分布式任务队列分享

目录

一、背景

- 公司技术栈PHP -> Go 迁移背景下，平台GO服务需要一套分布式任务队列承接新模块的异步任务。
- 现有阿拉丁平台、星图平台PHP代码存在大量异步调度任务，使用gearman作为任务调度队列，worker经常崩溃且运维困难，需要迁移。

因此调研设计一个分布式任务队列为PHP服务、GO服务提供任务调度功能。

方案选型与技术设计： platform-go分布式任务队列

https://github.com/hibiken/asynq

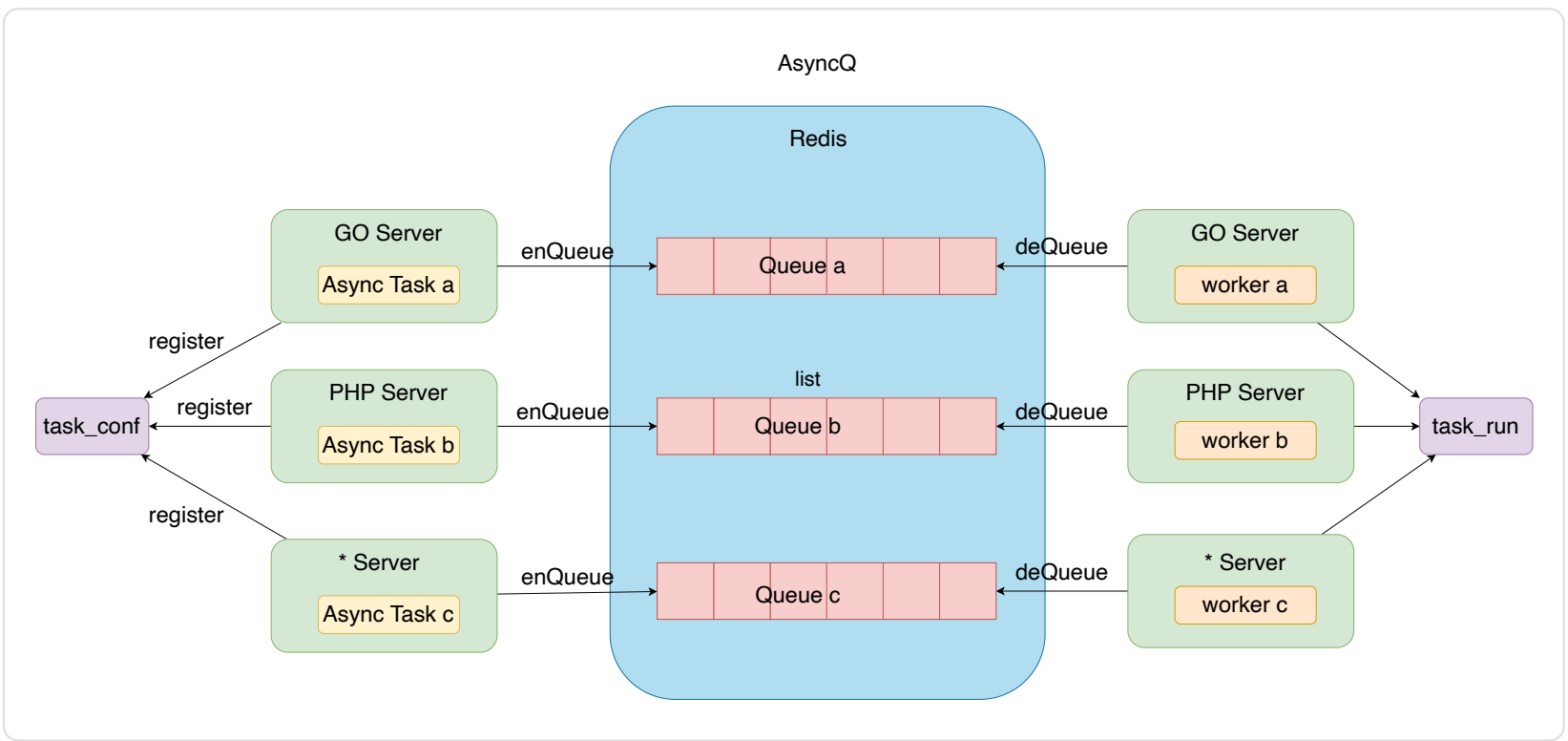
平台需求：

- 很多任务比如配送任务要求严格顺序执行，首先需保证消费的顺序性。
- 平台任务负载不高，性能要求不是特别高。
- 易于接入，需可视化界面便于管理运维。

基于平台需求与各方案的特点，选择基于Redis实现的分布式任务队列Asynq来实现平台的队列服务。

特点：

- 基于Redis的list数据结构实现任务队列，队列的FIFO保证了任务执行的顺序性，
- 基于Redis集群模式实现任务的分布式处理，高可用性。使用哈希槽保证任务的分片均衡。
- 基于Redis的高吞吐保证队列的高性能。



