

# MoonBit Workflow 与并行 FFI Runtime 的规范性规格

第一部分定义 workflow 图、capability、所有权与分离安全执行的工程合同

朱哲皓

Luna-Flow

GitHub: KCN-judu

## 摘要

本文把一个 MoonBit workflow 与并行 FFI runtime 规定为关于 workflow 图表示、capability 协议、所有权转移和并发 buffer 安全的合同。语义核心不是重新定义 map 与 reduce 的基础函数意义，而是精确回答三个跨边界致命问题：MoonBit 如何构造纯 workflow 图，该图如何以 ABI 受约束的 descriptor 族跨越 FFI，以及 native runtime 如何在不产生数据竞争和生命周期错误的前提下执行任务、同步与 compute 节点。第一部分定义 workflow ADT、compute 子 IR、布局同构约束、线性所有权状态机，以及分离安全执行义务。第二部分给出 native scheduler 层、native compute lane 与 JavaScript 兼容面的实现化与符合性义务。第三部分把这些义务细化为实现层面的一致性检查，覆盖异常控制流、字节级布局校验、安全算子准入与 fail-closed 验证场景。英文文本是权威版本，中文文本是严格镜像。

**关键词**：MoonBit, workflow runtime, FFI, OpenMP, pthread, separation safety, linear ownership, ABI, engineering specification

## 第一部分：语义核心

### 规范性范围

本规格把一个 MoonBit workflow 并行库的 version-1 合同划分为三层：

- 构造纯 workflow 图的 MoonBit facade；
- 将 workflow 图与 compute 子计划下放为 ABI 受约束 payload / descriptor 的后端特化 FFI 边界；
- 分为 workflow scheduler 层与 compute lane 的 native runtime。

具有规范约束力的内容仅限于：

- MoonBit 侧 workflow 图的代数形状；
- 允许穿越 FFI 的值类别；
- zero-copy 或借用视图所要求的布局同构条件；
- MoonBit 堆和 C 可见堆之间的线性所有权状态转移；
- race-free chunk 调度与安全 workflow capability 使用所要求的分离逻辑义务；
- native 与 JavaScript 目标的 faithful realization 所必须满足的符合性义务。

调度启发式、队列策略、OpenMP 调优细节和局部优化策略均不属于规范性内容，除非它们影响外部可观察的所有权、布局或竞争安全合同。

### 记号与元层约定

本文使用如下记号:

- $H_M$  表示 MoonBit 堆,  $H_C$  表示 C / 宿主可见堆;
- $\Gamma$  表示跨 FFI 边界携带的所有权资源上下文;
- $b$  表示 buffer 身份,  $v$  表示 view,  $p$  表示 plan,  $d$  表示 runtime descriptor;
- $\text{mapsto}$  表示元层部分求值;
- $\text{Own}_M(b)$ 、 $\text{Borrow}_C(b)$ 、 $\text{Own}_C(b)$  表示所有权状态;
- $\Gamma \rightarrow \Gamma'$  表示一步所有权转移;
- $P * Q$  表示不相交堆片段上的分离合取;
- $\text{Interval}(i, j)$  表示半开区间  $[i, j)$ ;
- $\text{Chunks}(n, m)$  表示把  $[0, n)$  划分为  $m$  个子区间的分块;
- $\text{Den}(p)$  表示 plan 指称;
- $t \dashv p \mapsto q$  表示在目标  $t$  上将 MoonBit plan  $p$  下放为 payload  $q$ ;
- $\text{parse}(t, q) \mapsto d$  表示包装层将 payload 解析为 runtime descriptor  $d$ ;
- $\text{exec}(d) = y$  表示 runtime descriptor  $d$  执行得到可观察结果  $y$ 。

## 语义 Workflow 演算

facade 是纯 workflow 代数。它的构造子携带足够信息, 使下层可以解释任务、同步与 compute 工作, 而无需跨越函数值或运行时状态边界。

**workflow 优先规则。** version 1 只把 Workflow 视为顶层执行请求。历史上的 MapPlan、ReducePlan、ScanPlan 与 MapReducePlan 仍然是规范性内容, 但只作为 ComputeNode 中可携带的 compute 子 IR。

**执行入口规则。** version 1 同时准入 workflow 图的异步与阻塞执行入口。阻塞式 submit 在外延上定义为 `submit_async` 后接 `wait` 与资源销毁, 并且它的返回状态以最终 `WorkflowResult.status` 为唯一语义来源。

**barrier group 规则。** barrier group 由 workflow 结构推导, 而不是由显式参与者数字段给出。在 version 1 中, barrier group 的键固定为 (barrier capability id, node depth)。若某个导出 barrier group 的参与者数小于 2, 则它是非法 group, 必须以 `BARRIER_BROKEN` fail closed。

```
ScalarKind ::= Int
              | UInt
              | Int64
              | UInt64
              | Float
              | Double
```

图 1 标量家族。

```
Schedule ::= Static
            | Dynamic
            | Guided
```

图 2 调度类别。

```

RuntimeConfig ::= {
    threads : Nat ,
    chunk : Nat ,
    schedule : Schedule
}

```

图 3 运行时配置记录。

```

AccessMode ::= ReadOnly
              | WriteOnly
              | ReadWrite

```

图 4 边界访问模式。

```

BufferRef ::= ( BufId , ScalarKind , Len , AccessMode , LayoutWitness )

```

图 5 facade 层 buffer 引用。

```

OutRef ::= ( BufId , ScalarKind , Len , LayoutWitness )

```

图 6 facade 层输出引用。

```

LawClass ::= StrictAssoc
            | DeterministicTree

```

图 7 规约律类别。

```

MonoidLaw ::= ( LawClass , OpId , IdentityId , AssocWitness )

```

图 8 么半群描述子。

```

Plan ::= MapPlan ( RuntimeConfig , OpId , BufferRef , BufferRef )
        | ReducePlan ( RuntimeConfig , MonoidLaw , BufferRef , OutRef )
        | ScanPlan ( RuntimeConfig , MonoidLaw , BufferRef , BufferRef )
        | MapReducePlan ( RuntimeConfig , OpId , MonoidLaw , BufferRef , OutRef )

```

图 9 并行 Plan ADT。

version 1 禁止任意 MoonBit 函数值与任意 JavaScript 宿主函数进入 Plan。

**定义（构造子形状义务）.** 记  $\text{ShapeOK}(p)$  当且仅当：

- 对  $\text{MapPlan}(\text{cfg}, \text{op}, \text{src}, \text{dst})$ ,  $\text{Len}(\text{src}) = \text{Len}(\text{dst})$  且  $\text{dst}$  可写；
- 对  $\text{ReducePlan}(\text{cfg}, \text{mu}, \text{src}, \text{out})$ ,  $\text{Len}(\text{out}) = 1$ 、 $\text{ScalarKind}(\text{out}) = \text{ScalarKind}(\text{src})$ , 且  $\text{out}$  是 reduction 的唯一结果位置；
- 对  $\text{ScanPlan}(\text{cfg}, \text{mu}, \text{src}, \text{dst})$ ,  $\text{Len}(\text{src}) = \text{Len}(\text{dst})$ 、 $\text{ScalarKind}(\text{src}) = \text{ScalarKind}(\text{dst})$ , 且  $\text{dst}$  可写；
- 对  $\text{MapReducePlan}(\text{cfg}, \text{op}, \text{mu}, \text{src}, \text{out})$ ,  $\text{Len}(\text{out}) = 1$ 、 $\text{out}$  可写, 且  $\text{op}$  产生的映射元素类型与  $\text{mu}$  要求的 reduction 元素类型一致。

