复习

海苔脆

- 异步并发
- 扩容
- 队列

进程与线程
- 异步并发 (Promise和async/await)
- TaskPool机制和Worker机制
- 进程模型
- IPC机制

关注点
- 秒杀系统
- 可视化全链路日志追踪

系统工具库
- 相机开发服务
- 传感器运作机制
- 卡片实现原理
- 卡片渲染原理
- 卡片数据交换

从微服务到云原生
- 微服务架构
- Docker、K8s、Istio架构
- Serverless
- 云原生
- AI原生

高级特性
- 脚本基础流程运行图
- UI测试用例
- NAPI对比JNI

智能体
- 智能体的组成
- Planning
  - Chain-of-Thoughts
  - React
  - Reflection / Self-Refine
  - Subgoal decomposition
- RAG
- MCP

Timeless的东西
- 需求的涌现
  - spec
- 现实的复杂度
  - 层次性skill+知识
- 不确定性问题
  - 各司其职
  - 规模化
  - 检查智能体

ArkCompiler和ArkRuntime

方舟编译运行时 (ArkCompiler & ArkRuntime)

- **核心理念**：方舟编译运行时的核心在于**将传统脚本语言在运行时（Runtime）才进行的源码解析、编译等高耗时操作，提前到开发端的编译阶段完成**。
- **运行模式对比**：
    - **普通 JS 运行时**：在编译阶段仅进行源码打包，具体的源码解析和字节码编译都在设备端运行时进行，这会占用大量的运行资源并拖慢首屏加载。
    - **方舟编译运行时**：在编译阶段即完成源码解析和字节码生成。设备端只负责安装和最终执行，从而显著提升了应用启动性能。
- **按需组合策略**：方舟运行时支持根据不同硬件条件（如低内存 IoT 设备或高性能手机）灵活组合执行模式，确保在各种设备上都能达到最优的性能与开销平衡。

类型推导 (Type Inference)

方舟编译器通过类型推导技术，将动态类型的脚本语言转化为更高效率的执行格式。

- **编译流程**：
    - **输入**：支持 ECMAScript 和 TypeScript 源代码。
    - **TSC & 类型推导**：通过 TSC 进行语法检查，并进行类型推导生成抽象语法树 (AST)。
    - **优化器 (Optimizer)**：基于推导出的类型信息进行类型指导优化，生成中间表示 (IR)。
    - **字节码生成**：由 BC Generator 最终生成方舟字节码文件 (.abc)。
- **产物组成**：最终生成的可执行文件是一个包含静态类型字节码 (Static typed BC)、动态类型字节码、类型信息和调试信息的综合包。

执行引擎对比 (Interpreter, JIT, AOT)

方舟运行时根据代码执行的频率和性能需求，动态切换不同的执行引擎。

执行模式	启动速度	执行性能	内存占用	特点与机制
解释器 (Interpreter)	**快**	一般	**小**	直接运行字节码，无需等待编译，适合首次运行或低频代码。
JIT 编译器	较慢	**极高**	较高	在运行过程中实时编译热点代码，随着预热时间的增加，性能达到峰值。
AOT 编译器	**快**	**高**	高	运行前预先编译成机器码，实现“启动即高性能”。

- **逆优化 (Deopt)**：当 AOT 或 JIT 生成的优化代码所基于的假设（如变量类型）在运行时不再成立时，系统会自动触发逆优化，安全回退到解释器模式执行。
- **PGO 优化**：支持配置文件导向优化，通过收集运行时的采样数据来指导 AOT 编译器生成质量更高的机器码。

好的，我会严格按照你提供的框架，从 PPT 课件和截图中提取核心知识点进行内容填充，并剔除所有杂乱的编号和图标。

ArkUI

ArkUI 是一套为构建 HarmonyOS 应用界面而设计的 UI 开发框架。它采用声明式语法，将界面的描述与逻辑分离，通过极简的代码即可完成跨设备的高性能界面开发。

