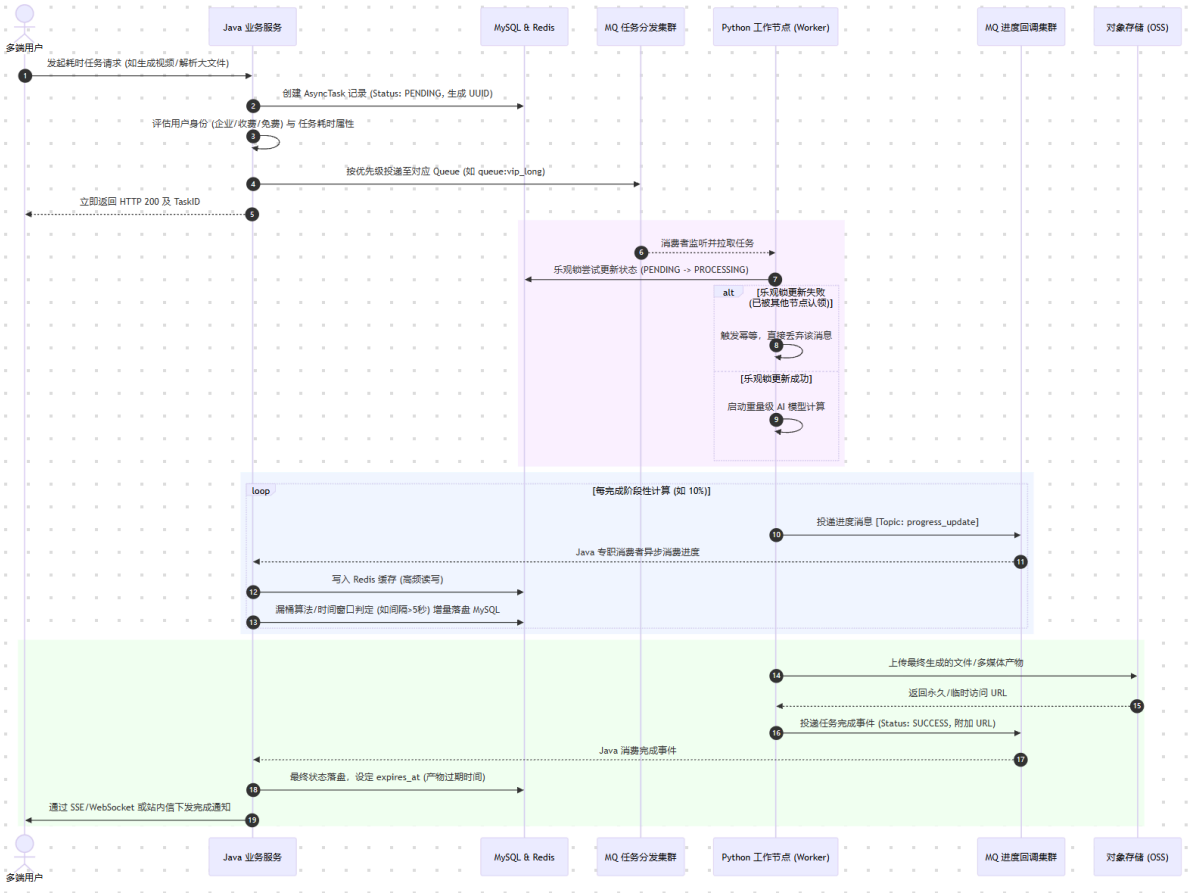
异步长耗时任务架构



2. 核心机制落地规范 (SOP)

2.1 任务分级与队列隔离 (Priority Routing)

为了防止免费用户的海量短任务阻塞了付费企业用户的长时任务，Java 端在投递任务前必须进行**路由判定**，将任务分发至隔离的 Kafka/RabbitMQ 队列：

- **Queue: enterprise_exclusive (企业独享)**：最高优先级。绑定专门的算力集群，保证 SLA。
- **Queue: vip_short (收费用户-短任务)**：如单页 OCR、简单数据汇编。优先调度，快速流转。
- **Queue: vip_long (收费用户-长任务)**：如高清视频生成、百页 PDF 向量化。
- **Queue: free_default (免费用户)**：利用空闲算力执行，允许较长的排队等待时间。

2.2 幂等性与防重复执行 (Idempotency)