**定义（Plan 指称）.** 记  $\text{Den}(p)$  为由 plan 构造子唯一决定的可观察结果：

- $\text{Den}(\text{MapPlan}(\text{cfg}, \text{op}, \text{src}, \text{dst}))$  是输出 buffer  $\text{dst}$ , 其中索引  $i$  处的元素为  $\text{Apply}(\text{op}, \text{src}[i])$ ;
- $\text{Den}(\text{ReducePlan}(\text{cfg}, \text{mu}, \text{src}, \text{out}))$  是  $\text{out}[0]$  中的标量结果, 它由  $\text{mu}$  对  $\text{src}$  进行 fold 得到;
- $\text{Den}(\text{ScanPlan}(\text{cfg}, \text{mu}, \text{src}, \text{dst}))$  是 inclusive prefix 输出 buffer  $\text{dst}$ , 其中索引  $i$  处的元素等于在  $\text{mu}$  下对  $\text{src}[0 \dots i]$  的规约结果;

- $\text{Den}(\text{MapReducePlan}(\text{cfg}, \text{op}, \text{mu}, \text{src}, \text{out}))$  定义为  $\text{Den}(\text{ReducePlan}(\text{cfg}, \text{mu}, \text{upright}(\text{"MapTmp"}) (\text{op}, \text{src}), \text{out}))$ , 其中  $\text{MapTmp}(\text{op}, \text{src})$  是对  $\text{src}$  逐点应用  $\text{op}$  得到的逻辑映射序列。

**定义 (Plan 良构性)** . 记  $\text{PlanWF}(p)$  当且仅当:

- $\text{threads} > 0$ ;
- 只要请求 chunk 化并行执行, 就必须满足  $\text{chunk} > 0$ ;
- 空输入 map 与 scan 只有在其 shape 义务仍能唯一确定空结果 buffer 时才可接受;
- 空输入 reduce 只有在 IdentityId 能解析到该 MonoidLaw 的受审计 identity 元素时才可接受, 否则必须在 lowering 前拒绝;
- 每个 buffer 引用都携带显式 scalar kind、长度、access mode 和 layout witness;
- 每个 OutRef 都携带显式 scalar kind、长度和 layout witness;
- 每个 operator id 都能解析到一个受信任的已注册 kernel descriptor;
- 每个 monoid law 描述子都携带显式 law class、operator id、identity id, 以及 associativity 或 deterministic-tree witness id;
- $\text{ShapeOK}(p)$  成立。

**定义 (规约律可接受性)** . 记  $\text{LawOK}(\mu, \tau)$  当且仅当:

- 若  $\text{ClassOf}(\mu) = \text{StrictAssoc}$ , 则  $\text{AssocWitness}(\mu)$  表示元素类型  $\tau$  上的严格结合律; 且
- 若  $\text{ClassOf}(\mu) = \text{DeterministicTree}$ , 则  $\tau \in \{\text{Float}, \text{Double}\}$ , 并且  $\text{AssocWitness}(\mu)$  表示与规范化 descriptor 绑定的固定规约树 witness。

浮点 reduce 与 scan 只能通过 DeterministicTree 被准入; 本规格不把 IEEE 754 加法或乘法视为无约束的代数么半群。

**定义 (浮点符合性 profile)** . 记  $\text{FPPProfile}(t)$  为目标  $t$  声明的目标局部浮点环境, 包括 rounding mode、contraction policy、NaN 与 infinity 处理、subnormal 处理, 以及解释 DeterministicTree 结果所需的其他条件。

## ABI 布局同构

只有在 MoonBit 布局与 C 布局同构时, 才允许 zero-copy 借用或表示保持不变的 transfer。

**定义 (边界视图布局)** . 对活动 backend 目标上的 layout-bearing buffer view, 定义边界视图记录:

```
lv_view ::= {
    addr : UIntPtr ,
    len : UInt64 ,
    kind : UInt32 ,
    mode : UInt32 ,
    stride : UInt64
}
```

图 10 规范边界 view。

规范 boundary ABI 不是从未说明的机器字宽  $w$  推导出来的; 它必须由一个受审计的目标 profile  $\text{ABIPProfile}(t)$  选定。

**定义 (边界 ABI profile)** .  $\text{ABIPProfile}(t)$  必须声明目标  $t$  上 boundary record 所使用的唯一宽度、对齐与 endian 合同, 并给出 UIntPtr、UInt64 与 UInt32 的映射。

并要求在  $\text{ABIPProfile}(t)$  下满足下列布局方程:

$$\begin{aligned} \text{sizeof}(\text{lv\_view}) &= \text{LVSize}(\text{ABIPProfile}(t)) \wedge \text{alignof}(\text{lv\_view}) = \text{LVAlign}(\text{ABIPProfile}(t)) \\ \text{offset}(\text{addr}) &= 0 \wedge \text{offset}(\text{len}) = \text{OffLen}(\text{ABIPProfile}(t)) \wedge \text{offset}(\text{kind}) = \text{OffKind}(\text{ABIPProfile}(t)) \\ \text{offset}(\text{mode}) &= \text{OffMode}(\text{ABIPProfile}(t)) \wedge \text{offset}(\text{stride}) = \text{OffStride}(\text{ABIPProfile}(t)) \end{aligned}$$

**定义 (布局记录)** . 对任意标量或 buffer 元素类型  $\tau$ , 定义:

$$\text{Layout}(\tau) = (\text{size}(\tau), \text{align}(\tau), \text{stride}(\tau), \text{endian}(\tau))$$

**定义 (布局同构)** . 记  $\text{Iso}(\tau_M, \tau_C)$  当且仅当:

$$\text{Layout}(\tau_M) = \text{Layout}(\tau_C)$$

且两边在标量解释、元素宽度和连续遍历语义上完全一致。

**约束 (边界布局 Gate)** .

$$\text{Iso}_{\frac{\tau_M, \tau_C}{\text{LayoutOK}}}(\tau_M, \tau_C)$$

若  $\text{LayoutOK}$  失败, 则实现只能显式拷贝到一个符合布局的表示, 或直接拒绝请求。不得静默重解释布局。

$\text{LayoutOK}$  还要求活动目标为承载该 view 的路径声明一个受审计的  $\text{ABIPProfile}(t)$ 。实现不得依据偶然的宿主 ABI 兼容性替换记录布局。

## 线性所有权状态机

边界合同被建模为一个关于所有权资源上下文的有限状态转移系统。

**所有权状态**。对任意 buffer 身份  $b$ , 允许的状态只有  $\text{Own\_M}(b)$ 、 $\text{Borrow\_MR}(b)$ 、 $\text{Borrow\_C}(b)$ 、 $\text{Own\_C}(b)$  和  $\text{Returned\_M}(b)$ 。

$\text{Returned\_M}(b)$  是一个可观察的“返还完成”中间态: C 可见 authority 已经放弃, 但在 rehydration 成功前, MoonBit 的可变 authority 尚未恢复。

