**定时任务服务端架构方案**

## 业务目标

服务器端满足用户设定定时任务的新增、删除、修改、关闭/打开

、查询等功能；

## 系统要求

1. 定时任务触发时，延迟不能大，例如10:00:00触发的任务，不能10:01:00还没有触发；
2. 分布式架构，能满足接近线性扩展；
3. 满足增、删、查、改、关闭、打开等业务功能；

## 总架构图

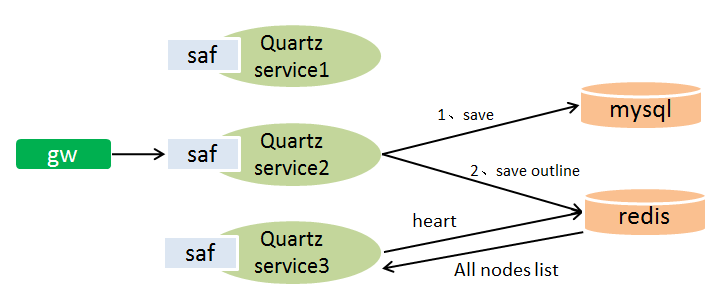


图1 总架构图

## 架构详解

如图1所示，每个quartz service 为一个node，各个node直接对等，无中心node。每个node提供saf服务。每个node需要与mysql、redis通信。

### 功能：

**Node：**

1. 接收用户的定时任务处理，并做相关存储；
2. 定期向redis发送自己的心跳；
3. 检测是否有其他node挂掉，如果有接收挂掉node的定时任务；
4. 检测是否有其他node给自己分配了定时任务；
5. 检测是否有需要删除的定时任务；
6. 具体定时任务的触发与执行

**Node上的Saf服务：**

gw收到app的定时任务后，调用saf服务，通过saf的负载均衡定向到某个quartz service上，quartz service将任务添加到自己服务中，并存储到mysql，将本次任务的概要信息存储到redis中。

**Mysql：**

主要存储各个用户设置的定时任务的具体信息与用户关系信息。

**Redis：**

1. 存储各个node的处理任务id与node的对应关系；
2. 接收各个node的心跳；

### Node实例

每个node实例id由：ip+timestamp组成，ip代码实例所在的机器ip，timestamp是实例启动时的时间戳，精确到毫秒。因此，启动一次就是一个新实例；

### Redis存储视图

Redis存储的数据包括四大类：

1. 各个node实例负责的定时任务，使用set结构存储，key为smart\_quartz+node实例id，set里面存储各个定时任务的id，id是任务存储到数据库中时分配的id；
2. 各个node实例的心跳,使用hashes结构存储，key为smart\_quartz\_node,field为各个node实例id，对应的value为各个node实例发送心跳时的时间戳和此时对应node的任务总数；
3. 各个node实例的附加定时任务，使用set存储结构，key为smart\_quartz+node实例id+addition，set里面存储其他node转发给此node的任务id；
4. 要删除任务队列，使用set结构存储，key为smart\_quartz\_task\_delete,set里面存储要删除的定时任务的id，各个node每次心跳时获取一次这个队列中的所有成员，遍历一次各个任务id，如果属于自己的任务，即删除此任务且同时将删除任务队列中的id也删除；

### node宕机检测方法：

每个node每隔一定时间向redis存储心跳的hashes结构中对应field中写入一个时间戳，同时获取各个这个hashes结构的所有field/value对，将每一个value与当前时间戳对比，如果超过心跳周期的2-3倍，则认为该node实例挂掉；

### 一个node宕机

由图1可知，某个node宕机并不影响新定时任务创建，当一个node宕机后，其他node会检测到，各个node立刻去pop这个实例对应的任务队列和附加任务队列，连续pop10个任务左右后根据获得的任务id查询数据库获得具体任务，再pop，直到这个两个队列为空，此时删除set任务队列、附加任务队列和心跳hashes中的field/value对；

### 全部node宕机

当其他node启动后，处理和一个node宕机情况是一致的。

### 新增node：

由图1可知，新node可以随时加入，不影响其他node。但是如果只借助saf的负载均衡会造成新node的负载低，而老node的负载高。因此，可以考虑saf在分配时先从redis中获取各个node的数量，如果其他的node比自己的负载低，接收到gw分配任务的node可以存储数据库，但不执行此任务，而是将任务添加到比自己负载低的node的附加任务队列中。

### Redis宕机

如果出现这种情况，只能等到redis恢复后，使用另外一个程序扫描一遍数据库，类似新增的业务将各个任务分配到各个node和存储到redis中。

### 优缺点

**优点：**

1. 具体任务执行者是分布式的且可横向接近线性扩展，理论上可以处理海量的定时任务；
2. 由于第一点，每个任务可以在触发时间点尽快得到执行，不会造成大的延时；
3. 对数据库压力低；
4. 无单点故障问题

**缺点：**

1. 稍微比较复杂；

## 业务实现简介

### 新增

用户通过超级app调用服务端gw，gw再通过saf调用后台某个node实例，node将此任务存入数据库，再根据上次心跳时获得的各个node的任务负载情况，决定是自己处理此node还是交给其他node处理。如果交给自己处理，则加入quartz服务且在redis中自己对应的任务set中，加入此任务的id；否则，将此id添加到要分配到的node在redis中对应的附加队列中；

### 删除

用户通过超级app调用服务端的gw，gw再通过saf调用后台某个node实例，该node实例判断此任务是否在自己任务队列中，如果在，则直接删除；否则，将此任务id存储到redis中的删除任务队列中，等待其他node实例检测；

### 修改

修改将分为删除与新增任务两个步骤进行

### 查询

用户通过超级app调用服务端的gw，gw再通过saf调用后台某个node实例，node查询数据库；

### 关闭/打开

数据库中设置一个标志，关闭时各个node、redis中按照删除处理，打开时数据库标志变为打开，各个node、redis按照新增处理。

## 附录

Quartz服务的线程分为两部分：trigger（触发时间）检测线程与job（定时任务，例如闹铃响）执行线程，其中job执行线程是个线程池，当某个trigger触发后，将trigger对应的job 分配给线程池中的某个线程去执行；

经过对单实例的quartz简单的压力测试结果如下：

1w定时任务均匀分布在每分钟中，即1w定时任务每分钟都会触发一次，job执行线程池使用默认的线程个数，quartz可以正常运行；

10w定时任务均匀分布在5分钟内，job执行线程池使用默认的线程个数时，大部分job的实际执行时间会比设定时间晚50s左右，例如设定10:00:00执行，真实执行时间为10:00:50；

当调整job执行线程池的线程个数为60个时，有部分job的实际执行时间会比设定时间晚3s左右，cpu占用率在150%（4核），内存占用在81%（4G内存）。