

# Serverless性能优化挑战赛

## 决赛答辩

弹性计算·虚拟化团队 二手玫瑰

于海鑫/徐意如

2023.08.15

# 目录

- 参赛目标
- 技术选型
- 方案设计
- 技术亮点

# 参赛目标

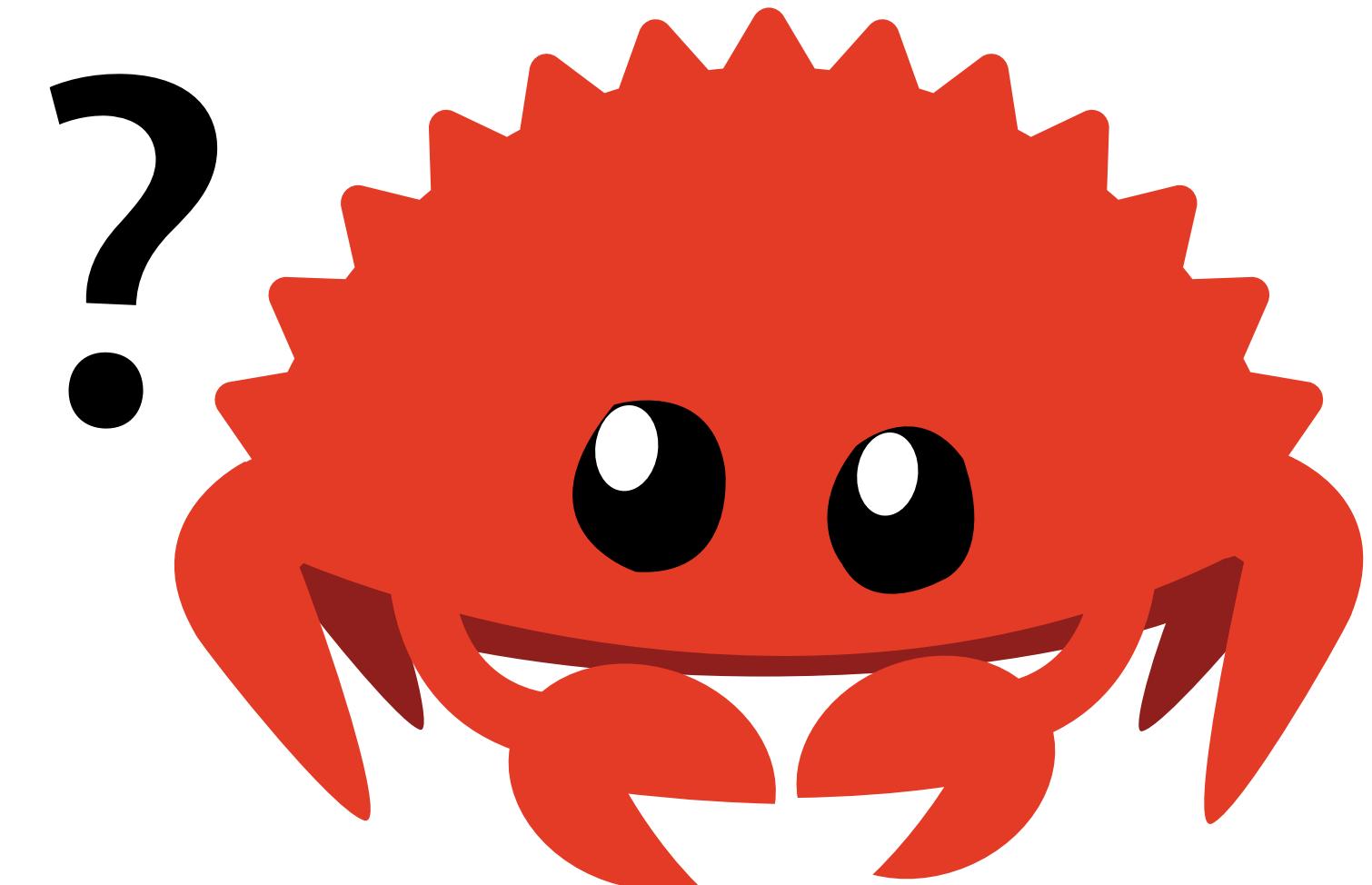
- 赛程短，且期间需要兼顾工作、娱乐
- → 时间有限，追求投入产出比
- = 在投入时间较少的情况下，做出最有效的优化