**转移规则**。

$$\begin{aligned} & \frac{\Gamma \vdash \text{Own\_M}(b) \wedge \text{LayoutOK}(\tau_M, \tau_C) \wedge \text{Live}(b) \wedge \text{ReadOnly}(q)}{\Gamma \rightarrow (\Gamma - \{\text{Own\_M}(b)\}) \cup \{\text{Borrow\_MR}(b), \text{Borrow\_C}(b)\}} \quad (\text{Borrow-Step}) \\ & \frac{\Gamma \vdash \text{Own\_M}(b) \wedge \text{LayoutOK}(\tau_M, \tau_C) \wedge \text{Exclusive}(b) \wedge \text{TransferOK}(q)}{\Gamma \rightarrow (\Gamma - \{\text{Own\_M}(b)\}) \cup \{\text{Own\_C}(b)\}} \quad (\text{Transfer-Step}) \\ & \frac{\Gamma \vdash \text{Own\_C}(b) \wedge \text{Received}(b) \wedge \text{Rebind}(b)}{\Gamma \rightarrow (\Gamma - \{\text{Own\_C}(b)\}) \cup \{\text{Returned\_M}(b)\}} \quad (\text{Return-Step}) \\ & \frac{\Gamma \vdash \text{Returned\_M}(b) \wedge \text{RehydrateOK}(b)}{\Gamma \rightarrow (\Gamma - \{\text{Returned\_M}(b)\}) \cup \{\text{Own\_M}(b)\}} \quad (\text{Rehydrate-Step}) \\ & \frac{\Gamma \vdash \text{Borrow\_MR}(b) \wedge \text{Borrow\_C}(b) \wedge \text{BorrowDone}(b)}{\Gamma \rightarrow (\Gamma - \{\text{Borrow\_MR}(b), \text{Borrow\_C}(b)\}) \cup \{\text{Own\_M}(b)\}} \quad (\text{Discharge-Step}) \end{aligned}$$

图 11 线性所有权转移。

**线性条件。** 任何可达上下文都不能同时包含同一  $b$  的  $\text{Own}_M(b)$  与  $\text{Own}_C(b)$ ；任何活动中的  $\text{Borrow}_C(b)$  都会阻止  $\text{MoonBit}$  侧突变和  $\text{free}$ ，直到  $\text{borrow token}$  通过  $\text{Discharge-Step}$  被释放。

$\text{version 1}$  中的  $\text{Borrow}$  只表示只读借用。任何可写边界  $\text{view}$  都必须使用  $\text{Transfer-Step}$ ；本规格不定义可写借用的状态转移。

## 边界承载 Payload

FFI 边界传输的是  $\text{Plan}$  与布局检查后的  $\text{view}$ ，而不是任意对象图。

按构造子的  $\text{Payload}$  骨架。

```
typedef struct { PlanTag plan_tag; CfgBits cfg_bits; ViewVec views; OpBits op_bits; } Payload_Map;
typedef struct { PlanTag plan_tag; CfgBits cfg_bits; ViewVec views; LawBits law_bits; } Payload_Reduce;
typedef struct { PlanTag plan_tag; CfgBits cfg_bits; ViewVec views; LawBits law_bits; } Payload_Scan;
typedef struct { PlanTag plan_tag; CfgBits cfg_bits; ViewVec views; OpBits op_bits; LawBits law_bits; }
Payload_MapReduce;
```

$\text{Payload\_Map}$  不携带  $\text{LawBits}$ 。  $\text{Payload\_Reduce}$  与  $\text{Payload\_Scan}$  不携带  $\text{OpBits}$ 。  $\text{Payload\_MapReduce}$  同时携带二者。任何构造子都不得依赖  $\text{absent}$  字段、 $\text{target-defined}$  哑元，或被忽略的  $\text{payload}$  字节。

每个  $\text{view}$  项都携带：

- $\text{buffer}$  身份或边界  $\text{view handle}$ ；
- $\text{scalar kind}$  标签；
- 长度；
- $\text{access mode}$ ；
- $\text{layout witness}$  或  $\text{layout class}$  标签；
- $\text{ownership mode}$  ( $\text{Borrow}$  或  $\text{Transfer}$ )。

在  $\text{version 1}$  中， $\text{ownership mode Borrow}$  只对只读  $\text{view}$  可接受。可写边界输出必须以  $\text{ownership mode Transfer}$  下放。

**边界可接受性。**  $\text{Adm}(t, q)$  仅当以下条件都成立：

- $\text{payload}$  标签属于目标  $t$  的  $\text{payload}$  语法；
- 每个承载  $\text{view}$  都满足  $\text{LayoutOK}$ ；
- $q$  请求的每个  $\text{ownership transition}$  都在线性状态机中合法；
- 每个以  $\text{Borrow}$  承载的  $\text{view}$  都是  $\text{ReadOnly}$ ；
- 每个  $\text{operator}$  或  $\text{monoid id}$  都能在目标  $t$  上解析到一个被准入的受信任  $\text{descriptor}$  或  $\text{law}$ 。

## 包装层解析与描述子形成

$\text{C wrapper}$  或  $\text{JS/N-API}$  边界是从  $\text{payload}$  语法到  $\text{runtime descriptor}$  的解释器。

按构造子的  $\text{Runtime descriptor}$ 。

```
typedef struct { PlanTag plan_tag; CfgNorm cfg_norm; ChunkPolicy chunk_policy; BufSliceVec buf_slices; KernelDesc
kernel_desc; } Descriptor_Map;
typedef struct { PlanTag plan_tag; CfgNorm cfg_norm; ChunkPolicy chunk_policy; BufSliceVec buf_slices; KernelDesc
kernel_desc; LawHandle law_handle; } Descriptor_Reduce;
typedef struct { PlanTag plan_tag; CfgNorm cfg_norm; ChunkPolicy chunk_policy; BufSliceVec buf_slices; KernelDesc
kernel_desc; LawHandle law_handle; } Descriptor_Scan;
typedef struct { PlanTag plan_tag; CfgNorm cfg_norm; ChunkPolicy chunk_policy; BufSliceVec buf_slices; KernelDesc
kernel_desc; LawHandle law_handle; } Descriptor_MapReduce;
```

Descriptor\_Map 不携带 LawHandle。 Descriptor\_Reduce、 Descriptor\_Scan 和 Descriptor\_MapReduce 各携带一个已解析的 LawHandle。

## Kernel descriptor 与 registry。

```
typedef struct {
    KernelId kernel_id;
    FnPtr fn_ptr;
    PurityWitness purity_witness;
    FootprintWitness footprint_witness;
    LayoutContract layout_contract;
    ArityMeta arity_meta;
    LawMeta law_meta;
    TargetMeta target_meta;
} KernelDesc;
```

KernelRegistry(t) 是目标 t 上从 KernelId 到 KernelDesc 的静态受审计映射。 KernelId 是 plan payload 中唯一允许出现的 kernel 级标识符； FnPtr 仅属于实现层， 绝不跨越 MoonBit plan 边界。

## 解析关系。

记 ParseCoreMap( $t, q, p, c, V, k$ ) 当且仅当：

- DecodePlan( $q$ ) =  $p$ ；
- NormCfg( $q$ ) =  $c$ ；
- NormViews( $q$ ) =  $V$ ；
- ResolveKernel(KernelRegistry( $t$ ),  $q$ ) =  $k$ ； 且
- plan\_tag( $p$ ) = MapPlan。

记 ParseCoreLaw( $t, q, p, c, V, k, \mu$ ) 当且仅当：

- DecodePlan( $q$ ) =  $p$ ；
- NormCfg( $q$ ) =  $c$ ；
- NormViews( $q$ ) =  $V$ ；
- ResolveKernel(KernelRegistry( $t$ ),  $q$ ) =  $k$ ； 以及
- ResolveLaw( $q$ ) =  $\mu$ 。

