

Machinery

author: asong 公众号: Golang梦工厂

最近想学习一下 `machinery` 的基本使用，利用搜索引擎搜索了一下，教程很少，官方文档又是英文的，所以就打算翻译一下官方文档，并出一篇入门教程，所以就有了官方中文文档。

我会定期维护的，小伙伴们可以放心观看，如果那里翻译有问题，可以提出来，我进行改进。可以加入我的交流群，一起学习技术。加我VX: `asong971011`，拉你入群哦。或者关注我的公众号: Golang梦工厂，第一时间观看优质文章~~~。



目录

Machinery

目录

- V2 实验

- 第一步

- 配置

 - Broker

 - AMQP

 - Redis

 - AWS SQS

 - GCP Pub/Sub

 - 默认队列 (DefaultQueue)

 - ResultBackend

 - redis

 - Memcache

 - AMQP

 - MongoDB

 - ResultsExpireIn

 - AMQP

 - DynamoDB

- Redis
 - GCPPubSub
- 自定义Logger
- Server
- Workers
- Tasks
 - 注册任务 (Registering Tasks)
 - 签名(Signatures)
 - 支持的类型(Supported Types)
 - 发送任务(Sending Tasks)
 - 延时任务(Delayed Tasks)
 - 重试任务(Retry Tasks)
 - 获取等待中的任务(Get Pending Tasks)
 - Keeping Results
 - 错误处理 (Error Handling)
- 工作流(Workflows)
 - Groups
 - Chords
- Chains
- 定时任务和工作流 (Periodic Tasks & Workflows)
 - 定时任务(Periodic Tasks)
 - 定时任务组(Periodic Groups)
 - 定时调用链(Periodic Chains)
 - Periodic Chord
- 开发(Development)
 - 环境(Requirements)
 - 依赖(Dependencies)
 - Testing

V2 实验

请注意使用V2版本直到其准备完成为止，因为V2正在开发之中且有可能发生重大更改。

您可以使用当前的V2来避免必须导入所有不使用的代理和后端依赖项。

使用factory代理，你将需要注入代理和后端对象到服务器构造函数：

```
import (
    "github.com/RichardKnop/machinery/v2"
    backendsiface "github.com/RichardKnop/machinery/v1/backends/iface"
    brokersiface "github.com/RichardKnop/machinery/v1/brokers/iface"
)

var broker brokersiface.Broker
var backend backendsiface.Backend
server, err := machinery.NewServer(cnf, broker, backend)
if err != nil {
    // do something with the error
}
```

第一步

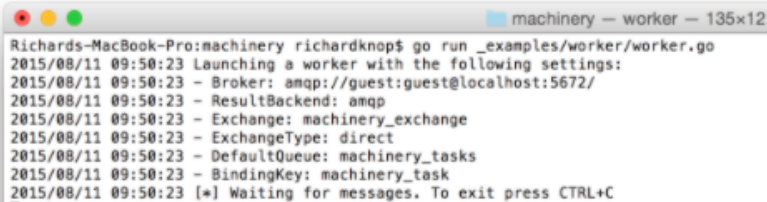
添加 `machinery` 库到你的 `$GOPATH/src`:

```
$ go get github.com/RichardKnop/machinery/v1
```

首先，我们需要定义一些任务。在 `example/tasks/tasks.go` 中查看示例任务。去看看几个例子吧。

第二步，你将需要启动一个工作进程:

```
go run example/machinery.go worker
```



```
machinery -- worker -- 135x12
Richards-MacBook-Pro:machinery richardknop$ go run _examples/worker/worker.go
2015/08/11 09:50:23 Launching a worker with the following settings:
2015/08/11 09:50:23 - Broker: amqp://guest:guest@localhost:5672/
2015/08/11 09:50:23 - ResultBackend: amqp
2015/08/11 09:50:23 - Exchange: machinery_exchange
2015/08/11 09:50:23 - ExchangeType: direct
2015/08/11 09:50:23 - DefaultQueue: machinery_tasks
2015/08/11 09:50:23 - BindingKey: machinery_task
2015/08/11 09:50:23 [*] Waiting for messages. To exit press CTRL+C
```

