自我介绍：

面试官，hr，您好。我是潘锐琦，暨南大学计算机科学系软件工程专业 2023届毕业生。我在校期间绩点年级前15%，获得过一年奖学金，积极参与各项比赛。获得过蓝桥杯算法程序大赛省三等奖。我大学的时候选修过移动端开发课程。

我的上一份工作是广州翼辉信息，我在里面负责处理内核bug，北航无人机项目和爱智app中短视频平台的验证工作。在工作中我取得了优异的成绩，成功解决内核设备树解析错误的bug，并合并到主分支中。北航无人机项目中我作为负责人，成功编写北航无人机项目消息中间件，和组员一起积极沟通，完成北航无人机第一阶段项目。后续则是完成短视频平台的验证工作。我在工作中尽心尽力，责任心强，曾被leader提名公司年度最佳新人。

我拥有1年工作经验，熟悉JAVA 类加载机制 / JVM 内存模型 / 对象的创建，存储和访问原理 / 垃圾回收机制 。熟悉安卓系统运行机制及各底层框架，熟悉View原理，自定义View，事件分发机制，动画机制，跨进程通信等。

日常生活中我热爱学习，喜欢写博客。具有强烈的求知欲、好奇心和进取心。

01.JAVA

1.1 GC源码

- GC机制的触发
  - 内存分配失败：当应用线程尝试分配对象（如 `new Object()`）时，若当前堆空间不足，会进入 `Heap::AllocateInternalWithGc()` 方法，触发 GC
    - ```
      mirror::Object* Heap::AllocateInternalWithGc(...) {
        // 尝试分配内存
        obj = TryToAllocate(...);
        if (obj == nullptr) {  // 分配失败
          CollectGarbageInternal(...);  // 触发GC
        }
        return obj;
      }
      ```
  - 根据堆内存状态选择 GC 类型（分代回收策略）：
      ```

```
void Heap::CollectGarbageInternal(GCType type, GcCause cause) {
  if (type == kGcTypeSticky) {  // 年轻代GC
    young_collector_->Run();    // 使用 Generational Concurrent
Copying
  } else {                      // 全堆GC
    concurrent_copying_->Run(); // 使用 Concurrent Copying GC
  }
}
```

- 并发标记阶段：
  - STW和工具整理：
    - **比喻**：大扫除前的准备工作，想象你是一个清洁队长（GC），要给整个房子（堆内存）做大扫除（垃圾回收）。但房子里有很多人（Mutator 线程）在活动。
    - **STEP 1 暂停所有人（STW）**

      ```
      Locks::mutator_lock_->ExclusiveLock(self);
      ```

      → 你吹哨子让所有人暂停动作，避免他们一边扔垃圾一边打扫（保证内存状态稳定）。
    - **STEP 2 准备工具**

      ```
      mark_stack_ = heap_->GetMarkStack();
      mark_stack_->Reset();
      is_marking_ = true;
      ```

      asdfsdaf→ 你拿出一个「待检查物品清单」（mark_stack），*并挂上「正在打扫」的牌子（is_marking*）。mark_stack中初始化有GC_root作为灰色标签。
    - **STEP 3 恢复活动**

      ```
      Locks::mutator_lock_->ExclusiveUnlock(self);
      ```

      → 大家继续活动，但你已经开始打扫（并发标记）。

      关键点：STW 时间非常短（仅吹哨子的时间），实际打扫工作和大家的活动是并行的。
  - 三色标记法：
    - **比喻**：整理房间的优先级标签
      - **白色标签（未处理）**：所有物品初始状态，暂时当作「垃圾」候选。
      - **灰色标签（待处理）**：放在「待检查清单」（mark_stack_）里的物品，需要检查它内部是否引用了其他物品。
      - **黑色标签（已处理）**：确认是「有用物品」，且其内部引用都已检查完毕。
    - **标记过程**：

```
void ProcessMarkStack() {
  while (!mark_stack_->IsEmpty()) {
    ObjectReference ref = mark_stack_->PopBack(); // 取出一个灰色物品
    for (每个子引用) {                          // 检查它内部包含的物品
      if (子物品是白色) {
        标记为灰色;                             // 改为灰色
        Push到mark_stack_;                      // 加入待检查清单
      }
    }
    标记为黑色;                                 // 此物品处理完毕
  }
}
```