记 ViewsOK( $V$ ) 当且仅当每个  $v$  in  $V$  都同时满足 LayoutOK( $v$ ) 与 OwnerOK( $v$ )。

记 ParseReadyMap( $t, p, V, k$ ) 当且仅当 KernelOK( $k, p, V, \text{emptyset}, t$ )。

记 ParseReadyLaw( $t, p, V, k, \mu$ ) 当且仅当 KernelOK( $k, p, V, \mu, t$ )。

$$\begin{array}{c}
 \frac{\text{ParseCoreMap}(t, q, p, c, V, k) \wedge \text{ViewsOK}(V) \wedge \text{ParseReadyMap}(t, p, V, k)}{\text{parse}(t, q) \mapsto \text{Descriptor\_Map}(p, c, \text{ChunkPolicy}(c), V, k)} \\
 \frac{\text{ParseCoreLaw}(t, q, p, c, V, k, \mu) \wedge \text{ViewsOK}(V) \wedge \text{ParseReadyLaw}(t, p, V, k, \mu) \wedge \text{plan\_tag}(p) = \text{ReducePlan}}{\text{parse}(t, q) \mapsto \text{Descriptor\_Reduce}(p, c, \text{ChunkPolicy}(c), V, k, \mu)} \\
 \frac{\text{ParseCoreLaw}(t, q, p, c, V, k, \mu) \wedge \text{ViewsOK}(V) \wedge \text{ParseReadyLaw}(t, p, V, k, \mu) \wedge \text{plan\_tag}(p) = \text{ScanPlan}}{\text{parse}(t, q) \mapsto \text{Descriptor\_Scan}(p, c, \text{ChunkPolicy}(c), V, k, \mu)} \\
 \frac{\text{ParseCoreLaw}(t, q, p, c, V, k, \mu) \wedge \text{ViewsOK}(V) \wedge \text{ParseReadyLaw}(t, p, V, k, \mu) \wedge \text{plan\_tag}(p) = \text{MapReducePlan}}{\text{parse}(t, q) \mapsto \text{Descriptor\_MapReduce}(p, c, \text{ChunkPolicy}(c), V, k, \mu)}
 \end{array}$$

图 12 包装层解析关系。

**描述子保真规则。** 产出的 descriptor 必须保持 plan 的构造子类别、每个 buffer 的 scalar kind 与 layout class、每个边界 view 所附带的 ownership mode、plan 所选择的 KernelId

与其从受信任 registry 中解析出的审计 descriptor、plan 构造子要求的 shape 义务，以及定义 scan 前缀语义和 map-reduce lowering 的那组规范选择。

此外：

- Descriptor\_Map 不保留任何 LawHandle；
- Descriptor\_Reduce 与 Descriptor\_MapReduce 各保留一个 reduction law handle；且
- Descriptor\_Scan 保留一个 scan law handle 以及其 carry correctness 所依赖的顺序 join contract。

## Fork-Join 操作语义

我们给出一个关于配置  $\text{Conf}(C, H, S)$  的标记小步语义，其中  $H$  是宿主堆， $s$  是 worker 本地 store。第三部分会把这台抽象机器细化为适用于物理 C/OpenMP 实现的显式 fail-closed 错误发布合同。

**Worker 语法。**

```
Worker ::= Read ( Addr )
        | Write ( Addr , Val )
        | MapStep ( Op , Addr , Addr )
        | ScanStep ( Law , Addr , Addr , State )
        | Seq ( Worker , Worker )
        | Done ( Val )
```

图 13 worker 语法。

**Read 规则。**

$$\frac{H(a) = v}{\text{Conf}(\text{Read}(a), H, S) \rightarrow \text{Conf}(\text{Done}(v), H, S)}$$

图 14 读步骤。

**Write 规则。**

$$\frac{\text{Writable}(a, S)}{\text{Conf}(\text{Write}(a, v), H, S) \rightarrow \text{Conf}(\text{Done}(v), H[a := v], S)}$$

图 15 写步骤。

**Map 规则。**

$$\frac{H(a_s) = x \wedge \text{Writable}(a_d, S) \wedge \text{Apply}(\text{op}, x) = y}{\text{Conf}(\text{MapStep}(\text{op}, a_s, a_d), H, S) \rightarrow \text{Conf}(\text{Done}(y), H[a_d := y], S)}$$

代码 1 映射步骤。

**Scan 规则。**

$$\frac{H(a_s) = x \wedge \text{Writable}(a_d, S) \wedge \text{Combine}(\mu, s, x) = s'}{\text{Conf}(\text{ScanStep}(\mu, a_s, a_d, s), H, S) \rightarrow \text{Conf}(\text{Done}(s'), H[a_d := s'], S)}$$

图 16 scan 步骤。

**Sequence 规则。**

$$\begin{array}{c}
\text{Conf}(C_1, H, S) \rightarrow \text{Conf}(C_1', H', S') \\
\hline
\text{Conf}(\text{Seq}(C_1, C_2), H, S) \rightarrow \text{Conf}(\text{Seq}(C_1', C_2), H', S') \\
\text{Conf}(C_2, H, S) \rightarrow \text{Conf}(C_2', H', S') \\
\hline
\text{Conf}(\text{Seq}(\text{Done}(v), C_2), H, S) \rightarrow \text{Conf}(C_2', H', S')
\end{array}$$

图 17 顺序组合步骤。

**Footprint 合同。**  $R(C)$  是  $C$  单步读取的地址集合,  $W(C)$  是  $C$  单步写入的地址集合,  $FP(C) = R(C) \cup W(C)$ 。

**Descriptor 元数据区域。** 对每个解析后的 descriptor  $d$ , 定义  $\text{MetaOf}(d)$  为只读元数据区域, 其中存放规范化的 runtime configuration、operator handle 及其参数、law handle, 以及 spawned workers 所需的每计划常量。

**Chunk 分块合同。**  $\text{Chunks}(n, k) = [I_0, \dots, I_{k-1}]$  只有在区间覆盖  $[0, n]$  且两两不相交时才成立。

**Worker 实例化合同。** 若 worker  $C_i$  由切片  $I_i$  实例化, 则  $FP(C_i)$  必须落在  $\text{AddrOf}(I_i) \cup \text{MetaOf}(d)$  之内, 且  $\text{MetaOf}(d)$  必须保持只读。

**顺序 join-state 关系。** 记  $\text{JoinState}(d, \sigma)$  为 worker 完成之后使用的构造子特化顺序状态。对  $\text{MapPlan}$ ,  $\text{JoinState}$  为空; 对  $\text{ReducePlan}$  与  $\text{MapReducePlan}$ , 它携带一个 fold accumulator; 对  $\text{ScanPlan}$ , 它携带从左到右的 chunk summaries 与 prefix carries。