最后，一旦你有一个任务正在运行并且等待消耗，你可以开启一些发送任务。

```
$ go run example/machinery.go send
```

你能看到这些任务正在被异步的处理。

```
machinery — worker — 142x39
cf37490f-17e3-442d-b580-e1b47626927d", "GroupTaskCount": 3, "Args": [{"Type": "int64", "Value": 2}, {"Type": "int64", "Value": 2}], "Immutable": false, "OnS
uccess": null, "OnError": null, "ChordCallback": {"UUID": "chord_3dae5136-baa0-417a-ab36-8355f55c3579", "Name": "multiply", "RoutingKey": "", "GroupUUID"
: "", "GroupTaskCount": 0, "Args": null, "Immutable": false, "OnSuccess": null, "OnError": null, "ChordCallback": null}}
2015/08/11 09:52:23 Processed task_f9bd00c0-092f-4e4f-b073-14fd7fad1f1a. Result = 4
2015/08/11 09:52:23 Received new message: {"UUID": "task_b8880b6b-4393-45c9-b80a-3b500a2ddc80", "Name": "add", "RoutingKey": "", "GroupUUID": "group_
cf37490f-17e3-442d-b580-e1b47626927d", "GroupTaskCount": 3, "Args": [{"Type": "int64", "Value": 5}, {"Type": "int64", "Value": 6}], "Immutable": false, "OnS
uccess": null, "OnError": null, "ChordCallback": {"UUID": "chord_3dae5136-baa0-417a-ab36-8355f55c3579", "Name": "multiply", "RoutingKey": "", "GroupUUID"
: "", "GroupTaskCount": 0, "Args": null, "Immutable": false, "OnSuccess": null, "OnError": null, "ChordCallback": null}}
2015/08/11 09:52:23 Processed task_b8880b6b-4393-45c9-b80a-3b500a2ddc80. Result = 11
2015/08/11 09:52:23 Received new message: {"UUID": "chord_3dae5136-baa0-417a-ab36-8355f55c3579", "Name": "multiply", "RoutingKey": "", "GroupUUID": "
", "GroupTaskCount": 0, "Args": [{"Type": "int64", "Value": 11}, {"Type": "int64", "Value": 4}, {"Type": "int64", "Value": 2}], "Immutable": false, "OnSuccess":
null, "OnError": null, "ChordCallback": null}}
2015/08/11 09:52:23 Processed chord_3dae5136-baa0-417a-ab36-8355f55c3579. Result = 88
2015/08/11 09:52:23 Received new message: {"UUID": "task_a1741d68-aa72-4909-aeb2-81494b33a281", "Name": "add", "RoutingKey": "", "GroupUUID": "", "Gro
upTaskCount": 0, "Args": [{"Type": "int64", "Value": 1}, {"Type": "int64", "Value": 1}], "Immutable": false, "OnSuccess": [{"UUID": "task_acce86a0-c89e-4153-
8e0e-757740582c92", "Name": "add", "RoutingKey": "", "GroupUUID": "", "GroupTaskCount": 0, "Args": [{"Type": "int64", "Value": 2}, {"Type": "int64", "Value": 2
}], "Immutable": false, "OnSuccess": [{"UUID": "task_16a1b7bb-6b33-431f-8824-f310fd31a90c", "Name": "add", "RoutingKey": "", "GroupUUID": "", "GroupTaskCo
unt": 0, "Args": [{"Type": "int64", "Value": 5}, {"Type": "int64", "Value": 6}], "Immutable": false, "OnSuccess": [{"UUID": "task_1fe09228-558f-4528-ba32-731