示例流程:

- 从根对象（如全局变量）出发，标记为灰色，压入栈
- 弹出灰色对象A，检查其引用了对象B和C
  - B是白色 → 标灰，入栈
  - C是白色 → 标灰，入栈
  - A处理完 → 标黑
- 弹出B，检查无引用 → 标黑
- 弹出C，检查引用D → D标灰，入栈
- 最终所有可达对象都会被标黑，白色对象即垃圾

- 写屏障:
  - 考虑一种情况，当并发标记时，你（GC）正在标记对象A为黑色（已处理），某人（Mutator 线程）偷偷把对象A的引用指向了一个新的白色对象X
  - 此时会触发写屏障
    - ```
      void WriteBarrier::Mark(Object* obj) {
        if (正在标记阶段 && 对象未被标记) {
          标记为灰色;
          压入标记栈;
        }
      }
      ```
    - 每当有人修改引用关系（`objA.field = objX`），写屏障会强制检查:
      - 如果 X 是白色 → 立即标灰，加入待检查清单
      - 保证不会漏掉这种「新增的引用」
  - **STW 初始化**：快速冻结世界，建立初始标记状态
  - **三色标记**：以「灰色清单」为驱动，波浪式推进标记
  - **写屏障**：在并发阶段实时监控引用变化，避免漏标
  - 这种组合使得 ART 的 GC 能够:
    - 极短暂停（通常 1-5ms）
    - 高并发效率（标记与 Mutator 线程并行）
    - 安全可靠（写屏障兜底）

- 引用处理
      ```

- Java 有四种引用类型，GC 对它们的处理策略不同：

引用类型	比喻	回收策略
强引用	必需品（如冰箱里的食物）	绝不回收
软引用	可丢弃的备用物资	内存不足时回收
弱引用	临时借用物品	本轮GC立即回收
虚引用	物品领取通知单	回收后通知（用于跟踪）

```
void ProcessReferences() {
  // 1. 处理软引用：根据内存压力决定是否回收
  soft_reference_queue_.ForwardSoftReferences(&visitor);

  // 2. 处理弱/虚引用：直接回收未被标记的对象
  weak_reference_queue_.ClearWhiteReferences(...);
  phantom_reference_queue_.ClearWhiteReferences(...);

  // 3. 将回收的引用加入队列（开发者可通过 ReferenceQueue 感知）
  EnqueueClearedReferences();
}
```

  - 软饮用：若堆空闲 50%，`SoftRefLRUPolicyMSPerMB=1000ms`，则保留最近 500ms 内访问过的软引用
  - **弱引用与虚引用**：如果引用的对象未被标记（白色），直接加入待清理队列，若拥有finalize方法

    ```
    if (object->HasFinalizer()) {    // 如果对象有 finalize() 方法
      AddToFinalizationQueue(object); // 延迟到下次GC回收
    }
    ```

- 空间压缩：
  - 存活对象定位：

    ```
    void ComputeLiveWords() {
      for (每个内存区域 region) {
        for (region 中的每个存活对象 obj) {
          live_words_bitmap_.Set(obj的偏移量); // 标记存活位置
        }
      }
    }
    ```

    - **位图作用**：快速定位哪些内存块需要保留（类似书架上的书签）
  - 对象移动：

    ```
    void MoveObjects() {
      for (每个存活对象 obj) {
        new_addr = 计算新位置;         // 向堆起始端靠拢
        memcpy(new_addr, obj, size); // 拷贝对象
        UpdateReferences(obj, new_addr); // 更新所有指向它的指针
      }
    }
    ```

```
void UpdateReferences(Object* old, Object* new) {
  atomic_store_release(&pointer, new); // 原子写入
  RememberOldAddress(old, new);        // 维护旧地址映射
}
```

