Microsoft

How Less Can Be More

少即是多

The War of Simplicity and Complexity in Dapr Workflow

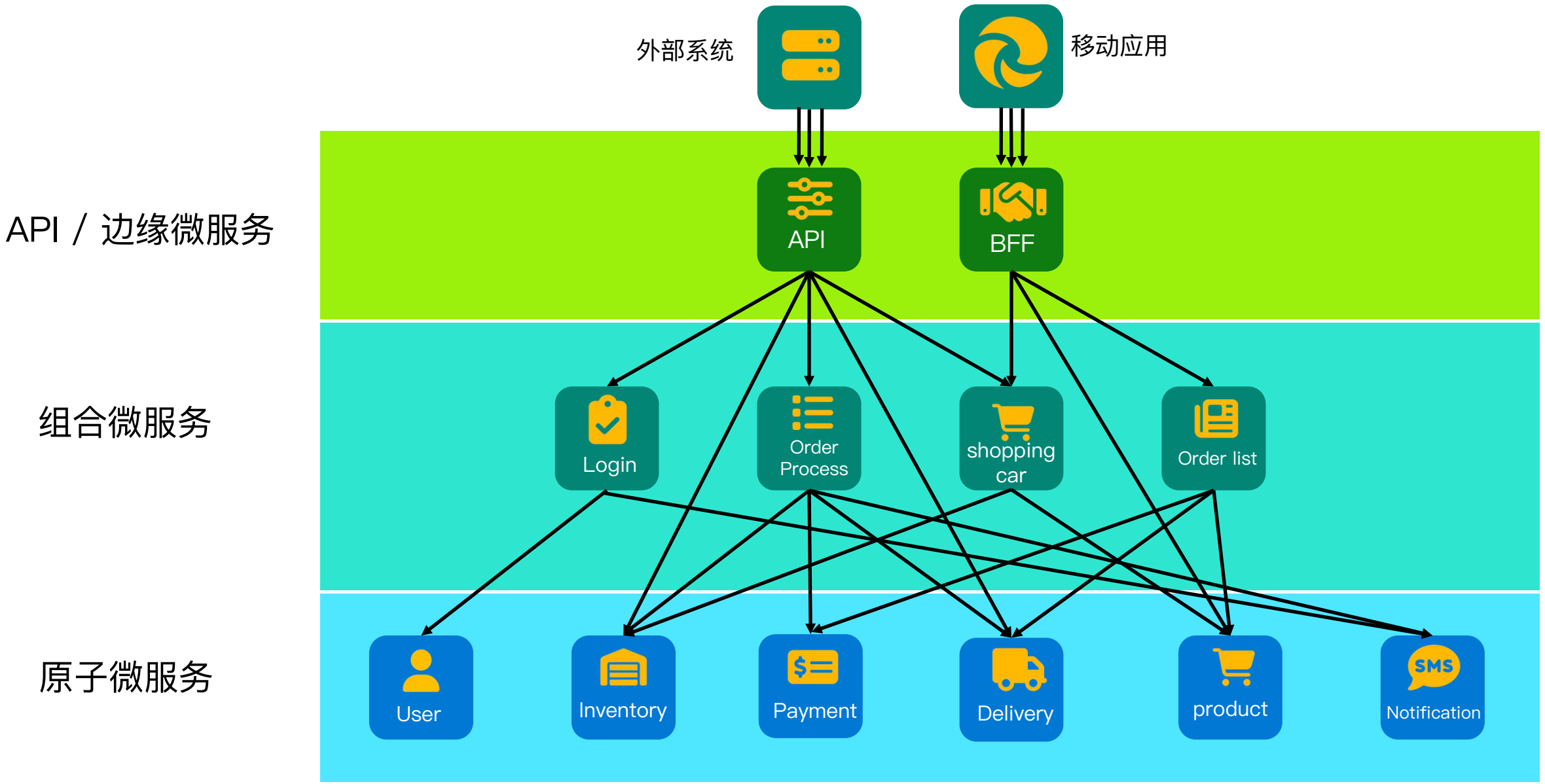
敖小剑 / Sky Ao

背景：微服务分层与工作流

1

- 微服务分层架构
- 工作流的通用模式
- 工作流的挑战

微服务分层架构

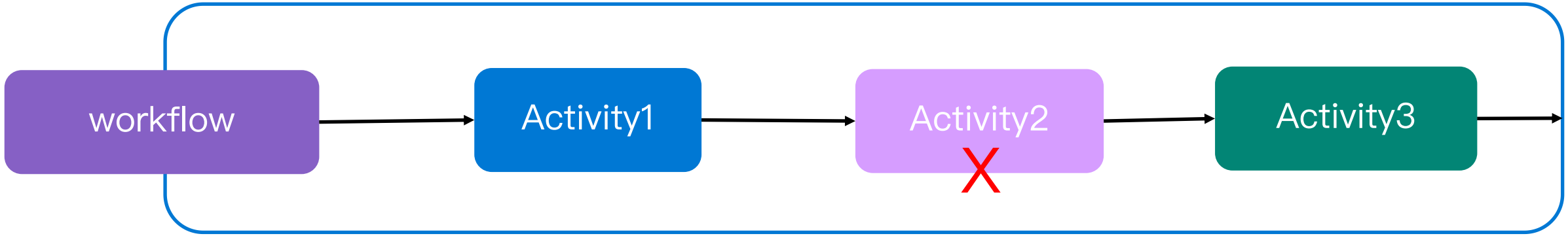

外部系统
移动应用
API / 边缘微服务
API
BFF
组合微服务
Login
Order Process
shopping car
Order list
原子微服务
User
Inventory
Payment
Delivery
product
SMS
Notification

工作流通用模式

工作流是实现服务编排/组合微服务的通用方案。

- 任务链模式
- Fan-out/Fan-in 模式
- 观察者模式
- 外部系统交互模式

任务链模式



挑战

- 如果其中一个活动因为 可恢复故障 而长时间失败，要如何处理?
- 可否自动重试工作流?
- 可否从上一次失败的点重新开始并取回所有的工作流状态?

可恢复故障

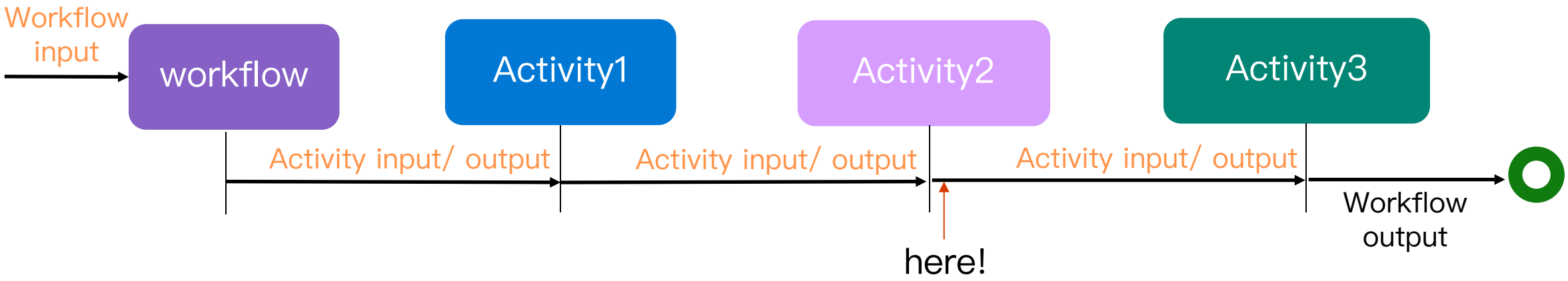
服务过载

- 慢响应
- 超时
- 临时性错误
- 熔断
- 限流

服务不可用

- 进程崩溃
- 网络异常

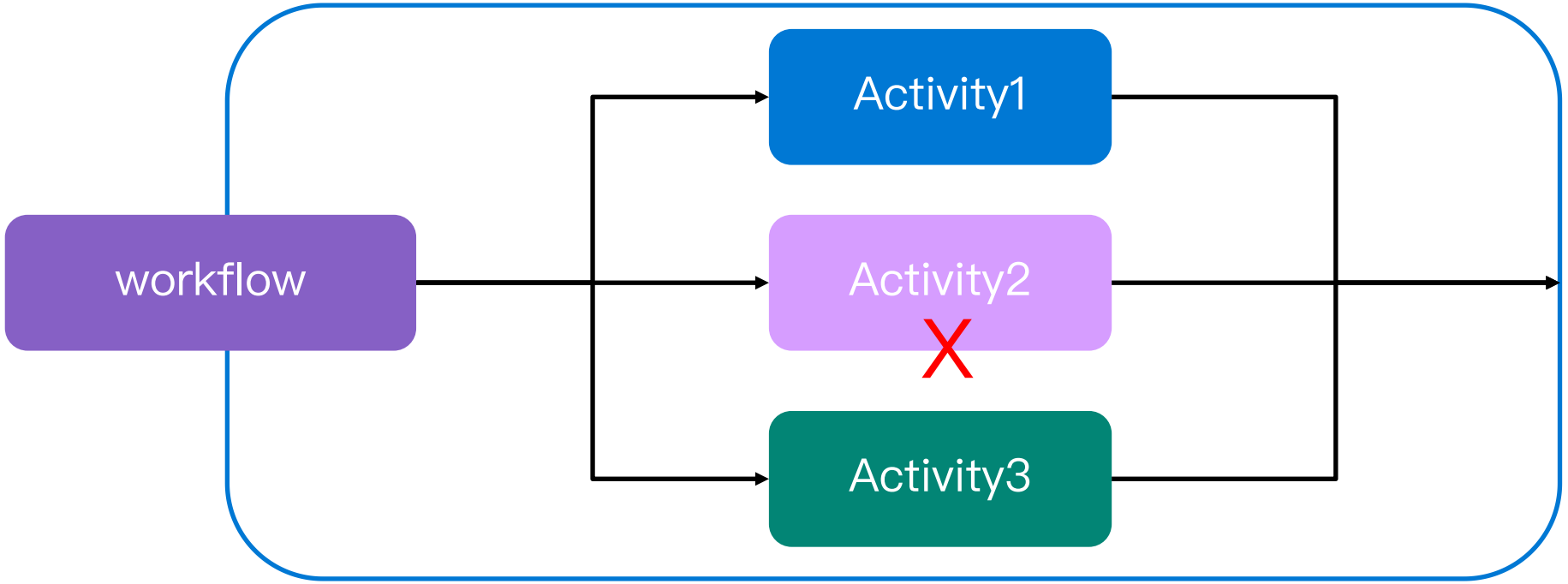
工作流状态



所有的变量都是状态

```
WorkflowInput workflowInput = ctx.getInput(WorkflowInput.class);
// …… some business logic
// local variables
String orderId = ctx.getInstanceId();
Object output1 = ctx.callActivity(Activity1.class.getName(), input1).await();
// …… some business logic
Object output2 = ctx.callActivity(Activity2.class.getName(), input2).await();
// …… some business logic
// here!
Object output3 = ctx.callActivity(Activity3.class.getName(), input3).await();
// …… some business logic
ctx.complete(workflowOutput);
```

