线程池用法

参考了 线程池

用法：在ThreadPollExecutorConfig有许多不同线程池的配置

eg：

```java
@Bean("commonExecutor")
public ExecutorService commonExecutor() {
    ExecutorService threadPoolExecutor = new ThreadPoolExecutor(
        Math.max(Runtime.getRuntime().availableProcessors() * 2, 4),
        Math.max(Runtime.getRuntime().availableProcessors() * 4, 16),
        60L,
        TimeUnit.SECONDS,
        new SynchronousQueue<>(),
        new ThreadFactoryBuilder().setNameFormat("common-commonExecutor-%d").build(),
        new ThreadPoolExecutor.CallerRunsPolicy()
    );
    return threadPoolExecutor;
}
```

1. corePoolSize 配置

```java
corePoolSize = Math.max(Runtime.getRuntime().availableProcessors() * 2, 4)
```

- Runtime.getRuntime().availableProcessors() 获取 CPU 核心数。
- availableProcessors() * 2 设置核心线程数为 CPU 核心数的 2 倍，以提高并发能力。
- Math.max(..., 4) 确保核心线程数至少为 4，即使在单核或双核机器上，仍然能保证一定的并发能力。

为什么这么设置：

- corePoolSize 决定了线程池中长期维持的活跃线程数。
- 对于 **CPU 密集型任务**，适合将核心线程数设置为与 CPU 核心数相等；而对于 **IO 密集型任务**，则适合将核心线程数设置为 CPU 核心数的 2 倍甚至更多。
- 这样的配置适应了不同硬件环境，确保多核 CPU 可以充分利用线程池，提高吞吐量。

2. maximumPoolSize 配置

```java
```

```
maximumPoolSize = Math.max(Runtime.getRuntime().availableProcessors() * 4, 16)
```

- availableProcessors() * 4 设置最大线程数为 CPU 核心数的 4 倍，提高峰值时的吞吐量。
- Math.max(..., 16) 确保最大线程数至少为 16，适用于小型服务器或本地开发环境。

为什么这么设置：

- maximumPoolSize 允许线程池在高并发压力下动态扩容。
- 适用于处理短时高峰流量，避免任务堆积过多。
- 但过大的 maximumPoolSize 可能导致线程上下文切换过于频繁，反而影响性能。

一般经验：

- **CPU 密集型任务**：最大线程数 ≈ CPU 核心数 + 1
- **IO 密集型任务**：最大线程数 ≈ CPU 核心数 * 2 ~ 4

这个配置更适用于 **IO 密集型场景**（如数据库查询、HTTP 请求等）。

3. keepAliveTime 配置

Java 复制代码

```
keepAliveTime = 60,
TimeUnit.SECONDS
```

- 当线程数超过 corePoolSize 时，空闲的多余线程在 60 秒后会被回收。

为什么设为 60 秒：

- 避免线程长期占用资源，节省系统开销。
- 适用于短时高峰负载，避免线程频繁销毁和创建。
- 1 分钟的空闲时间可以确保短期内的波动流量仍然能快速被处理。

4. 使用 SynchronousQueue 提高并发

这样会直接让新任务交给线程执行，如果没有空闲线程，会 **立刻创建新线程**，直到达到 maximumPoolSize。

5. CallerRunsPolicy拒绝策略

如果任务超出 maximunPoolSize = 200，**CallerRunsPolicy** 拒绝策略会让主线程执行：

- **优点**：可以 **防止任务丢失**，但会影响主线程速度。
- **适用场景**：适用于 **对任务可靠性要求较高** 的场景，如 **订单处理、日志收集**。

具体使用：

Java 复制代码

```
/**
     * 批量查询笔记历史版本id
     *
     * @param noteIdList
```

```
     * @return
     */
    public Map<String, List<Long>> multiGetNoteHistoryId(List<String> noteIdList) {
        List<CompletableFuture<Void>> futureList = new ArrayList<>();
        HashMap<String, List<Long>> result = new HashMap<>();
        if (CollectionUtils.isNotEmpty(noteIdList)) {
            noteIdList.forEach(e -> {
                CompletableFuture<Void> listCompletableFuture =
CompletableFuture.runAsync(() -> {
                    List<Long> noteHistoryId = this.getNoteHistoryId(e);
                    result.put(e, noteHistoryId);
                }, commonExecutor);
                futureList.add(listCompletableFuture);
            });
            try {
                CompletableFuture<Void> future =
CompletableFuture.allOf(futureList.toArray(new CompletableFuture[0]));
                future.get(3000, TimeUnit.MILLISECONDS);
            } catch (Exception e) {
                log.error("GetLiveRoomAuditInfo error: {}.", e.getMessage(), e);
            }
        }
        return result;
    }
```

在批量查询笔记历史版本id的时候，每个异步任务通过 CompletableFuture.runAsync() 提交给 commonExecutor 线程池执行。

异步任务内部执行 this.getNoteHistoryId(e) 查询历史ID，并将结果存入 result。

每个异步任务的 CompletableFuture 被添加到 futureList 中，方便后续进行统一的等待

再使用 CompletableFuture.allOf() 将所有的异步任务合并成一个新的 CompletableFuture，表示所有异步任务的完成状态。

future.get(3000, TimeUnit.MILLISECONDS) 让主线程阻塞，直到所有异步任务完成或超时（最大等待时间为 3000 毫秒，即 3 秒）。

如果所有任务在 3 秒内未完成，会抛出超时异常，并记录错误日志。

异步

20240807 Future超时不释放线程资源问题

CompletableFuture（优势）	Future（劣势）
1. **链式调用（Chaining）**： ○ CompletableFuture 支持链式调用，可以通过 thenApply、thenCompose、thenCombine 等方法将多个异步操作连接在一起，形成一条异步处理流水线，使代码结构更清晰易懂。 2. **组合多个 CompletableFuture**：	1. **同步阻塞**： ○ 传统的 Future 接口通常采用阻塞的方式获取异步任务的结果，如果任务未完成则会阻塞当前线程，导致程序效率下降。 2. **有限的功能**： ○ Future 接口提供的功能相对有限，不支持丰富的异步操作组合和异常处理机制。 3. **无法手动完成**：

- CompletableFuture 提供了多种方法（如 thenCombine、thenAcceptBoth 等）来组合多个 CompletableFuture 的结果，实现并行或串行执行，从而更灵活地处理多个异步操作的结果。

3. **异常处理**：
   - CompletableFuture 允许通过 exceptionally、handle 等方法处理异常，使得异常处理更加灵活和方便。
4. **异步执行**：
   - CompletableFuture 可以通过 supplyAsync、runAsync 等方法实现异步执行任务，提高系统性能和响应速度。
5. **超时处理**：
   - CompletableFuture 提供了 exceptionally 和 completeOnTimeout 等方法，支持对任务设置超时时间，避免任务执行时间过长导致系统阻塞。
6. **取消任务**：
   - CompletableFuture 可以通过 cancel 方法取消任务的执行，避免不必要的任务继续执行。

- 传统的 Future 无法手动完成（complete）一个任务，导致对异步任务的状态控制不够灵活。

💡 注意**CompletableFuture 的isCancel(true)，和超时并不会中断线程，需要手动调用当前线程并进行中断。**