ecab30212", "Name": "multiply", "RoutingKey": "", "GroupUUID": "", "GroupTaskCount": 0, "Args": [{"Type": "int64", "Value": 4}], "Immutable": false, "OnSucces
s": null, "OnError": null, "ChordCallback": null}], "OnError": null, "ChordCallback": null}], "OnError": null, "ChordCallback": null}}
2015/08/11 09:52:23 Processed task_a1741d68-aa72-4909-aeb2-81494b33a281. Result = 2
2015/08/11 09:52:23 Received new message: {"UUID": "task_acce86a0-c89e-4153-8e0e-757740582c92", "Name": "add", "RoutingKey": "", "GroupUUID": "", "Gro
upTaskCount": 0, "Args": [{"Type": "int64", "Value": 2}, {"Type": "int64", "Value": 2}, {"Type": "int64", "Value": 2}], "Immutable": false, "OnSuccess": [{"UUID
": "task_16a1b7bb-6b33-431f-8824-f310fd31a90c", "Name": "add", "RoutingKey": "", "GroupUUID": "", "GroupTaskCount": 0, "Args": [{"Type": "int64", "Value": 5
}, {"Type": "int64", "Value": 6}], "Immutable": false, "OnSuccess": [{"UUID": "task_1fe09228-558f-4528-ba32-731ecab30212", "Name": "multiply", "RoutingKey
": "", "GroupUUID": "", "GroupTaskCount": 0, "Args": [{"Type": "int64", "Value": 4}], "Immutable": false, "OnSuccess": null, "OnError": null, "ChordCallback": n
ull}], "OnError": null, "ChordCallback": null}], "OnError": null, "ChordCallback": null}}
2015/08/11 09:52:23 Processed task_acce86a0-c89e-4153-8e0e-757740582c92. Result = 6
2015/08/11 09:52:23 Received new message: {"UUID": "task_16a1b7bb-6b33-431f-8824-f310fd31a90c", "Name": "add", "RoutingKey": "", "GroupUUID": "", "Gro
upTaskCount": 0, "Args": [{"Type": "int64", "Value": 6}, {"Type": "int64", "Value": 5}, {"Type": "int64", "Value": 6}], "Immutable": false, "OnSuccess": [{"UUID
": "task_1fe09228-558f-4528-ba32-731ecab30212", "Name": "multiply", "RoutingKey": "", "GroupUUID": "", "GroupTaskCount": 0, "Args": [{"Type": "int64", "Val
ue": 4}], "Immutable": false, "OnSuccess": null, "OnError": null, "ChordCallback": null}], "OnError": null, "ChordCallback": null}}
2015/08/11 09:52:23 Processed task_16a1b7bb-6b33-431f-8824-f310fd31a90c. Result = 17
2015/08/11 09:52:23 Received new message: {"UUID": "task_1fe09228-558f-4528-ba32-731ecab30212", "Name": "multiply", "RoutingKey": "", "GroupUUID": ""
", "GroupTaskCount": 0, "Args": [{"Type": "int64", "Value": 17}, {"Type": "int64", "Value": 4}], "Immutable": false, "OnSuccess": null, "OnError": null, "ChordCa
llback": null}}
2015/08/11 09:52:23 Processed task_1fe09228-558f-4528-ba32-731ecab30212. Result = 68
```