# 技术选型

## 编程语言 A.K.A Why Rust?

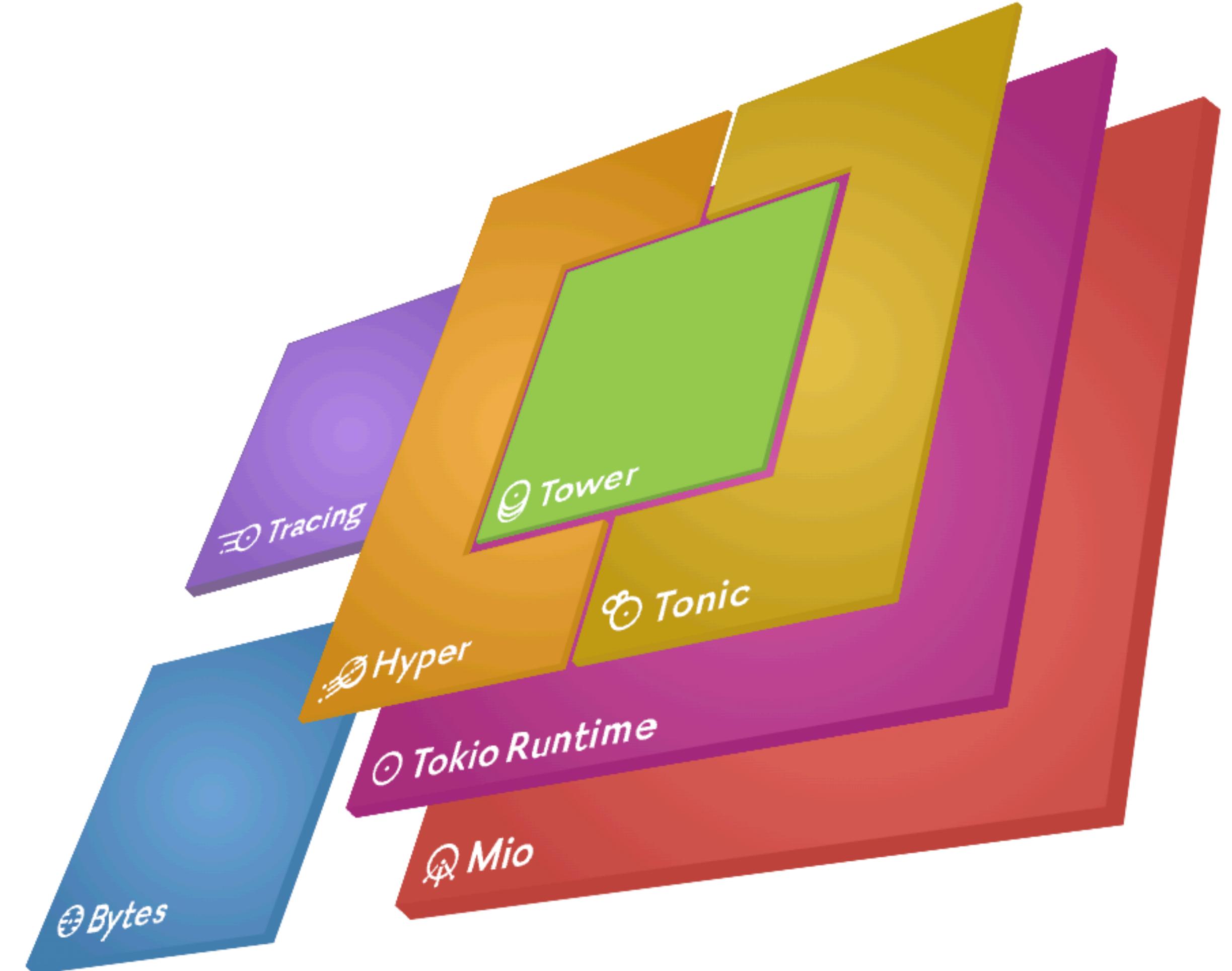
- 主办方提供的模版
- Java? 语法繁琐、古板，运行时依赖尤其重 🤦
- Go? 「大道至简」，语法不够 expressive 🤦
- Rewrite it in **Rust!**
- 优势：高性能、高可靠性、对开发者友好
- 问题：入门曲线陡峭，需要从头搭建框架



