## #概况

hook平台:业务端对钩子端的数据进行检查，阻止非法提交
因数据丢失问题现引入定时任务自动修复丢失数据

## #环境

python3.7, Flask-APScheduler=1.11.0, APScheduler=3.6.0, uWSGI=2.0.18及其兼容版本

## #关于

wangliyang@dobest.com hook.qa.com

## #Bug

原因:1.业务逻辑和数据库设计引发数据频繁丢失;2.容灾设计，服务器宕机后，钩子端和业务端可能不能同步上线，这时为了防止阻塞提交业务端“允许”这些提交，这部分被“允许”的提交没有经过业务端检查也没有数据留存，所以需要复现这部分数据

## #影响范围

"提交记录，mysql"， “定时任务，异步多线程”

## #设计

- 优化业务逻辑和数据结构，减少数据插入失败和中途执行失败的现象
- 使用凭证获取项目的真实提交记录，对比业务端数据库的提交记录，将缺失的部分生成记录保存到数据库中

## #横向对比

Python中常用的定时任务框架有以下几种：

1. `schedule` 库
2. `APScheduler` 库
3. `Celery` 库
4. `RQ (Redis Queue)` 库

横向对比：

1. `schedule` 库：
   - 优点：简单易用，适用于轻量级定时任务。
   - 缺点：不支持分布式，不具备任务持久化功能。
2. `APScheduler` 库：
   - 优点：功能强大，支持多种调度策略，支持分布式和任务持久化。
   - 缺点：配置相对复杂，学习成本较高。
3. `Celery` 库：
   - 优点：支持分布式，任务持久化，与Django等Web框架集成良好，社区活跃。
   - 缺点：依赖于消息队列（如RabbitMQ、Redis等），配置和管理相对复杂。
4. `RQ (Redis Queue)` 库：
   - 优点：基于Redis，支持分布式，任务持久化，易于与其他Python库集成。
   - 缺点：依赖于Redis，需要额外部署和维护Redis服务。

总结：

- 如果需求较为简单，可以选择`schedule`库。
- 如果需要支持分布式和任务持久化，可以选择`APScheduler`或`Celery`库。
- 如果项目已经使用了Redis，可以考虑使用`RQ`库。

当前项目使用flask框架并且需要持久化数据选择APScheduler

## #APScheduler

APScheduler的全称是Advanced Python Scheduler。它是一个轻量级的 Python 定时任务调度框架。APScheduler 支持三种调度任务：固定时间间隔，固定时间点（日期），Linux 下的 Crontab 命令。同时，它还支持异步执行、后台执行调度任务。
APScheduler官网文档

### APScheduler基本功能

1. 简单使用：
   - 安装pip install apscheduler。示例，每5秒输出时间

```
from apscheduler.schedulers.blocking import BlockingScheduler
from datetime import datetime

def timed_task():
    print(datetime.now().strftime("%Y-%m-%d %H:%M:%S"))

if __name__ == '__main__':
    # 创建当前线程执行的schedulers
    scheduler = BlockingScheduler()
    # 添加调度任务(timed_task),触发器选择interval(间隔性),间隔时长为5秒
    scheduler.add_job(timed_task, 'interval', seconds=5)
    # 启动调度任务
    scheduler.start()
```

2. 调度器（scheduler）：
   - BlockingScheduler: 调度器在当前进程的主线程中运行，会阻塞当前线程。
   - AsyncIOScheduler: 结合asyncio模块一起使用。
   - GeventScheduler: 程序中使用gevent作为IO模型和GeventExecutor配合使用。
   - TornadoScheduler: 程序中使用Tornado的IO模型，用 ioloop.add_timeout 完成定时唤醒。
   - TwistedScheduler: 配合TwistedExecutor，用reactor.callLater完成定时唤醒。
   - QtScheduler: 应用是一个Qt应用，需使用QTimer完成定时唤醒。
3. 触发器（trigger）：
   - `date`是最基本的一种调度，作业任务只会执行一次。参数详见
   - `interval`触发器，固定时间间隔触发。参数详见
   - `cron`触发器，在特定时间周期性地触发，和Linux crontab格式兼容。它是功能最强大的触发器。参数详见
4. 作业存储（job store）：
   - 添加任务，有两种添加方法，一种`add_job()`， 另一种是`scheduled_job()`修饰器来修饰函数。

```
from datetime import datetime
from apscheduler.schedulers.blocking import BlockingScheduler
scheduler = BlockingScheduler()

# 第一种
@scheduler.scheduled_job(job_func, 'interval', seconds=10)
def timed_task():
```

```
    print(datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
# 第二种
scheduler.add_job(timed_task, 'interval', seconds=5)

scheduler.start()
```