配置

[Config](#) package 可以从环境变量或 `yaml` 文件中进行加载，例如，从环境变量中加载配置：

```
cnf, err := config.NewFromEnvironment()
```

或者从 `YAML` 文件中加载：

```
cnf, err := config.NewFromYaml("config.yaml", true)
```

第二个布尔类型标志允许每10秒实时重新加载配置。使用`false`则禁用实时重新加载。

`Machinery` 的配置由 `Config` 结构体封装，并使用它作为对象的依赖项注入。

Broker

消息代理。当前支持的代理如下：

AMQP

使用AMQP URL，格式如下：

```
amqp://[username:password@]host[:port]
```

例如：

```
1. amqp://guest:guest@localhost:5672
```

AMQP还支持多个代理的url。您需要在 `MultipleBrokerSeparator` 字段中指定URL分隔符。

Redis

使用如下格式之一的Redis URL:

```
redis://[password@]host[port][/db_num]
redis+socket://[password@]/path/to/file.sock[:db_num]
```

例如:

```
1. redis://localhost:6379, or with password redis://password@localhost:6379
2. redis+socket://password@/path/to/file.sock:/0
```

AWS SQS

使用 `AWS SQS` URL 格式如下:

```
https://sqs.us-east-2.amazonaws.com/123456789012
```

获取更多信息可以看 [AWS SQS 文档](#)。此外，还需要配置AWS_REGION，否则将抛出错误。

手动配置 `sqs` 客户端如下:

```
var sqsClient = sqs.New(session.Must(session.NewSession(&aws.Config{
    Region:      aws.String("YOUR_AWS_REGION"),
    Credentials: credentials.NewStaticCredentials("YOUR_AWS_ACCESS_KEY",
"YOUR_AWS_ACCESS_SECRET", ""),
    HTTPClient:  &http.Client{
        Timeout: time.Second * 120,
    },
})))
var visibilityTimeout = 20
var cnf = &config.Config{
    Broker:      "YOUR_SQS_URL"
    DefaultQueue: "machinery_tasks",
    ResultBackend: "YOUR_BACKEND_URL",
    SQS: &config.SQSConfig{
        Client: sqsClient,
        // if VisibilityTimeout is nil default to the overall visibility timeout
        setting for the queue
    }
}
```

```
//  
https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-visibility-timeout.html  
    VisibilityTimeout: &visibilityTimeout,  
    WaitTimeSeconds: 30,  
},  
}
```

GCP Pub/Sub

使用 GCP Pub Sub URL 格式如下:

```
gcppubsub://YOUR_GCP_PROJECT_ID/YOUR_PUBSUB_SUBSCRIPTION_NAME
```

使用配置 Pub/Sub 客户端如下:

```
pubsubClient, err := pubsub.NewClient(  
    context.Background(),  
    "YOUR_GCP_PROJECT_ID",  
    option.WithServiceAccountFile("YOUR_GCP_SERVICE_ACCOUNT_FILE"),  
)  
  
cnf := &config.Config{  
    Broker:  
    "gcppubsub://YOUR_GCP_PROJECT_ID/YOUR_PUBSUB_SUBSCRIPTION_NAME"  
    DefaultQueue:    "YOUR_PUBSUB_TOPIC_NAME",  
    ResultBackend:    "YOUR_BACKEND_URL",  
    GCPPubSub: config.GCPPubSubConfig{  
        Client: pubsubClient,  
    },  
}
```

默认队列 (DefaultQueue)

默认队列的名字: e.g. `machinery_tasks`.

ResultBackend

`Result backend` 是为了保存任务的状态和结果集。

目前支持的 `backend` 如下:

redis

使用如下格式之一的 Redis URL:

```
redis://[password@]host[port][/db_num]
redis+socket://[password@]/path/to/file.sock[:db_num]
```

例如：

1. `redis://localhost:6379`，或者带密钥 `redis://password@localhost:6379`
2. `redis+socket://password@/path/to/file.sock:/0`
3. 集群 `redis://host1:port1,host2:port2,host3:port3`
4. 集群带密钥 `redis://pass@host1:port1,host2:port2,host3:port3`

Memcache

使用 Memcache URL 格式如下：

```
memcache://host1[:port1][,host2[:port2],...[,hostN[:portN]]]
```

例如：

1. `memcache://localhost:11211` 一个实例
2. `memcache://10.0.0.1:11211,10.0.0.2:11211` 集群使用

AMQP

使用 AMQP URL 格式如下：

```
amqp://[username:password@]host[:port]
```

例如：

1. `amqp://guest:guest@localhost:5672`

不推荐使用AMQP作为 `result backend`。更多信息可以查看[keeping Results](#)

MongoDB

使用 MongoDB URL 格式如下：

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]
```

例如：

1. `mongodb://localhost:27017/taskresults`

更多信息请查看 [MongoDB文档](#)

ResultsExpireIn

任务结果的存储时间，以秒为单位。默认值为3600(1小时)。

AMQP

`RabbitMQ` 的相关配置如下。如果你使用其他 broker 或者 backend 则不需要配置。

- `Exchange`: exchange name, e.g. `machinery_exchange`
- `ExchangeType`: exchange type, e.g. `direct`
- `QueueBindingArguments`: 绑定到AMQP队列时使用的附加参数的可选映射
- `BindingKey`: 队列用这个键绑定到交换器，例如 `machinery_task`
- `PrefetchCount`: 预取多少任务(如果有长时间运行的任务，设置为1)

DynamoDB

`DynamoDB` 相关联配置。如果你使用其他 backend 可以不配置。

- `TaskStatesTable`: 用于保存任务状态的自定义表名。默认是 `task_states`，确保首先在AWS管理中创建这个表，使用TaskUUID作为表的主键。
- `GroupMetasTable`: 用于保存组元数据的自定义表名。默认选项是 `group_metas`，并确保首先在AWS管理中创建这个表，使用GroupUUID作为表的主键。例如:

