JobRun 的作用

1. **任务实例化**:

- **Job Definition Store** 中保存的是任务的定义（metadata），而 **JobRun** 是任务的具体执行实例。
- 每次任务触发时，Scheduler 会创建一个新的 **JobRun** 记录，表示该任务的一次运行。

2. **Worker 使用**:

- Worker 从消息队列中消费任务时，会根据 **JobRun** 的记录执行任务。
- Worker 会更新 **JobRun** 的状态（如 RUNNING、SUCCESS、FAILURE）以及相关的日志信息。

3. **监控和追踪**:

- **JobRun** 记录用于监控任务的执行状态。
- 系统可以通过 **JobRun** 的状态（如 PENDING、RUNNING、TIMEOUT）来判断任务是否正常完成。
- 如果任务失败或超时，系统可以触发重试或警报机制。

JobRun 的生命周期

1. **Scheduler 创建 JobRun**:

- 当 Scheduler 触发任务时，会在 **JobRun Store** 中创建一个记录，初始状态为 PENDING。

2. **Worker 更新状态**:

- Worker 消费任务后，将 **JobRun** 状态更新为 RUNNING。
- 执行完成后，状态更新为 SUCCESS 或 FAILURE，并记录相关日志。

3. **Monitoring Service 检查**:

- 如果任务长时间处于 RUNNING 状态，Monitoring Service 会检查是否超出 SLA。
- 超时的任务会被标记为 TIMEOUT，并触发警报或重试机制。

4. **Job Definition Store**:

- 保存任务的元数据（如任务类型、触发时间、参数等）。
- 任务定义通过用户或API创建。

5. **Scheduler Service**:

- 定时触发任务，根据预定义的时间或事件。
- Use set interval or timer in js or python or Airflow

- 创建一个新的 JobRun 条目，状态为 PENDING。
- 发布任务运行消息到消息队列（如 Kafka、RabbitMQ）。

6. **Worker Pool**:

- 消费消息队列中的任务运行消息。
- 更新 JobRun 状态为 RUNNING。
- Use kafka or rabbitMQ
- 异步执行任务逻辑（如调用外部服务或运行脚本）。
- 执行完成后更新 JobRun 状态为 SUCCESS 或 FAILURE。

7. **JobRun Store**:

- 存储任务运行的状态和日志（如开始时间、结束时间、错误信息等）。

8. **Monitoring Service**:

- 定期扫描 RUNNING 状态的任务。
- 如果任务超过预定义的 SLA（服务级别协议），将其标记为 TIMEOUT。
- 触发警报或重试机制。

9. **UI/API for Logs and Status**:

- 提供用户界面或 API，供用户查询任务状态和日志。

10. **Notification/Alert Systems**:

- 发送通知或警报（如邮件、短信、Slack消息）以告知任务状态或异常。

11. **Scheduler 定期扫描 Job Run Table**:

- Scheduler 每隔固定时间（例如每分钟）扫描 Job Run Table。
- 检查所有任务的 next_run_time 是否符合当前时间。
- 如果任务到点需要执行，Scheduler 将任务发布到消息队列。

12. **Worker 监听消息队列并消费任务**:

- Worker 持续监听消息队列，消费任务消息。
- 根据任务的参数和逻辑执行任务。

13. **Worker 更新 Job Run Table 的状态**:

- Worker 执行任务后，将任务的状态更新到 Job Run Table。
- 状态可以是 SUCCESS、FAILURE 或 TIMEOUT。
- 同时记录任务的运行日志（如开始时间、结束时间、错误信息等）。

14. **Job Run Table 作为日志存储**:

- Job Run Table 中的记录不仅用于任务调度，还可以作为任务运行的日志存储。
- 系统可以通过查询 Job Run Table 查看任务的历史运行状态和日志。

1. Task Scheduler

15. **职责**:

- 负责生成任务（如定时任务、用户触发的任务）。
- 将任务推送到消息队列。
- 定期扫描数据库，检查需要重试或超时的任务。

16. **实现**:

- 可以使用定时器（如 Cron）或事件驱动的方式触发任务生成。
- 将任务的元数据（如任务 ID、参数、优先级）写入消息队列。

2. Message Queue

4. **职责**:

- 存储任务并将其分发给 Worker。
- 支持任务重试机制（如未确认的任务重新入队）。

5. **实现**:

- 使用 RabbitMQ、Kafka 或 AWS SQS 等消息队列。
- 配置任务的重试策略（如最大重试次数、延迟重试）。

3. Worker

- **职责**:
- 从消息队列中消费任务。
- 执行任务逻辑（如调用外部服务、处理数据）。

- 更新任务状态到数据库（如 RUNNING、SUCCESS、FAILED）。
- **实现**:
- 分布式部署，支持水平扩展。
- 每个 Worker 独立监听队列，确保高可用性。
- 设置超时时间，防止长时间运行的任务阻塞。

4. Database

- **职责**:
- 存储任务的元数据、状态、运行日志和历史记录。
- 提供查询接口，供用户查看任务状态和日志。
- **表设计**:
- **Jobs Table**: 存储任务的基本信息（如任务 ID、类型、参数、状态）。
- **Job Runs Table**: 存储任务的运行记录（如开始时间、结束时间、错误信息）。
- **Logs Table**: 存储任务的详细日志。

任务生命周期

1. **任务创建**:
- Scheduler 生成任务并推送到消息队列。
- 在数据库中记录任务的初始状态（PENDING）。
2. **任务分发**:
- 消息队列将任务分发给空闲的 Worker。
- Worker 从队列中拉取任务并开始执行。
3. **任务执行**:
- Worker 执行任务逻辑。
- 在任务开始时更新状态为 RUNNING。
- 执行完成后更新状态为 SUCCESS 或 FAILED，并记录日志。
4. **任务重试**:
- 如果 Worker 执行任务失败或超时，消息队列会将任务重新入队。
- Scheduler 或 Worker 根据重试策略决定是否继续重试。
5. **任务查询**:
- 用户可以通过查询数据库查看任务的状态和运行日志。

扩展功能

1. **任务优先级**:
- 在消息队列中设置优先级队列，高优先级任务优先分发给 Worker。

2. **任务超时处理**:
- Worker 设置超时时间，超时后将任务状态更新为 TIMEOUT。
- Scheduler 定期扫描超时任务并重新入队。

3. **动态扩展**:
- 使用容器化（如 Docker）和编排工具（如 Kubernetes）动态扩展 Worker 数量。
- 根据队列长度或 Worker 的负载自动调整规模。

4. **监控与告警**:
- 监控任务的执行状态（如失败率、超时率）。
- 通过告警系统（如 Prometheus + Alertmanager）通知管理员。

技术选型

组件	**技术选型**
Task Scheduler	Python (Celery, APScheduler) 或 Java (Quartz)
Message Queue	RabbitMQ, Kafka, AWS SQS
Worker	Python, Java, Go 等语言实现
Database	PostgreSQL, MySQL, MongoDB
容器化与扩展	Docker, Kubernetes
监控与告警	Prometheus, Grafana