# 技术选型

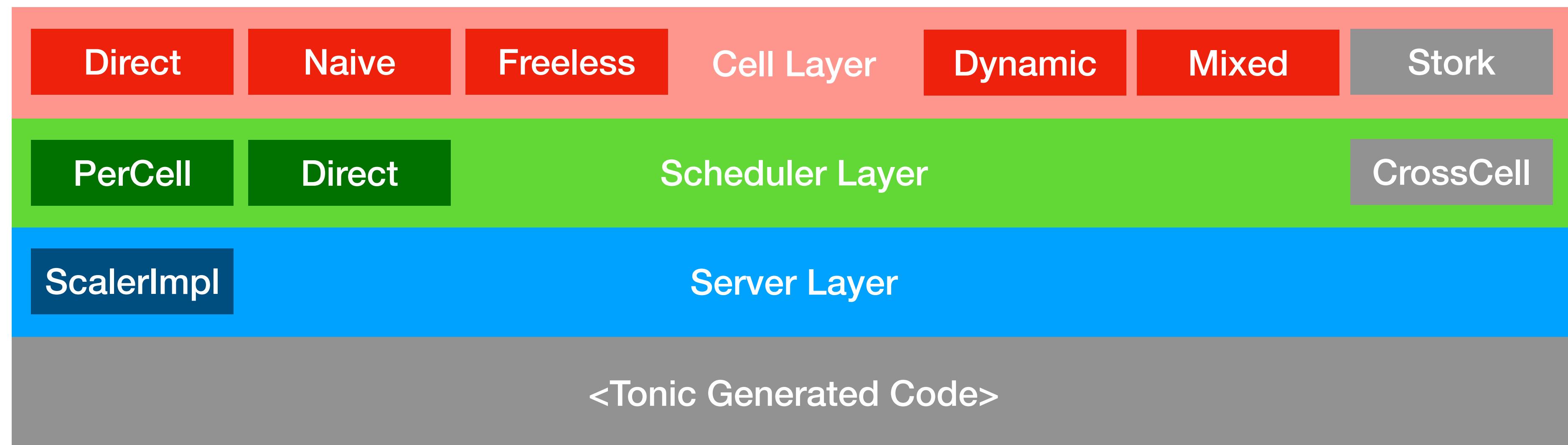
## 技术栈

# 「The Tokio Stack」



# 方案设计

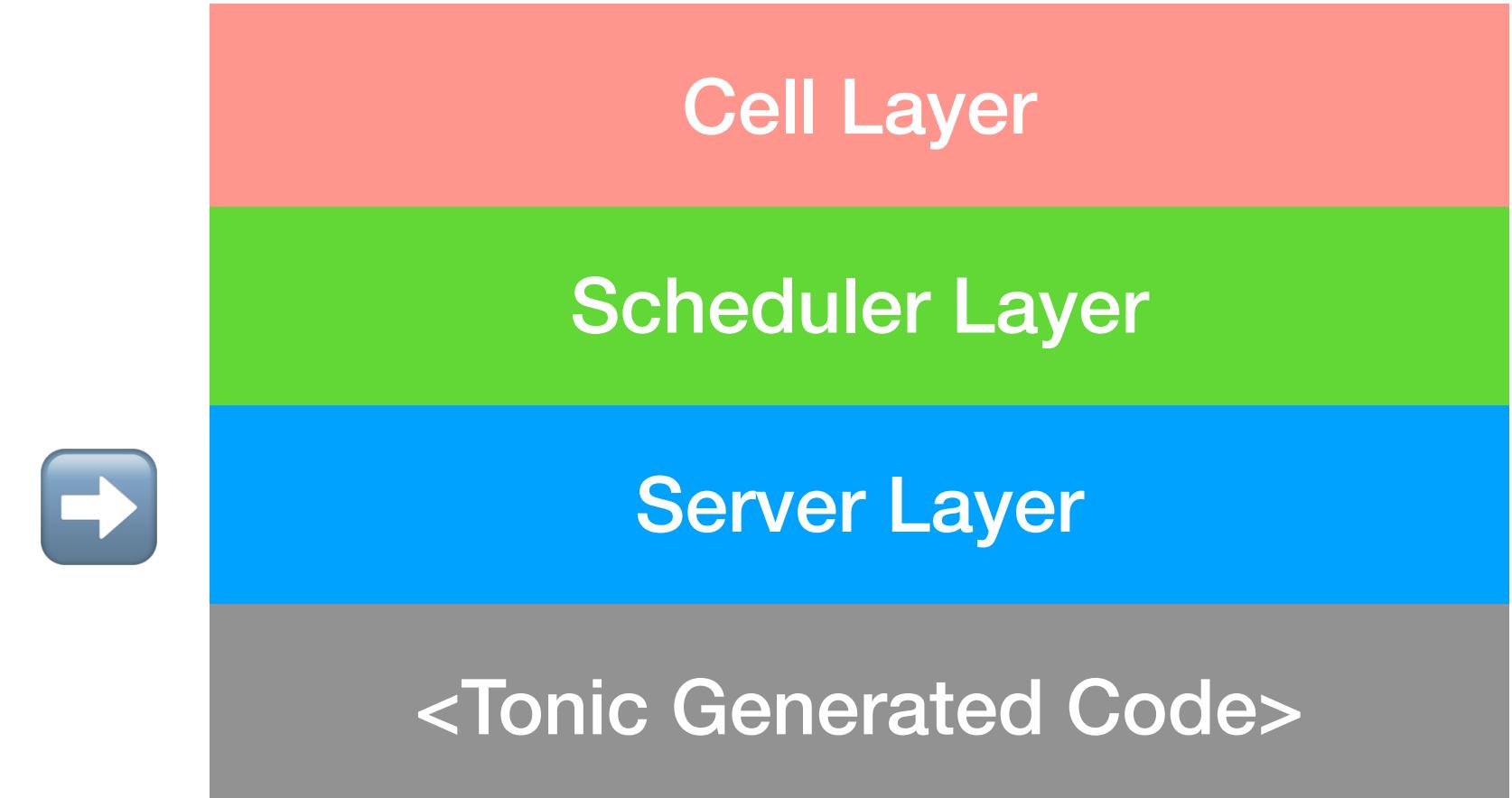
## 总体架构



- **Server Layer**: 对接 Tonic，实现内部错误到 Tonic 错误的转换
- **Scheduler Layer**: 总体调度策略、提取参数，对 Cell Layer 简化输入
- **Cell Layer**: 对单个 Meta Key 的调度策略