```
dynamodb:
  task_states_table: 'task_states'
  group_metas_table: 'group_metas'
```

如果没有找到这些表，将抛出一个致命错误。

如果希望使记录过期，可以在AWS admin中为这些表配置TTL字段。TTL字段是根据服务器配置中的ResultsExpireIn值设置的。查看更多信息点击该文档<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/howitworks-ttl.html>。

Redis

`redis` 相关配置。如果使用其他的 backend 则不需要配置。

查看: [config](#) (TODO)

GCPPubSub

`GCPPubSub` 相关配置。如果使用其他的 backend 则不需要配置。

查看: [config](#) (TODO)

自定义Logger

你可以通过实现以下接口来定义一个自定义Logger:

```
type Interface interface {
    Print(...interface{})
    Printf(string, ...interface{})
    Println(...interface{})

    Fatal(...interface{})
    Fatalf(string, ...interface{})
    Fatalln(...interface{})

    Panic(...interface{})
    Panicf(string, ...interface{})
    Panicln(...interface{})
}
```

然后通过调用 `github.com/RichardKnop/machinery/v1/log` 包中的 `Set` 函数:

```
log.Set(myCustomLogger)
```

Server

`Machinery` 库必须在使用前实例化。实现方法是创建一个 `Server` 实例。

`Server` 是 `Machinery` 配置和注册任务的基本对象，例如:

```
import (
    "github.com/RichardKnop/machinery/v1/config"
    "github.com/RichardKnop/machinery/v1"
)

var cnf = &config.Config{
    Broker:      "amqp://guest:guest@localhost:5672/",
    DefaultQueue: "machinery_tasks",
    ResultBackend: "amqp://guest:guest@localhost:5672/",
    AMQP: &config.AMQPConfig{
        Exchange:      "machinery_exchange",
        ExchangeType: "direct",
        BindingKey:     "machinery_task",
    },
},

server, err := machinery.NewServer(cnf)
if err != nil {
```

```
// do something with the error
}
```

Workers

为了消费任务，你需要有一个或多个worker正在运行。运行worker所需要的只是一个具有已注册任务的 `Server` 实例。例如：

```
worker := server.NewWorker("worker_name", 10)
err := worker.Launch()
if err != nil {
    // do something with the error
}
```

每个worker将只使用已注册的任务。对于队列中的每个任务，`Worker.Process()`方法将在一个goroutine中运行。可以使用 `server.NewWorker` 的第二参数来限制并发运行的`worker.Process()`调用的数量(每个worker)。

示例：参数1将序列化任务执行，而参数0将使并发执行的任务数量不受限制(默认情况下)。

Tasks

tasks是 `Machinery` 应用的一个构件块。任务是一个函数，它定义当worker收到消息时发生的事情。

每个任务至少要返回一个 `error` 作为返回值。除了错误任务现在可以返回任意数量的参数。

一个任务例子如下：

```
func Add(args ...int64) (int64, error) {
    sum := int64(0)
    for _, arg := range args {
        sum += arg
    }
    return sum, nil
}

func Multiply(args ...int64) (int64, error) {
    sum := int64(1)
    for _, arg := range args {
        sum *= arg
    }
    return sum, nil
}

// You can use context.Context as first argument to tasks, useful for open
tracing
```

```
func TaskWithContext(ctx context.Context, arg Arg) error {
    // ... use ctx ...
    return nil
}

// Tasks need to return at least error as a minimal requirement
func DummyTask(arg string) error {
    return errors.New(arg)
}

// You can also return multiple results from the task
func DummyTask2(arg1, arg2 string) (string, string, error) {
    return arg1, arg2, nil
}
```

注册任务（Registering Tasks）

在你的 `workders` 能消费一个任务前，你需要将它注册到服务器。这是通过给任务分配一个唯一的名称来实现的：

```
server.RegisterTasks(map[string]interface{}{
    "add":      Add,
    "multiply": Multiply,
})
```

`tasks` 可以一个接一个的注册：

```
server.RegisterTask("add", Add)
server.RegisterTask("multiply", Multiply)
```

简单地说，当worker收到这样的消息时：

