MQ运维开发，全栈工作，需求、设计、开发、上线、管理。推行、监控：MQadmin协议，获取broker的运行状态，send，pull，admin线程池，耗时监控，操作系统监控，disState，zabbix，grafana。

需求，消息可靠性，分片式，一主多备，同步刷盘，不丢消息。延迟消息，查询和通知。异步请求。减轻其他应用系统重量，压力转移到MQ。

削峰填谷。消息分发。

消息存本地，原生MQ无，嵌入式数据库，调度，新消息来通知进行补发。

MQ查询消息的能力比较弱，commitLog形式存储消息，解析消息，转储到mangDb里，可以进行带有业务含义的查询。出报表，消息可以多存一些时间。

开发运维一体化。

Python，ansible

重新回答：

在wl负责RocketMQ运维、开发、技术储备、推广等工作，即所有跟RMQ相关的工作都由我主导。

首先为实现自主可控，详细阅读了RocketMQ的源码，熟悉主干代码的实现逻辑，在组内做了几次源码分析。在此基础上书写了《RMQ开发者手册》，《参数明细》和几篇源码分析，贡献给了开源社区，也参与了开源社区文档的编写。阅读、调试源码发现的bug提给官方，并尝试提交pr解决问题。

运维，包括编写运维发布脚本；日报、运维周报脚本；上线前检查脚本；6IDC服务搭建、版本更新、版本升级，同运维120余台虚机。脚本依赖python和ansible。还做了同步、异步刷盘集群的压测，参数调优。生产维护两套集群。

同时做了较为完善的监控，监控包括实现原生监控、二次开发、操作系统的监控增强。做独立的监控系统，java编写，调用broker提供的mqadmin接口，获取broker运行状态和topic、consumerGroup积压和消费状态等，聚合汇总后通过zabbix sender发给zabbix。同时依赖了zabbix api，实现生产环境创建topic、consumerGroup，新增broker时自动创建主机、监控项，更新模板操作。迭代多次，在原有数据基础上做二次计算，拓展监控，聚合数据实现集群监控。

操作系统监控增强，由于虚机本身的OS监控是zabbix每隔一段时间采集一个数据，漏掉了大量数据，所以在本地依赖dstat获取cpu、memory、net、swap等10个大项，28个小项每秒监控，计算一个时间段内的最大、最小、平均值，发给zabbix，提高OS监控的精确度。python编写。

开发工作，首先做了消息转储工具，部署在slave上，读取并解析commitLog，将消息剥离出来并转储到MongoDB上，依赖了apache的FIleAlterationMonitor。java编写。产品亮点：进程间共享commitLog，线程间通知策略，可靠性高，重复数据少。

还做了消息存本地功能，修改生产者源码，当出现发送失败时将消息存入本地数据库中，用了BerkeleyDB，再定时的重新投递出去。产品亮点：客户端影响小，客户端可自定义规则和Hook，决定和感知存本地的执行逻辑和结果，提供同步、异步两种写库操作。以上工程或工具均通过sonar单侧扫描，且通过测试。

知识储备，同时还做了对相关技术的调研，例如G1的工作原理；pdflush、swap工作原理；Netty主干源码的理解；sentinel匀速器调研和接入。

推广，在公司做推广宣讲，了解业务逻辑，帮助交易系统更换接入RMQ，帮助解决流控问题。

【了解业务、对MQ其他性能的理解可以更深入、需要整理一下思路】

1. JDK原生线程池有哪些参数？队列有哪些？拒绝处理器？线程池工作顺序？在Executors类里面提供了常用的线程池？

回答：

答案：详见模拟面试20181024问题2

corePoolSize -> 任务队列 -> maximumPoolSize -> 拒绝策略

|  |  |
| --- | --- |
| xecutorService | 真正的线程池接口。 |
| ScheduledExecutorService | 能和Timer/TimerTask类似，解决那些需要任务重复执行的问题。 |
| ThreadPoolExecutor | ExecutorService的默认实现。 |
| ScheduledThreadPoolExecutor | 继承ThreadPoolExecutor的ScheduledExecutorService接口实现，周期性任务调度的类实现。 |

1. newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

2.newFixedThreadPool

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

3. newCachedThreadPool

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，

那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

4.newScheduledThreadPool

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

快捷方法创建线程池，会使用无界的任务队列，为避免OOM，还是不建议

·向线程池提交的任务有两种：Runnable和Callable，二者的区别如下：

方法签名不同，void Runnable.run(), V Callable.call() throws Exception

是否允许有返回值，Callable允许有返回值

是否允许抛出异常，Callable允许抛出异常。

排队有三种通用策略：

直接提交。工作队列的默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

无界队列。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize的值也就无效了。）当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

有界队列。当使用有限的 maximumPoolSizes时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。