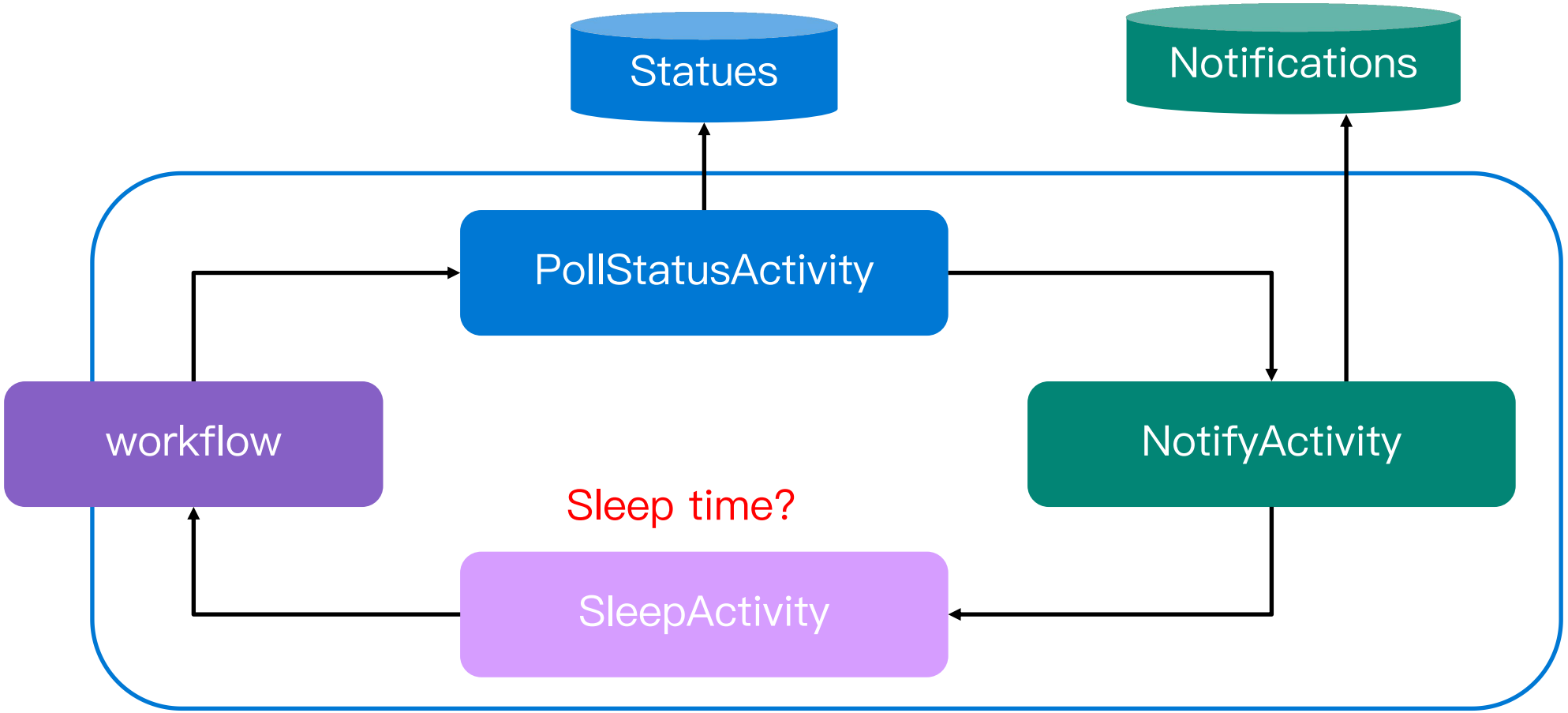
Fan–out/Fan–in 模式



挑战

- 如何控制并发度?
- 如何触发后续的聚合步骤?

观察者模式

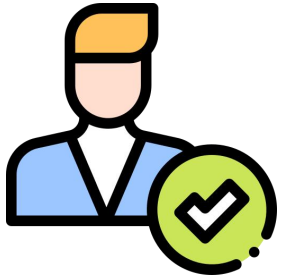


挑战：

- 休眠时间可能是动态的，取决于业务逻辑
- 长时间观察

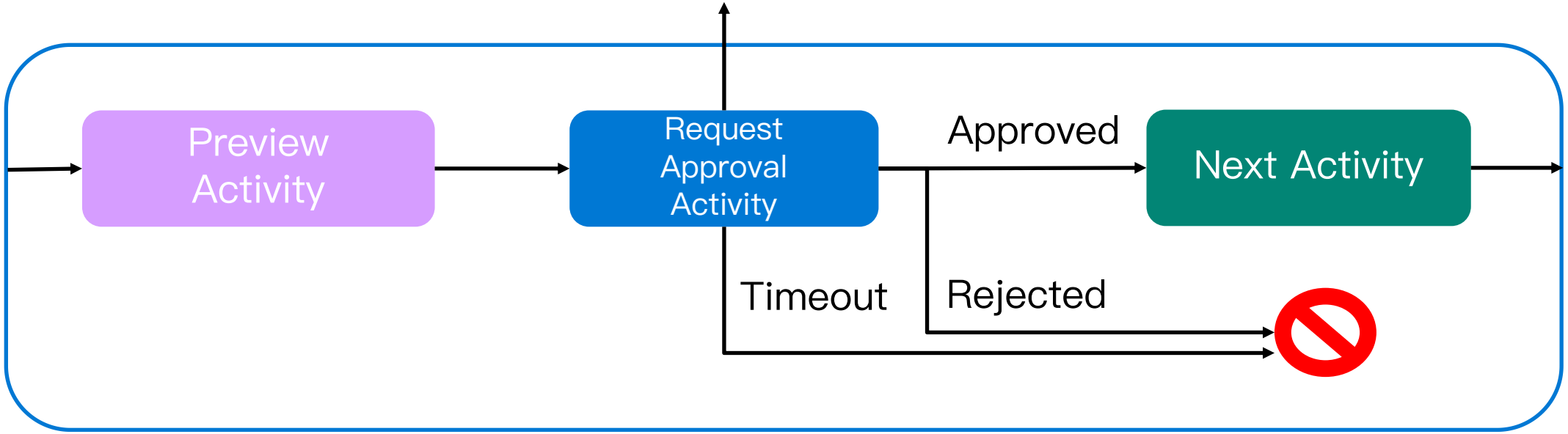
外部系统交互模式

人工操作



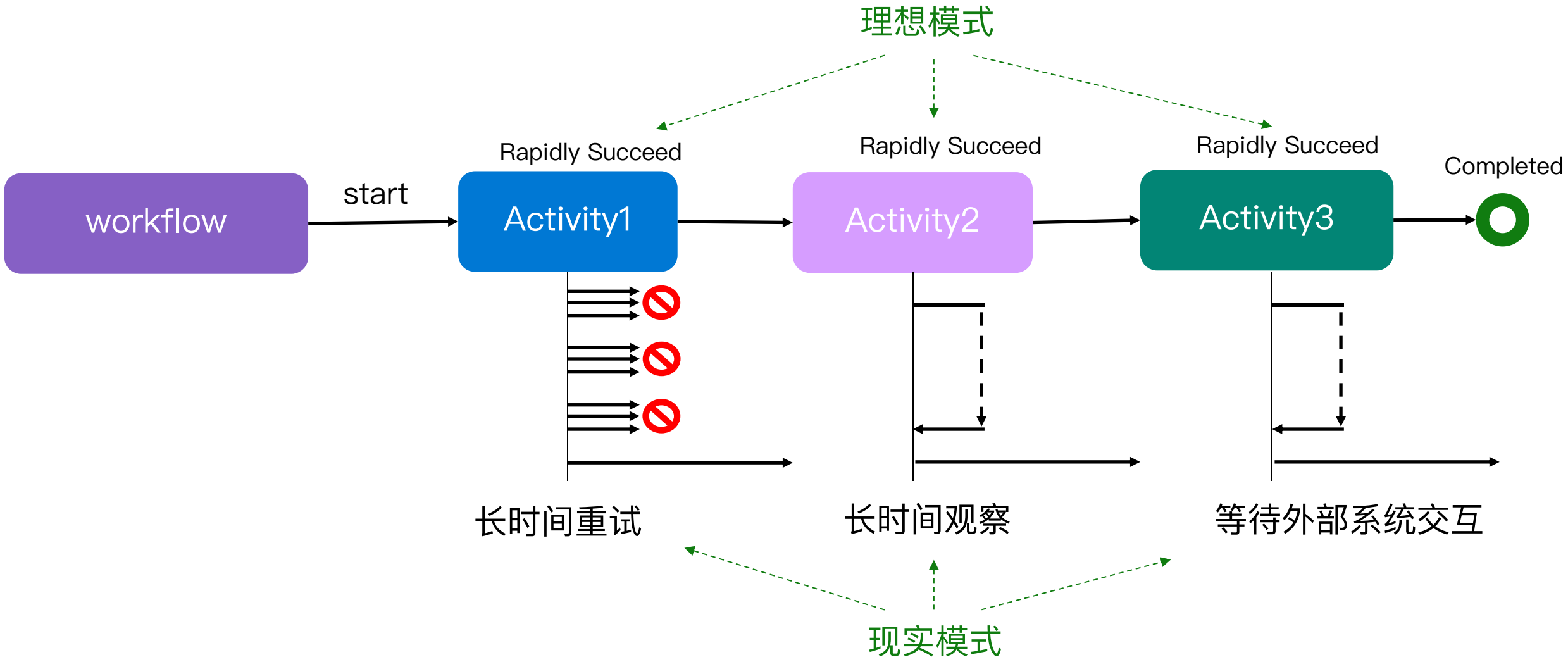
挑战

- 人工操作的等待时间非常长

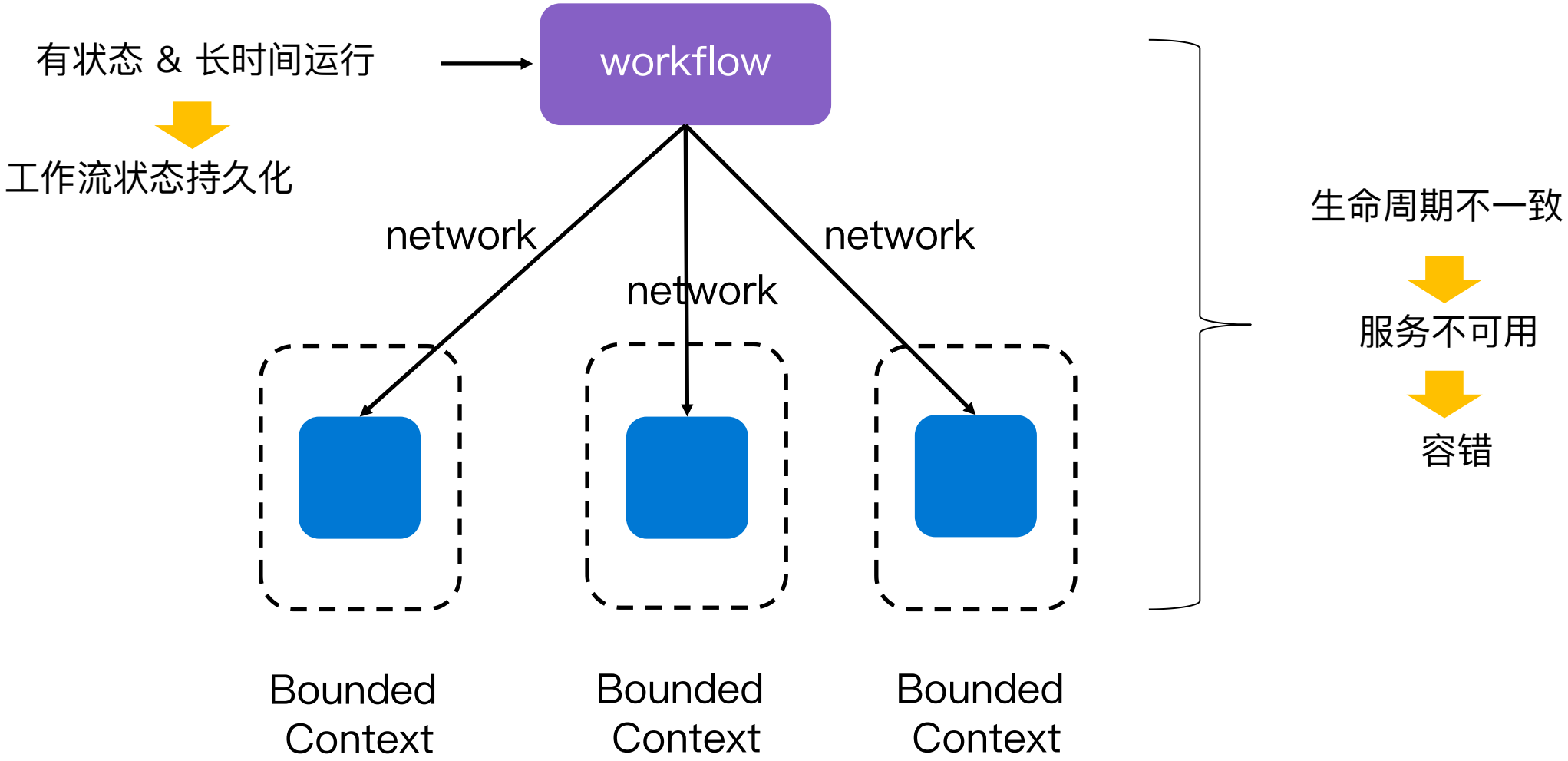


- 暂停并等待外部系统执行某些操作
- 常见场景：暂停并等待人工操作
- 案例：等待请求审核，等待用户付款

结论：理想很美好，实现很骨感



工作流挑战：容错 & 长时间运行 & 有状态



不友好的用户体验

技术实现细节 **混杂** 工作流业务逻辑

如何触发工作流的重复执行?