**Fork 与 Join。**

$$\begin{array}{c}
\text{Chunks}(n, k) = [I_0, \dots, I_{k-1}] \wedge \forall i. \text{Spawn}(d, I_i) = C_i \\
\hline
\text{Conf}(\text{Fork}(d), H, S) \rightarrow \text{Conf}(\text{Par}([C_0, \dots, C_{k-1}]), H, S) \\
\text{plan\_tag}(d) = \text{MapPlan} \wedge \forall i. C_i = \text{Done}(y_i) \\
\hline
\text{Conf}(\text{Par}([C_0, \dots, C_{k-1}]), H, S) \rightarrow \text{Conf}(\text{Done}(\text{MapResult}(d)), H, S) \\
\text{plan\_tag}(d) \in \{\text{ReducePlan}, \text{MapReducePlan}\} \wedge \forall i. C_i = \text{Done}(y_i) \\
\hline
\text{Conf}(\text{Par}([C_0, \dots, C_{k-1}]), H, S) \rightarrow \text{Conf}(\text{Done}(\text{Fold}(\mu, \text{TreeOf}(d), [y_0, \dots, y_{k-1}])), H, S) \\
\text{plan\_tag}(d) = \text{ScanPlan} \wedge \forall i. C_i = \text{Done}(y_i) \wedge \text{JoinScan}(d, [y_0, \dots, y_{k-1}]) = \sigma \\
\hline
\text{Conf}(\text{Par}([C_0, \dots, C_{k-1}]), H, S) \rightarrow \text{Conf}(\text{Done}(\text{ScanResult}(d, \sigma)), H, S)
\end{array}$$

图 18 fork 与 join 步骤。

**Scan 与 map-reduce 的实现化规则。** 若  $\text{plan\_tag}(d) = \text{ScanPlan}$ , 则  $\text{Spawn}(d, I_i)$  生成的 worker 只能产出 chunk-local scan 输出和每个 chunk 的一个 summary。  $\text{JoinScan}(d, \dots)$  必须按从左到右的顺序组合这些 summary, 并据此导出  $\text{Den}(\text{ScanPlan}(\dots))$  所要求的 inclusive prefix 结果。若  $\text{plan\_tag}(d) =$

$\text{MapReducePlan}$ , 则 descriptor normalization 必须保持  $\text{Den}(\text{MapReducePlan}(\dots))$  使用的“先 map 再 reduce”的逻辑 lowering; 只有在可观察结果不变时, 实现才可以对物理执行做融合。

对  $\text{ScanPlan}$ , carry correctness 依赖  $\text{JoinScan}$  这一顺序 join contract; 它既不编码为可写 metadata, 也不在 version 1 中授权单独的 audited scratch region。

**浮点组合规则。** 若  $\text{ClassOf}(\mu) = \text{DeterministicTree}$ , 则  $\text{TreeOf}(d)$  属于 descriptor normalization 的一部分, 并唯一决定 split-combine 顺序。 worker 调度上的重排不会改变由  $d$  选定的组合树。该规则保证在单一目标局部  $\text{FPPProfile}(t)$  下重复执行的结果稳定; 跨目标相等只在被比较目标声明了兼容的受审计浮点 profile 时才被要求。

**并行侧条件。** 若  $\text{Conf}(C_i, H, S) \rightarrow \text{Conf}(C_i', H', S')$ , 则提升到  $\text{Par}(\dots)$  的步骤只在  $W(C_i) \cap FP(C_j) = \text{emptyset}$  对所有  $j \neq i$  都成立时才可接受。

该并行侧条件不适用于 JoinState，因为 JoinState 只在 worker 并行阶段结束后才被求值。

## JS ArrayBuffer 边界合同

JavaScript 边界被建模为宿主视图上的目标特化转移系统。

JS 视图语法。

JSView ::= ( ArrayBufferKind , Offset , Len , ScalarKind , AccessMode , Share )

图 19 JavaScript 边界视图。

边界规则。

$$\frac{\text{Kind}(v) = \text{SharedArrayBuffer} \wedge \text{Access}(v) = \text{ReadOnly} \wedge \text{LayoutOK}(v) \wedge \text{AtomicOK}(v)}{(\text{Own\_M}(b), v) \rightarrow (\text{Borrow\_MR}(b) * \text{Borrow\_C}(b), v)}$$

$$\frac{\text{Kind}(v) = \text{ArrayBuffer} \wedge \text{CopyBytes}(b) = b' \wedge \text{CopyOwner}(b') = \text{HostRuntime}}{(\text{Own\_M}(b), v) \rightarrow (\text{Own\_M}(b) * \text{Own\_C}(b'), v)}$$

$$\frac{\text{TransferOK}(v) \wedge \text{Tombstone}(v)}{(\text{Own\_M}(b), v) \rightarrow (\text{Own\_C}(b), v)}$$

图 20 JS 边界上的 borrow、copy 与 transfer。

**锁定侧条件。** 共享借用的每个推导都携带宿主侧锁义务：若 Share 中不包含相应原子纪律，则共享区域上的写入不可接受。

**共享快照义务。** 一条被准入的 SharedArrayBuffer 借用必须针对只读输入区域的一个逻辑快照来解释。实现可以通过复制、冻结视图，或其他在可观察行为上等价的机制实现该快照；但一旦执行被接受，它就必须对被快照时刻的输入保持 Den(p)，而不是对后续宿主写入保持 Den(p)。

若快照通过复制实现，该复制在规范层不可观察：它不会分配新的规范 buffer identity，不会改变 ownership fidelity，也不会把该被准入路径从 borrow 重分类为 copy。

**JS 生命周期义务。** 对 copy 规则，宿主 runtime 持有 b' 并且必须恰好释放一次，可通过显式 release 或 finalizer 完成。对 transfer 规则，宿主对象在 transfer 准入后立即进入 tombstone 状态；explicit return 优先于 finalizer 清理，而 finalizer 清理优先于 fault 情况下的泄漏式处理。若 transfer 之后 runtime 准入或 worker 执行 fail closed，则目标必须为仍由 C 可见侧持有的 buffer 发布一条确定性的清理路径。

## 分离与数据竞争检查

空间分离谓词。

$$\text{SepChunks}(b, [I_0, \dots, I_{k-1}]) = \forall i \neq j. \text{AddrOf}(I_i) \cap \text{AddrOf}(I_j) = \emptyset$$

并行可接受性。

$$\text{ParOK}([C_0, \dots, C_{k-1}]) = \forall i \neq j. W(C_i) \cap \text{FP}(C_j) = \emptyset$$

**定理（数据竞争自由）。** 若

1. KernelSafe(d) 成立，即已解析的受审计 kernel descriptor 满足 PurityWitness、FootprintWitness、LayoutContract 与 LawMeta；
2. SepChunks(b, [I\_0, ..., I\_{(k-1)}]),

3. 每个 worker  $c_i$  都由  $I_i$  生成,
4.  $FP(C_i) \text{ subset.eq } \text{AddrOf}(I_i) \text{ union } \text{MetaOf}(d)$  对所有  $i$  成立, 且
5.  $\text{MetaOf}(d)$  是只读的,

则每个可达配置  $\text{Conf}(\text{Par}([C_0, \dots, C_{(k-1)}]), H, S)$  都满足  $\text{ParOK}([C_0, \dots, C_{(k-1)}])$ 。

**证明。** 对推导长度归纳。基例由 chunk-spawn 不变式和 SepChunks 的定义直接得到。前提 (1) 使该定理在“受审计 kernel 可接受性”假设下闭合, 从而 worker footprint 的可推导性仅依赖 plan 数据。归纳步中, 每次只有一个 worker  $c_i$  进行小步。由前提 (4),  $c_i$  的写入落在  $\text{AddrOf}(I_i)$  中; 对于任意  $j \neq i$ , 前提 (4) 给出  $FP(C_j) \text{ subset.eq } \text{AddrOf}(I_j) \text{ union } \text{MetaOf}(d)$ 。前提 (2) 保证  $\text{AddrOf}(I_i)$  与  $\text{AddrOf}(I_j)$  不相交, 前提 (5) 禁止写入  $\text{MetaOf}(d)$ 。因此  $W(C_i) \text{ inter } FP(C_j) = \text{emptyset}$  对所有  $j \neq i$  仍成立, 故并行配置保持 ParOK。