# 方案设计

## 总体架构：Server Layer

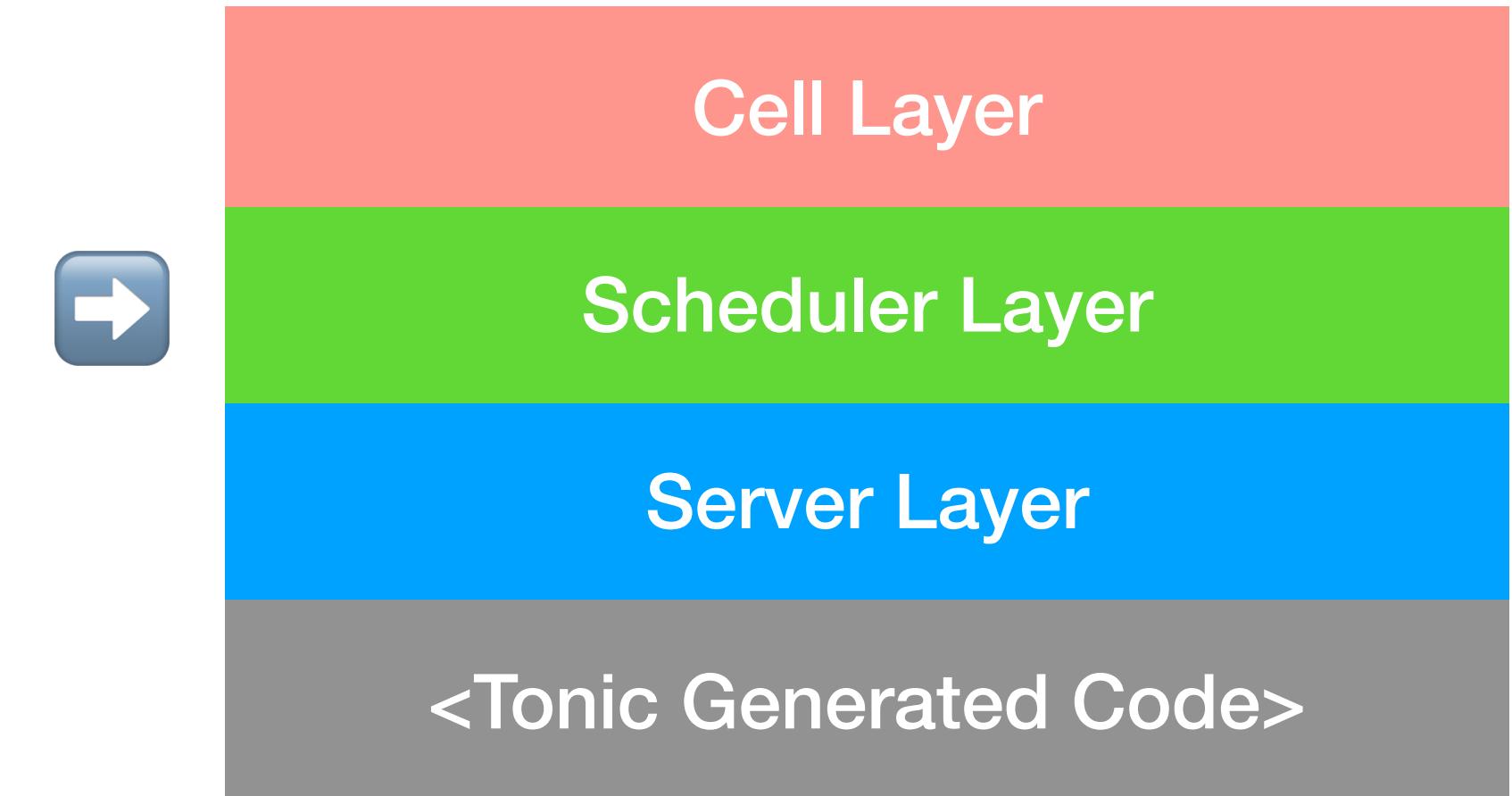


```
// Server Layer Input
pub trait Scaler: Send + Sync + 'static {
    async fn assign(&self, req: Request<AssignRequest>) -> TonicResult<AssignReply>;
    async fn idle(&self, req: Request<IdleRequest>) -> TonicResult<IdleReply>;
}

// Server Layer Output
pub trait Scheduler: Send + Sync + 'static {
    async fn assign(&self, request_id: String,
                    timestamp: u64, meta: Option<model::Meta>) -> Result<model::Assignment>;
    async fn idle(
        &self, assignment: Option<model::Assignment>,
        idle_reason: Option<model::IdleReason>) -> Result<()>;
}
```

# 方案设计

## 总体架构：Scheduler Layer



```
// Scheduler Layer Input
pub trait Scheduler: Send + Sync + 'static {
    async fn assign(&self, request_id: String,
                    timestamp: u64, meta: Option<model::Meta>) → Result<model::Assignment>;
    async fn idle(
        &self, assignment: Option<model::Assignment>,
        idle_reason: Option<model::IdleReason>) → Result<()>;
}
```

```
// Scheduler Layer Output, Cell Layer Input
pub trait CellFactory<T: Cell>: Send + Sync + 'static {
    fn new(&self, meta: model::Meta, client: Arc<Platform>) → Arc<T>;
}
```

```
pub trait Cell: Send + Sync + 'static {
    async fn assign(self: Arc<Self>, request_id: String,
                    timestamp: u64) → Result<model::Assignment>;
    async fn idle(self: Arc<Self>, assignment: model::Assignment,
                  idle_reason: model::IdleReason) → Result<()>;
}
```

# 方案设计

## 调度策略 v0.1: Direct

- 最简单的调度策略
- 作为整个框架的 PoC 开发

```
assign = create_slot  
idle   = destroy_slot
```