1、概念：

哈希槽

Redis 哈希槽（Hash Slot）是 Redis Cluster（集群模式）中分片和数据分布的核心概念，用于将数据均匀地分布在集群的多个节点上。

机制

- 在 Redis Cluster 中，整个键空间被划分为 16384 个哈希槽，编号从 0 到 16383。
- 每个键通过 CRC16 哈希函数计算得到一个值，然后对 16384 取模（`hash(key) % 16384`），得到该键对应的哈希槽编号。
- Redis Cluster 中的每个节点（主节点）都负责管理一部分哈希槽，比如节点 A 负责哈希槽 0-5460，节点 B 负责哈希槽 5461-10922，节点 C 负责哈希槽 10923-16383。
- 当集群启动或发生拓扑变化时，哈希槽会动态地分配或重新分配给不同的节点。

数据存储与访问

- 当一个键值对（Key-Value）被存储到 Redis Cluster 中时，首先通过哈希函数计算该键对应的哈希槽编号，然后将数据存储到负责该哈希槽的节点上。
- 当客户端需要访问某个键时，集群通过相同的哈希计算找到对应的哈希槽，从而定位到负责该槽的节点，进而完成数据访问。

优点

- **负载均衡**：哈希槽机制确保了键值对在集群中均匀分布，避免某些节点的负载过重。
- **扩展性**：当增加或减少节点时，集群可以通过重新分配哈希槽来调整节点之间的数据分布，实现平滑扩展。
- **容错性**：Redis Cluster 可以通过副本机制（即主从复制）确保哈希槽的数据冗余，当主节点故障时，副本节点可以快速接管相应的哈希槽。

哈希槽的设计确保了 Redis Cluster 在大规模分布式场景下的高可用性和高性能，同时提供了灵活的数据分片和管理能力。

任务队列使用 `asynq:{dump_feature}:pending 作为具体任务的队列名`

任务队列通过Redis集群模式以及哈希槽的机制保证了任务队列本身的负载均衡、高可用性以及高性能。

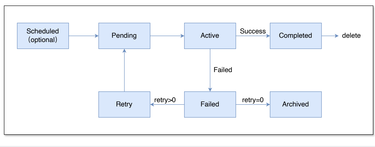
分片效果对比：

键名模式	实际哈希计算部分	集群分布情况
`asynq:dump_feature:pending`	整个字符串	可能分散在不同节点
`asynq:{dump_feature}:pending`	`dump_feature`	所有相关键位于同一节点

```
10.11.133.62:8859> CLUSTER KEYSLOT asynq:dump_feature:pending
(integer) 9998
10.11.133.62:8859> CLUSTER KEYSLOT asynq:dump_feature:active
(integer) 9412
10.11.133.62:8859>
10.11.133.62:8859> CLUSTER KEYSLOT asynq:{dump_feature}:pending
(integer) 8726
10.11.133.62:8859> CLUSTER KEYSLOT asynq:{dump_feature}:active
(integer) 8726
```

二、实现

1. 任务生命周期




```
-- 核心队列键
asynq:{dump_feature}:pending        -- List 类型，待处理任务队列
asynq:{dump_feature}:active         -- List 类型，处理中任务队列
asynq:{dump_feature}:retry          -- List 类型，处理中任务队列
asynq:{dump_feature}:lease          -- ZSET 类型，到期时间，(key=任务id, score=到期时间戳)
asynq:{dump_feature}:t:task123      -- Hash 类型，任务详细信息存储
```