ArkUI 框架

ArkUI 框架由以下层次组成：

- **前端框架层**：包含声明式 UI 范式、内置组件、API 和状态管理机制。
- **桥接层**：负责前端框架与底层引擎的对接，实现语法解析和属性转换。
- **引擎层**：包括 UI 后端引擎和语言运行时。后端引擎负责布局、动画、事件及渲染管线；运行时负责代码的执行。
- **平台抽象层**：实现对底层操作系统的对接，确保在不同硬件上的渲染一致性。
- **技术升级**：在 ArkUI 3.1 中，UI 更新机制从 COMPONENT 和 ELEMENT 树形结构对比，升级为基于单节点 NODE 的函数式更新，极大提升了渲染性能。

UI示例

以下是一个标准的 ArkUI 声明式开发代码示例：

```
@Entry
@Component
struct Index {
  @State message: string = 'Hello World'

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
        Button('点击更新')
          .onClick(() => {
            this.message = 'ArkUI 已更新'
          })
          .width('150vp')
          .height('50vp')
          .margin({ top: 20 })
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

页面生命周期 组件生命周期

- **页面生命周期**（仅限于被 `@Entry` 装饰的组件）：
  - **onPageShow**：页面显示时触发。
  - **onPageHide**：页面隐藏时触发。
  - **onBackPress**：用户点击返回按钮时触发。
- **组件生命周期**（适用于所有 `@Component` 装饰的组件）：
  - **aboutToAppear**：组件实例创建后、执行 `build()` 渲染前触发。
  - **aboutToDisappear**：组件即将销毁前触发。

状态管理

状态管理是通过装饰器驱动 UI 自动更新的核心机制：

- **@State**：组件内私有变量，改变时触发当前组件重新渲染。
- **@Prop**：父组件向子组件单向传递数据。
- **@Link**：父子组件之间双向同步数据。
- **@Provide / @Consume**：跨层级组件之间双向同步数据。
- **@Observed / @ObjectLink**：用于监控嵌套对象或数组内部属性的变化。

渲染控制

- **条件渲染**：使用 `if/else` 语句根据逻辑状态决定是否创建或渲染特定的 UI 组件。
- **循环渲染**：
  - **ForEach**：从数组中迭代数据并创建相应组件。
  - **LazyForEach**：数据懒加载机制，仅在滚动到可视区域时创建组件，适用于大数据量列表，能显著优化内存消耗。

布局结构

ArkUI 提供了多种容器组件来构建复杂的界面：

- **线性布局**：`Column`（纵向排列）、`Row`（横向排列）。
- **层叠布局**：`Stack`（组件在 Z 轴上堆叠）。
- **弹性布局**：`Flex`（提供更灵活的对齐和分布方式）。
- **相对布局**：`RelativeContainer`（基于锚点定位）。
- **网格布局**：`Grid` / `GridItem`。
- **布局属性**：包括组件区域（Size）、填充（Padding）、边距（Margin）以及比例分配（layoutWeight）。

UI渲染

UI 渲染是将描述性的代码转化为屏幕像素的过程。ArkUI 通过数据绑定机制实现逻辑与 UI 的分离，将传统的 7 个复杂流转步骤简化为 2 个，从而降低了跨端协作的开发代码量。

Component树、Element树、Render树

渲染管线中维护着三棵核心树：