# 方案设计

## 调度策略 v1.0: Pool + GC

```
impl Cell for NaiveCell {
    async fn assign(
        self: Arc<Self>, request_id: String, _: u64
    ) -> Result<model::Assignment> {
        let slot = self.clone().get_or_create_free_slot().await?;
        let instance_id = slot.slot.instance_id.clone();
        self.occupied_slots.insert(request_id.clone(), slot);
        Ok(model::Assignment { ... })
    }
    async fn idle(
        self: Arc<Self>, assignment: model::Assignment, idle_reason: model::IdleReason,
    ) -> Result<()> {
        let slot = { ... }; // Get slot by id
        if idle_reason.need_destroy {
            self.destroy_slot(&slot).await
        } else {
            self.put_free_slot_fresh(slot);
            Ok(())
        }
    }
}
```

# 方案设计

## 调度策略 v1.0: Pool + GC

```
async fn get_or_create_free_slot(self: Arc<Self>) → Result<SlotInfo> {
    if let Some(slot) = self.try_get_free_slot() {
        return Ok(slot);
    }

    if let Some(slot) = timeout(ONE_SEC, self.wait_for_free_slot()).await.ok().flatten() {
        return Ok(slot);
    }

    match select(self.wait_for_free_slot(), self.create_free_slot()).await {
        Either::Left((slot, fut)) => {
            if let Some(slot) = slot {
                self.create_slot_in_background(fut);
                return Ok(slot);
            }
            fut.await
        }
        Either::Right((slot, _)) => slot,
    }
}
```

# 方案设计

## 调度策略 v1.0: Pool + GC

```
async fn destroy_outdated_slots(self: Arc<Self>) -> usize {
    // Select slots that unused for a long time
    let to_free = self.select_outdated_slots();
    let result = to_free.len();
    tokio::spawn(async move {
        for slot in to_free {
            if let Err(e) = self.destroy_slot(&slot).await {
                log::error!("destroy slot failed: {:?}", e, will retry);
                self.put_free_slot(slot);
            }
        }
    });
    result
}

tokio::spawn(async move {
    while let Some(cell) = weak_cell.upgrade() {
        let key = cell.meta.key.clone();
        let count = cell.destroy_outdated_slots().await;
        tokio::time::sleep(OUTDATED_SLOT_GC_INTERVAL_SEC).await;
    }
});
```

# 方案设计

## 调度策略 v1.0: Pool + GC

- 优势: 结构简洁, 效果不算差
- 问题: 不感知 meta 自身信息以及请求数量的变化走向

# 方案设计

## 调度策略 v1.5: Dynamic GC Timeout



```
const OUTDATED_SLOT_GC_SEC: f64 = 15; // v1.0
```



```
// v1.5, factor = [1, self.meta.memory_in_mb / MEM_MB_BASE]
fn outdated_slot_gc_sec(&self) → f64 {
    MEM_MB_BASE * GC_SEC_BASE * self.factor / self.meta.memory_in_mb
}
```

- 在 v1.0 的基础上，增加动态超时时间的支持
  - 相同情况下，内存越大的实例，超时时间越短
  - (本轮回收周期中的 assign 请求数 - 上轮周期的 assign 请求数) 越大/小，factor 越大/小

# 方案设计

## 调度策略 v1.5: Dynamic GC Timeout

- 测试结果: DataSet1/DataSet2 负向收益明显, DataSet3 正向收益明显, 总体负收益
- 优势: 感知 meta 的信息以及请求数的变动
- 问题: 需要假设各个 key 的请求数完全独立, 不适用于前两个数据集的场景

# 方案设计

## 调度策略 v2.0: Dynamic Dispatch

- 想要保留 v1.5 带来的性能提升
- + 不想降低前两个数据集的性能
- = 动态派发，只对数据集 3 使用 v1.5 的策略



# 方案设计

## 调度策略 v2.0: Dynamic Dispatch

```
pub struct MixedCell {
    inner: Arc<dyn Cell>,
}

impl Cell for MixedCell {
    async fn assign(
        self: Arc<Self>, request_id: String, timestamp: u64,
    ) -> Result<model::Assignment> {
        self.inner.clone().assign(request_id, timestamp).await
    }

    async fn idle(
        self: Arc<Self>, assignment: model::Assignment, idle_reason: model::IdleReason,
    ) -> Result<()> {
        self.inner.clone().idle(assignment, idle_reason).await
    }
}
```

