F-S-05-01： WMS和WCS 服务自动停止

典型现象

- 运行中的 WMS 或 WCS 服务进程突然消失，客户端无法连接。
- 系统日志显示服务异常退出，需要手动重启。
- 服务停止后无自动恢复机制，导致业务中断。

可能原因

1. **内存溢出（OOM）**：JVM 堆内存不足，或物理内存耗尽，操作系统杀死进程。
2. **未捕获的致命异常**：代码中出现 Error 级异常（如 StackOverflowError 、 NoClassDefFoundError ）导致进程崩溃。
3. **依赖服务强行断开**：数据库或中间件主动断开连接，服务处理逻辑不当导致崩溃。
4. **人为误操作**：运维人员误执行 kill 或停止命令。

排查思路

1. **查看服务日志**：搜索 OutOfMemoryError 、 Fatal error 、 Shutdown 等关键词，定位退出前的最后几行日志。
2. **分析堆转储文件（Heap Dump）**：若启用了 -XX:+HeapDumpOnOutOfMemoryError ，使用 MAT 或 JProfiler分析内存泄漏对象。
3. **监控内存使用趋势**：通过监控工具（如 Prometheus、Zabbix）查看服务退出前的内存增长曲线，判断是否存在内存泄漏。
4. **检查是否有 core dump**：执行 ulimit -c 确认 core 文件是否生成，用 gdb 分析崩溃点。

保养提示

- 为 JVM 设置合理的堆内存（如 -Xms4g -Xmx4g ），并启用 -XX:+ExitOnOutOfMemoryError 让服务退出后可由守护进程重启。
- 部署进程守护工具（如 systemd、supervisor）实现服务自动重启。
- 配置内存告警：当 JVM 堆内存使用率持续超过 85% 时，触发预警并分析泄漏。

F-S-05-02：服务启动失败

典型现象

- 执行启动命令后，进程立即退出或报错退出。
- 日志中出现错误信息，服务无法进入正常运行状态。
- 服务端口未能监听。

可能原因

1. **端口冲突**：服务配置的端口（如 8080）已被其他进程占用。
2. **配置文件错误**：配置文件（appsettings.json、 Program.cs）格式错误、缺少必填项、引用了不存在的文件。
3. **依赖服务未就绪**：数据库、 Redis、注册中心（如 Nacos、 Eureka）不可访问，且服务未配置重试或降级。
4. **权限不足**：服务用户没有写日志目录、临时目录的权限，或无法绑定低端口（<1024）。

排查思路

1. **查看启动日志**：从日志中找到第一个 ERROR 或 FATAL 信息，定位根本原因。
2. **检查端口占用**：执行 `netstat -tulnp | grep <端口号>` ，若占用则 kill 冲突进程或更改服务端口。
3. **验证配置文件**：使用 YAML/JSON 校验工具检查格式；逐一核对数据库连接串、 Redis 地址等关键配置。
4. **测试依赖服务连通性**：从服务所在服务器 telnet 数据库、 Redis 等服务的 IP 和端口。若不通过，修复网络或先启动依赖项。
5. **检查文件和目录权限**：确认日志目录、数据目录可写；如果需要使用 80/443 端口，使用 `setcap` 或反向代理转发。
6. **分析依赖冲突**：使用 `mvn dependency:tree` 或 `gradle dependencies` 查看依赖树，排除重复版本。

保养提示

- 编写启动脚本，在启动前自动检测端口占用和依赖服务健康状态，给出友好提示。
- 配置中心化管理配置文件（如 Apollo、 Nacos），避免因本地配置错误导致启动失败。
- 建立服务启动失败的监控告警， 一旦失败立即通知运维。

F-S-05-03：服务假死（进程存在但无响应）

典型现象

- ps -ef 或 jps 能看到进程还在，但接口长时间无响应。
- 日志停止输出，健康检查失败。
- 无法正常关闭服务（ kill -15 无效）。

可能原因

1. **死锁**：多个线程互相等待对方释放资源，导致所有工作线程阻塞。
2. **无限循环**：代码中出现 while(true) 且无跳出条件，消耗 CPU 但不响应请求。
3. **GC 停顿过长**： Full GC 持续数分钟甚至更久，导致服务完全暂停。
4. **文件或网络 I/O 阻塞**：读取文件、数据库连接或网络 Socket 进入阻塞状态且未设置超时。
5. **线程池队列满**：请求全部排队，但也可能是队列中的任务本身无法完成。

排查思路

1. **查看 CPU 使用率**：执行 top -p <pid> ，观察 %CPU。
    - 若 CPU 接近 0% → 可能是死锁或 I/O 阻塞。
    - 若 CPU 接近 100% → 可能是无限循环。
2. **打印线程堆栈**：使用 jstack <pid> 输出线程状态，分析：
    - 大量线程处于 BLOCKED → 死锁或锁竞争。
    - 某线程长时间 RUNNABLE 且执行同一行代码 → 无限循环。
    - 线程处于 WAITING 或 TIMED_WAITING 但无进展 → 资源等待。
3. **检查 GC 活动**：使用 jstat -gcutil <pid> 1000 观察 GC 情况。若 FGC 频繁且 FGCT 快速增加，说明 Full GC 导致停顿。
4. **检查 I/O 阻塞**：查看线程堆栈中是否有 java.net.SocketInputStream.read 或 FileChannel.read 且无超时设置。
5. **尝试生成 dump**： jmap -dump:live,format=b,file=heap.hprof <pid> 可能因假死而失败，可改用gcore 或 Linux 的 kill -3 生成 javacore。

保养提示

- 为所有网络和文件读写出设置超时时间（如 connectionTimeout 、 socketTimeout ）。
- 部署监控系统定期探测服务健康端点， 一旦假死立即自动重启。
- 编写脚本定期执行 jstack 并分析是否有长时间阻塞的线程，触发告警。