1. 常用mvn命令？版本控制？git命令

回答：

答案：

mvn compile，编译类文件

mvn install，包含mvn compile，mvn package，然后上传到本地仓库

mvn deploy,包含mvn install,然后，上传到私服

mvn install -Dmaven.test.skip=true

首先需要编译java工程：mvn compile 不存在参数的情况下：mvn exec:java -Dexec.mainClass="\*\*\*.Main" 存在参数：mvn exec:java -Dexec.mainClass="\*\*\*.Main" -Dexec.args="arg0 arg1 arg2" 指定运行时库：mvn exec:java -Dexec.mainClass="\*\*\*.Main" -Dexec.classpathScope=runtime

mvn help:describe

你是否因为记不清某个插件有哪些goal而痛苦过,你是否因为想不起某个goal有哪些参数而苦恼,那就试试这个命令吧,它会告诉你一切的.

参数: 1. -Dplugin=pluginName   2. -Dgoal(或-Dmojo)=goalName:与-Dplugin一起使用,它会列出某个插件的goal信息,

如果嫌不够详细,同样可以加-Ddetail.(注:一个插件goal也被认为是一个 “Mojo”)

下面大家就运行mvn help:describe -Dplugin=help -Dmojo=describe感受一下吧!

mvn tomcat:run

用了maven后,你再也不需要用eclipse里的tomcat来运行web项目(实际工作中经常会发现用它会出现不同步更新的情况),只需在对应目录里运行 mvn tomat:run命令,然后就可在浏览器里运行查看了.如果你想要更多的定制,可以在pom.xml文件里加下面配置:

01 02 03 04 org.codehaus.mojo 05 tomcat-maven-plugin 06 07 /web 08 9090 09 10 11 12 当然你也可以在命令里加参数来实现特定的功能,

下面几个比较常用:

　　1>. 跳过测试:-Dmaven.test.skip(=true)

　　2>. 指定端口:-Dmaven.tomcat.port=9090

　　3>. 忽略测试失败:-Dmaven.test.failure.ignore=true 当然,如果你的其它关联项目有过更新的话,一定要在项目根目录下运行mvn clean install来执行更新,再运行mvn tomcat:run使改动生效.

mvnDebug tomcat:run

这条命令主要用来远程测试,它会监听远程测试用的8000端口,在eclipse里打开远程测试后,它就会跑起来了,设断点,调试,一切都是这么简单.上面提到的那几个参数在这里同样适用.

mvn dependency:sources

故名思义,有了它,你就不用到处找源码了,运行一下,你项目里所依赖的jar包的源码就都有了

一、新建代码库

# 在当前目录新建一个Git代码库

$ git init

# 新建一个目录，将其初始化为Git代码库

$ git init [project-name]

# 下载一个项目和它的整个代码历史

$ git clone [url]

二、配置

Git的设置文件为.gitconfig，它可以在用户主目录下（全局配置），也可以在项目目录下（项目配置）。

# 显示当前的Git配置

$ git config --list

# 编辑Git配置文件

$ git config -e [--global]

# 设置提交代码时的用户信息

$ git config [--global] user.name "[name]"

$ git config [--global] user.email "[email address]"

三、增加/删除文件

# 添加指定文件到暂存区

$ git add [file1] [file2] ...

# 添加指定目录到暂存区，包括子目录

$ git add [dir]

# 添加当前目录的所有文件到暂存区

$ git add .

# 添加每个变化前，都会要求确认

# 对于同一个文件的多处变化，可以实现分次提交

$ git add -p

# 删除工作区文件，并且将这次删除放入暂存区

$ git rm [file1] [file2] ...

# 停止追踪指定文件，但该文件会保留在工作区

$ git rm --cached [file]

# 改名文件，并且将这个改名放入暂存区

$ git mv [file-original] [file-renamed]

四、代码提交

# 提交暂存区到仓库区

$ git commit -m [message]

# 提交暂存区的指定文件到仓库区

$ git commit [file1] [file2] ... -m [message]

# 提交工作区自上次commit之后的变化，直接到仓库区

$ git commit -a

# 提交时显示所有diff信息

$ git commit -v

# 使用一次新的commit，替代上一次提交

# 如果代码没有任何新变化，则用来改写上一次commit的提交信息

$ git commit --amend -m [message]

# 重做上一次commit，并包括指定文件的新变化

$ git commit --amend [file1] [file2] ...