- 简单拿 Arc<dyn Cell> 做一个动态派发 😊

# 方案设计

## 调度策略 v2.0: Dynamic Dispatch

```
impl CellFactory<MixedCell> for SimpleMixedCellFactory {
    fn new(&self, meta: model::Meta, client: Arc<Platform>) → Arc<MixedCell> {
        if meta.key.len() ≥ 30 {
            // Use v1.5 when key is likely to from dataset 3
            MixedCell::new(MemoryCellFactory.new(meta, client))
        } else {
            // otherwise use v1.0
            MixedCell::new(NaiveCellFactory.new(meta, client))
        }
    }
}
```

- 如何区分数据集?
  - key 长的像是 md5 值的就是数据集 3

# 方案设计

## 调度策略 v2.0: Dynamic Dispatch

- 测试结果: DataSet1/DataSet2 分数同 v1.0, DataSet3 分数同 v1.5
- 优势: 结合了前两个版本的优势
- 问题: 分数不够高

日期: 2023-08-05 10:32:58 [查看日志](#)  
score: 51.0852  
dataSet\_1\_score: 47.0228  
dataSet\_2\_score: 49.0614  
dataSet\_3\_score: 57.1715  
dataSet\_1\_resourceUsage\_score: 2.5179  
dataSet\_1\_coldStartTime\_score: 44.5049  
dataSet\_2\_resourceUsage\_score: 4.3555  
dataSet\_2\_coldStartTime\_score: 44.7059  
dataSet\_3\_resourceUsage\_score: 14.9999  
dataSet\_3\_coldStartTime\_score: 42.1716

# 方案设计

## 调度策略 v3.0: Pre-allocate Slot

- 回顾评分方式
  - 总分 = (请求执行总消耗/Slot 总消耗 + 请求执行总时间/(调度总时间 + 请求执行总时间)) \* 100/2
  - 假设在评测开始时申请  $\infty$  个 Slot ...
    - 总分 = 0(资源分) + 50(调度分) = 50
  - 热知识:

在性能优化挑战赛背景下，总分小于 50 的优化没有必要考虑。

大哥你玩优化，~~你玩他有啥用啊~~~

# 方案设计

## 调度策略 v3.0: Pre-allocate Slot

- 在 v1.0 之上的优化：
  - **预创建**: 在 Cell 创建之初，预创建一些 Slot 备用
  - **长周期**: GC 超时时间/GC 周期拉长，避免 Slot 过早释放
  - **快释放**: 在一段时间没有任何请求后，释放所有 Slot

```
let pre_allocate = req_per_min_to_slots(&meta.key, expected_max_req);
for _ in 0..pre_allocate {
    let h = cell
        .clone()
        .create_slot_in_background(cell.clone().create_free_slot());
    handles.push(h);
}
```

# 方案设计

## 调度策略 v3.0: Pre-allocate Slot

- 预创建 Slot 数计算
- $\text{Slot 数} = (\text{最大分钟请求数} * \text{平均请求处理时间}) / 1\text{min} + \text{裕量}$
- 其中：
  - 最大分钟请求数、平均请求处理时间来自对数据集的处理
  - 裕量是试出来的经验值

# 方案设计

## 调度策略 v3.0: Pre-allocate Slot

- 测试结果: DataSet1/DataSet2 分数大幅提升, DataSet3 分数同 v1.5
- 优势: 通过预创建 Slot, 规避了冷启动时带来的调度开销以及资源开销
- 问题: 不够智能

◦ 日期: 2023-08-08 00:07:23 [查看日志](#)

score: 55.7583

dataSet\_1\_score: 55.3504

dataSet\_2\_score: 54.1914

dataSet\_3\_score: 57.7331

dataSet\_1\_resourceUsage\_score: 5.5205

dataSet\_1\_coldStartTime\_score: 49.8298

dataSet\_2\_resourceUsage\_score: 5.0264

dataSet\_2\_coldStartTime\_score: 49.1650

dataSet\_3\_resourceUsage\_score: 15.3556

dataSet\_3\_coldStartTime\_score: 42.3775

# 技术亮点

- 基于 Rust 构建项目
  - 代码表达整洁，二进制文件依赖少，性能高
  - 更加的「云原生」
- 分层架构设计：可以方便地拓展功能、测试新策略
- 动态派发：根据不同数据集特征选用不同策略
- 预分配 Slot：避免按需创建时带来的初始化开销

# 分工

- 于海鑫：算法实现，PPT
- 徐意如：数据集分析

Q & A

