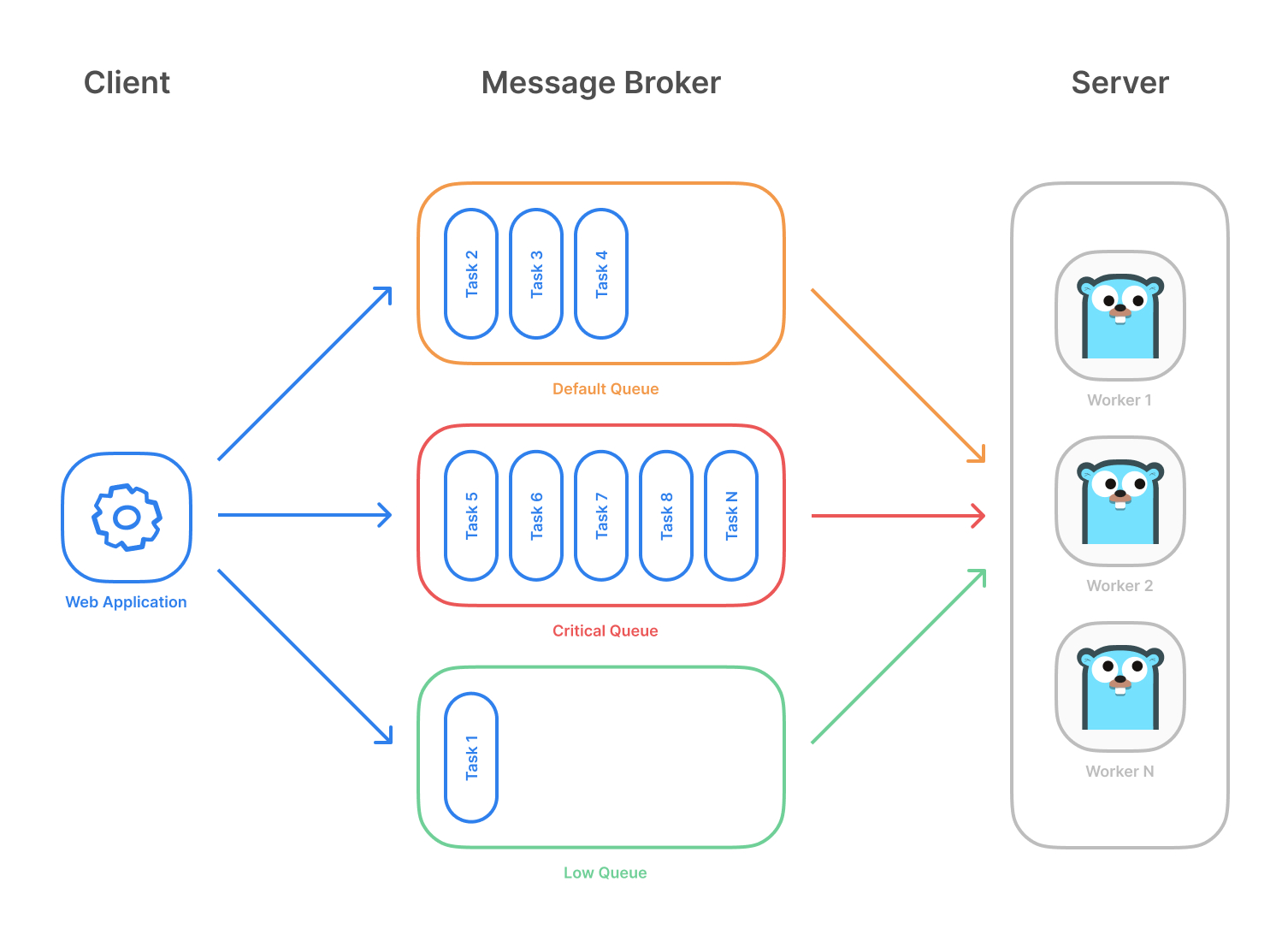
基于redis的mq组件 Asynq使用小结

## 概述：

Asynq 是一个 Go 库，用于对任务进行排队并与工作线程异步处理它们。它由[Redis](https://redis.io/)支持，旨在可扩展且易于入门。并且有对应webUi界面可以进行图形化管理。相比kafka、rabbitMq等传统mq消息中间件，更加轻量化

任务队列用作跨多台机器分配工作的机制。一个系统可以由多个工作服务器和代理组成，从而实现高可用性和水平扩展。支持单点redis和redis集群



主要功能：

1. 队列
2. 优先级队列
3. 定时任务
4. 周期性任务
5. 自动重试
6. 超时、取消

任务的基本状态：

* **Scheduled**：任务正在等待将来处理（仅适用于带有ProcessAt或ProcessIn选项的任务）。
* **待处理**：任务已准备好处理，并将由空闲的工作人员接手。
* **Active**：任务正在由工作人员处理（即随任务调用处理程序）。
* **重试**：worker 处理任务失败，等待将来重试。
* **已存档**：任务达到最大重试次数并存储在存档中以供手动检查。
* **已完成**：任务已成功处理并保留，直到保留 TTL 过期（仅适用于带有选项的任务Retention）。