**推论 (所有权安全)。** 若  $\text{Borrow}_C(b)$  活动, MoonBit 不能导出释放或突变  $b$  的步骤; 若  $\text{Own}_M(b) \rightarrow \text{Own}_C(b)$  已经发生, 则 MoonBit 不能在无显式 Return-Step 与后续 Rehydrate-Step 的情况下恢复对  $b$  的可变 authority; 若  $\text{Borrow}_M(b) * \text{Borrow}_C(b)$  活动, 则  $\text{Own}_M(b)$  只能通过 Discharge-Step 恢复。

## 第二部分：实现化与符合性

### Native 实现化

native 实现化路径为:

MoonBit facade  $\rightarrow$  workflow descriptor  $\rightarrow$  pthread scheduler runtime  $\rightarrow$  compute lane  $\rightarrow$  OpenMP compute runtime

一个符合性的 native 实现必须:

- 只下放第一部分允许的 workflow 图;
- 对每条被准入的边界路径保持一个受审计的  $\text{ABIPProfile}(t)$ , 或在布局不同时显式执行拷贝;
- 按线性所有权状态机实现只读 borrow 与 transfer 边;
- 只在满足第一部分分离合同的前提下调度可写 chunk;
- 维护一个处理任务、同步与 capability 状态迁移的 workflow scheduler 层;
- 让 compute 节点经由专用 compute lane 执行, 而不是在普通 scheduler worker 中直接递归开 OpenMP team;
- 暴露阻塞式 workflow submit, 并保证其返回状态在定义上等于底层异步 workflow result 的最终 status。

### JavaScript 实现化

JavaScript 实现化路径为:

MoonBit facade  $\rightarrow$  MoonBit JS FFI  $\rightarrow$  JS wrapper  $\rightarrow$  N-API addon  $\rightarrow$  runtime descriptor  $\rightarrow$  C runtime

一个符合性的 JavaScript 实现必须:

- 把 typed-array 兼容视图下放为携带显式元素种类标签的 payload view;
- 拒绝所有不满足布局与所有权义务的宿主对象;
- 与 native 实现保持相同的 workflow / compute 指称和 split-combine 合同; 只有当双方声明兼容的  $\text{FPProfile}$  时, 才要求更强的跨目标相等性;
- 对被准入的 SharedArrayBuffer 借用, 以逻辑快照的方式保证指称语义;
- 在 runtime 执行开始前, 为 copy 与 transfer 路径定义 owner、tombstone、finalizer 与 explicit-return 行为;

- 在 runtime 执行开始之前完成 borrow 或 transfer 纪律检查。

## Facade 义务

一个符合性的 facade 必须:

- 只暴露纯 workflow 构造子、compute 子计划构造子和显式执行入口;
- 拒绝把任意函数值环境或任意宿主函数编码进 workflow 图或 compute 子计划;
- 为每个导出的边界 view 绑定显式 ownership mode 与 access mode;
- 拒绝任何试图通过 ownership mode Borrow 进行可写 lowering 的构造路径;
- 暴露与真实目标支持在外延上完全一致的 capability 查询。

## Runtime 义务

一个符合性的 runtime 必须:

- 只通过第一部分定义的 workflow / compute 构造子类别解释 descriptor;
- 拒绝任何把可写 access 与 ownership mode Borrow 组合在一起的 descriptor;
- 对所有不满足分离或 side condition 的 descriptor 进行执行前拒绝;
- 保持每个承载 view 所附带的 ownership mode;
- 将失败提升为显式状态类别, 而不是返回部分成功;
- 禁止异常逃逸、panic 传播或栈展开跨越 FFI 边界;
- 在 worker 执行开始前, 对非同构视图执行确定性的布局降级或拒绝;
- 把“同步条件暂未满足”解释为 blocking 状态, 而不是立即失败;
- 当遇到 descriptor 级允许、但本版本尚未真实执行的 runtime 类别时 fail closed, 包括 RwLock、Semaphore 与 OpaqueCapability。

## 不支持表面

以下内容不属于 version-1 合同:

- 任意 MoonBit 函数值在 foreign worker 线程上的执行;
- 任意 JavaScript 宿主函数在 native worker 执行期间的运行;
- 对非同构 ABI 记录或数组进行隐式布局重解释;
- 不经过边界状态转移的隐式所有权变更;
- 无法通过分离与 split-combine 约束验证的 chunk 调度;
- 在普通 workflow scheduler worker 内递归开 OpenMP team 来执行 compute 节点;
- 默默接受 RwLock、Semaphore 或 OpaqueCapability 作为可执行 native workflow 原语。

## 符合性

一个实现只有在满足以下条件时才可以声称符合本规格:

- 它的 MoonBit-facing API 只构造第一部分允许的 workflow 图与 compute 子 IR;
- 它的边界 payload 保持显式 layout witness 与 ownership witness;
- 它的 wrapper 满足解析保真与所有权保真;
- 它的 scheduler/runtime 满足 workflow capability 安全、chunk 正交性和 split-combine 健全性;
- 所有非法、非同构或生命周期不安全请求都必须在执行前 fail closed。

## 符合性传递

**忠实实现。** 只有当实现满足以下条件时，才可称为忠实的：

- 每个被接受 payload 都是某个良构 workflow 图的 lowering；
- 每个 borrow / transfer 边都遵守线性所有权状态机；
- 每个可写 chunk 调度都满足 SepChunks；
- 每个 reduction / scan combine 路径都由有效 monoid 描述子支撑；且
- 每条构造子特化的 payload、workflow descriptor、compute descriptor 与 join 路径都保持 第一部分准入的构造子类别。

**符合性规则。** 如果忠实实现接受某个 workflow  $w$ ，则执行  $w$  的可观察结果和所有权轨迹必须留在本规格范围内。如果它拒绝  $w$ ，则该拒绝必须是本规格允许的 fail-closed 结果。对浮点 compute 节点，“可观察结果”的含义是：在单一已准入目标 profile 下重复执行结果稳定；跨目标相等只在显式兼容的受审计浮点 profile 之间被要求。

## 验证场景

1. 提交一个 workflow，其中某个 compute 节点携带源视图与目标视图布局不同构的请求。预期结果：显式拷贝到符合布局的表示，或直接 fail-closed 拒绝。
2. 提交一个包含只读 borrow 路径的 workflow，并在 borrow 释放前尝试从 MoonBit 侧突变该 buffer。预期结果：该 mutation 被拒绝或被 borrow 纪律阻塞。
3. 提交一个 workflow，其中某个 compute 节点要求的可写 chunk 调度发生重叠。预期结果：descriptor 形成或 runtime 准入在执行前失败。
4. 提交一个 workflow，其中某个 reduce 或 scan compute 节点使用了无有效 monoid 描述子的算子。预期结果：该 compute 节点在 fork-join 执行前被拒绝。
5. 提交一个 transfer workflow，并尝试通过旧 alias 在 MoonBit 侧释放该 buffer。预期结果：可变别名权限不存在，因此该操作被拒绝。
6. 在 native 与 JavaScript 两个目标上执行等价的浮点 compute 节点，但两边的浮点 profile 不兼容。预期结果：每个目标内部都保持确定性，但不声称跨目标相等。
7. 在声明兼容浮点 profile 的目标上执行等价的浮点 compute 节点。预期结果：两边保持相同的已声明可观察结果和所有权纪律。
8. 提交一个 transfer-return-reuse workflow，并在 Rehydrate-Step 之前尝试 reuse。预期结果：在 rehydration 成功之前，reuse 被拒绝。
9. 提交一个 JS copy 或 transfer workflow，并分别触发 explicit return、finalizer cleanup 和 fail-closed runtime admission。预期结果：文档声明的生命周期优先级与单次释放义务成立。
10. 提交一个节点之间存在环的 workflow。预期结果：facade 校验或 native 准入在执行前拒绝。
11. 提交一个把 LockNode 绑定到 Channel 的 workflow。预期结果：facade 校验或 native 准入拒绝 capability 不匹配。
12. 提交一个使用 RwLock、Semaphore 或 OpaqueCapability 的 workflow。预期结果：descriptor 结构上可被接受，但 native runtime 在执行前以显式 unsupported runtime status fail closed。
13. 提交一个只包含单个 BarrierNode 的导出 barrier group。预期结果：该 workflow 的最终状态为 BARRIER\_BROKEN。