```
{
  "UUID": "48760a1a-8576-4536-973b-da09048c2ac5",
  "Name": "add",
  "RoutingKey": "",
  "ETA": null,
  "GroupUUID": "",
  "GroupTaskCount": 0,
  "Args": [
    {
      "Type": "int64",
      "Value": 1,
    },
    {
      "Type": "int64",
      "Value": 1,
    }
  ]
}
```

```

],
"Immutable": false,
"RetryCount": 0,
"RetryTimeout": 0,
"OnSuccess": null,
"OnError": null,
"ChordCallback": null
}

```

他将调用 `Add(1,1)`。每个任务也应该返回一个错误，这样我们就可以处理失败。理想情况下，任务应该是幂等的，这意味着当使用相同的参数多次调用任务时，不会出现意外的结果。

签名(Signatures)

签名包装了任务的调用参数，执行选项（比如不可变性）和成功/错误回调任务。所以可以通过网络发送到 `workers`。任务签名实现了一个简单的接口：

```

// Arg represents a single argument passed to invocation fo a task
type Arg struct {
    Type    string
    Value   interface{}
}

// Headers represents the headers which should be used to direct the task
type Headers map[string]interface{}

// Signature represents a single task invocation
type Signature struct {
    UUID           string
    Name           string
    RoutingKey     string
    ETA            *time.Time
    GroupUUID      string
    GroupTaskCount int
    Args           []Arg
    Headers        Headers
    Immutable      bool
    RetryCount     int
    RetryTimeout   int
    OnSuccess      []*Signature
    OnError        []*Signature
    ChordCallback  *Signature
}

```

`UUID` 是任务的唯一ID。您可以自己设置，也可以自动生成。

`Name` 是在服务器实例中注册任务的唯一任务名称。

`RoutingKey` 用于将任务路由到正确的队列。如果将其保留为空，则默认行为是将其设置为直接交换类型的默认队列的绑定键，以及其他交换类型的默认队列名。

`ETA` ETA是用于延迟任务的时间戳。如果填为nil，任务被推送到worker讲立即执行。

`GroupUUID`，`GroupTaskCount` 对于创建任务组很有用。

`Args` 是在worker执行任务时传递给任务的参数列表。

`Headers` 是将任务发布到AMQP队列时使用的 `headers` 列表。

`Immutable` 是一个标志，它定义了执行任务的结果是否可以被修改。这对于OnSuccess回调是很重要的。不可变任务不会将其结果传递给它的成功回调，而可变任务会将其结果提前发送给回调任务。长话短说，如果您想将调用链中的第一个任务的结果传递给第二个任务，那么将不可变设置为false。

`RetryCount` 指定应该重试失败的任务的次数(缺省值为0)。失败尝试是在一定的时间间隔内，在每一次失败后都会等待下一次的调度。

`RetryTimeout` 指定在将任务重新发送到队列进行重试之前需要等待多长时间。默认行为是在每次重试失败后使用斐波那契序列增加超时。

`OnSuccess` 定义了任务成功执行后将被调用的任务。他是一个 `signature` 类型的切片。

`OnError` 定义任务执行失败后将被调用的任务。传递给错误回调函数的第一个参数是失败任务返回的错误字符串。

`ChordCallback` 用于创建对一组任务的回调。

支持的类型(Supported Types)

`Machinery` 在将任务发送到代理之前将其编码为JSON。任务结果也作为JSON编码的字符串存储在 `backend`。因此，只能支持带有原生JSON表示的类型。目前支持的类型有：

- `bool`
- `int`
- `int8`
- `int16`
- `int32`
- `int64`
- `uint`
- `uint8`
- `uint16`
- `uint32`
- `uint64`
- `float32`
- `float64`
- `string`
- `[]bool`
- `[]int`
- `[]int8`
- `[]int16`

- []int32
- []int64
- []uint
- []uint8
- []uint16
- []uint32
- []uint64
- []float32
- []float64
- []string

发送任务(Sending Tasks)

可以通过将 `Signature` 实例传递给 `Server` 实例来调用任务。例句：