- **Component 树**：由开发者编写的代码构成，描述了 UI 的结构和属性。
- **Element 树**：Component 的实例，负责维护 UI 的状态信息。
- **Render 树**：负责最终的显示信息，包含每个节点的坐标、大小和绘制指令。

布局的步骤

布局过程遵循由上至下的约束传递和由下至上的尺寸反馈：

1. **测量**：父节点向下传递布局约束，子节点计算自身所需的空间。
2. **计算**：根据子节点的反馈，父节点确定所有节点的最终大小。
3. **定位**：父节点根据布局策略计算子节点的相对位置坐标。

绘制

绘制阶段由 UI 线程完成：

- 遍历 Render 树中标记为“脏（Dirty）”的节点。
- 执行 `Paint` 操作，将绘制命令记录到图层（Layer Tree）中。
- 每个组件的绘制内容最终被转化为一系列底层绘图指令。

光栅化合成机制

- **合成（Compositing）**：UI 线程生成图层树后发送给 GPU 线程。
- **光栅化**：GPU 线程将绘图指令转换为屏幕上的像素点（位图）。
- **帧显示**：通过 `compositeFrame()` 将最终合成的画面发送给显示硬件。

大前端框架演进

1. **Web 渲染类**：利用 WebView 容器渲染，如 PhoneGap。
2. **原生渲染类**：使用原生控件渲染 UI，但通过桥接调用，如 React Native、Weex。
3. **自渲染技术**：不依赖原生控件，通过底层绘图引擎（如 Skia）自行绘制 UI，如 Flutter。
4. **ArkUI**：代表了最新的全场景声明式开发范式，深度集成在系统底层。

React、Flutter、ArkUI对比

- **React (React Native)**：JS 驱动，通过 Bridge 映射到原生控件，性能受限于 JS 引擎与原生通信。
- **Flutter**：基于 Dart 语言，通过 Skia 引擎自渲染，脱离原生控件限制，性能表现优秀。
- **ArkUI**：HarmonyOS 原生框架。相比 React，它减少了跨端通信损耗；相比 Flutter，它在系统级有更深的优化，且支持声明式和类 Web 双范式，开发效率与性能更均衡。

根据你提供的《05-移动互联网需求分析.pdf》课件内容，我为你补全了这份笔记。我会严格遵守你的要求，不修改标题，仅补充内容，并去除所有杂乱的编号和图标。

需求分析

需求分析在移动互联网时代已经从单纯的技术实现转向了多维度的思维融合。它不仅要求理解“怎么做”，更要求开发者具备资本、商业、产品和技术四种思维。其核心目标是判断商业模式是否跑通、实际可获得市场规模大小、客户终身价值是否大于获客成本，以及产品是否具备足够的护城河。

层次性

需求的层次性体现了从宏观战略到微观功能的逐层转化，主要包含以下三个维度：

- **业务需求（战略层）**：对应公司的发展战略和业务目标。
- **用户需求（产品层）**：关注用户在特定场景下的目标和痛点。
- **系统需求（功能层）**：具体的功能实现和技术指标。

涉众

涉众分析是挖掘需求的起点，核心关注点在于：

- **分析范围的变化**：从关注“可见用户”转变为关注“不确定的用户”，深入探索潜在的群体。
- **公司基因的影响**：理解公司的原生基因（如淘宝的电商基因、抖音的短视频/社交基因）如何决定需求的优先级和产品走向。
- **多视角博弈**：平衡资本、商业、产品、技术等不同涉众的思维模式，确保产品既符合商业逻辑又具备技术可行性。

需求获取

需求获取的方法正随着互联网的发展而演进：

- **传统方法**：通过访谈、调查等方式获取需求。
- **上线原型法**：这是目前主流的获取方式，通过快速发布最小可行产品（MVP）并收集真实用户的反馈，进行快速迭代。
- **实时反馈机制**：利用数据罗盘、实时监控等工具，从用户的真实行为数据中捕捉需求。

关注点

在移动互联网环境下，需求的关注点发生了根本性的位移：

- **由内向外的转变**：从关注内部的“工作流程、用户界面”转向关注外部的“移动场景、互联网特性、跨端协同、用户体验”。
