Antdeck 归档平台技术架构设计

目录

系统简介 2

整体架构 2

核心模块 3

    任务调度模块 3

    数据归档模块 4

前端界面 7

版本	作者	内容	时间
1.0	茹作军	文档发布	2019-10-08

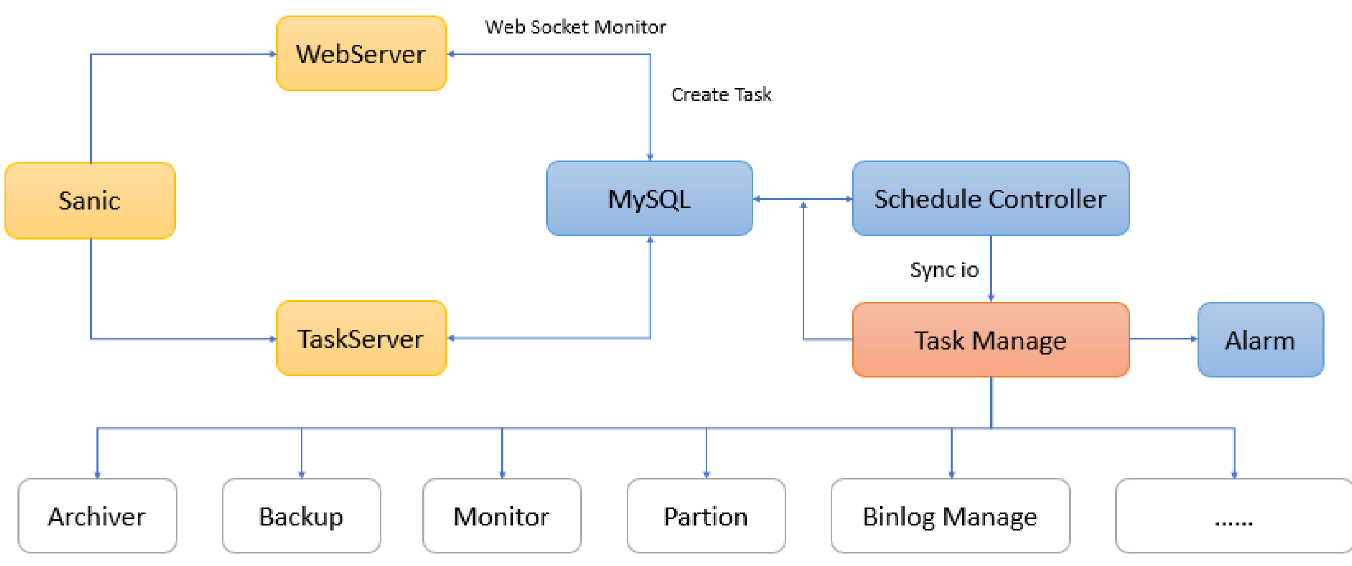
系统简介

Antdeck 是平安好医生 DBA 团队开发的一套系统任务调度和数据库归档的系统平台，用于解决数据库管理任务统一调度管理和大表数据归档问题。

Antdeck 的底层核心模块为我们使用 Python3 自己开发的任务调度模块，曾经调研和测试过 Python 的 Celery 任务调度模块，也使用 Celery 搭建过一个分布式秒级监控任务的模型，不得不说 Celery 是一个非常棒的分布式任务调度方案，配合 MQ 或者 Redis 可以达到很高的性能。但是对于我们日常服务器的任务调度来说，不需要太高的任务调度性能，每天几千次几万次调度已经可以满足我们的需要，我们更加注重调度任务的整体功能是否满足我们的需求，比如任务状态和执行百分比的实时监控、任务队列和异常任务的监控告警、可视化的数据分析、如何将任务调度和目前我们的元数据中心打通等等。面对如此多样化的需求，所以我考虑自己写一个定制化并且轻量级的任务调度模块。