- 删除任务，两种方法：`remove_job()` 和 `job.remove()`。`remove_job()` 是根据任务的id来移除，所以要在任务创建的时候指定一个 id。`job.remove()` 则是对任务执行remove方法。

```
scheduler.add_job(job_func, 'interval', seconds=20, id='one')
scheduler.remove_job(one)

task = add_job(task_func, 'interval', seconds=2, id='job_one')
task.remvoe()
```

- 关闭任务，使用`scheduler.shutdown()`默认情况下调度器会等待所有正在运行的作业完成后，关闭所有的调度器和作业存储。

```
scheduler.shutdown()
scheduler.shutdown(wait=false)
```

- 获取任务列表，通过`scheduler.get_jobs()`方法能够获取当前调度器中的所有任务的列表

```
tasks = scheduler.get_jobs()
```

5. `执行器`（executor）：
   - 执行器是执行调度任务的模块。最常用的 executor 有两种：`ProcessPoolExecutor` 和 `ThreadPoolExecutor`

## Flask_APScheduler解决方案

1. `基本任务`：
   - new线程执行业务

```
scheduler = BackgroundScheduler(daemon=True)
scheduler.start()

@scheduler.scheduled_job('cron', day_of_week='sun', hour=0, minute=0, timezone='Asia/Shanghai')
def my_task():
        # 业务线程启动
        job = threading.Thread(target=do_job)
        job.start()

def do_job():
    print(datetime.now().strftime('%Y-%m-%d %H:%M:%S'))
```

2. `uWSGI多进程模式下,每个进程启动定时任务,导致重复启动`
   - 添加互斥锁保证只有一个任务正常执行

```
scheduler = BackgroundScheduler(daemon=True)
scheduler.start()

# 补全日志的定时任务
@scheduler.scheduled_job('cron', day_of_week='sun', hour=0, minute=0, timezone='Asia/Shanghai')
```

```
        def my_task():
            # 获取互斥锁
            lock = get_lock('hook:{}:job:history_complete'.format(Config.TAG))
            if lock.acquire(blocking=False):
                job_id = str(uuid.uuid1())
                job_key = 'hook:{}:job:history_complete:{}'.format(Config.TAG, job_id)
                job_info = {"status": JobStatusEnum.RUNNING.code,
                            'msg': '{} : {}'.format(JobStatusEnum.RUNNING.msg, job_id)}
                redis_set_expire(job_key, job_info, TimeOutConstant.MULTI_JOB_LIVE_TIME_OUT)
                # 业务线程启动
                job = threading.Thread(target=do_job, args=(job_key, lock))
                job.start()
                return {'data': {'job_key': job_key}}
            else:
                return {'data': ErrorInfoEnum.LOCK_OCCUPIED_ERROR.msg}
```

3. 当任务需要调用数据库时关联上下文
    - 添加互斥锁保证只有一个任务正常执行

```
def do_job(job_key, lock):
        ins_app = create_app(Config)
        # 关联flask的数据库上下文
        with ins_app.app_context():
                try:
                        result_info = history_completion()
                        data = {"status": JobStatusEnum.OK.code, 'msg': result_info}
                except RuntimeError as e:
                        print('history_completion_by_project failed:', e)
                        data = {"status": JobStatusEnum.FAILED.code, 'msg': JobStatusEnum.FAILED.msg}
                redis_set_expire(job_key, data, TimeOutConstant.JOB_LIVE_TIME_OUT)
                lock.release()
```