- **涌现性**：关注那些无法由单一组件解决、必须依赖系统各组件相互作用才能产生的整体属性，如呼叫中心的吞吐量。
- **场景化**：重视用户在碎片化时间、移动状态下的即时需求。

需求的组织

需求组织形式逐步从重型文档转向轻量化、敏捷化的描述：

- **从用例到用户故事**：需求的组织方式从复杂的“用例”演变为更加简洁、关注用户价值的“用户故事（User Story）”。
- **核心表达**：强调以用户为中心，描述“作为一个[角色]，我希望[做某事]，以便实现[某种价值]”。
- **资产沉淀**：通过有条理的组织方式，将琐碎的需求转化为可交付的产品功能列表。

根据你提供的多份 PPT 课件内容，我为你补全了这份笔记。笔记严格遵循你提供的框架，补充了核心知识点，并剔除了所有杂乱的编号、图标和冗余信息。

AbilityKit、Network Kit 、ArkData和ArkWeb

启动UIAbility

- **启动机制**：通过 `context.startAbility(want)` 发起。`Want` 是系统内的信息传递载体，用于指定目标组件的包名、类名或行为。
- **权限校验**：系统会检查调用方是否有权启动目标应用，并根据设备类型和配置决定启动方式。

冷启动和热启动

- **冷启动**：当应用进程尚未运行，系统需要先创建进程、初始化运行环境（如 ArkTS 运行时），再加载并实例化 UIAbility 组件，最后触发生命周期回调。
- **热启动**：应用进程已在后台运行，且 UIAbility 实例依然存在。此时系统直接将该实例切换到前台，不重新创建进程，仅触发 `onNewWant` 等回调，响应速度更快。

用户首选项原理

- **存储机制**：提供非关系型数据库形式的键值对（Key-Value）存储。
- **运作原理**：数据会加载到内存中以实现极速读写。在修改数据后，系统异步地将内存快照持久化到本地 XML 或二进制文件中。适用于保存配置信息、用户偏好设置。

Web组件生命周期

- **关键阶段**：
    - **onControllerAttached**：Web 控制器与组件绑定完成。
    - **onPageBegin**：H5 页面开始加载。
    - **onProgressChange**：页面加载进度更新。
    - **onPageEnd**：页面加载完成。
    - **onRefresh**：页面触发刷新动作。

H5页面和ArkTS交互

- **ArkTS 调用 H5**：使用 `WebviewController.runJavaScript()` 执行 JS 脚本。
- **H5 调用 ArkTS**：通过 `registerJavaScriptProxy()` 注入原生对象到 H5 窗口。H5 端通过约定的对象名直接调用 ArkTS 侧定义的方法。

架构设计视角

业务、应用、数据、技术

- **业务架构**：定义业务战略、组织结构和核心业务流程，是所有架构的源头。
- **应用架构**：明确系统功能的蓝图，包括子系统划分、应用间的交互协议。
- **数据架构**：规划数据资产的存储、分布、流转和治理模型。
- **技术架构**：选择底层技术栈（基础设施、中间件、开发框架），支撑上层应用运行。

整体横向分层抽象，局部纵向贯穿分解

- **横向分层**：将系统分为接入层（网关）、服务层（业务逻辑）、中间件层（缓存/消息队列）和存储层，实现关注点分离。
- **纵向贯穿**：对日志监控、安全防护、配置管理等公共能力进行纵向集成，确保这些能力能覆盖所有层级。

高可用 高并发

大型网站架构演化历程

- **路径**：单机架构 -> 应用与数据分离 -> 引入本地与分布式缓存 -> 应用服务器集群化（负载均衡） -> 数据库读写分离 -> 使用反向代理与 CDN -> 分布式文件系统与 NoSQL -> 业务拆分与服务化（微服务）。

高可用

- **目标**：通过冗余和自动故障切换确保服务持续可用（如 99.99%）。