## Closure Checklist

只有在以下条件全部成立时，本规格才算闭合：

1. 每个导出执行请求都能由 workflow ADT 表示，而 compute 请求则作为 compute 子节点表示。
2. 每个边界 view 都携带显式 layout witness 与 ownership witness。

3. 每条 zero-copy 路径都满足布局同构。
4. 每条 borrow 路径都冻结 MoonBit 侧突变直到 borrow 释放。
5. 每条 transfer 路径都消费 MoonBit 侧的可变 authority，并且只通过显式 return 加 rehydration 恢复。
6. 每个可写 chunk 调度都经过分离检查验证。
7. 每条 reduction 或 scan combine 路径都由有效 monoid law 支撑。
8. 不存在任何异常、panic 或 abort 路径跨越 FFI 边界逃逸。
9. 每条 zero-copy 准入路径都完成字节级布局与对齐检查。
10. 每条 JS copy 与 transfer 路径都具有一个闭合的清理故事，包括 finalizer 与 fail-closed 情况。
11. 每个 witness 义务都映射到一个可审计的 artifact 格式。
12. 每个非法输入向量都通过显式状态失败，而不是崩溃或触发未定义行为。
13. 文档中引用的每项保证都已显式陈述。

### 第三部分：实现一致性验证

#### 异常控制流与 Fail-Closed 语义

第一部分把 worker 执行建模为成功的局部小步。忠实的物理实现还必须闭合 MoonBit、C wrapper 与 native runtime 中可能出现的所有异常控制流路径。

**状态分类。** 每个实现可见的失败都必须属于一个封闭的状态家族：

```
RejectReason ::= ParseReject
               | LayoutReject
               | OwnershipReject
               | DescriptorReject
               | OperatorReject
               | KernelLookupReject
               | KernelWitnessReject
               | KernelLayoutReject
               | KernelArityReject
               | KernelLawReject
               | BorrowLifecycleReject
               | TransferLifecycleReject
               | JSLifecycleReject
               | DeterministicTreeReject
               | TargetSupportReject
```

图 21 封闭的拒绝原因家族。

```
Status ::= Ok
         | Reject ( RejectReason )
         | Fault ( RuntimeFault )
         | Worker ( WorkerFault )
```

图 22 带标签的符合性状态代数。

**边界结果承载。**每个供运行时执行使用的 MoonBit 暴露 native kernel 与 wrapper 入口，都必须返回一个 ABI-safe 的结果封装：

```
typedef struct {  
    Status status;  
    Payload payload;  
} FFIResult_Payload;
```

或者返回目标特化但在可观察行为上等价的记录。任何栈展开、panic 传播或宿主异常传播都不得跨越 FFI 边界。

**阶段映射义务。**每个非 ok 状态都必须对应以下符合性阶段之一：

- facade 构造拒绝；
- wrapper parse 拒绝；
- descriptor 准入拒绝；
- 并行域前的 runtime 拒绝；或
- 并行域内 worker 的 fail-closed 发布。

该阶段映射属于可观察合同的一部分，MoonBit 侧必须能够通过模式匹配，或通过可观察行为等价的 目标 API 恢复这一信息。

**共享 worker 状态。**在 native 并行域内，worker 只能通过一个共享的原子状态字通信失败，该状态字初始为 ok。

```
typedef struct {  
    _Atomic uint32_t code;  
} AtomicStatusWord;
```

**Worker 错误发布规则。**若某 worker 检测到错误状态  $e \neq \text{Ok}$ ，则它必须：

1. 以 release 语义把  $e$  写入 AtomicStatusWord，除非先前已经可见某个非 ok 状态；
2. 除线程本地清理外，不再继续发出新的写入；
3. 只执行清理或 early-exit 步骤，直到并行域 join。

其余 worker 必须在准入点或循环检查点以 acquire 语义观察共享状态字；一旦观察到非 ok 值，就必须降级为确定性的 early exit。

**CFG 闭包义务。**对于 #pragma omp parallel 内每一条可达控制流路径，实现必须通过静态分析、代码审计或等价论证证明：它只能以下列方式之一终止：

- 以 ok 正常完成 worker；
- 显式发布一个非 ok 状态并完成本地清理；
- 因观察到先前发布的非 ok 状态而 early exit。

不得存在任何通过 abort()、未捕获异常逃逸、跨边界栈展开或未定义控制转移终止的路径。

## 布局同构的实现化与安全降级

第一部分把 LayoutOK 抽象定义为公理式条件。符合性的实现必须通过字节级静态断言与动态准入检查 来将它实现化。

**C 边界记录义务。**对 lv\_view，C 侧实现必须通过属性、packing 指令或等价的 ABI 稳定机制，保持第一部分给出的 规范 size、alignment 与 field offset 方程。

这些检查必须绑定到一个已声明的 ABIPProfile(t)，而不是绑定到推断出来的宿主字宽。

**编译器布局出清。**native wrapper 必须在编译期出清下列条件：

- `sizeof(lv_view)` 与 `alignof(lv_view)`;
- `addr`、`len`、`kind`、`mode` 与 `stride` 的字段偏移;
- 活动目标 `profile` 下 `UIntPtr`、`UInt64` 与 `UInt32` 的宽度检查;
- 活动目标合同所要求的标量宽度与 `endian` 假设。

这些检查必须由 `static_assert` 或可观察行为等价的编译期机制表达。

**动态 zero-copy gate。** 在某个边界 `view` 以无拷贝方式被准入为 `borrow` 或 `transfer` 之前, `descriptor` 形成必须检查:

- `base pointer` 对所需元素对齐的模运算结果为零;
- `stride` 等于该标量种类声明的连续遍历规则;
- 总字节跨度有界且不发生算术溢出;
- 源与目标的 `layout witness` 在标量解释和宽度上一致。

若任一检查失败, 实现不得原地重解释该 `view`。

**确定性降级规则。** 对每个目标, 实现必须为非同构视图声明下列结果之一:

- `CopyConforming`: 复制字节到符合布局的表示后继续; 或
- `RejectNonIsomorphic`: 在 `runtime` 准入前直接拒绝。

该选择必须对给定目标路径稳定成立, 不得依赖偶然的宿主 ABI 兼容性。

对于每条被准入的目标路径, 实现都必须把 `ABIProfile(t)` 及所选降级规则文档化为一个可审计的 边界合同。

## 算子纯度与 Plan 准入

数据竞争自由定理依赖于 `worker footprint` 可以仅由 `plan` 数据推出。因此 `facade` 与 `lowering` 边界必须拒绝任何把不可验证 `effect` 偷运过 FFI 边界的可执行内容。