2. 核心代码

核心操作代码

文件页：baidu/ps-aladdin/asynq *master

以任务dump_feature为例

入队

```
// enqueueCmd enqueues a given task message.
//
// Input:
// KEYS[1] -> asynq:{dump_feature}:t:123
// KEYS[2] -> asynq:{dump_feature}:pending
// --
// ARGV[1] -> task message data
// ARGV[2] -> task ID
// ARGV[3] -> current unix time in nsec
//
// Output:
// Returns 1 if successfully enqueued
// Returns 0 if task ID already exists
var enqueueCmd = redis.NewScript(`
if redis.call("EXISTS", KEYS[1]) == 1 then
    return 0
end
redis.call("HSET", KEYS[1],
           "msg", ARGV[1],
           "state", "pending",
           "pending_since", ARGV[3])
redis.call("LPUSH", KEYS[2], ARGV[2])
return 1
`)
```

出队:

```
// Input:
// KEYS[1] -> asynq:{dump_feature}:pending
// KEYS[2] -> asynq:{dump_feature}:paused
// KEYS[3] -> asynq:{dump_feature}:active
// KEYS[4] -> asynq:{dump_feature}:lease
// --
// ARGV[1] -> initial lease expiration Unix time
// ARGV[2] -> task key prefix
//
// Output:
// Returns nil if no processable task is found in the given queue.
// Returns an encoded TaskMessage.
//
// Note: dequeueCmd checks whether a queue is paused first, before
// calling RPOPLPUSH to pop a task from the queue.
var dequeueCmd = redis.NewScript(`
if redis.call("EXISTS", KEYS[2]) == 0 then
    local id = redis.call("RPOPLPUSH", KEYS[1], KEYS[3])
    if id then
        local key = ARGV[2] .. id
        redis.call("HSET", key, "state", "active")
        redis.call("HDEL", key, "pending_since")
        redis.call("ZADD", KEYS[4], ARGV[1], id)
        return redis.call("HGET", key, "msg")
    end
end
return nil`)
```

RPOPLPUSH、RPOP

RPOP

用 LPUSH把消息入队，用 RPOP获取消息，绝大部分的情况下，这些操作都是没问题的，但并不能保证绝对安全。

当 LPOP 返回一个元素给客户端的时候，会从 list 中把该元素移除，这意味着该元素就只存在于客户端的上下文中，如果客户端在处理这个返回元素的过程崩溃了，那么这个元素就永远丢失了。

RPOPLPUSH

原子性地返回并移除 source 列表的最后一个元素， 并把该元素放入 destination 列表的头部

使用 RPOPLPUSH 获取消息时，RPOPLPUSH 会把消息返给客户端，同时把该消息放入一个备份消息列表，并且这个过程是原子的，可以保证消息的安全，当客户端成功的处理了消息后，就可以把此消息从备份列表中移除了。

3. 改造

```
// Input:
// KEYS[1] -> asynq:{dump_feature}:pending
// KEYS[2] -> asynq:{dump_feature}:paused
// KEYS[3] -> asynq:{dump_feature}:active
// KEYS[4] -> asynq:{dump_feature}:lease
// --
// ARGV[1] -> initial lease expiration Unix time
// ARGV[2] -> task key prefix
//
var dequeueCmd = redis.NewScript(`
if redis.call("EXISTS", KEYS[2]) > 0 then
    return nil
end
local count = redis.call("ZCARD", KEYS[4])    --  Zcard 命令用于计算集合中元素的数量。
if (count >= tonumber(ARGV[3])) then
    return nil
end
local id = redis.call("RPOPLPUSH", KEYS[1], KEYS[3])
if id then
    local key = ARGV[2] .. id
    redis.call("HSET", key, "state", "active")
    redis.call("HDEL", key, "pending_since")
    redis.call("ZADD", KEYS[4], ARGV[1], id)
    return redis.call("HGET", key, "msg")
end
return nil`)
```

实现了队列的并发控制功能。

初始化时加载任务列表中各任务的MaxRun（最大运行数量），实现不同任务的并发运行数量限制。补齐队列的功能。

```

    queues := map[string]int{}
    queueConcurrency := map[string]int{}
    for _, tc := range taskConfs {
        queues[tc.TaskName] = 1
        queueConcurrency[tc.TaskName] = int(tc.MaxRun)
    }

    srv := asynq.NewServer(r, asynq.Config{
        Queues:           queues,
        QueueConcurrency: queueConcurrency,
    })

```

4. 队列服务

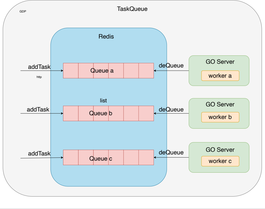
基于GDP框架实现了一个微服务task-queue，提供队列服务，支持两种使用方式