- 原子操作：保证其他线程看到的是更新后的指针。

- 并发与线程管理
  - **实现原理**：
    线程在安全点会主动检查是否需要暂停（类似红绿灯）：

```
void Thread::CheckSuspend() {
  if (suspend_count_ > 0) {  // 收到暂停信号
    EnterSuspendedState();   // 进入暂停状态
  }
}
```

02.Andriod

2.1 同步消息屏障

- 消息机制：同步屏障（SyncBarrier）如何实现消息优先级调度？
  - 本质是向消息队列插入一条特殊消息（`target=null`），消息队列会跳过所有同步消息（即使其时间已到），仅处理异步消息。
  - 为什么要有同步消息屏障？：**确保高优先级任务及时执行**：例如在UI渲染时，系统通过同步屏障优先处理VSYNC信号触发的界面绘制消息，避免因同步消息堆积导致界面卡顿。
- 执行过程：
  - 我们通过queue.postSyncBarrier()，插入一个`target`字段为`null`的Message（屏障标记）。
  - ```
    // MessageQueue.java
    int postSyncBarrier() {
        Message msg = Message.obtain();
        msg.when = SystemClock.uptimeMillis();
        msg.arg1 = token;  // 唯一标识符
        synchronized (this) {
            // 插入屏障到合适位置（按时间排序）
            Message prev = null;
            Message p = mMessages;
            while (p != null && p.when <= msg.when) {
                prev = p;
                p = p.next;
            }
            // 插入操作...
        }
        return token;
    }
    ```
  - 在`MessageQueue.next()`中，若检测到当前消息是同步屏障（`target == null`），则跳过后续所有同步消息，仅处理异步消息。
    ```

```
Message next() {
    for (;;) {
        //...
        synchronized (this) {
            Message prevMsg = null;
            Message msg = mMessages;
            if (msg != null && msg.target == null) {
                // 遇到同步屏障，寻找下一个异步消息
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null && !msg.isAsynchronous());
            }
```

- 标记异步消息：Message通过`setAsynchronous(true)`标记为异步。
- 移除方式：**移除方式**：调用`MessageQueue.removeSyncBarrier(token)`，根据插入时返回的`token`移除屏障。

2.2 `ActivityThread` 和 `AMS` 如何协作启动 Activity?

- `ActivityThread` 和 `AMS` 如何协作启动 Activity?
  - 启动请求发起：当前 Activity 调用 `startActivity()`
    - ```
      // Activity.java
      startActivity() → startActivityForResult() →
      Instrumentation.execStartActivity() →
      // 通过Binder调用AMS的startActivity
      ActivityManager.getService().startActivity()
      ```
    - `ActivityManager.getService()` 获取 AMS 的 Binder 代理对象（`IActivityManager`），通过 `transact()` 将启动请求发送到 AMS。
  - AMS处理请求：
    - **权限与合法性校验**：AMS 检查目标 Activity 是否存在、权限是否满足等。
    - **创建ActivityRecord**：记录 Activity 信息，并确定目标进程：
      - **若目标进程未启动**：通过 `Process.start()` 请求 Zygote 创建新进程。
      - **若进程已存在**：直接复用。
    - **暂停当前Activity**：若需要，AMS 通知当前 Activity 进入 `onPause()`。
  - 应用进程初始化：我们前面只是创建了进程，并没有创建Activity啊?
    - Zygote 创建进程后，执行 `ActivityThread.main()`。，创建一个主线程
      - ```
        // ActivityThread.java
        public static void main(String[] args) {
            Looper.prepareMainLooper();
            ActivityThread thread = new ActivityThread();
            thread.attach(false); // 关键：绑定到AMS
            Looper.loop();
        }
        ```

- **绑定到AMS**：调用IActivityManager 代理对象的 `attach()` 方法，将应用进程的 `ApplicationThread`（Binder对象）注册到 AMS：
  - ```
    // ActivityThread.java → attach()
    final IActivityManager mgr = ActivityManager.getService();
    mgr.attachApplication(mAppThread); // mAppThread是ApplicationThread实例
    ```
- AMS触发Activity启动
  - **发送启动指令**：AMS 通过 `ApplicationThread` 的 Binder 代理，调用 `scheduleLaunchActivity()`：
    - ```
      // ApplicationThread.java
      public void scheduleLaunchActivity(Intent intent, IBinder token,
          int ident, ActivityInfo info, Configuration curConfig, ...) {
          // 封装参数为ActivityClientRecord
          ActivityClientRecord r = new ActivityClientRecord();
          r.token = token;
          r.activityInfo = info;
          r.intent = intent;
          // 发送消息到主线程Handler
          sendMessage(H.LAUNCH_ACTIVITY, r);
      }
      ```
  - **应用进程处理启动（ActivityThread主线程）**
    - 主线程的 `H`（Handler）收到 `LAUNCH_ACTIVITY` 消息，调用 `handleLaunchActivity()`：