## #具体实现

1. 关联实际记录：
    - 业务端的代码仓库会关联用户的凭证的信息，该凭证可访问此仓库的全部提交信息
2. 执行业务：
    - 使用凭证查询仓库的提交id列表，使用数据库查询已经保存的提交id列表，找出数据库缺失的部分id子集
3. 持久化数据：
    - 根据id列表查询对应的提交信息，将这些信息保存到数据库并打上特殊标记"未检查"
4. 外部调用：
    - 定时任务每隔一周自动执行一次全仓库修复，前端暴露单个仓库的异步修复接口，客户端可以选择自行调用

## #关于uWSGI多进程多线程

1. uWSGI：
    - python是通常是单线程的，所以会借助webserver增加并发能力，但是在工作中多次遇到多进程的坑，所以查询了一下uWSGI相关原理
2. 工作流程：

    - uwsgi是用c语言写的一个webserver，可以启动多个进程，进程里面可以启动多个线程来服务。进程分为主进程和worker进程，worker里面可以有多个线程。
    - 一开始进入main函数，启动这个就是主进程了，uwsgi_setup函数（主进程）里面针对选项参数做一些处理，执行环境设置，执行一些hook，语言环境初始化（python），如果没有设置延迟加载app，则app在主进程加载；如果设置了延

迟加载，则在每一个worker进程中都会加载一次。uwsgi_setup函数还执行了插件的初始化最后uwsgi主进程fork了指定worker进程用来接收(accept)请求。

- 默认情况下，uWSGI在第一个生成的进程中加载你的应用，然后多次 fork() 自身。这意味着，你的应用被单次加载，然后被拷贝。虽然这个方法加速了服务器的启动，但是有些应用在这项技术下会出问题 (特别是那些在启动时初始化db连接的应用，因为将会在子进程中继承连接的文件描述符)。如果你对uWSGI使用的粗暴的preforking不确定，那么只需使用 --lazy-apps 选项来禁用它。它将会强制uWSGI在每个worker中完全加载你的应用一次。
- 在uwsgi.ini中设置支持懒加载,保证每个进程加载数据库连接正常

```
thunder-lock=True
```

- wsgi_run函数就是真正的开始执行了，这个函数分为俩个部分，主进程执行部分和worker进程执行部分。主进程执行部分是一个无限循环，他可以执行特定的hook以及接收信号等，总之是用来管理worker进程以及一些定时或者事件触发任务。worker部分：注册型号处理函数，执行一些hook，循环接收(accept)请求。在启动woker时可以根据--threads参数指定要产生的线程个数，否则只在当前进程启动一个线程。这些线程循环接收请求并处理。
- 在worker中接收请求的函数wsgi_req_accept有一个锁：thunder_lock，这个锁用来串行化accept，防止“惊群”现象：参考这里
- 惊群现象出现在这样的情况：主进程绑定并监听socket，然后调用fork，在各个子进程进行accept。无论任何时候，只要有一个连接尝试连接，所有的子进程都将被唤醒，但只有一个会连接成功，其他的会得到一个EAGAIN的错误，浙江导致巨大的CPU资源浪费，如果在进程中使用线程，这个问题被再度放大。一个解决方法是串行化accept，在accept前防止一个锁。
- 在uwsgi.ini中设置锁

```
thunder-lock=True
```

如果你在不使用线程的情况下启动uWSGI，那么Python GIL将不会启动，因此，你的应用生成的线程将不会运行。但是uWSGI是一个语言无关的服务器，因此它的大多数选择都是为了维护它的“不可知论”如果你想要维护Python线程支持，而不为你的应用启动多线程，那么仅需添加 `--enable-threads` 选项或者在ini风格的文件中添加 `enable-threads = true`

- 在uwsgi.ini中设置多线程

```
enabled-threads=True
```

参考uwsgi文档