- **核心策略**：消除单点故障，实现故障自愈。

负载均衡与反向代理

- **反向代理**：隐藏后端服务器，统一接收外部请求。
- **负载均衡**：根据算法（轮询、权重、最少连接等）将请求分发到健康的服务器集群。

隔离

- **机制**：通过线程池隔离、进程隔离或集群隔离，防止某个局部功能的故障或高负载拖垮整个系统（如核心业务与非核心业务隔离）。

限流

- **作用**：在流量暴增时，通过算法（令牌桶、漏桶）限制进入系统的请求量，保护系统不因过载而崩溃。

降级

- **策略**：当系统压力过大或依赖的服务故障时，暂时关闭非核心功能（如评价、推荐），仅保障核心流程（如搜索、下单）正常运行。

超时与重试机制

- **原则**：设置合理的超时时间防止请求堆积；对具有幂等性的接口进行有限次数的重试，以应对网络抖动。

回滚

- **场景**：当新代码上线出现异常或数据库事务执行失败时，迅速恢复到之前的正确状态（版本回滚、事务回滚）。

压测和预案

- **压测**：通过模拟高并发流量找出系统的容量瓶颈。
- **预案**：根据压测结果制定 SLA，编写自动化或手动的故障应对流程。

高并发

- **核心思想**：通过分治、缓存和异步化手段提升系统单位时间内的处理能力。

应用级缓存

- **形式**：本地缓存（Guava/Caffeine）与分布式缓存（Redis）。通过减少对数据库的物理读取来降低响应延迟。

HTTP缓存

- **机制**：利用浏览器缓存、CDN 边缘缓存，减少请求到达应用后端的次数。

连接池

- **对象**：数据库连接池、线程池、HTTP 连接池。通过复用长连接，减少创建和销毁连接带来的高昂系统开销。

异步并发

- **模式**：使用消息队列实现请求的解耦和削峰填谷，允许系统异步处理耗时任务。

扩容

- **垂直扩容**：升级单台机器的硬件配置。
- **水平扩容**：增加服务器节点数量，通过集群化提升整体吞吐量。

队列

- **用途**：作为缓冲区，存储待处理请求。在处理能力不足时进行任务排队，确保请求不丢失。

进程与线程

异步并发 (Promise和async/await)

- **特性**：属于 ArkTS 语言层面的单线程异步机制。通过将 I/O 等待任务挂起，允许主线程继续处理其他 UI 事件，待任务完成后再执行回调。

TaskPool机制和Worker机制

- **TaskPool**：系统自动管理线程生命周期，支持任务优先级排队和负载均衡。适用于大量且耗时较短的独立任务。
- **Worker**：开发者手动维护生命周期，适用于常驻后台的长时任务。Worker 拥有独立的运行环境。

进程模型

- **结构**：应用通常运行在独立的进程中。HarmonyOS 区分应用进程、系统服务进程。应用进程崩溃不会直接导致系统或其他应用受损。

IPC机制

- **通信**：通过 IPC Kit 实现跨进程的数据传输。基于 Binder 或消息序列化技术，支持同步和异步调用。RPC 则进一步支持跨设备的进程间通信。

关注点

秒杀系统

- **挑战**：瞬时流量极高。
- **对策**：前端限流、静态化页面、库存预扣机制、异步写入数据库、系统排队等待。

可视化全链路日志追踪

- **功能**：在分布式微服务架构中，通过 TraceID 将请求在各个节点（网关、服务 A、服务 B、数据库）产生的日志串联，实现故障的快速定位和性能瓶颈分析。

系统工具库

相机开发服务

- **能力**：提供预览、拍照、录像、元数据识别（如人脸检测）。通过 Camera Kit 实现对底层硬件的流式调用。

传感器运作机制

- **模式**：基于订阅机制。应用向系统注册感兴趣的传感器（加速度、陀螺仪），系统在数据变化时通过回调实时推送。

卡片实现原理

- **架构**：元服务（Atomic Service）的一种表现形式。卡片作为 Provider 提供 UI，由系统的 FormManager 统一调度并渲染在桌面上。

卡片渲染原理

- **方式**：ArkTS 卡片支持在后台直接渲染出 UI 描述，由系统 Host 侧负责显示，具有极低的内存和功耗开销。