      ```
      // ActivityThread.java
      private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {
          // 1. 创建Activity实例
          Activity activity = performLaunchActivity(r, customIntent);
          if (activity != null) {
              // 2. 执行生命周期：onCreate(), onStart(), onResume()
              handleResumeActivity(r.token, false, r.isForward, ..., r.activity);
          }
      }
      ```
    - **创建Activity**：通过反射调用 `Activity` 的构造函数，并关联 `Context`。
    - **生命周期调用**：依次触发 `onCreate()` → `onStart()` → `onResume()`。
- `ApplicationThread` 的作用是什么?
  - **跨进程通信的桥梁**
    AMS（运行在`system_server`进程）通过 `ApplicationThread` 的 Binder 接口，向应用进程发送指令（如启动 Activity、绑定 Service、处理广播等）。
  - **封装 AMS 的请求**
    将 AMS 的跨进程调用封装为消息（`Message`），通过 `Handler` 发送到应用主线程执行，**确保 UI 操作和生命周期的线程安全。**

- 为何需要 Binder 线程到主线程的切换?
  - **Binder 调用默认在 Binder 线程池执行**：当 AMS 通过 `ApplicationThread` 的 Binder 接口调用应用进程时，请求会进入应用进程的 Binder 线程池（非主线程）。
  - **UI 操作的线程限制**：Android 要求 UI 操作（如更新控件、处理生命周期）必须在主线程执行，直接在其他线程操作 UI 会抛出 `CalledFromWrongThreadException`。

2.3 启动优化

- 冷启动：冷启动优化的关键手段？如何检测启动耗时?
- 冷启动优化的核心目标是减少从用户点击应用图标到首帧渲染完成的时间。那么请问，整个过程是什么样的？它分为系统阶段和程序员写的代码阶段
  - 首先，应用进程的创建，到ActivityThread方法的执行，到主线程的创建，都是系统阶段。到了Application的onCreate，主Activity的onCreate阶段是我们可以控制的了。
- 那么我们该怎么处理呢，常规优化是什么样子?
  - 我们首先可以减少不必要的三方库的加载。
  - 或者开个工作线程后台执行数据的初始化工作。
  - 我们可以减少Activity的嵌套层级和显示逻辑，只采用简单的SplashActivity进行启动页的显示。
- 除此之外呢？我们有没有更优雅的形式，更底层的形式？我们可以优化代码和资源的物理布局。
  - 我们知道代码执行，首先是从内存中取出一个块放到cache中。假如，他们很分散，cache命中率会很低，我们就会cache miss，不断访问内存。
  - 所以我们可以通过**PGO（Profile Guided Optimization）** 优化代码顺序，将高频执行的启动代码集中在连续内存页，减少 **Cache Miss**。
  - 我们还可以优化Dex文件的结构，
    - **Dex 文件的结构与问题**
      - **Dex 文件**：Android 将 Java/Kotlin 代码编译为 Dex（Dalvik Executable）文件，包含类、方法、字段等信息。
      - **类加载顺序**：默认情况下，Dex 文件中的类按编译顺序排列，若启动阶段用到的类分散在 Dex 不同位置，会导致：
        - **磁盘 I/O 效率低**：需要多次随机读取才能加载所有启动类。
        - **内存碎片化**：加载后的类在内存中不连续，增加 CPU 缓存失效（Cache Miss）的概率。
    - **优化原理：类重新排列**
      - **目标**：将启动阶段高频使用的类集中在 Dex 文件的前部，并尽量在内存中连续存放。
      - **效果**：
        - **减少 Page Fault**：连续读取类数据，减少磁盘寻址次数。
        - **提升缓存命中率**：CPU 缓存更可能命中连续内存块。
  - 还有呢，`read()` 调用需将文件数据从 **磁盘** → **内核缓冲区** → **用户空间缓冲区**，存在两次拷贝。我们通过mmap 加载 Dex，替代传统的 `read`，这样我们访问只需要一次了，对吧。
  - 我还听过一种方法，但是我没去实际了解过。就是大页内存，