```
import (
    "github.com/RichardKnop/machinery/v1/tasks"
)

signature := &tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 1,
        },
        {
            Type: "int64",
            Value: 1,
        },
    },
}

asyncResult, err := server.SendTask(signature)
if err != nil {
    // failed to send the task
    // do something with the error
}
```

延时任务(Delayed Tasks)

可以通过在任务 `signature` 上设置ETA时间戳字段来延迟任务。

```
// Delay the task by 5 seconds
eta := time.Now().UTC().Add(time.Second * 5)
signature.ETA = &eta
```

重试任务(Retry Tasks)

在将任务声明为失败之前，可以设置多次重试尝试。斐波那契序列将用于在一段时间内分隔重试请求。(详细信息请参见RetryTimeout)。

```
// If the task fails, retry it up to 3 times
signature.RetryCount = 3
```

或者，你可以使用 `return.tasks.ErrRetryTaskLater` 返回任务并指定重试的持续时间，例如：

```
return tasks.NewErrRetryTaskLater("some error", 4 * time.Hour)
```

获取等待中的任务(Get Pending Tasks)

当前在队列中等待被worker消耗的任务可以被检查到，例如。

```
server.GetBroker().GetPendingTasks("some_queue")
```

当前只支持redis队列代理

Keeping Results

如果配置了 `result backend`，则任务状态和结果将被持久化。可能的状态：

```
const (
    // StatePending - initial state of a task
    StatePending = "PENDING"
    // StateReceived - when task is received by a worker
    StateReceived = "RECEIVED"
    // StateStarted - when the worker starts processing the task
    StateStarted = "STARTED"
    // StateRetry - when failed task has been scheduled for retry
    StateRetry = "RETRY"
    // StateSuccess - when the task is processed successfully
    StateSuccess = "SUCCESS"
    // StateFailure - when processing of the task fails
    StateFailure = "FAILURE"
)
```

当使用AMQP作为 `result backend` 时，任务状态将持久化到每个任务的单独队列中。尽管 RabbitMQ 可以扩展到数千个队列，但当你希望运行大量并行任务时，强烈建议使用一个更好的 `result backend` (例如 Memcache)。

```
// TaskResult represents an actual return value of a processed task
type TaskResult struct {
    Type string    `bson:"type"`
    Value interface{} `bson:"value"`
}

// TaskState represents a state of a task
type TaskState struct {
    TaskUUID string    `bson:"_id"`
    State    string    `bson:"state"`
    Results  []*TaskResult `bson:"results"`
    Error    string    `bson:"error"`
}

// GroupMeta stores useful metadata about tasks within the same group
// E.g. UUIDs of all tasks which are used in order to check if all tasks
// completed successfully or not and thus whether to trigger chord callback
type GroupMeta struct {
    GroupUUID string    `bson:"_id"`
    TaskUUIDs []string `bson:"task_uuids"`
    ChordTriggered bool    `bson:"chord_triggered"`
    Lock      bool    `bson:"lock"`
}
```

`TaskResult` 表示已处理任务返回值的切片类型。

每当任务状态改变时，`TaskState` 结构将被序列化并存储。

`GroupMeta` 存储关于同一组内任务的有用元数据。例如，所有任务的uuid，用于检查所有任务是否成功完成，从而是否触发回调。

`AsyncResult` 对象允许你检查一个任务的状态：

```
taskState := asyncResult.GetState()
fmt.Printf("Current state of %v task is:\n", taskState.TaskUUID)
fmt.Println(taskState.State)
```

有两个方便的方法来检查任务状态：

```
asyncResult.GetState().IsCompleted()
asyncResult.GetState().IsSuccess()
asyncResult.GetState().IsFailure()
```

你也可以做一个同步阻塞调用来等待一个任务结果。


```

results, err := asyncResult.Get(time.Duration(time.Millisecond * 5))
if err != nil {
    // getting result of a task failed
    // do something with the error
}
for _, result := range results {
    fmt.Println(result.Interface())
}

```

错误处理 (Error Handling)

当一个任务返回错误时，默认的行为首先尝试重试该任务如果是开启了重试功能，否则记录错误，然后最终调用任何错误回调。

可以自定义错误处理，你可以设置一个自定义的错误处理程序上的 `worker`，它可以做更多的记录，重试失败和错误回调的触发：

