concurrency:
warmup / measurement / forks:
external stubs and store:
raw result path:
single changed axis:
stop rule:
claim and non-claim:
```

benchmarks/run.sh 与 compare.py 提供运行和比较入口；阈值必须来自当前脚本或团队基线，不能从书中复制后假定仍适用。

0.6 精确入口

- benchmarks/run.sh
- benchmarks/compare.py
- RecoveryLoopConfig.java
- RuntimeRoutingShardingTest.java

章节

—

附录 I ——Spring 生产参考

请在第 *20* 章已经跑通后使用本附录。这里是生产装配的查询面，不是第二条教程主线。

如果组件所有权仍不清楚，请回到第 20 章。

0.1 拆开来看

0.1.1 starter 自动配置了什么

自动配置通过 Spring Boot 3.x 的 `AutoConfiguration.imports` 文件注册，其中列出了十个顶层自动配置入口。下表还展开了 `BlogeAutoConfiguration` 内部导入的 `BlogeCoreAutoConfiguration`：

配置类	提供的内容	激活条件
`BlogeAutoConfiguration`	入口点—导入 `BlogeCoreAutoConfiguration`	始终激活
`BlogeCoreAutoConfiguration`	`OperatorRegistry`、`GraphLoader`、`List<Graph>`、租户解析器、租户过滤器	始终激活
`BlogeDurableMigrationConfiguration`	显式的加密 checkpoint 迁移工具	`bloge.decrypt-checkpoints=true` 且 classpath 上存在迁移类
`BlogeGraphEngineAutoConfiguration`	通过 builder 创建 `GraphEngine`，包含所有 `ObjectProvider` 协作者	未启用 checkpoint 解密迁移模式时激活
`BlogeDurableAutoConfiguration`	所有持久化存储、`RecoveryConfig`、`RoutingStore`、Flyway 迁移	classpath 上有 `bloge-durable` 且存在 `DataSource` bean
`BlogeAuditAutoConfiguration`	`AuditJournalListener`	`spring.bloge.audit.enabled=true` 且存在 `AuditJournalStore` bean
`SessionExecutorAutoConfiguration`	纯内存 `SessionExecutor`；可选的 durable session adapter 与 manager	classpath 上有 `bloge-session-ext`；durable bean 还要求对应类、store 与 definition lookup
`BlogeStateMachineAutoConfiguration`	durable state-machine manager 与恢复接线	classpath 上有 durable state-machine 类且 `spring.bloge.state-machine.enabled=true`
`BlogeEndpointAutoConfiguration`	`/actuator/bloge`、`/actuator/blogeStats`	classpath 上有 `spring-boot-actuator`

配置类	提供的内容	激活条件
`BlogeObservabilityAutoConfiguration`	指标、链路追踪、MDC、日志、健康指示器	classpath 上有 `bloge-metrics-otel`
`BlogeEventJournalAutoConfiguration`	事件日志保留和生命周期	classpath 上有 `bloge-event-journal`

如果你要接入多轮 session 流程，可以配合阅读第 14 章——多轮 Session。那一章会把 `SessionExecutorAutoConfiguration` 背后真正接起来的运行时模型讲清楚：session graph、phase / round 执行、空闲超时，以及可选的持久恢复。

0.1.2 `@BlogeOperator` 注解

定义在 `BlogeOperator.java` 中：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Component
public @interface BlogeOperator {
    @AliasFor(annotation = Component.class, attribute = "value")
    String value() default "";
    String description() default "";
    String owner() default "";
    String[] tags() default {};
    String promptHint() default "";
    String usageExample() default "";
    String constraintsDescription() default "";
}
```

它底层就是 `@Component`，所以 Spring 能发现它。`BlogeAutoConfiguration.operatorRegistry()` bean 方法会扫描所有标注了 `@BlogeOperator` 的 bean，并以注解的 `value()` 为键将每一个注册到 `DefaultOperatorRegistry` 中。