如何保存和取回工作流状态?

在长时间等待时如何跳出工作流?

如何重新开始工作流

……

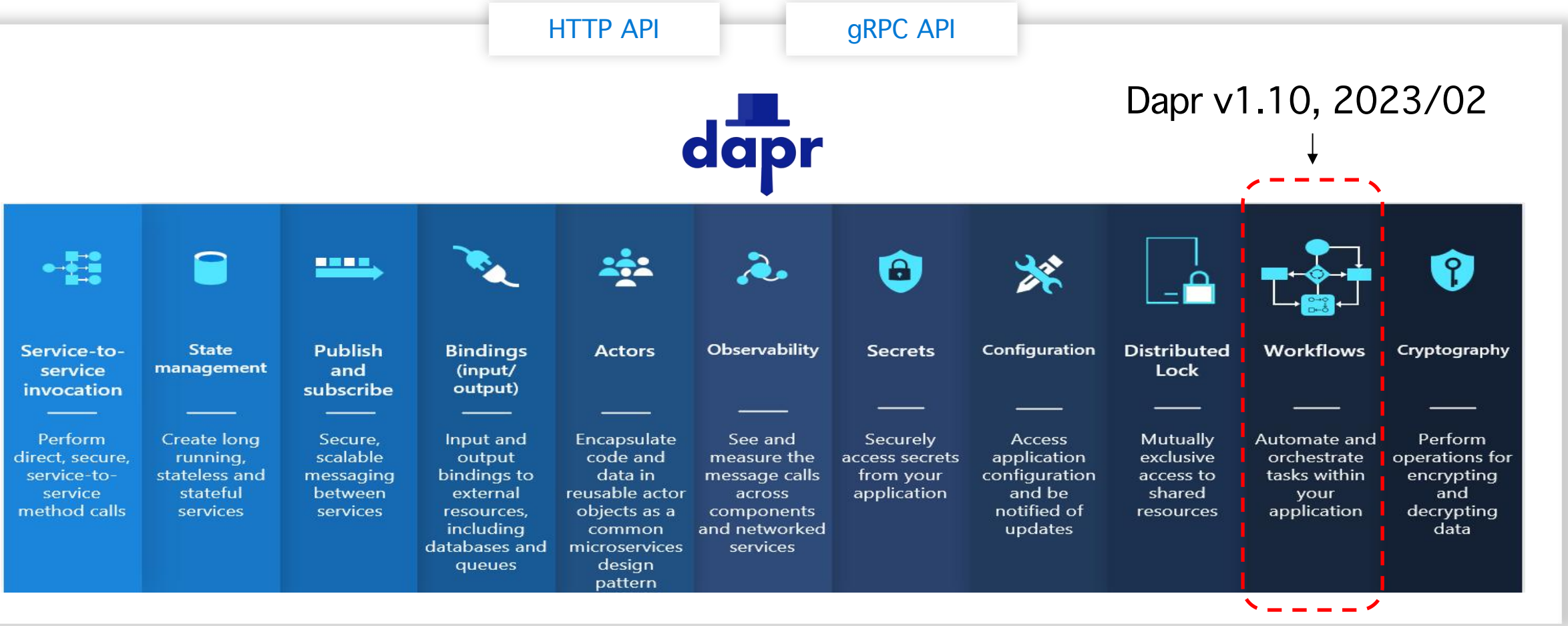
```
WorkflowInput workflowInput = ctx.getInput(WorkflowInput.class);

String orderId = ctx.getInstanceId();
Input1 input1 = …… // some business logic
Object output1 = ctx.callActivity(
            Activity1.class.getName(), input1).await();

Input2 input2 = …… // some business logic
Object output2 = ctx.callActivity(
            Activity2.class.getName(), input2).await();

Input3 input3 = ……; // some business logic
Object output3 = ctx.callActivity(
            Activity3.class.getName(), input3).await();

// …… some business logic
ctx.complete(workflowOutput);
```

介绍： Dapr Workflow

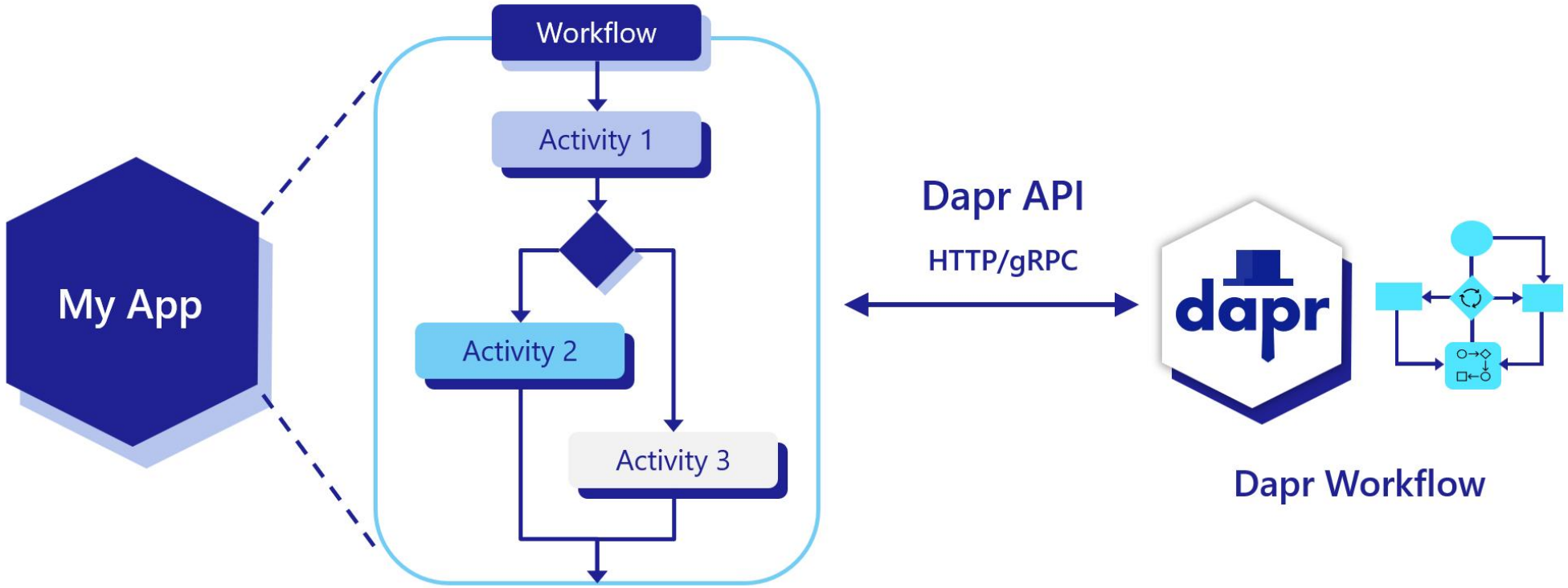
2

- Dapr worflow 简介
- 设计和实现
- 简化工作流的编写

2023年新加入的 Dapr Workflow 构建块

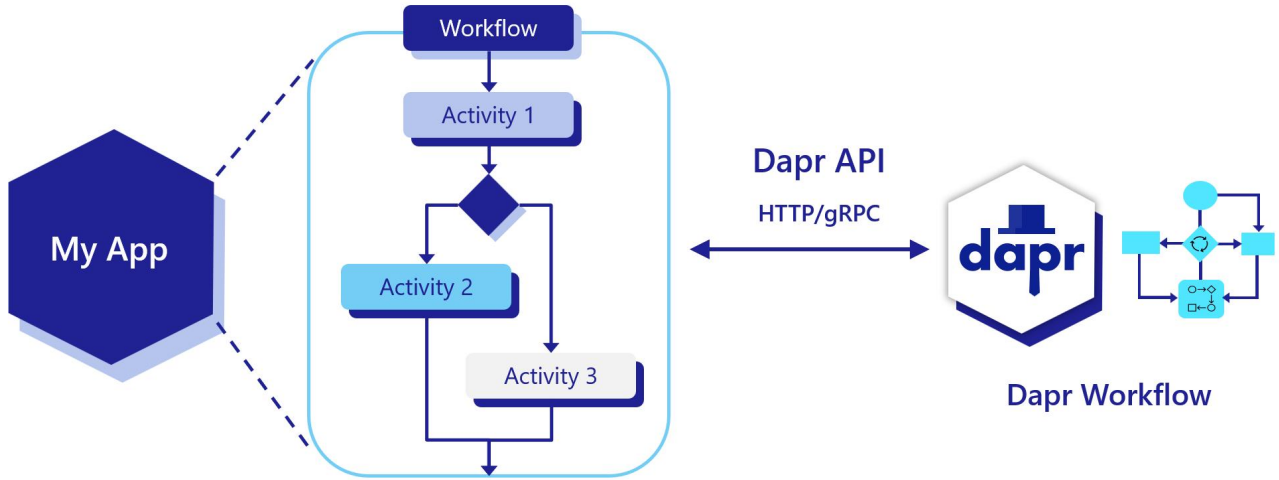


Dapr workflow 介绍



“Since Dapr workflows are stateful, they support long-running and fault-tolerant applications, ideal for orchestrating microservices.”