整体架构

Antdeck 的整体技术方案选型中，后端使用 Python3+sanic 框架实现，Python3 的优势是可以支持异步 IO，可以更好的完成一些异步任务。Sanic 是一个由 python3 开发的高性能的异步 IO 框架，开发的模型参考了 flask,使用起来非常的轻量级，但是因为支持异步 IO，比起 flask 有者更好的性能。前端的设计开发是基于 recat 设计思路，使用了蚂蚁金服 ant.design 的一整套解决方案。整体架构图如下所示：



Sanic 默认是运行 1 个 WebServer，在此基本上，我们使用 Sanic 创建了另一个 TaskServer。WebServer 用于接收上层用户的信息，创建调度任务后将任务存储在 MySQL 中。同时 TaskServer 实时监控 MySQL 任务队列，并在 Schedule Controller 中解析任务（手动执行、定时执行、周期执行），在到达对应执行时间点后将任务分发到 Task Manage。Task Manage 在接收到任务后放入内存缓存队列中，并为每个任务 Fork 出对应的任务子进程进行执行，对超过设置的 timeout 时间或者执行异常的任务发送短信告警。另外 WebServer 也会通过 web socket 方式连接实时监控后端任务的运行状态和执行百分比。

核心模块

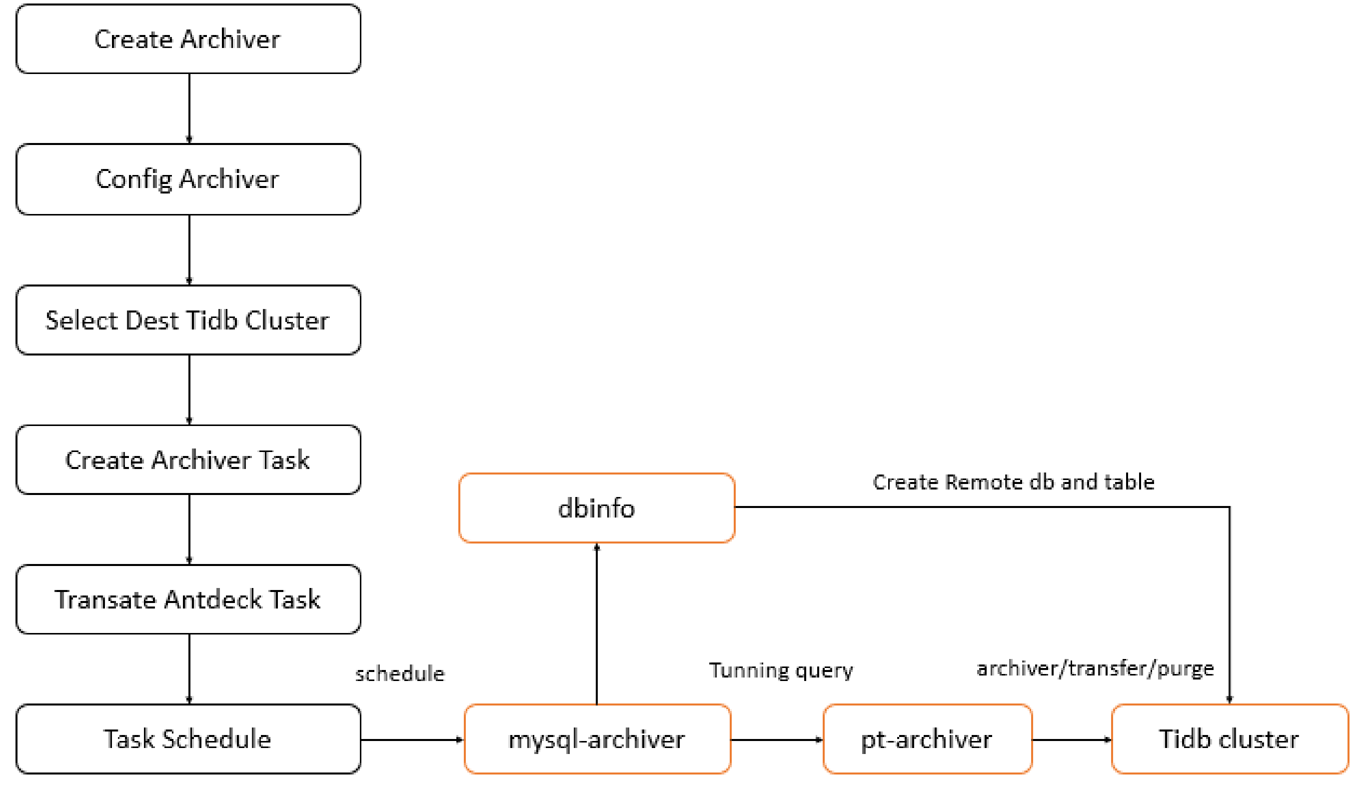
任务调度模块

任务调度模块的主要功能如下：

- 多种调度类型：支持配置手工任务、定时任务、秒级周期任务三种任务类型，并支持随时手动触发任务及手动终止正在运行的任务。
- 定时任务格式兼容：定时任务兼容 Linux 的 crontab 任务类型，可以简单的将 Linux 的计划任务格式 copy 到 antdeck 里面。
- 任务运行实时监控：在任务运行时通过 websocket 技术和历史任务数据计算实时监控任务的执行状态和执行进度百分比。
- 告警支持：支持任务在运行失败、运行超时、任务队列异常时发送短信报警。
- 历史跟踪：可以跟踪每个任务的历史执行状况，以及运行日志。
- 支持多任务执行：每个任务项可以支持最多三个任务的顺序调度运行，在第一个程序运行成功后才会进行后续任务。