> 面向 ***agent/LLM*** 集成时，三个额外属性向 *AI agent* 描述该 *operator*：`promptHint` 告诉 *agent* 何时选择此 *operator*；`usageExample` 给出展示该 *operator* 用法的 *DSL* 片段；`constraintsDescription` 记录了从 *schema* 中不易察觉 的约束。这三个属性均为可选。

0.1.3 `GraphEngine` 是如何组装的

`BlogeGraphEngineAutoConfiguration` 负责创建引擎 bean，接受 `tenantContextResolver` 和 `tenantResourcePolicy` 用于多租户部署，并提供 `GraphEngineCustomizer` 回调钩子，让你无需替换整个引擎 bean 即可调整它。

`BlogeGraphEngineAutoConfiguration` 中的 `graphEngine()` bean 方法对每个可选协作者都使用了 `ObjectProvider`：

```
@Bean
@ConditionalOnMissingBean
public GraphEngine graphEngine(
        OperatorRegistry registry,
        ObjectProvider<OperatorInterceptor> interceptors,
        ObjectProvider<ExecutionListener> listeners,
        ObjectProvider<CallerContextCarrier> contextCarriers,
        ObjectProvider<CheckpointCodec> checkpointCodec,
        ObjectProvider<VersionRoutingPolicy> versionRoutingPolicy,
        ObjectProvider<TenantContextResolver> tenantContextResolver,
        ObjectProvider<TenantResourcePolicy> tenantResourcePolicy,
        ObjectProvider<GraphEngineCustomizer> customizers,
        /* ... 更多 ObjectProvider ... */) {
    GraphEngine.Builder builder = GraphEngine.builder()
            .registry(registry)
            .interceptors(interceptors.orderedStream().toList())
            .listeners(listeners.orderedStream().toList())
            .contextCarriers(contextCarriers.orderedStream().toList())
            .checkpointCodec(checkpointCodec.getIfAvailable())
            .versionRoutingPolicy(versionRoutingPolicy.getIfAvailable())
            .tenantContextResolver(tenantContextResolver.getIfAvailable())
            .tenantResourcePolicy(tenantResourcePolicy.getIfAvailable());

    if (properties.getEngineMode() == BlogeProperties.EngineMode.REQUEST_RESPONSE) {
        builder.requestResponseDefaults();
    }

    customizers.orderedStream().forEach(c -> c.customize(builder));
    return builder.build();
}
```

这意味着你声明为 Spring bean 的 任何 `OperatorInterceptor`、`ExecutionListener`、`CallerContextCarrier`、`TenantContextResolver` 或 `TenantResourcePolicy` 都会被引擎自动拾取。你永远不需要手动操作引擎的 builder。

`GraphEngineCustomizer` 是一个回调接口，让你在所有自动配置默认值和 engine-mode 预设应用完毕后微调 `GraphEngine.Builder` ——无需替换整个引擎 bean：

```
@Bean
public GraphEngineCustomizer myCustomizer() {
    return builder -> builder
            .schedulerTimerSupport(myTimerSupport);
}
```

0.1.4 核心配置属性

声明在 BlogeProperties.java 中，前缀为 spring.bloge：

```yaml
spring:
  bloge:
    dsl-locations: classpath:bloge/    # .bloge 文件的扫描位置
    hot-reload: false                  # 文件监听并在变更时重新编译
    default-timeout: 30s               # 默认节点超时
    accessor-mode: method-handle       # 表达式编译模式
    engine-mode: auto                  # 引擎预设（auto | request-response）
    streaming-default-buffer-size: 16  # 流式执行缓冲区大小
```

engine-mode 和 ***streaming-default-buffer-size***。*engine-mode* 为引擎 *builder* 选择高层预设。*AUTO*（默认）使用标准设置；*REQUEST_RESPONSE* 为短生命周期、请求范围内的 *graph* 执行应用优化默认值。*streaming-default-buffer-size* 控制流式 *graph* 执行的默认背压缓冲区。