五、分支

# 列出所有本地分支

$ git branch

# 列出所有远程分支

$ git branch -r

# 列出所有本地分支和远程分支

$ git branch -a

# 新建一个分支，但依然停留在当前分支

$ git branch [branch-name]

# 新建一个分支，并切换到该分支

$ git checkout -b [branch]

# 新建一个分支，指向指定commit

$ git branch [branch] [commit]

# 新建一个分支，与指定的远程分支建立追踪关系

$ git branch --track [branch] [remote-branch]

# 切换到指定分支，并更新工作区

$ git checkout [branch-name]

# 切换到上一个分支

$ git checkout -

# 建立追踪关系，在现有分支与指定的远程分支之间

$ git branch --set-upstream [branch] [remote-branch]

# 合并指定分支到当前分支

$ git merge [branch]

# 选择一个commit，合并进当前分支

$ git cherry-pick [commit]

# 删除分支

$ git branch -d [branch-name]

# 删除远程分支

$ git push origin --delete [branch-name]

$ git branch -dr [remote/branch]

六、标签

# 列出所有tag

$ git tag

# 新建一个tag在当前commit

$ git tag [tag]

# 新建一个tag在指定commit

$ git tag [tag] [commit]

# 删除本地tag

$ git tag -d [tag]

# 删除远程tag

$ git push origin :refs/tags/[tagName]

# 查看tag信息

$ git show [tag]

# 提交指定tag

$ git push [remote] [tag]

# 提交所有tag

$ git push [remote] --tags

# 新建一个分支，指向某个tag

$ git checkout -b [branch] [tag]

七、查看信息

# 显示有变更的文件

$ git status

# 显示当前分支的版本历史

$ git log

# 显示commit历史，以及每次commit发生变更的文件

$ git log --stat

# 搜索提交历史，根据关键词

$ git log -S [keyword]

# 显示某个commit之后的所有变动，每个commit占据一行

$ git log [tag] HEAD --pretty=format:%s

# 显示某个commit之后的所有变动，其"提交说明"必须符合搜索条件

$ git log [tag] HEAD --grep feature

# 显示某个文件的版本历史，包括文件改名

$ git log --follow [file]

$ git whatchanged [file]

# 显示指定文件相关的每一次diff

$ git log -p [file]

# 显示过去5次提交

$ git log -5 --pretty --oneline

# 显示所有提交过的用户，按提交次数排序

$ git shortlog -sn

# 显示指定文件是什么人在什么时间修改过

$ git blame [file]

# 显示暂存区和工作区的代码差异

$ git diff

# 显示暂存区和上一个commit的差异

$ git diff --cached [file]

# 显示工作区与当前分支最新commit之间的差异

$ git diff HEAD

# 显示两次提交之间的差异

$ git diff [first-branch]...[second-branch]

# 显示今天你写了多少行代码

$ git diff --shortstat "@{0 day ago}"

# 显示某次提交的元数据和内容变化

$ git show [commit]

# 显示某次提交发生变化的文件

$ git show --name-only [commit]

# 显示某次提交时，某个文件的内容

$ git show [commit]:[filename]

# 显示当前分支的最近几次提交

$ git reflog

# 从本地master拉取代码更新当前分支：branch 一般为master

$ git rebase [branch]

八、远程同步

$ git remote update --更新远程仓储

# 下载远程仓库的所有变动

$ git fetch [remote]

# 显示所有远程仓库

$ git remote -v

# 显示某个远程仓库的信息

$ git remote show [remote]

# 增加一个新的远程仓库，并命名

$ git remote add [shortname] [url]

# 取回远程仓库的变化，并与本地分支合并

$ git pull [remote] [branch]

# 上传本地指定分支到远程仓库

$ git push [remote] [branch]

# 强行推送当前分支到远程仓库，即使有冲突

$ git push [remote] --force

# 推送所有分支到远程仓库

$ git push [remote] --all

九、撤销

# 恢复暂存区的指定文件到工作区

$ git checkout [file]

# 恢复某个commit的指定文件到暂存区和工作区

$ git checkout [commit] [file]

# 恢复暂存区的所有文件到工作区

$ git checkout .

# 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变

$ git reset [file]

# 重置暂存区与工作区，与上一次commit保持一致

$ git reset --hard

# 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变

$ git reset [commit]

# 重置当前分支的HEAD为指定commit，同时重置暂存区和工作区，与指定commit一致

$ git reset --hard [commit]

# 重置当前HEAD为指定commit，但保持暂存区和工作区不变

$ git reset --keep [commit]

# 新建一个commit，用来撤销指定commit

# 后者的所有变化都将被前者抵消，并且应用到当前分支

$ git revert [commit]

# 暂时将未提交的变化移除，稍后再移入

$ git stash

$ git stash pop

十、其他

# 生成一个可供发布的压缩包

$ git archive

1. <https://leetcode-cn.com/problems/teemo-attacking/>

回答：python 大法好

class Solution:

def findPoisonedDuration(self, timeSeries, duration):

"""

:type timeSeries: List[int]

:type duration: int

:rtype: int

"""

res = 0

for i, n in enumerate(timeSeries):

if i + 1 < len(timeSeries):

res += min(duration, timeSeries[i + 1] -timeSeries[i])

else:

res += duration

return res