网络抖动极易导致 MQ 消息重复投递。Python 节点在拉取到任务后，**严禁直接开始计算**，必须先通过 Java 提供的内部 API 或直接连库执行**乐观锁防重**校验：

SQL

- `- Python 触发状态流转时的底层 SQL 逻辑`
  `UPDATE async_task`
  `SET status = 'PROCESSING', version = version + 1WHERE task_id = #{taskId} AND status = 'PENDING'AND version = #{oldVersion};`

如果影响行数为 0，说明任务已被其他 Worker 抢占或已取消，Python 节点立即 `return` *结束流程。*

2.3 进度回传与数据库减压 (DB Protection)

Python 节点**禁止通过同步 HTTP/gRPC 直连 Java 更新进度**。所有进度更新通过 `progress_update` MQ 队列异步解耦。
Java 侧消费该队列时，采取**"Redis 高频更新 + MySQL 降频落盘"**策略：

- **高频写缓存：** 进度第一时间写入 Redis (如 `Key: task_progress:{task_id}`)，供前端随时通过短轮询接口高速读取。
- **降频写 DB：** Java 消费者判断，仅当 `(当前时间 - MySQL上次更新时间) > 5秒` 或者 `progress == 100` 时，才执行 MySQL 的 `UPDATE` 语句。极大降低数据库写压力。

2.4 死锁检测与自动纠错 (Deadlock Watchdog)

为应对 Python 节点 OOM 崩溃导致任务永久卡死在 `PROCESSING` 状态，Java 侧必须部署后台调度任务（`@Scheduled`）：

- **扫描规则：** 每 5 分钟扫描一次 MySQL，查找 `status = PROCESSING` 且 `update_time < NOW() - 2小时` 的僵尸任务。
- **干预策略：** 视业务要求，将其状态强制扭转为 `FAILED` 并附注"执行超时"，或重新将其投递至重试队列。

2.5 产物清理与成本控制 (Cost Control)

AI 任务（特别是音视频）会产生巨大的 OSS 存储费用。系统采用生命周期管理：

- 任务标记为 `SUCCESS` 时，必须同时写入 `expires_at` (例如：免费用户产物保留 3 天，VIP 保留 30 天)。
- Java 定时任务每日凌晨扫表：寻找 `expires_at < NOW()` 且 `cleanup_status = 0` 的记录。

- 调用 OSS SDK 物理删除文件，随后将 `cleanup_status` 置为 1。

二、 长效异步任务的状态流转与持久化设计 (超长耗时任务)

针对“AI 执行异步任务（如音视频制作、几百页 PDF 的知识解析）”场景，HTTP 连接超时不可避免，系统采用**全异步事件驱动与最终一致性架构**。

在这个场景中，Java (MySQL) 是状态流转与调度的唯一真理源（Source of Truth），Python 是无状态、可随时扩缩容的执行工人（Worker）。

1. 任务创建与动态路由 (Task Creation & Priority Routing)

- 用户发起耗时任务请求。Java 服务接收请求，在 MySQL 的 `AsyncTask` 表中创建记录。
- 核心字段包括：`TaskID`, `UserID`, `TaskType`, `Status` (PENDING / PROCESSING / SUCCESS / FAILED), `Progress` (0-100), `Result_URL`, 以及用于防并发的 `Version`（版本号）和控制成本的 `Expires_At`（产物过期时间）。
- Java 根据用户身份（VIP/免费）与任务预估耗时，将带有 `TaskID` 的消息投递至 **MQ 分层隔离队列**（如 `queue:vip_long`, `queue:default`），实现紧急任务与普通任务的物理隔离。随后立即给前端返回 HTTP 200 及 `TaskID`。

2. 幂等认领与防重执行 (Idempotent Worker Processing)

- Python FastAPI 底层的消费者从 MQ 拉取任务。
- 为防止网络抖动导致的 MQ 消息重复消费，Python 节点在启动重度 AI 计算前，必须先进行**乐观锁校验**（如：`UPDATE async_task SET status='PROCESSING', version=version+1 WHERE task_id=X AND status='PENDING'`）。
- 若更新失败（影响行数为 0），说明该任务已被其他节点抢占，Python 触发幂等逻辑直接丢弃该消息；若更新成功，则正式启动 AI 推理计算，并利用 MQ 手动 ACK 机制防止任务在宕机时丢失。