Dapr Workflow 设计目标



“Dapr workflow makes it easy for developers to write business logic and integrations in a reliable way. ”

工作流编写

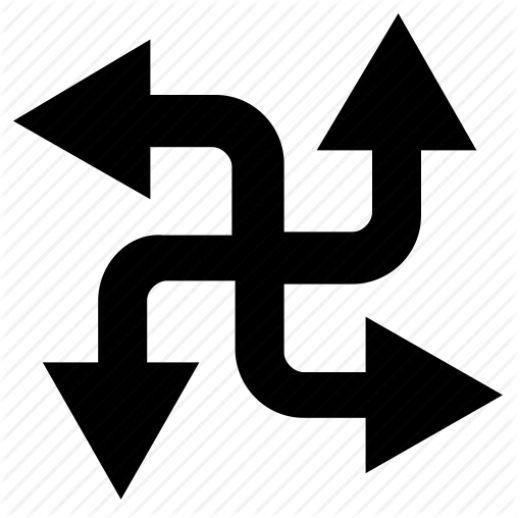
工作流执行

回顾：工作流的挑战

工作流的挑战

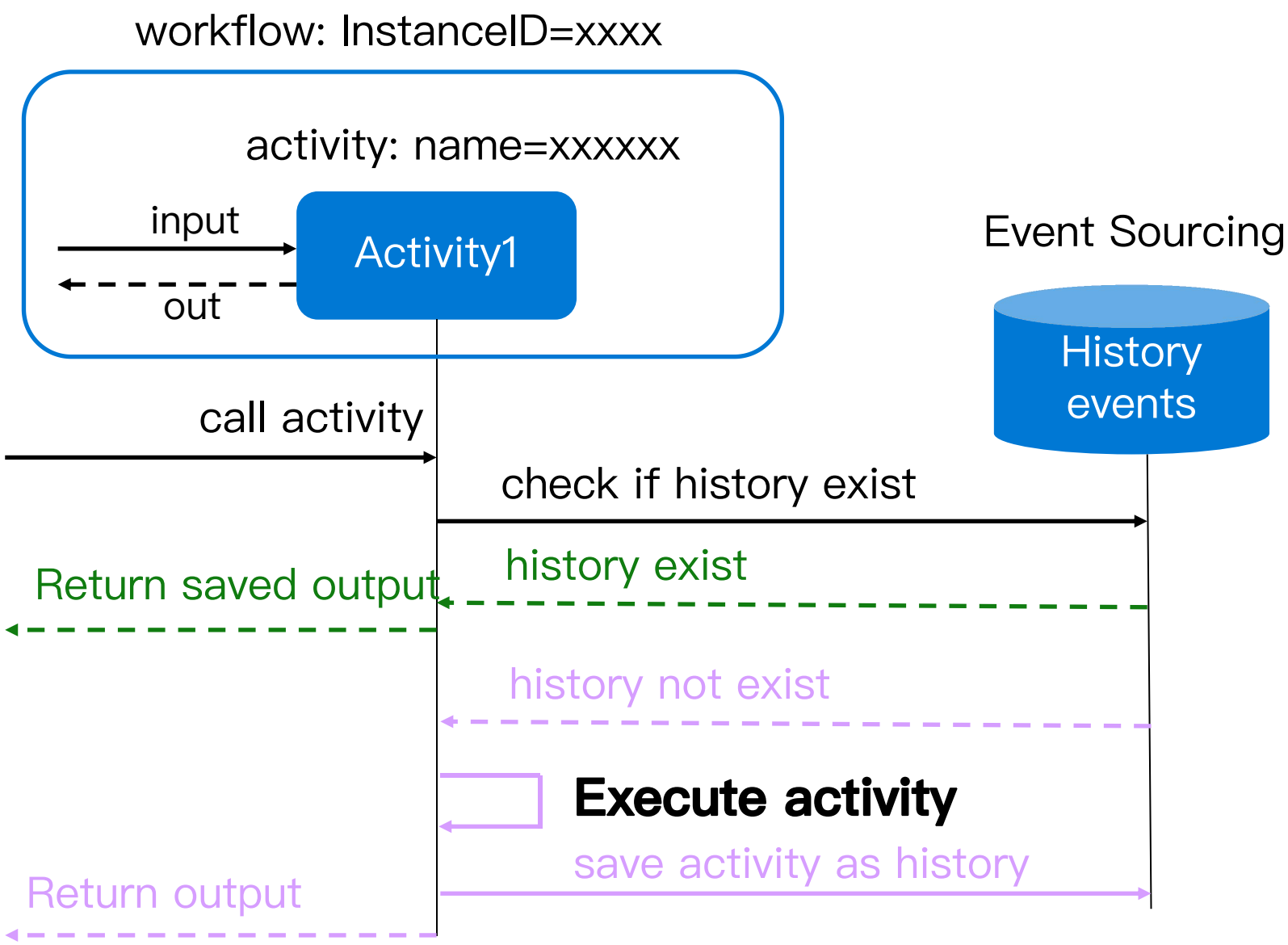
- 容错
- 长时间运行
- 有状态

用户体验不友好

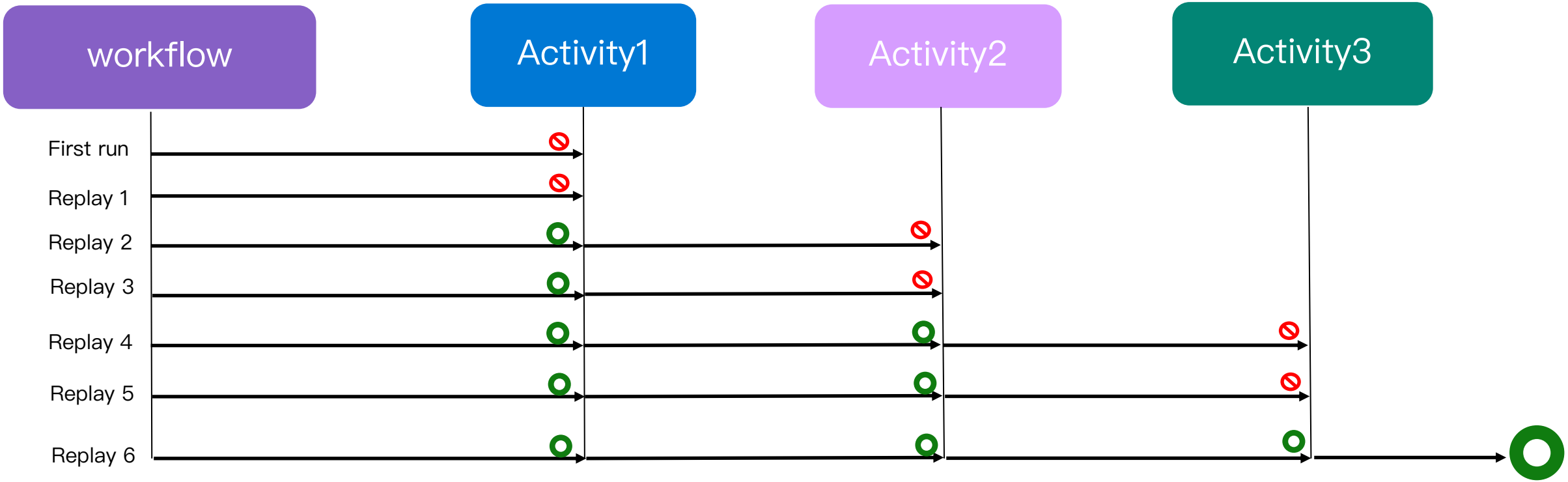


工作流业务逻辑混杂技术实现细节

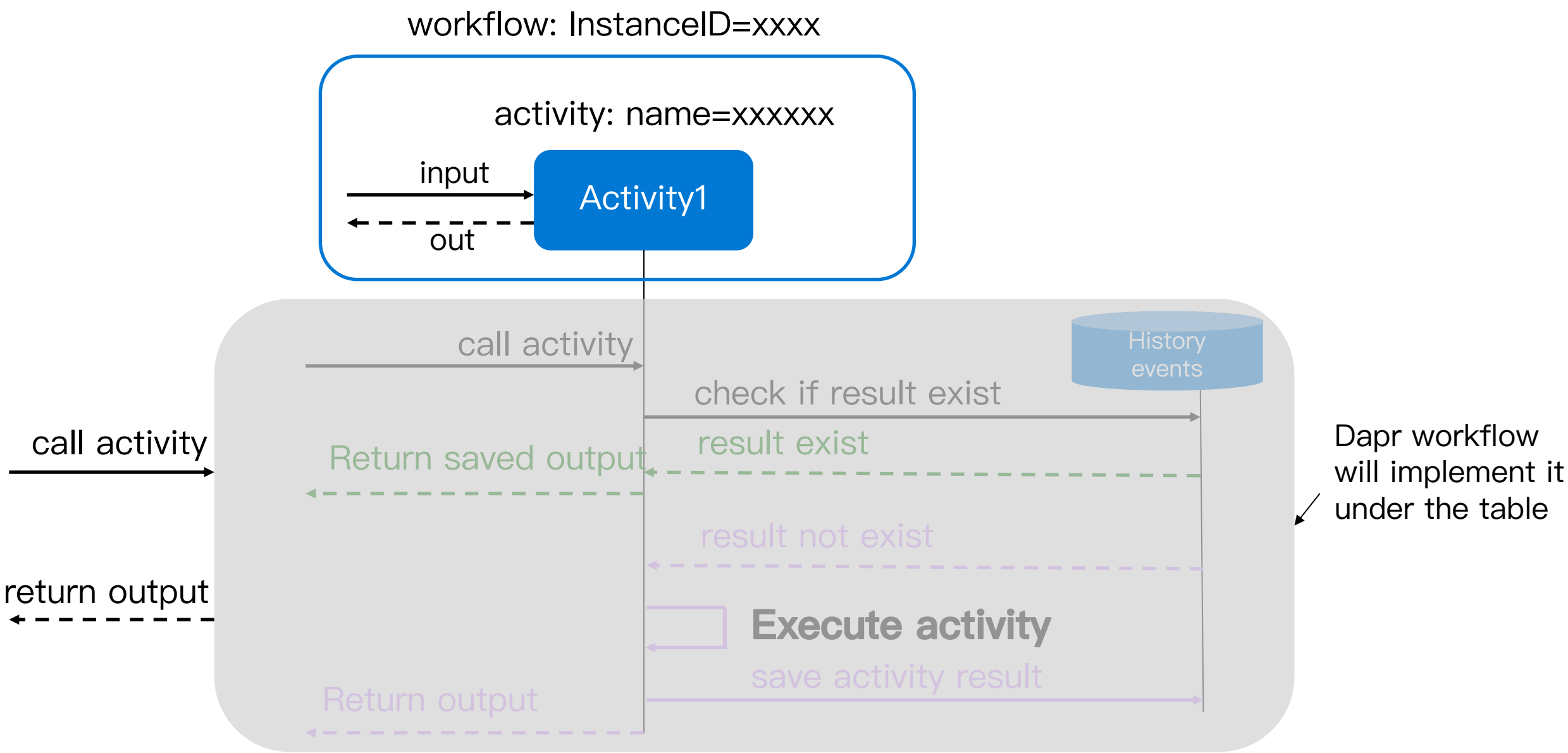
Dapr Workflow设计(1): Activity History Events