- Dashboard 大盘：通过今日任务、本周任务、24H 调度监控、历史任务分布等不同维度监控和分析任务运行状态。

数据归档模块

数据归档模块可以理解为基于调度模块上层的应用层，当我们通过 WEB 界面配置归档后，会最终转换成标准的调度任务进行执行，在到达执行时间点后，归档任务会调用我们使用 Python 封装好的 mysql_archiver 归档程序，mysql_archiver 会首先做相应的参数检测，检测通过后通过归档的参数读取 DBINFO 元数据，远创检测 tidb 是否存在归档库和表，如果不存在会自动创建，然后调用 pt-archiver 进行数据归档，将数据归档到 tidb 集群。数据归档的流程如下：



那么为什么任务调度模块不直接调度 pt-archiver 进行数据归档，而是通过封装 mysql_archiver 再进行调用呢，是不是觉得很多余呀，答案肯定是不，毕竟任何事物存在就有价值，我也不是时间多的需要靠多写代码来实现项目的价值。我们知道 pt-archiver 归档要必须在目标端建立对应的数据表，所以我们通过 mysql-archiver 的封装来实现 tidb 的自动建库和建表，简化人工操作。另一个优化点是如果归档条件是非 id 是字段时，我们会自动增加归档的 max id 防止全表扫描，因为我们在时间字段配置按天归档时，配置到一个 case 就是由于归档的数据量满足不了设置的 bulk size，导致全表扫描。这个 case 中 pt-archiver 第一次查询用了半小时，扫描了三亿多行数据。

Query_time: 1832.913922 Lock_time: 0.000155 Rows_sent: 0 Rows_examined: 337587083

SELECT /*!40001 SQL_NO_CACHE */ `id`,...... FROM `skyeye`.`im_message` FORCE INDEX(`PRIMARY`) WHERE (create_time < DATE_FORMAT(DATE_SUB(NOW(), INTERVAL 70 DAY),'%Y-%m-%d')) ORDER BY `id` LIMIT 6000;