卡片数据交换

- **机制**：通过 `postCardAction` 发起刷新请求，利用 `LocalStorage` 进行卡片与应用间的数据共享和状态更新。

从微服务到云原生

微服务架构

- **特征**：单一职责、团队自治、去中心化治理。将大型应用拆分为多个可独立部署的小型服务。

Docker、K8s、Istio架构

- **Docker**：解决环境一致性问题，实现轻量级虚拟化。
- **Kubernetes (K8s)**：负责容器的自动编排、扩容和自愈。
- **Istio**：服务网格（Service Mesh），处理微服务间的通信、安全、可观测性和流量治理。

Serverless

- **定义**：无服务器计算。开发者只需关注业务逻辑，无需管理服务器。系统按需自动分配资源并根据流量计费。

云原生

- **愿景**：充分利用云平台的弹性、弹性和自动化能力，实现高可用和极致的按量伸缩。

AI原生

- **范式**：应用从设计之初就以 AI 智能体（Agent）为核心驱动，利用大模型的能力进行任务编排和决策，而非传统的硬编码逻辑。

高级特性

脚本基础流程运行图

- **流程**：脚本加载 -> 编译为字节码 -> VM 运行 -> 结果反馈。在 HarmonyOS 中，JSVM 提供了独立且高性能的脚本执行环境。

UI测试用例

- **工具**：使用 `UiTest` 框架。通过选择器（Selector）查找组件，模拟点击、滑动等交互，验证 UI 状态的正确性。

NAPI对比JNI

- **NAPI**：ArkTS 与 C/C++ 互调的标准接口。设计更现代化，提供了更安全的内存管理和生命周期控制，相比 Android 的 JNI 更简洁且性能损耗更低。

智能体

智能体的组成

- **框架**：大语言模型（大脑） + 感知单元（输入） + 记忆单元（长期/短期） + 规划单元 + 执行工具（手脚）。

Planning

- **定义**：智能体将复杂目标拆解为可执行的子任务序列。

Chain-of-Thoughts

- **逻辑**：思维链。通过显式引导模型逐步推理，显著提升解决复杂逻辑问题的能力。

React

- **模式**：推理与行动结合（Reasoning and Acting）。模型先思考下一步做什么，执行后再根据反馈调整后续计划。

Reflection / Self-Refine

- **机制**：自我反思。智能体审视自己的执行结果，发现错误并进行自我修正。

Subgoal decomposition

- **操作**：子目标分解。将宏大、模糊的任务不断细化，直到成为可以直接调用的工具指令。

RAG

- **全称**：检索增强生成。通过从私有知识库中检索相关文档，辅助大模型生成更准确、不产生幻觉的答案。

MCP

- **协议**：模型上下文协议。标准化 AI 模型连接各种资源、工具和数据的接口，解决孤岛问题。

Timeless的东西

需求的涌现

- **原理**：系统整体的属性（如响应速度、稳定性）不是单一组件能决定的，而是所有组件相互作用后产生的涌现结果。

spec

- **重要性**：软件规格说明书是开发的基石。在不确定的环境中，明确的 spec 是减少沟通成本和技术债务的关键。

现实的复杂度

- **根源**：复杂度来自于业务的非线性、网络环境的不确定性以及多任务并发处理。

层次性skill+知识

- **结构**：知识是静态的，Skill（技能）是动态的。架构师需要具备分层思维，将复杂问题转化为可管理的层次结构。

不确定性问题

- **应对**：通过原型法快速迭代、上线反馈机制，在不确定中寻找确定的用户需求。

各司其职

- **原则**：在微服务和团队架构中，确保每个组件和每个成员都有清晰的边界和职责，避免职能重叠导致的效率低下。

规模化

- **挑战**：当用户规模从万级增长到亿级，架构必须经历从量变到质变的飞跃。

检查智能体

- **闭环**：对智能体的输出建立监控和评估机制，确保其行为符合业务逻辑和安全规范。