0.1.5 租户属性

当 spring.bloge.tenant.enabled=true 时，starter 会创建 SpringTenantContextResolver 和 SpringTenantResourcePolicy bean，注入引擎的多租户准入控制：

```yaml
spring:
  bloge:
    tenant:
      enabled: false                   # 主开关
      mode: header                     # header | security-context | static
      header-name: X-Tenant-Id         # 租户解析用的 HTTP 头
      namespace-header-name: X-Namespace # 命名空间解析用的 HTTP 头
      default-tenant: default          # 回退租户 ID
      default-namespace: default       # 回退命名空间
      max-concurrent-executions: 100   # 每租户并发上限
      max-starts-per-minute: 0         # 每租户速率限制（0 = 无限制）
```

mode 属性决定租户的解析方式：

模式	如何解析租户
HEADER	从 HTTP 请求头中读取 X-Tenant-Id 和 X-Namespace
SECURITY_CONTEXT	从 Spring Security 的 Authentication.getDetails() 中提取 tenantId / namespace
STATIC	始终返回 default-tenant / default-namespace

启用租户支持后，starter 还会注册一个 TenantContextFilter ——一个 servlet 过滤器，通过 TenantContextHolder.callWith() 将解析后的 TenantContext 绑定到当前请求线程，使整个 HTTP 请求期间的存储查询和默认 GraphContext 构造 都能看到同一个租户。

0.1.6 版本路由属性

```yaml
spring:
  bloge:
    version-routing:
      policy: latest                  # latest | pinned | canary
      pinned-version: null            # policy=pinned 时必填
      canary-primary: null            # canary 的主版本
      canary-candidate: null          # canary 的候选版本
      canary-percentage: 10           # 路由到候选版本的流量百分比
```

0.1.7 持久化路由属性

```yaml
spring:
  bloge:
    durable:
      routing:
        mode: single                  # single | hash | tenant
        tenant-strategy: null         # shared-table | separate-schema | separate-database
        default-shard-id: default
        fallback-shard-id: null
        shard-ids: []
        tenant-shards: {}             # tenantId → shardId 的映射
```

0.1.8 持久化存储接线

当 classpath 上有 `bloge-durable` 且存在 `DataSource` bean 时，starter 会创建一个 `durableSqlSessionFactory` bean，运行 Flyway 迁移（如果 `spring.bloge.durable.flyway.enabled=true`，这是默认值），并注册十多个运行时存储 bean——每个都带有 `@ConditionalOnMissingBean`，所以你可以覆盖其中任何一个。完整列表包括 `ExecutionStore`、`ExecutionCheckpointStore`、`WaitStore`、`WorkItemStore`、`GraphRegistryStore`、`RoutingStore`、`TaskInboxStore`、`EventMatcherStore`、`EventDeduplicationStore`、`ArchiveService` 和 `DurableControlPlaneService`。

0.1.9 健康指示器

`BlogeHealthIndicator` 在至少一个 graph 被加载时报告 `UP`，在列表为空时报告 `DOWN` 并附带 `"reason": "No graphs loaded"`。它出现在 `/actuator/health` 的 `bloge` 组件下。

0.2 错误响应——RFC 9457 ProblemDetail

`bloge-spring-web` 在 classpath 上时，所有 BLOGE 异常都会被映射到 **RFC 9457 `ProblemDetail`**，这样客户端看到的是统一、机器可读的错误格式，而不是堆栈。