这个情况发生在老的存储过程归档到新归档平台迁移测试的过程中，在凌晨的时候，老的存储过程已经按

create_time < DATE_FORMAT(DATE_SUB(NOW(), INTERVAL 70 DAY),'%Y-%m-%d')的条件进行了数据归档，也就是说 70 天之前的数据已经被归档，数据量为 0，然后当天使用了新的归档平台进行测试，配置了相关的归档条件 create_time < DATE_FORMAT(DATE_SUB(NOW(), INTERVAL 70 DAY),'%Y-%m-%d')。所以上面 pt-archiver 归档其实归档不到任何数据。

那么归档不到数据，上面那个 SQL 为什么会慢呢，原因是 pt-archiver 的设计思路是分批次查询归档，在查询语句中加入了 FORCE INDEX(`PRIMARY`)强制使用主键进行查询，同时使用了 ORDER BY `id` LIMIT 6000 用于保证查询的 id 是顺序的，在每批次查询时会记录当前归档数据的 max id,并在下一批次查询过程中将 max id 加入到查询条件中(and id>max id)，用于提高查询性能 。由于强制使用主键，所以不会使用 create_time 时间索引，而是直接使用 id 排序后再通过 limit 6000 取出 6000 条数据，同时记录 max id。正常情况下，如果查询能通过 id 直接查询拿到 6000 条数据的时候，性能是比较高的。但是如果归档条件包含非 id 字段，同时根据 id 顺序查询的数据无法满足非 id 字段时，查询就会持续向后扫描，持续寻找满足条件的数据，直到找到满足条件的 6000 条数据为止，可想而知，上面的 SQL 查到数据的最后一条也满足不了条件，所以就变更了全表扫描，对于上亿的数据，全表扫描必将是非常慢的。

那么这个问题是如何解决的呢，我们在 mysql-archiver 里面对非 ID 字段加入了一层逻辑，在归档前连接到数据库检索出 max ID 并带入到归档参数中，用于防止全表扫描，主要逻辑代码如下：

```
 query_max_id_sql = "select max(id) from %s.%s where %s" %(source_db,source_table,arch_where)

 my_conn=MySQLdb.connect(host=source_host,user='dbadmin',passwd='xxxxxx',port=int(source_port),connect_time
 out=3,charset='utf8')
my_cur=my_conn.cursor()
my_conn.select_db(source_db)
my_cur.execute(query_max_id_sql)
result = my_cur.fetchone()
max_id = result[0]
if max_id==None or max_id=='' or max_id=='none':
```

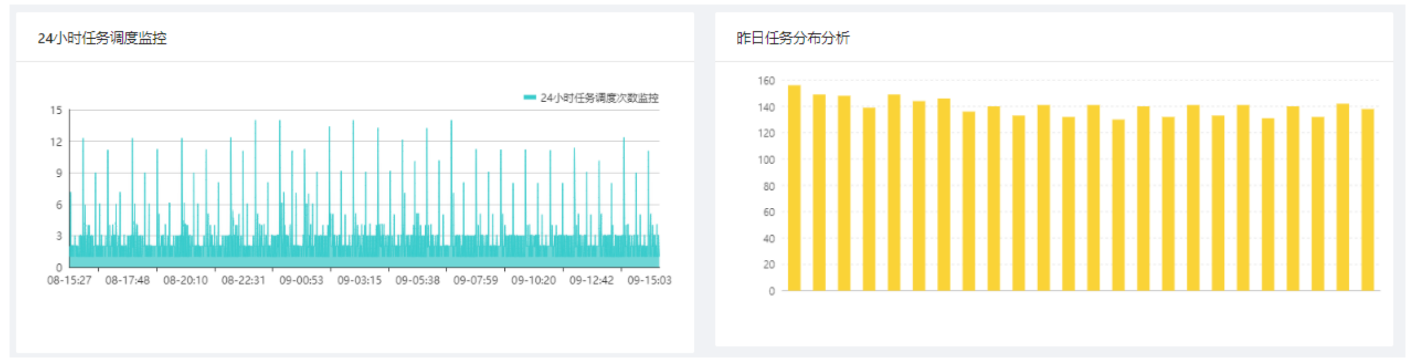
```
        print ('Waring: Arch max id id null. cant find any arch rows. so pt-archiver not work.')
        sys.exit(0)

    arch_where = "id <= %s and %s" %(max_id,arch_where)


    cmd=(""" /usr/bin/pt-archiver   --source h=%s,P=%s,u=dbadmin,p=xxxxxx,D=%s,t=%s --dest
h=%s,P=%s,u=dbadmin,p=xxxxxx,D=%s,t=%s   --where="%s" %s
""")%(source_host,source_port,source_db,source_table,dest_host,dest_port,dest_db,dest_table,arch_where,ext_options)
    stat,output=subprocess.getstatusoutput(cmd)
```