- **传统分页（4KB）**：
  操作系统将物理内存划分为 4KB 的页，通过 **页表** 记录虚拟地址到物理地址的映射。
- **TLB（Translation Lookaside Buffer）**：
  CPU 缓存页表项的硬件组件，加速地址转换。若 TLB 未命中（TLB Miss），需从内存加载页表项，增加延迟。
- ** 大页内存的优势**
  - **大页尺寸**：2MB 或 1GB（对比传统 4KB）。

```
- **减少 TLB Miss**：
  单页覆盖更大内存范围，相同 TLB 容量下可映射更多物理内存。

  ```
  示例：TLB 有 512 项
  - 4KB 页 → 覆盖 512 * 4KB = 2MB
  - 2MB 页 → 覆盖 512 * 2MB = 1GB
  ```

- 我记得底层的surfaceFlinger会采取这种措施，他需要内核的优化。
```

2.4 什么是ANR？如何定位

- ANR如何定位：

2.5 内存泄漏如何检测

2.6 如何监控主线程卡顿?

- 卡顿优化：
  - 如何监控主线程卡顿？Choreographer 的作用?
    - 主线程卡顿的根源是 **主线程（UI 线程）的消息处理耗时过长**，导致无法及时响应 UI 渲染（16.6ms/帧）。以下是一种核心监控方案：
    - 主线程的 `Looper` 在处理每条消息（`Message`）前后会打印日志，通过替换 `Looper` 的 `Printer`，计算消息处理的耗时。**实现步骤**：**Hook Looper 的 Printer**：

      ```
      // 获取主线程的 Looper
      Looper.getMainLooper().setMessageLogging(new Printer() {
          private long mStartTime = 0;
          @Override
          public void println(String log) {
              if (log.startsWith(">>>>> Dispatching")) {
                  mStartTime = SystemClock.uptimeMillis();
              } else if (log.startsWith("<<<<< Finished")) {
                  long cost = SystemClock.uptimeMillis() - mStartTime;
                  if (cost > 16) { // 阈值设为 16ms（一帧时间）
                      reportBlock(cost); // 上报卡顿
                  }
              }
          }
      });
      ```

- `Looper` 类提供了 `setMessageLogging(Printer printer)` 方法，允许设置一个日志打印机。该打印机在 **消息处理前** 和 **消息处理后** 会输出特定日志，用于调试消息处理流程。
- 用户提供的代码通过 **替换主线程 Looper 的 Printer**，监控每条消息的处理耗时：
- 当检测到日志以 `>>>>> Dispatching` 开头时，记录当前时间戳（`mStartTime`）。
- 当检测到日志以 `<<<<< Finished` 开头时，计算耗时（当前时间 - `mStartTime`）。
- 若耗时超过阈值（如 16ms），触发卡顿上报（`reportBlock`）。

2.4