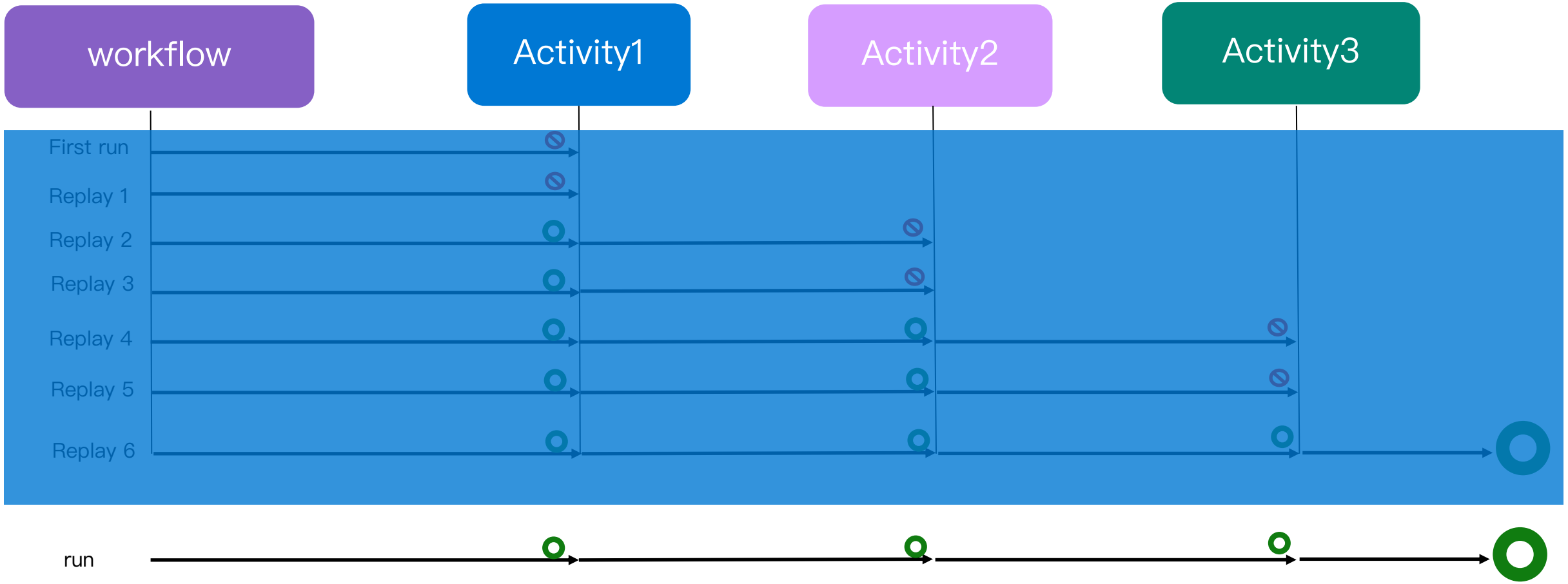
Dapr Workflow设计(2): Replay



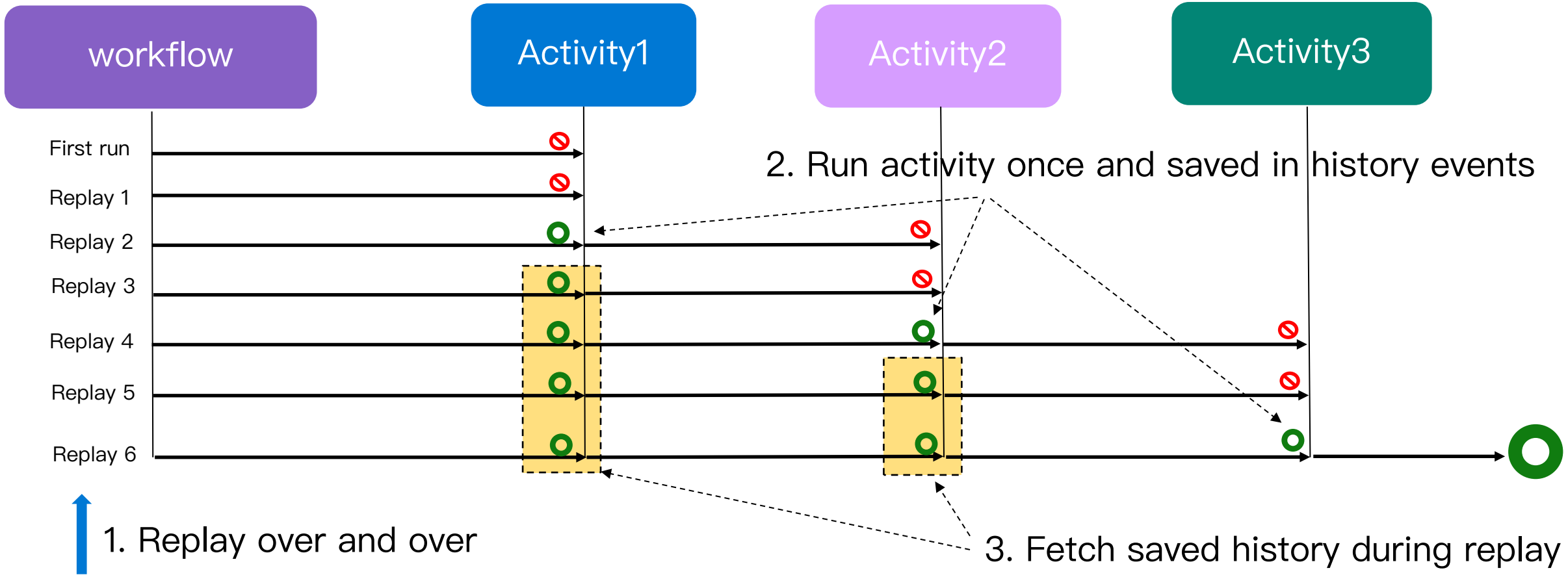
Dapr Workflow设计(3): 隐藏工作流执行细节



Dapr Workflow 设计(4): Hide Replay

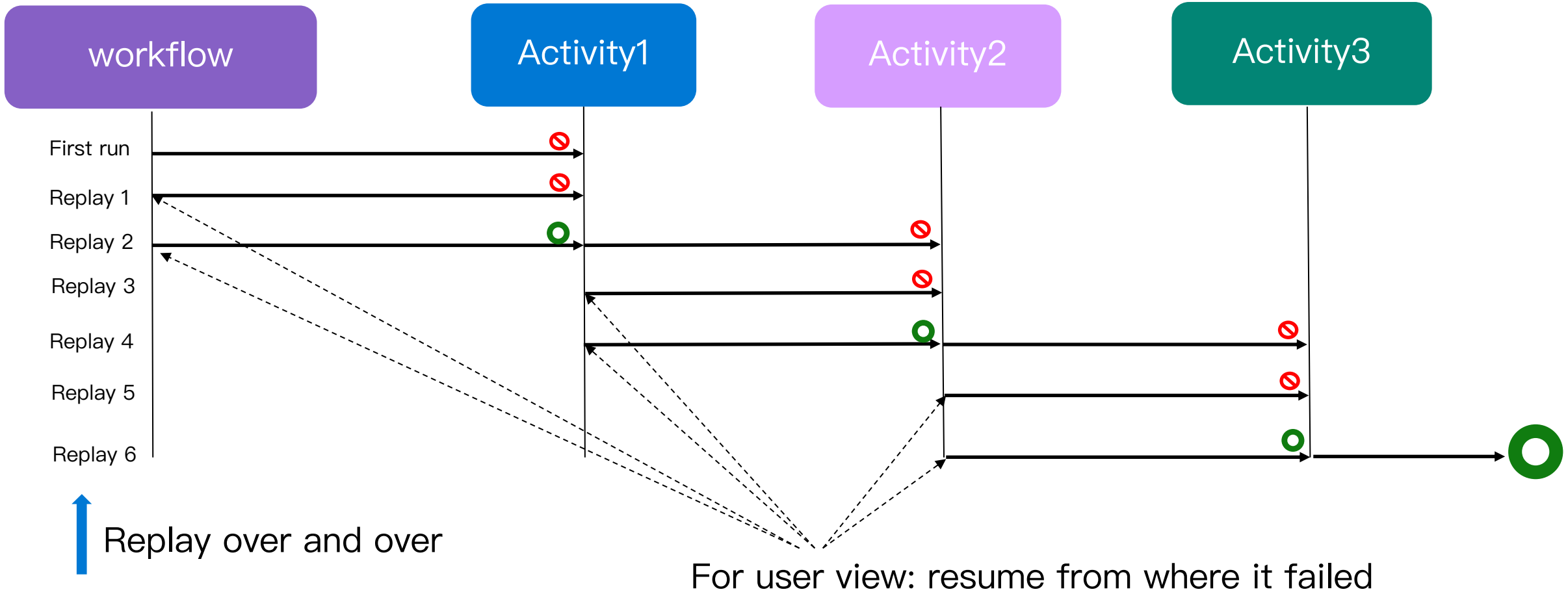


完整视角：Dapr Workflow实现细节



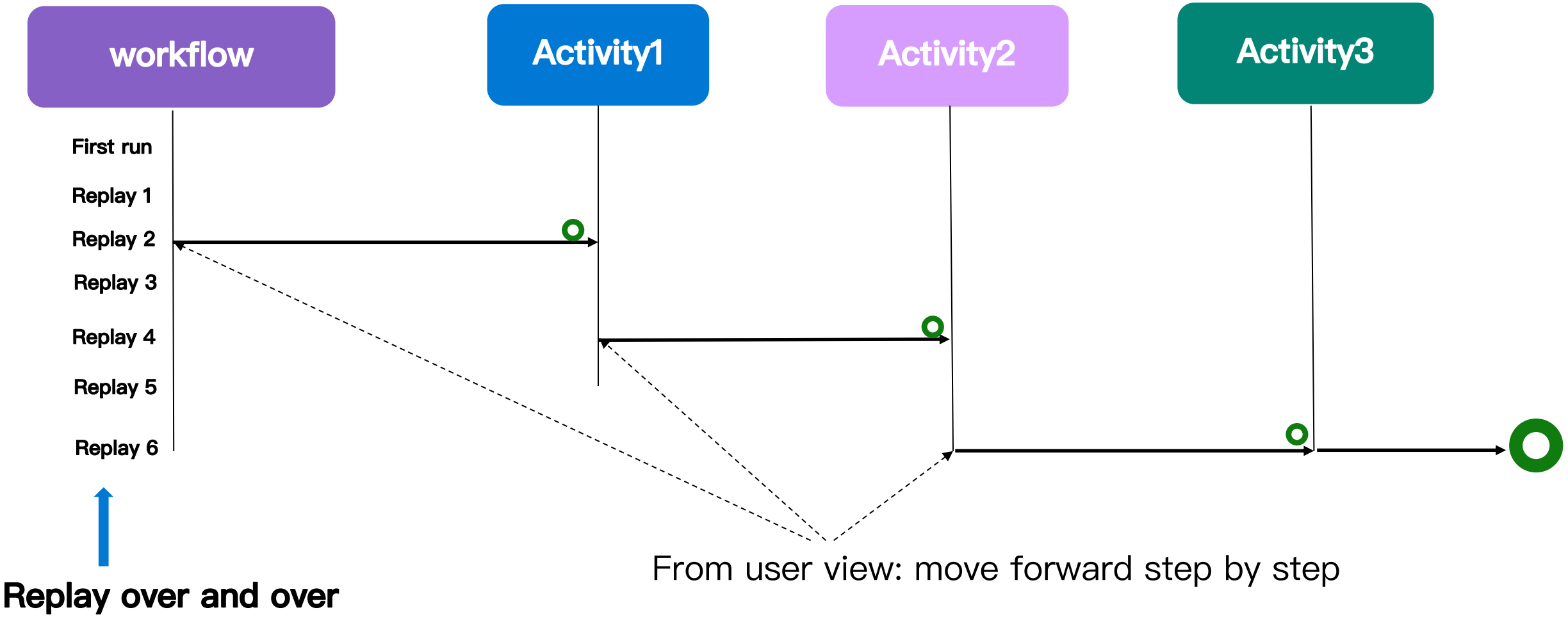
“Dapr workflow makes it …… in a **reliable** way. ”

用户简化视角(1): 隐藏历史记录事件的处理



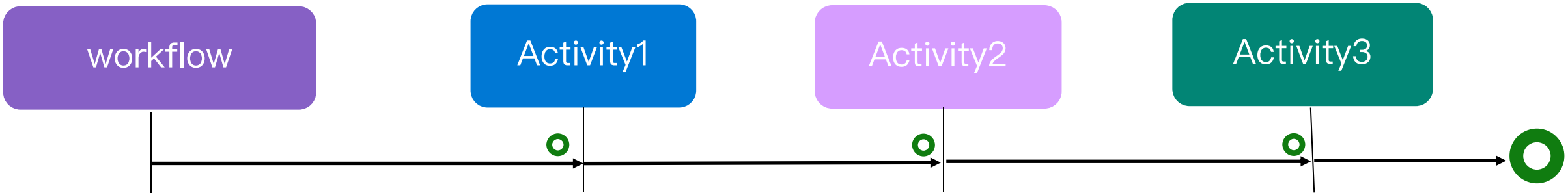

```
Object output1 = ctx.callActivity(Activity1.class.getName(), input1).await();
```

用户简化视角(2): 隐藏可恢复故障的重放




```
Object output1 = ctx.callActivity(Activity1.class.getName(), input1).await();
```