前端界面

Dashboard 界面



创建任务

* 任务类型 ⑦：

* 所在分组：分组规则：MySQL为group_name, Redis为cluster_name

标签：Default

* 任务名称：

任务描述：描述任务的作用，负责人,注意事项等信息

* 执行命令：任务的执行命令或脚本

执行命令：任务的执行命令或脚本

执行命令：任务的执行命令或脚本

调度类型 ⑦：手动触发

超时时间：10800

是否启用：

取消 提交

任务管理

任务名	DB标签	所在分组	调度类型	状态	下次运行时间	启用	操作
redis_expire_keys	Redis	dbmanage	定时	运行成功	2019-10-03 07:30	停用	修改 | 详情
trans_vpr_device_user_status	MySQL	dbmanage	定时	运行成功	2019-10-10 02:03	正常	修改 | 详情
fix_filebeat	Default	dbmanage	定时	运行成功	2019-10-10 06:30	正常	修改 | 详情
fix_phenix_key	MySQL	dbmanage	定时	运行成功	2019-10-10 06:00	正常	修改 | 详情
sync_innotop_conf	MySQL	dbmanage	定时	运行成功	2019-10-10 07:00	正常	修改 | 详情
manager_scripts_totoal_backup	Default	dbmanage	定时	运行成功	2019-10-10 05:05	正常	修改 | 详情
purge_mysql_xtrabackup_120days	MySQL	dbmanage	定时	运行成功	2019-10-09 22:30	正常	修改 | 详情
purge_relay_log	MySQL	dbmanage	定时	正在运行	2019-10-09 18:10	正常	修改 | 详情

任务执行监控

历史执行状态跟踪

刷新任务列表

全部任务 | 执行成功 | 执行失败

运行开始时间	运行结束时间	运行时长	运行状态	操作
2019-10-09 15:10:02	---------------------	1367秒	正在运行	运行日志
2019-10-09 12:10:01	2019-10-09 13:43:49	5628秒	运行成功	运行日志
2019-10-09 09:10:03	2019-10-09 10:43:50	5627秒	运行成功	运行日志
2019-10-09 06:10:08	2019-10-09 07:44:16	5648秒	运行成功	运行日志
2019-10-09 03:10:03	2019-10-09 04:44:16	5653秒	运行成功	运行日志
2019-10-09 00:10:10	2019-10-09 01:43:43	5613秒	运行成功	运行日志
2019-10-08 21:10:02	2019-10-08 22:43:42	5620秒	运行成功	运行日志

归档界面

源数据库：

源数据表：

归档类型 ⑦ ：数据归档

目标TiDB集群：

归档条件：归档条件(不要写where)，示例：deleted=1 and id<100000

批次查询量：2000

事务提交量：1000

休眠时间(秒)：1

忽略重复数据：否

启用批量插入：否

启用批量删除：否

启用延迟监控：否

调度类型 ⓘ：单次任务

备注说明：归档任务的作用，负责人注意事项等信息

是否启用：

取消 提交