3. 异步状态回传与数据库保护 (Async State Synchronization)

- **解耦汇报：** Python 在执行过程中（如每推进 10%），不再通过同步 API 直连 Java，而是将进度消息投递至专门的 MQ 进度队列（`Topic: progress_update`）。
- **高频写缓存与降频落盘：** Java 专职消费者监听进度队列。收到进度后，第一时间更新至 **Redis**（满足前端高频查询需求）；同时采用时间窗口或漏桶机制（如：距离上次更新大于 5 秒，或进度达到 100%），才执行 **MySQL** 落盘，极大程度保护数据库免受写压垮。
- **完成回调与生命周期控制：** 任务完成时，Python 将产物（如 MP4）传到 OSS，将 OSS URL 及 SUCCESS 状态推入 MQ。Java 消费后更新最终状态，并根据业

务逻辑为产物设定清理过期时间（`Expires_At`）。

4. 异常兜底与死锁检测 (Deadlock Detection & Compensation)

- Java 端部署定时调度任务（Watchdog），周期性扫描 MySQL 中长期处于 `PROCESSING` 状态（如超过 2 小时未更新）的“僵尸任务”。
- 对这类由 Python 节点 OOM 或网络分区造成的卡死任务，系统自动将其标记为 `FAILED`，或重新投入重试队列，保障系统状态机的自我修复能力。

5. 前端状态获取 (Frontend Status Sync)
此时前端有两套丝滑的展现方式配合：

- **方式 A（长链接主动推送 - 推荐）：** 如果用户一直在线，Java 在监听到 MQ 完成事件并更新 MySQL 为 SUCCESS 的同时，通过用户的全局 WebSocket 或 SSE 通道，主动向前端下发系统级通知：“您的视频已生成完毕”。
- **方式 B（高频缓存轮询 - 兜底）：** 如果用户刷新了页面，前端重新进入时，可通过 `GET /api/tasks?userId={id}` 接口获取列表。此接口优先从 **Redis** 读取实时进度，提供极低延迟的列表状态渲染。

三、 Java 与 Python 跨语言通信基座总结 (Enterprise Edition)

综合以上两大核心业务场景（智能问答与耗时任务），为了充分发挥 Java 在高并发调度与事务管理上的优势，以及 Python 在 AI 计算上的敏捷性，系统内部正式确立以下**三种标准化的跨语言通信规范**：

1. 实时流式问答 (低延迟、打字机体验)

- **通信链路：** `Java -> (异步指令) -> Python -> (流式追加 Redis Streams) -> Java (阻塞消费) -> (SSE 透传) -> Frontend`
- **架构约束：** 彻底弃用传统 HTTP 等待与易丢消息的 Redis Pub/Sub。Python 侧化身无状态生产者，逐字向 **Redis Streams** 写入（`XADD`）；Java 侧利用轻量级线程（如虚拟线程）进行阻塞监听（`XREAD`）。实现零丢失、抗闪断、全异步的打字机透传体验。

2. 长耗时异步计算 (高可靠、最终一致性)

- **通信链路：** `Java -> (投递 Task MQ) -> Python执行 -> (投递 Progress MQ) -> Java (缓冲消费) -> DB 持久化`
- **架构约束：** 严禁 Python 在高耗时任务中通过 HTTP/gRPC 直连回调 Java 刷新进度。双端必须通过 **Kafka/RabbitMQ 进行双向解耦**。Python 产出状态推入 MQ，Java 消费后执行“高频写 Redis 缓存 + 降频写 MySQL 落盘”策略，极大限度保护核心数据库免受流量冲击。

3. 内部强一致同步查询 (低延迟、强约束)

- **适用场景：** 如 Python 在执行敏感工具前，需向 Java 实时校验该用户的资产权限；或 Python 查询系统的某项全局配置。
- **通信链路：** `Python <-> (gRPC) <-> Java`
- **架构约束：** 针对必须等待结果的同步调用，全面采用 **gRPC** 替代传统 REST API。借助底层 HTTP/2 的多路复用特性提升并发性能；同时依靠 **Protobuf (Protocol Buffers)** 强制约束双端的强类型数据契约，从根源上杜绝跨语言联调时常见的字段拼写错误与类型不匹配隐患。