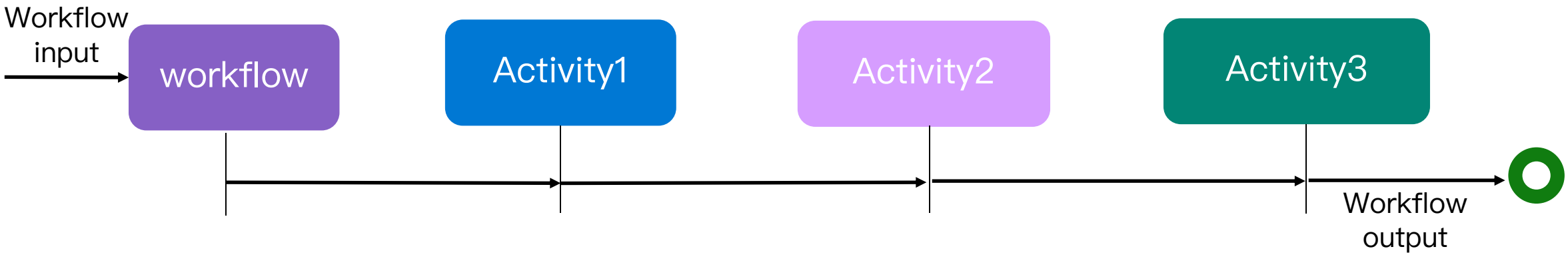
用户简化视角(3): 隐藏重放



从用户视角看：

- 没有发生任何事情
- 不需要处理任何事情
- **可以专注于纯业务逻辑**

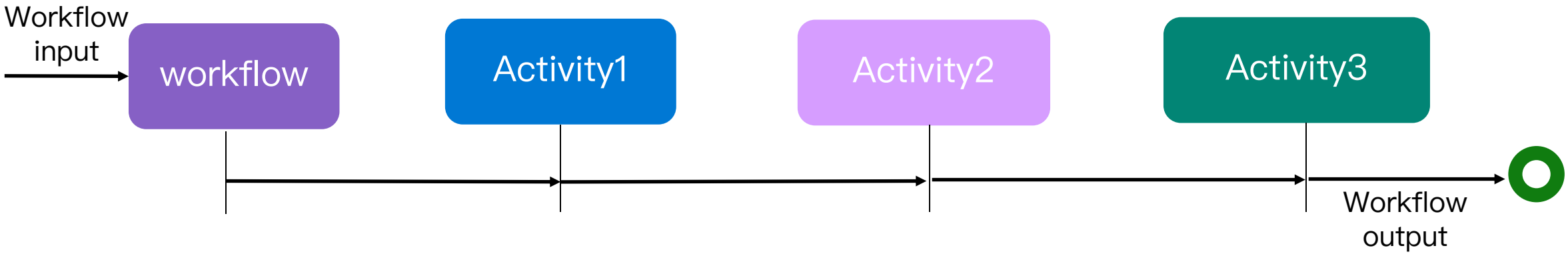
翻译翻译：什么叫做专注于纯业务逻辑?



工作流业务逻辑
(伪代码):

```
get input of workflow as workflowInput;

Build up input of Activity1 as input1
Call Activity1 with input1 and get activity output as output1

Build up input of Activity2 as input2
Call Activity2 with input2 and get activity output as output2


Build up input of Activity3 as input3
Call Activity3 with input3 and get activity output as output3

Build up workflowOutput
Return workflowOutput as workflow output
```

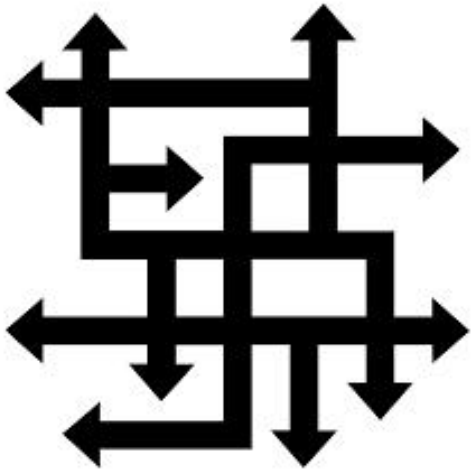
这就是专注于纯业务逻辑



工作流编写
(Java 代码示例)

```java
WorkflowInput workflowInput = ctx.getInput(WorkflowInput.class);

String orderId = ctx.getInstanceId();
Input1 input1 = …… // some business logic
Object output1 = ctx.callActivity(Activity1.class.getName(), input1).await();

Input2 input2 = …… // some business logic
Object output2 = ctx.callActivity(Activity2.class.getName(), input2).await();

Input3 input3 = ……; // some business logic
Object output3 = ctx.callActivity(Activity3.class.getName(), input3).await();

// …… some business logic
ctx.complete(workflowOutput);
```

Dapr workflow 优势



只需掌握基本的语言编程技巧，就能完成非常复杂的工作流程编排工作

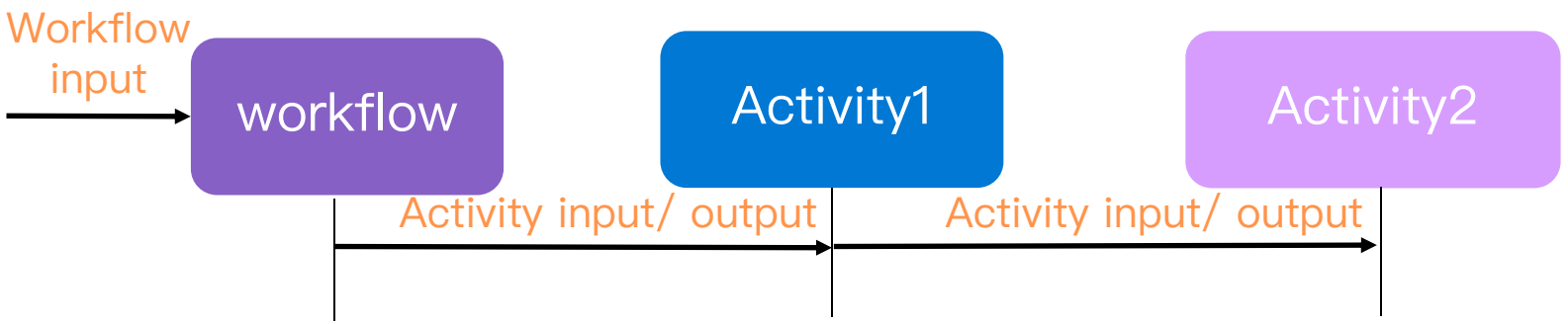
领域专家可直接参与工作流程的编写。

还有一些关键问题需要回答：

如何在工作流执行期间保持状态?

“Workflow is long-running and stateful”

Dapr workflow 设计: 确定性约束



Is it deterministic?

workflow Instanceld, yes

workflow input, yes

activity1 input, yes

activity2 input, yes

activity2 output, yes

business logic, yes

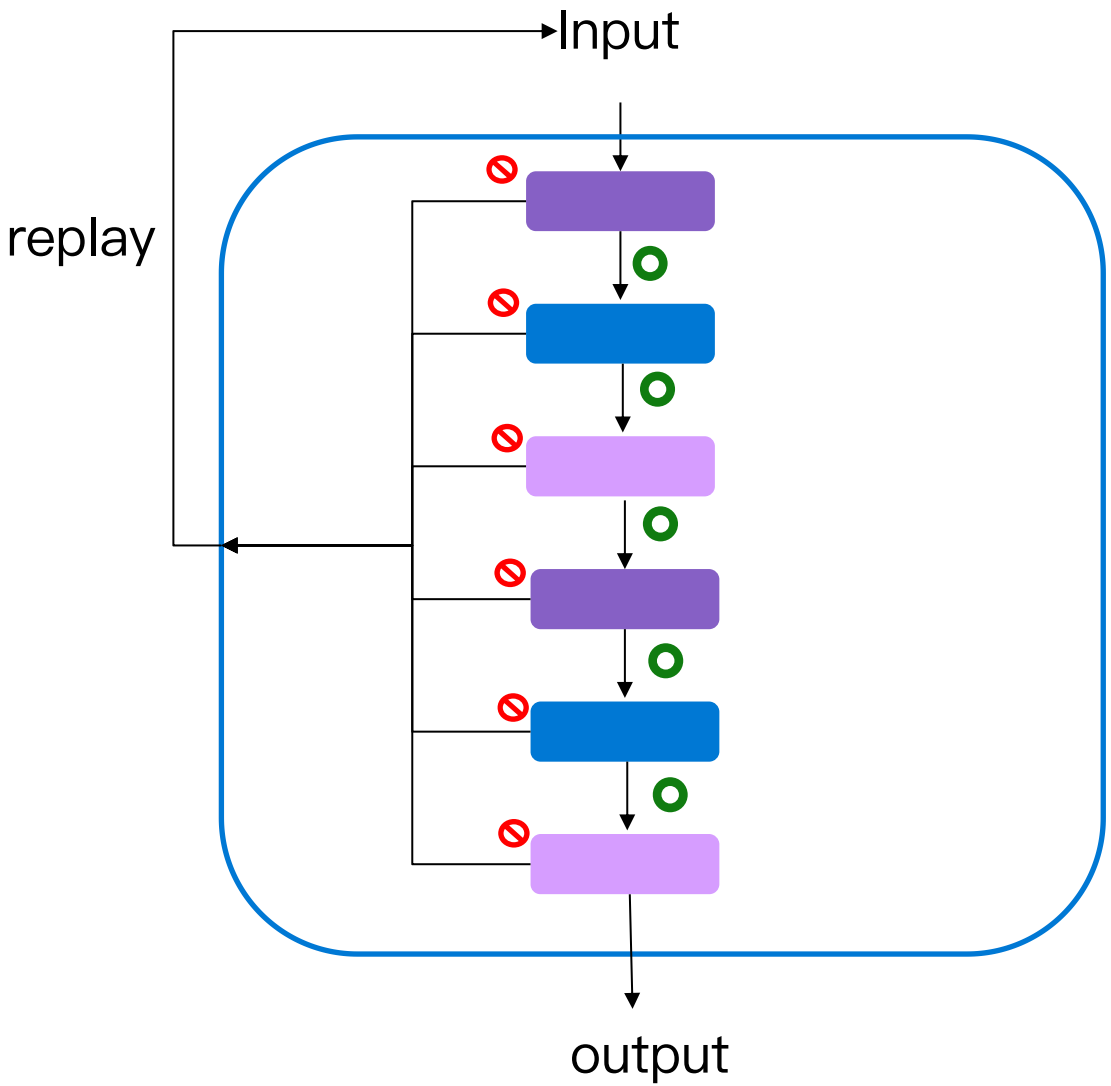
activity3 input, yes

……(keep yes)

workflow output, **yes**

```
String orderId = ctx.getInstanceId();
OrderPayload order = ctx.getInput(OrderPayload.class);

// step1: notify the user that an order has come through
Notification notification = new Notification();
notification.setMessage("Received Order: " + order.toString());
ctx.callActivity(NotifyActivity.class.getName(), notification).await();

InventoryRequest inventoryRequest = new InventoryRequest();
inventoryRequest.setRequestId(orderId);
inventoryRequest.setItemName(order.getItemName());
inventoryRequest.setQuantity(order.getQuantity());
InventoryResult inventoryResult = ctx.callActivity(ReserveInventoryActivity.class.getName(),
inventoryRequest, InventoryResult.class).await();

if (!inventoryResult.isSuccess()) {
     notification.setMessage("Insufficient inventory for order : " + order.getItemName());
     ctx.callActivity(NotifyActivity.class.getName(), notification).await();
     ctx.complete(orderResult);
     return;
}
```