**Kernel registry 表面。** `version 1` 的可执行 `Plan` 只能携带能在活动目标的静态受审计 `KernelRegistry(t)` 中解析的 `KernelId` 引用。任意 `MoonBit` 闭包、任意 `JavaScript` 宿主函数、未入册的 `native` 函数指针, 以及捕获了可变环境的值, 都不是可接受的 `plan payload`。

每个受信任的 `KernelDesc` 都必须携带:

- `KernelId`, 即 `plan` 唯一可见的 `kernel` 级标识符;
- 隐藏的 `native FnPtr`, 其不得跨越 `MoonBit plan` 边界;
- `PurityWitness`;
- `FootprintWitness`;
- `LayoutContract`;
- `ArityMeta`;
- `LawMeta`; 以及
- `TargetMeta`。

动态的任意 `kernel` 注册不属于本合同。符合性声明只能依赖静态受审计 `registry` 中已有的 `descriptor`。

对浮点 `reduce` 与 `scan`, 额外的准入条件是必须提供 `DeterministicTree` 律 `witness`, 使 `runtime descriptor` 为被接受的执行固定唯一的组合树。

**安全抽象义务。** 一个符合性的 `MoonBit-facing API` 必须确保: 任何通过类型检查并被准入的执行请求, 都会 `lowering` 为一个其 `worker footprint` 仅由以下信息决定的 `plan`:

- plan 构造子;
- 显式 buffer view 与 layout witness;
- 受信任的 KernelId 与已解析的 KernelDesc;
- 被 PlanWF 允许的 runtime configuration 字段。

任何允许隐藏可写别名、逃逸可变引用，或携带无法依据 SepChunks、所有权线性与边界 access mode 证明其安全性的 effectful 算子体的构造路径，都必须被拒绝。

**Witness 义务。** 对每个被准入的 descriptor  $k = \text{KernelDesc}(\dots)$ :

- PurityWitness(k) 必须证明执行在声明的输入/输出 view 和 MetaOf(d) 之外没有副作用;
- FootprintWitness(k) 必须证明 worker 写入被限制在其分配到的 chunk 内，而任何超出 chunk 的读取都仅限于声明的只读输入或 MetaOf(d);
- LayoutContract(k) 必须在 runtime 执行前针对所有承载的边界 view 被出清;
- ArityMeta(k) 必须与 plan 的构造子类别和 access mode 匹配; 且
- LawMeta(k) 必须满足  $\mu$  所要求的规约律义务，包括浮点归约所需的 deterministic-tree 要求。

这些义务若不满足，执行必须通过 Status 中专用的拒绝分支被拒绝。

**可审计 witness 格式。** 每个携带 witness 的 registry 条目都必须暴露一个可独立复审的 artifact bundle。最小可接受 bundle 包含:

- 稳定的 kernel 标识符，以及与被审计实现绑定的版本号或哈希;
- 对 purity、footprint、layout、arity 与 law 声明的机器可检查摘要或表格化摘要;
- 这些声明成立时所依赖的目标或 ABIProfile(t) 与浮点 profile 假设;
- 每项声明所采用的审计方法: 证明 artifact、静态分析结果、测试证书，或人工审计记录; 以及
- 以第三方可区分“缺失”“拒绝”“接受”证据的形式记录的审计结论。

语义义务本身具有规范性; artifact 格式则是使符合性声明可复审而非自证的最小可接受证据。

## 基于契约的验证矩阵

符合性声明必须由一组可执行的验证矩阵支撑，该矩阵覆盖接受、拒绝与 fail-closed 执行结果。

**验证维度。**

1. 正向实现化: 等价的 map、reduce、scan 与 map-reduce Plan 在已准入的 native 与 JavaScript 目标上保持相同的 plan 指称与所有权轨迹。
2. 异常控制流: 注入的算子故障、分配失败、畸形 payload 与 runtime 准入失败都以显式状态码出现，而不是以进程终止暴露。
3. 布局与字节级安全: 错误对齐、不一致 stride、非同构 scalar kind 与溢出的字节跨度计算，都会按照目标规则触发 CopyConforming 或显式拒绝。
4. 所有权与 chunk 安全: 重叠可写 chunk、transfer 后的陈旧 alias、活动 borrow 期间的突变，以及 double-borrow 尝试，都必须在健全执行发生前失败。
5. 算子准入: 携带闭包、宿主函数、未知 KernelId，或其他不可验证内容的 Plan，必须在 facade 或 wrapper 准入阶段被拒绝。
6. 浮点确定性规约: 对同一个已准入的浮点 reduce 或 scan descriptor，在 worker 调度不同的情况下重复执行，都必须在单一目标 profile 下得到相同的组合树与相同的可观察结果; 跨目标相等性只对已声明兼容的受审计浮点 profile 做测试。

7. 借用生命周期: 成功 discharge 会恢复 `Own_M(b)`; 而过早 discharge、重复 discharge, 或在没有活动借用时 尝试 discharge, 都必须被拒绝。
8. Transfer-return 生命周期: 成功的 `Return-Step` 加 `Rehydrate-Step` 会恢复 `Own_M(b)`; 而在 `Returned_M(b)` 上 reuse、重复 return, 或在没有活动 `Own_C(b)` 时 return, 都必须通过 `TransferLifecycleReject` 被拒绝。
9. JS 生命周期: copy 出来的 buffer 只有一个 owner 且只有一条释放路径; transfer tombstone 会阻止陈旧宿主复用; finalizer 与 explicit-return 的顺序遵守文档声明的目标规则; 缺失 snapshot 义务或违反 生命周期时, 必须通过 `JSLifecycleReject` 拒绝。
10. Borrow 纪律: 任何以 ownership mode Borrow 下放的 view 都必须是 `ReadOnly`; 可写 borrow 尝试必须在 执行前被拒绝。
11. 元数据纪律: worker 在其 chunk footprint 之外只能读取 `MetaOff(d)`; 任何可写或畸形的元数据区域都必须 在不健全执行发生前被拒绝。
12. Registry 解析: 已知 `KernelId` 会解析为受审计 descriptor; 未知 id 必须通过 `KernelLookupReject` 拒绝。
13. Witness 校验: 缺失 purity、footprint、layout、arity、law、profile 或 target-support 证据时, 必须通过 对应的专用拒绝分支拒绝。
14. 审计 artifact 完整性: 验证矩阵使用到的每个 registry 条目都必须标识其证据 bundle 与审计结论。

**拒绝阶段合同。** 每个负向验证用例都必须说明:

- 被违反的前置条件;
- 拒绝或失败发布发生的阶段: facade 构造、wrapper parse、descriptor 准入、并行域前的 runtime 检查, 或并行域内 worker 的 fail-closed 发布;
- 预期的带标签 `Status` 分支, 以及适用时的拒绝子原因。

其中, borrow discharge 违规映射到 `BorrowLifecycleReject`, transfer-return 违规映射到 `TransferLifecycleReject`, 而 JS snapshot / copy / transfer 生命周期违规映射到 `JSLifecycleReject`。

**覆盖义务。** 只有当验证矩阵中的每个非法输入向量都收敛为本规格定义的显式状态, 并且没有任何非法向量能够产生 段错误、静默数据损坏或未定义行为时, 实现才可称为一致性完备。