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/problem+json
```

```
{
  "type":   "https://bloge.dev/problems/graph-version-mismatch",
  "title":  "Graph version mismatch",
  "status": 422,
  "detail": "Execution started against orderProcess@v3 but live registry has @v4.",
  "instance": "/executions/exec-12af9",
  "executionId": "exec-12af9",
  "graphId":     "orderProcess",
  "expectedHash": "sha256:1f3a…",
  "actualHash":   "sha256:9c0e…"
}
```

`BlogeProblemDetailAdvice` 内置的映射:

异常	HTTP 状态	type URI 后缀
`GraphDefinitionException`	400	`/problems/graph-definition`
`GraphValidationException`	422	`/problems/graph-validation`
`GraphVersionMismatchException`	422	`/problems/graph-version-mismatch`
`SessionAccessDeniedException`	403	`/problems/session-access-denied`
`StaleLeaseException`	409	`/problems/stale-lease`
`OperatorTimeoutException`	504	`/problems/operator-timeout`
其他	500	`/problems/internal`(不泄漏异常细节)

要禁用或覆盖, 自己声明一个 `@RestControllerAdvice` bean 即可——自动配置 通过 `@ConditionalOnMissingBean(BlogeProblemDetailAdvice.class)` 让位。

0.3 引导式重写

从工单分流示例出发，逐层添加生产特性。

0.3.1 1. 添加可观测性

在 POM 中添加 `bloge-metrics-otel`。无需修改任何配置——starter 会检测到 classpath 上的 `MetricsExecutionListener`，并将其注册到你现有的 `MeterRegistry` 中。

验证:

```
GET /actuator/bloge
```

你应该看到:

```
{
  "graphCount": 1,
  "graphs": [
    {
```

```
      "name": "springTicketTriage",
      "nodeCount": 2,
      "edgeCount": 1,
      "sourceNodes": ["classifyTicket"],
      "terminalNodes": ["draftReply"]
    }
  ]
}
```

0.3.2 2. 启用审计日志

在 `application.yml` 中添加：

```
spring:
  bloge:
    audit:
      enabled: true
      capture-input: true
      capture-output: false
      async-flush: true
      flush-interval: 100ms
      batch-size: 64
```

这需要一个持久化 `DataSource`，以便 `AuditJournalStore` 可用。`BlogeAuditAutoConfiguration` 会创建一个 `AuditJournalListener`，引擎会自动拾取它。

0.3.3 3. 启用崩溃恢复

```
spring:
  bloge:
    recovery:
      enabled: true
      scan-interval: 30s
      batch-size: 10
      claim-timeout: 5m
      max-recovery-attempts: 3
```

starter 会创建一个 `RecoveryConfig` bean 并将其接入 `GraphEngine`。引擎在后台管理执行租约，并扫描过期的 RUNNING 执行以进行恢复。

> 恢复需要 *classpath* 上有 `bloge-durable`，以及版本 *V11* 或更高的 *Flyway* 迁移。

0.3.4 4. 添加自定义拦截器

```
@Bean
public OperatorInterceptor loggingInterceptor() {
    return invocation -> {
        log.info("Executing node: {}", invocation.nodeId());
```

```
            return invocation.proceed();
        };
    }
```

`graphEngine()` bean 通过 `ObjectProvider<OperatorInterceptor>` 收集所有 `OperatorInterceptor` bean，并按 Spring 排序注入。无需编写注册代码。

0.3.5 5. 生产部署模式

生产部署通常再增加 graph 版本路由、按 Spring profile 切换 operator 实现，以及共享 durable store 的滚动升级。它们都属于部署参考，不改变本章的最短 接线主线；先固定单一变化轴，再分别验证版本、环境绑定或恢复行为。

0.4 生产就绪快速检查

边界	发布前必须回答的问题
组件所有权	哪些 bean 来自自动配置，哪些被 override，谁拥有 override?
Operator 发现	应用能否证明每个必需 operator 恰好注册一次?
持久化	声称重启恢复时，execution、wait 与 work-item store 是否真正持久?
路由	Tenant、version 与 durable routing 策略是否显式且经过测试?
错误契约	客户端是否得到稳定 ProblemDetail type，并且看不到内部异常?
可观测性	一个 execution identity 能否连接日志、指标、trace 与业务事件?
恢复	是否针对当前 store 与版本完成过冷重启测试?

请记录答案与证据位置。没有 owner、测试和运维后果的属性值，只是配置知识，不是生产就绪证据。