确定性的设计哲学

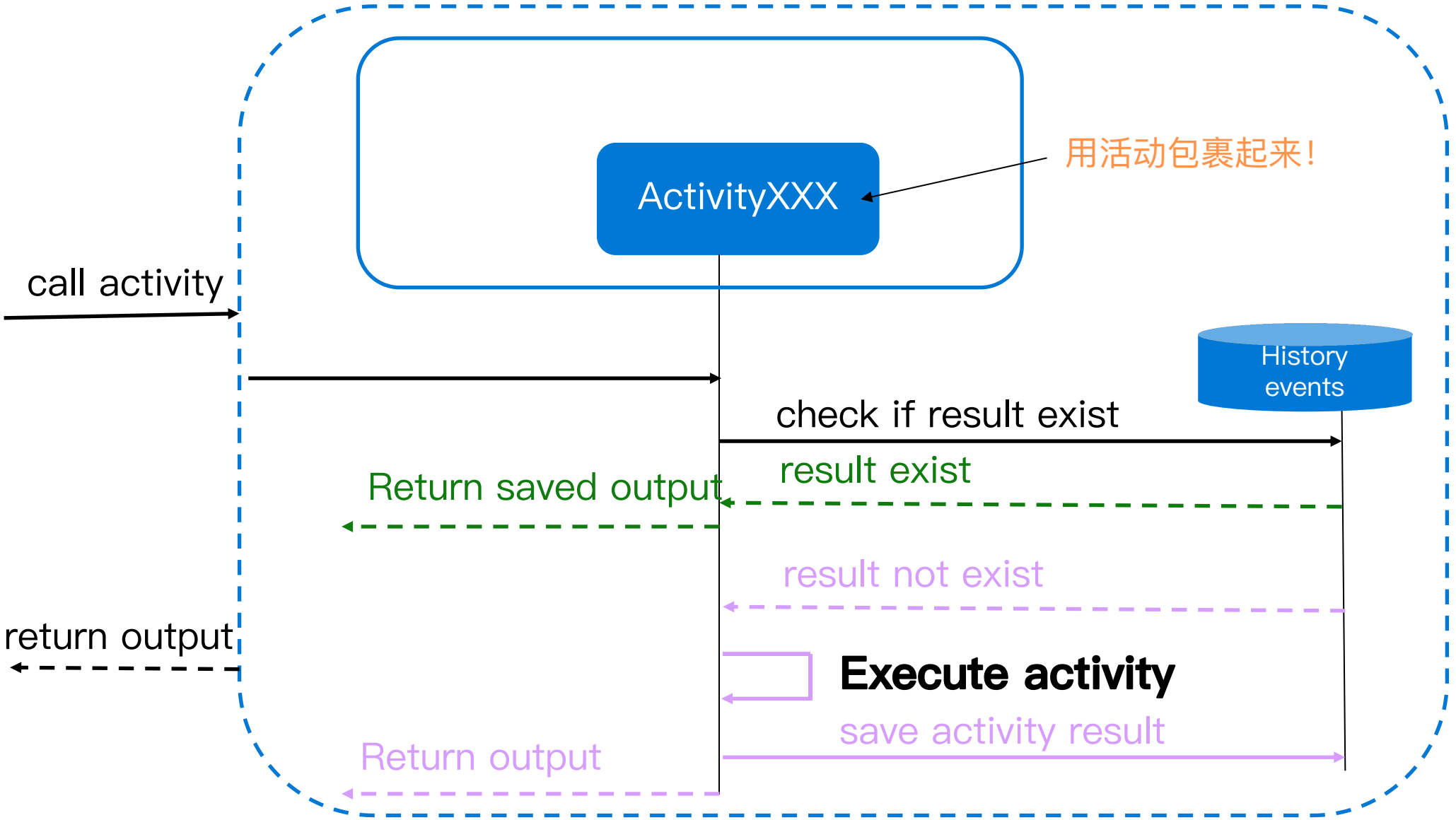


如果步骤的输入是确定的，步骤的执行过程也是确定的，那么输出也是确定的吗?

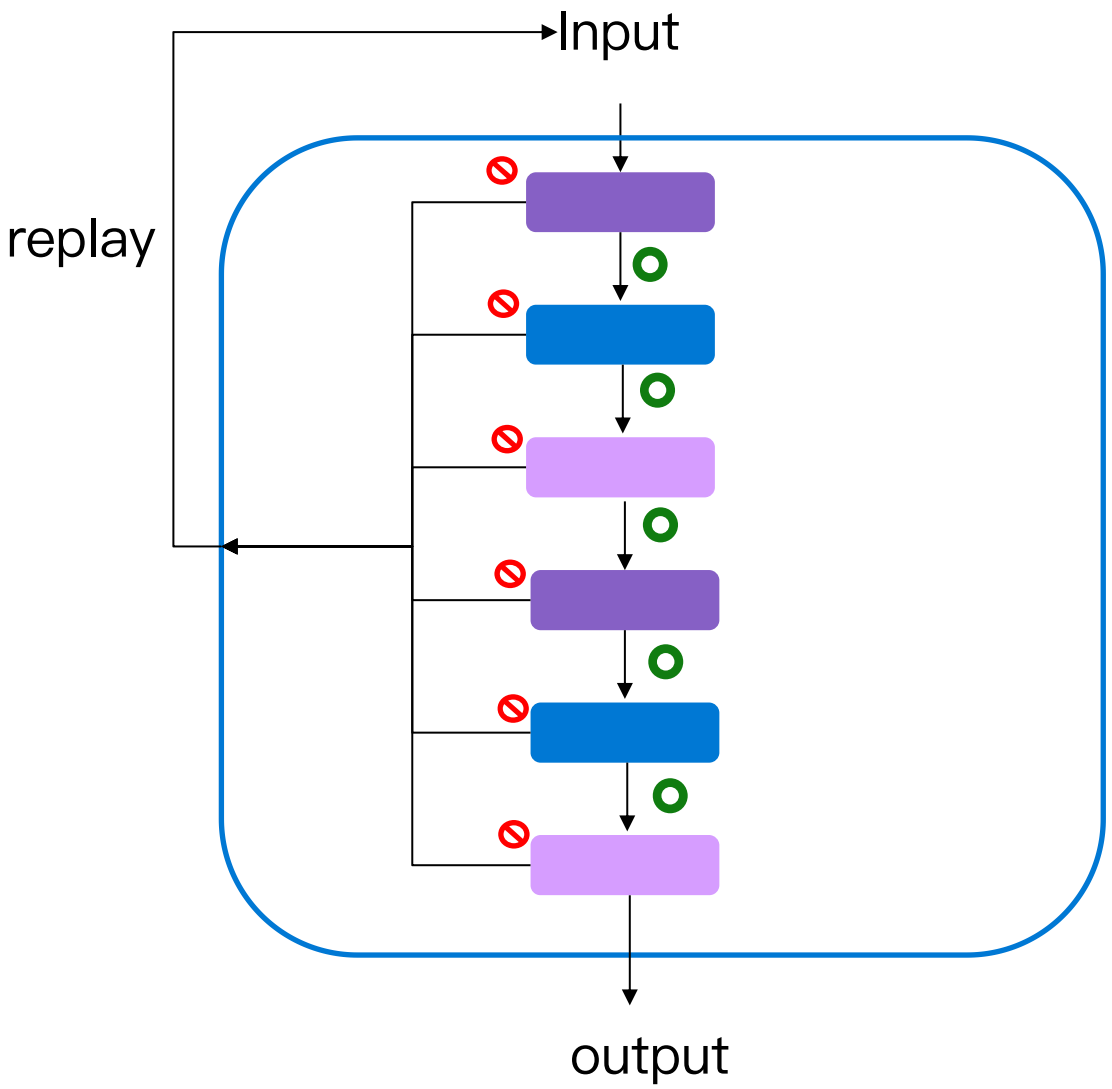
当多个步骤组合在一起时，如果组合的逻辑是确定的，那么组合执行的结果也是确定的吗?

如果执行一次的结果是确定的，那么再次/重复执行的结果也是确定的吗?

如果工作流程中的某些行为不是确定的怎么办?



确定性约束极大的简化了重放过程



可以从头开始回放整个工作流程

不需要关心如何保存和加载工作流程的状态。

在编写工作流业务逻辑时，不需要考虑重放之间的区别。

确定性是 Dapr Workflow 的灵魂

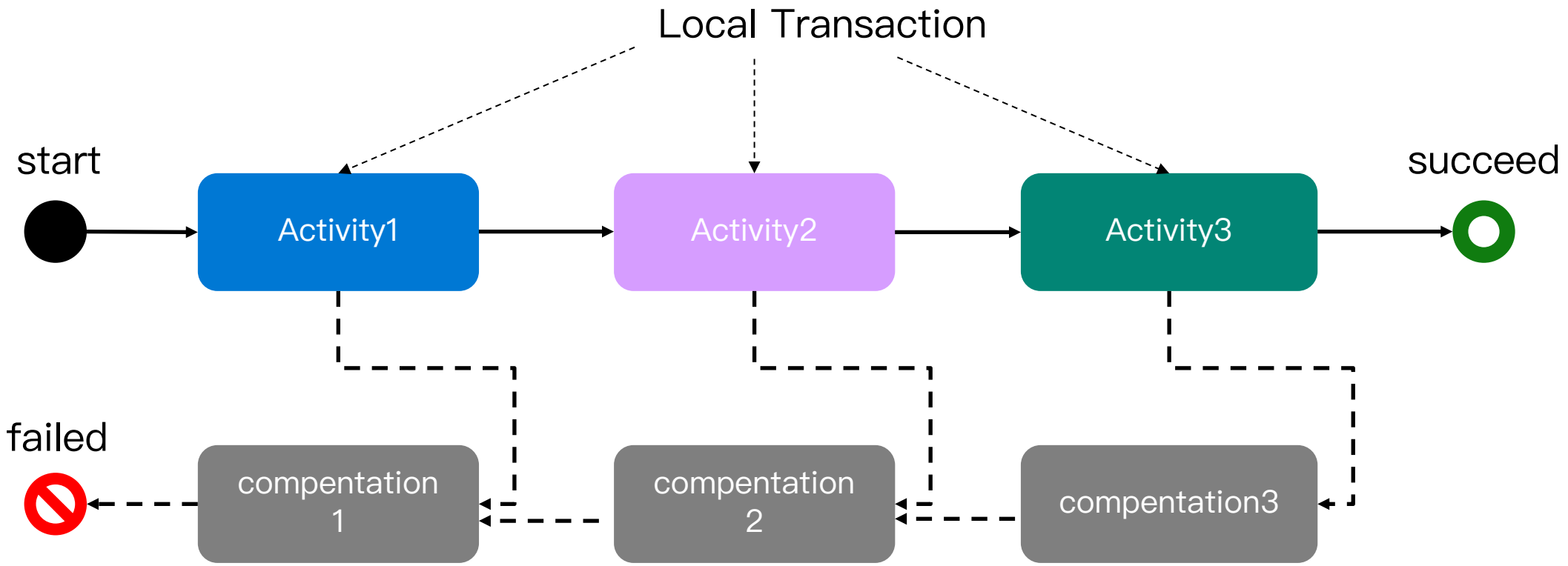
理解了确定性，也就理解了 Dapr 工作流

扩展：Dapr 的 Saga 支持

3

- Saga 模式介绍
- 如何在 Dapr workflow 中支持 Saga

什么是 Saga?