### 普通队列使用

* 1. Client（相当于生产者）

1、asynq.NewClient(),获取客户端实例

2、func NewTask(typename string, payload []byte, opts ...Option) \*Task 新建任务

Typename：为队列任务名

Payload：消息内容

3、func (c \*Client) Enqueue(task \*Task, opts ...Option) (\*TaskInfo, error)

func (c \*Client) EnqueueContext(ctx context.Context, task \*Task, opts ...Option) (\*TaskInfo, error)

将前面创建的任务加入队列

相当于延时队列任务：

使用enqueue的配置项：

func ProcessIn(d time.Duration) Option在时间d后执行

func ProcessAt(t time.Time) Option 在t时间执行

* 1. server（相当于消费者）

1、func NewServer(r RedisConnOpt, cfg Config) \*Server

创建server实例

2、新建消费者逻辑处理方法，需要为type HandlerFunc func(context.Context, \*Task) error

实现 Handler方法

3、func (srv \*Server) Run(handler Handler) error

传入hanler方法实现，开启消费者监听

升级版用法：

将client和server中用到的payload、typename、redis配置等单独封装出来，方便使用。

在server中，使用func NewServeMux() \*ServeMux 方法聚合所有的消费者handler方法，再使用run方法开启监听

### 错误重试机制

任务在handler返回error时会记为任务失败，进入重试。如果任务用尽其所有重试次数（默认值：25），该任务将移至存档以进行调试和检查，并且不会自动重试（您仍然可以使用 CLI 或 WebUI 手动运行任务）

#### 每个任务的最大重试次数（asynq.MaxRetry(5)）

在NewTask函数传入可选参数，asynq.MaxRetry(n int)，配置重试次数

不传入该参数，则使用默认重试次数25

#### 失败的任务可以再次重试之前等待的持续时间（即延迟）

* + 默认为指数退避
  + 自定义延迟：使用asynq.Config配置项的RetryDelayFunc RetryDelayFunc，在创建server时传入，可自定义单个task的重试次数。
  + type RetryDelayFunc func(n int, e error, t \*Task) time.Duration

#### 是否消耗任务的重试次数

指非故障错误。可指定对应的错误类型，但不触发重试

Config中的IsFailure func(error) bool

#### 是否跳过重试并将任务直接发送到存档

如果Handler.ProcessTask返回SkipRetry错误，则无论剩余重试次数是多少，任务都将被存档。返回的错误可以是SkipRetry错误，也可以是包装SkipRetry错误的错误。

### 优先级队列：

Asynq默认创建名为default的单个队列来处理所有任务。

使用优先级队列需要为每个任务分配优先级，可以创建多个不同优先级的队列。

消费者：

在server的asynq.Config的queues进行创建

srv := asynq.NewServer(redis, asynq.Config{

Concurrency: 10,

Queues: map[string]int{

"critical": 6,

"default": 3,

"low": 1,

},

})

生产者：

在创建任务时，传入`asynq.Queue` 的option.

err := client.Enqueue(task, asynq.Queue("critical"))

队列名与消费者的队列名要一致。

严格优先级：

如果需要创建多个队列，并且需要将一个队列中的所有任务优先于其他队列处理，可以使用StrictPriorityoption.

srv := asynq.NewServer(redis, asynq.Config{

Concurrency: 10,

Queues: map[string]int{

"critical": 3,

"default": 2,

"low": 1,

},

StrictPriority: true, // strict mode!

})

这将创建一个Background具有三个队列的实例：critical、default和具有严格优先级的low 。在严格优先级模式下，始终首先处理具有较高优先级的队列，只有当所有其他具有较高优先级的队列都为空时，才会处理具有较低优先级的队列。

所以在这个例子中，关键队列中的任务总是首先被处理。如果关键队列为空，则处理默认队列。如果关键队列和默认队列都为空，则处理低队列。

### 周期性任务

定时向队列中添加任务，支持spec表达式和@every的写法

scheduler := asynq.NewScheduler(redisConnOpt, nil)

task := asynq.NewTask("example\_task", nil)

// You can use cron spec string to specify the schedule. entryID, err := scheduler.Register("\* \* \* \* \*", task)if err != nil {

log.Fatal(err)

}log.Printf("registered an entry: %q\n", entryID)

// You can use "@every <duration>" to specify the interval.entryID, err = scheduler.Register("@every 30s", task)if err != nil {

log.Fatal(err)

}log.Printf("registered an entry: %q\n", entryID)

// You can also pass options.entryID, err = scheduler.Register("@every 24h", task, asynq.Queue("myqueue"))if err != nil {

log.Fatal(err)

}log.Printf("registered an entry: %q\n", entryID)

if err := scheduler.Run(); err != nil {

log.Fatal(err)

}

在调用的同时，需要对应的操作，定时关闭

如果调度程序无法将任务排入队列，您可以提供一个函数来处理错误。

func handleEnqueueError(task \*asynq.Task, opts []asynq.Option, err error) {

// your error handling logic

}

scheduler := asynq.NewScheduler(

redisConnOpt,

&asynq.SchedulerOpts{

EnqueueErrorHandler: handleEnqueueError,

},

)

动态添加周期性任务:

https://github.com/hibiken/asynq/wiki/Dynamic-Periodic-Task