4.1 任务托管服务

文件页：baidu/ps-aladdin/task-queue *master

任务执行Handler托管在队列服务，worker由task-queue统一管理、执行

适合一些通用的任务类型，比如cmd、http等类型的任务。跨语言的任务也可托管，参考下文的php接入。



4.2 sdk

平台提供sdk供其他服务接入，使用平台的专有BDRP作为队列载体，供统一监控管理。

三、接入

异步任务队列（task-queue）使用文档

1、Go服务接入

推荐使用sdk方式接入，成本非常低

同样使用dump_feature举例

文件页：baidu/ps-aladdin/starmap *master

```go
var taskConf = map[string]func(context.Context, *asynq.Task) error{
    "dumpfeature": HandlerDumpFeature,
}

// 中间件列表配置
var mids = []asynq.MiddlewareFunc{
    InitLoggerMiddleware,
    UpdateDatasetStatusMiddleware,
    asynqserver.TaskRunMiddleware,
}

func initAsyncTaskWorker(ctx context.Context) {
    go func() {
        if err := asynqserver.StartAsynqWorker(ctx, mids, taskConf); err != nil {
            log.Printf("failed to start asynq server: %v\n", err)
            panic(err)
        }
    }()
    log.Println("start asynq.Serve")
}

func HandlerDumpFeature(ctx context.Context, t *asynq.Task) error {...}
```

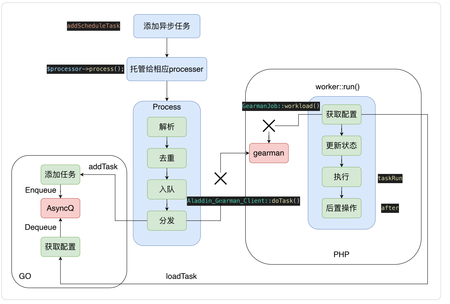
任务入队

```go
TaskReq := &asynqserver.TaskReq{
        TaskID:   1,
        TaskName: "dumpfeature",
        Params: map[string]any{
            "path": tmpFile,
            "id":   id,
        },
    }

    err = asynqserver.TaskEnqueue(ctx, TaskReq)
    if err != nil {
        resp.Msg = "添加任务失败:" + err.Error()
        resp.Code = types.RuntimeError
        return ghttp.NewJSONResponse(200, resp)
    }

```

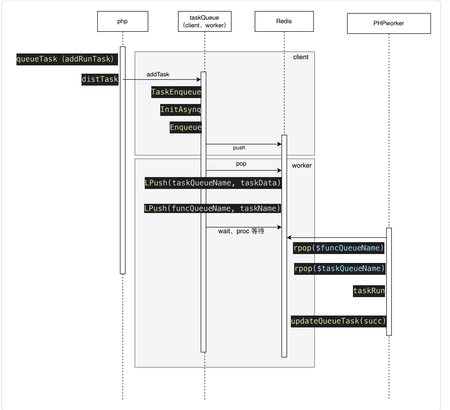
2、PHP接入（跨语言服务接入）

平台PHP服务有较多异步任务，且有一套完整的调度前后的任务执行框架，调度能力由gearman提供。现将gearman替换为task-queue。

阿拉丁平台 gearman任务迁移



GO worker 与PHP worker 通信，完成跨语言服务的队列接入：



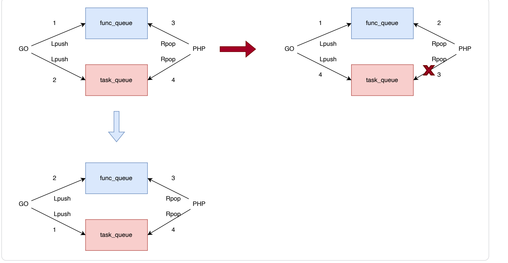
坑：

高并发情况下，

php worker持续在func_queue中rpop。

预期 左上角顺序，实际偶尔发生右上角顺序，导致任务堆积无法消费。

调换 Lpush顺序解决



原解决方案：

1、2 改造为原子操作，原子操作需要操作的key在同一个哈希槽，而func_queue对应很多个task_queue, 这样所有任务的跨服务调度都会被分配到同一个分片，导致负载不均衡，因此无法使用。

3、监控

http://10.11.133.62:8090/