本地事务是 Saga 参与者执行工作的单位。Saga 中的每个操作都可以通过补偿事务回滚。
此外，Saga 模式保证所有操作都能成功完成，或者运行相应的补偿事务来撤销之前完成的工作。

Saga 模式的核心设计

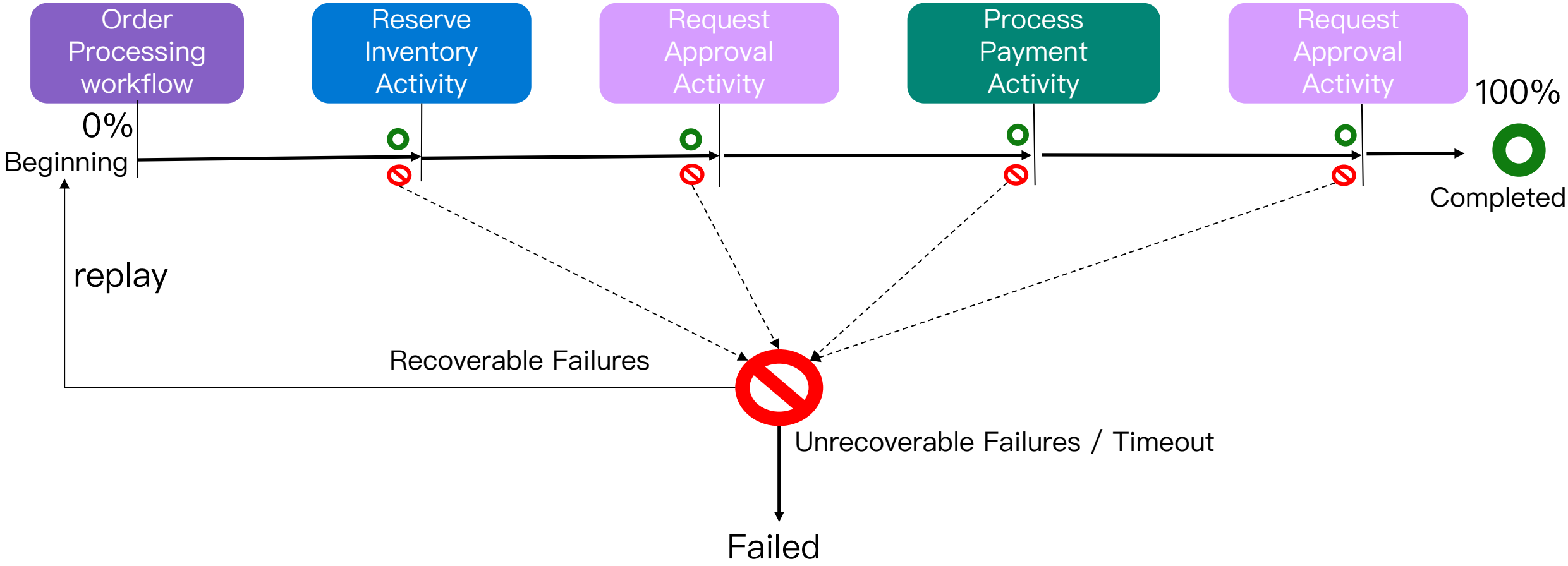
努力前行

- 保持状态
- 从上次失败处继续开始

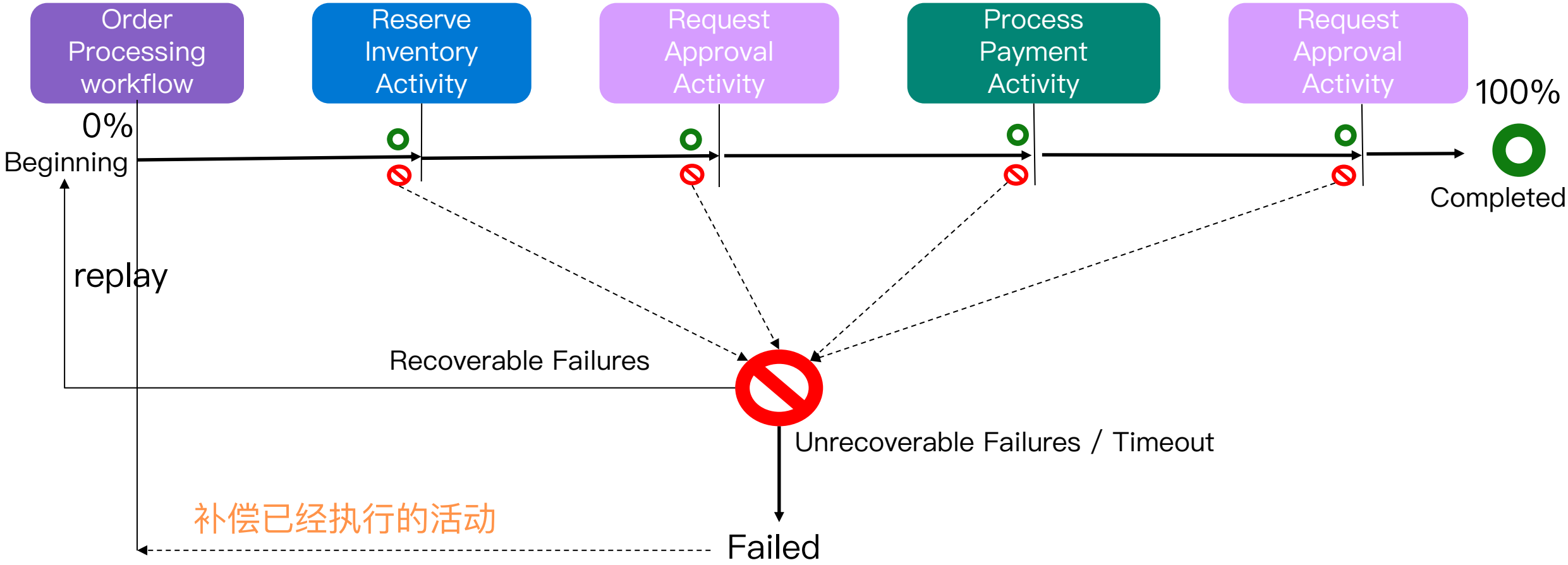
准备回退

- 准备补偿信息
- 必要时进行补偿

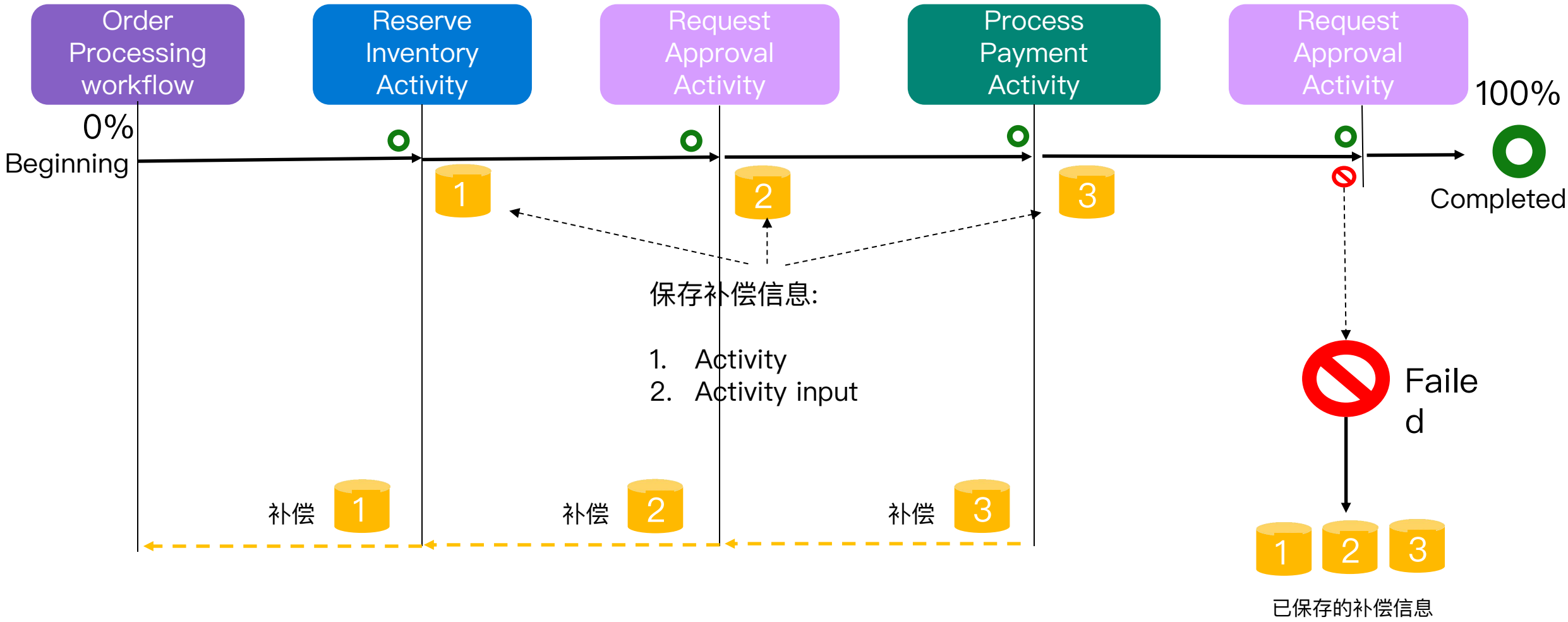
Dapr workflow: 努力前行



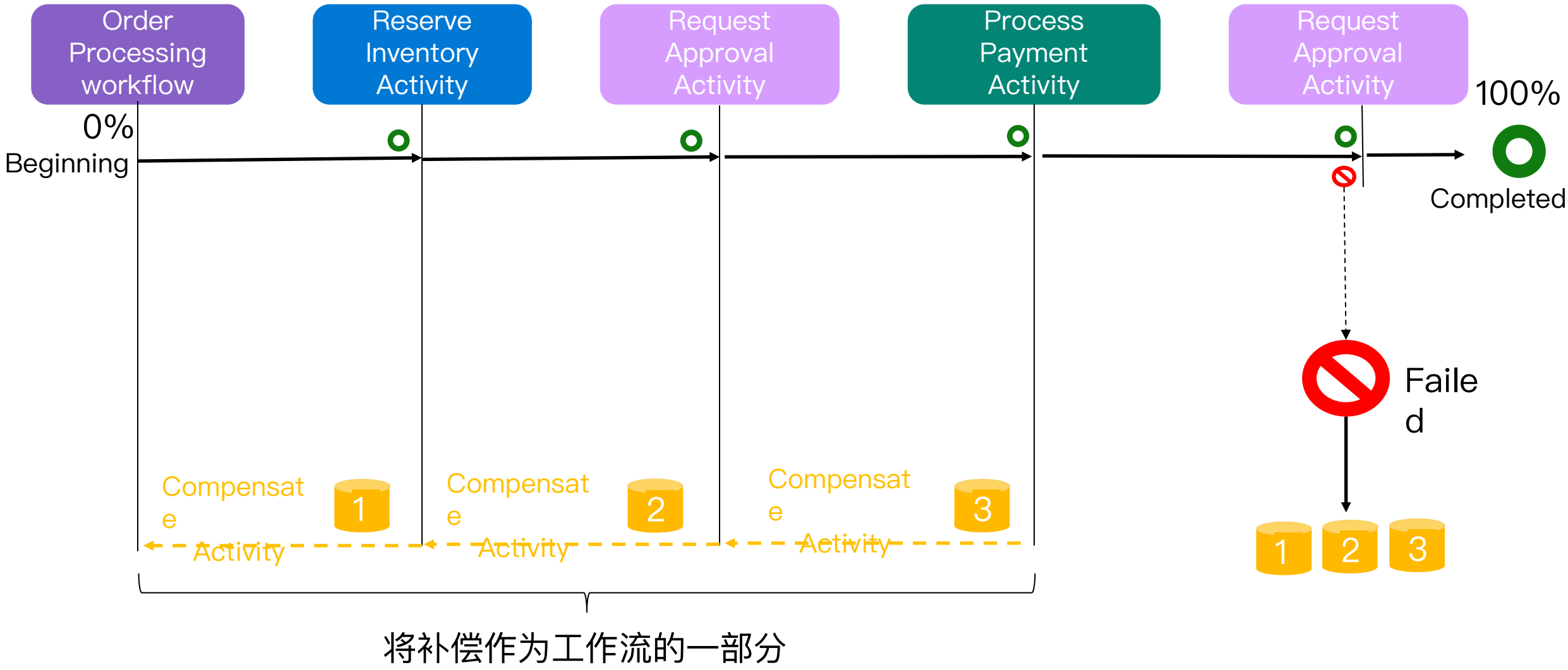
Dapr Saga：通过补偿进行回退



补偿的实现


Order Processing workflow
Reserve Inventory Activity
Request Approval Activity
Process Payment Activity
Request Approval Activity
0%
Beginning
100%
Completed
1
2
3
保存补偿信息:
1. Activity
2. Activity input
Faile
d
补偿 1
补偿 2
补偿 3
1
2
3
已保存的补偿信息

如何确保补偿操作的执行?



Talk is cheap, show me the code

Demo Time

Dapr workflow quickstart

https://docs.dapr.io/getting-started/quickstarts/workflow-quickstart/

Talk is cheap, show me the code

Demo Time

Dapr saga quickstart

https://github.com/dapr/quickstarts/pull/957


Microsoft
Make stateful application programed like stateless
Thank you