```

worker.SetErrorHandler(func (err error) {
    customHandler(err)
})

```

工作流(Workflows)

运行单个异步任务是不错的，但您通常希望设计一个任务工作流，以编排好的方式执行。有两个有用的函数可以帮助您设计工作流。

Groups

`Group` 是一组任务，它们将相互独立地并行执行。例如：

```

import (
    "github.com/RichardKnop/machinery/v1/tasks"
    "github.com/RichardKnop/machinery/v1"
)

signature1 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 1,
        },
        {

```

```

        Type: "int64",
        Value: 1,
    },
},
}

signature2 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 5,
        },
        {
            Type: "int64",
            Value: 5,
        },
    },
}

group, _ := tasks.NewGroup(&signature1, &signature2)
asyncResults, err := server.SendGroup(group, 0) //The second parameter
specifies the number of concurrent sending tasks. 0 means unlimited.
if err != nil {
    // failed to send the group
    // do something with the error
}

```

`SendGroup` 返回一个 `AsyncResult` 对象的切片。所以你可以做一个阻塞调用，等待组 `groups` 任务的结果：

```

for _, asyncResult := range asyncResults {
    results, err := asyncResult.Get(time.Duration(time.Millisecond * 5))
    if err != nil {
        // getting result of a task failed
        // do something with the error
    }
    for _, result := range results {
        fmt.Println(result.Interface())
    }
}

```

Chords

`Chord` 允许你定一个回调任务在 `groups` 中的所有任务执行结束后被执行。

```

import (
    "github.com/RichardKnop/machinery/v1/tasks"
    "github.com/RichardKnop/machinery/v1"
)

signature1 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 1,
        },
        {
            Type: "int64",
            Value: 1,
        },
    },
}

signature2 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 5,
        },
        {
            Type: "int64",
            Value: 5,
        },
    },
}

signature3 := tasks.Signature{
    Name: "multiply",
}

group := tasks.NewGroup(&signature1, &signature2)
chord, _ := tasks.NewChord(group, &signature3)
chordAsyncResult, err := server.SendChord(chord, 0) //The second parameter
specifies the number of concurrent sending tasks. 0 means unlimited.
if err != nil {
    // failed to send the chord
    // do something with the error
}

```

上面的例子并行执行task1和task2，聚合它们的结果并将它们传递给task3。因此最终会发生的是：

```
multiply(add(1, 1), add(5, 5))
```

更明确的表达：

```
(1 + 1) * (5 + 5) = 2 * 10 = 20
```

`SecodChord` 返回的 `ChordAsyncResult` 包含着异步结果的值。

所以你可以做一个阻塞调用，等待回调的结果：

```
results, err := chordAsyncResult.Get(time.Duration(time.Millisecond * 5))
if err != nil {
    // getting result of a chord failed
    // do something with the error
}
for _, result := range results {
    fmt.Println(result.Interface())
}
```

Chains

`chain` 就是一个接一个执行的任务集，每个成功的任务都会触发 `chain` 中的下一个任务。例如：

```
import (
    "github.com/RichardKnop/machinery/v1/tasks"
    "github.com/RichardKnop/machinery/v1"
)

signature1 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 1,
        },
        {
            Type: "int64",
            Value: 1,
        },
    },
}

signature2 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
```

```

        Type: "int64",
        Value: 5,
    },
    {
        Type: "int64",
        Value: 5,
    },
},
}

signature3 := tasks.Signature{
    Name: "multiply",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 4,
        },
    },
}

chain, _ := tasks.NewChain(&signature1, &signature2, &signature3)
chainAsyncResult, err := server.SendChain(chain)
if err != nil {
    // failed to send the chain
    // do something with the error
}

```

上面的例子执行task1，然后是task2，最后是task3。当一个任务成功完成时，结果被附加到 `chain` 中下一个任务的参数列表的末尾。因此最终会发生的是：

```
multiply(4, add(5, 5, add(1, 1)))
```

更准确的表达是：

```

4 * (5 + 5 + (1 + 1))    # task1: add(1, 1)        returns 2
= 4 * (5 + 5 + 2)        # task2: add(5, 5, 2)      returns 12
= 4 * (12)                # task3: multiply(4, 12)   returns 48
= 48

```

`SendChain` 返回的 `ChainAsyncResult` 包含着异步结果的值。

所以你可以做一个阻塞调用，等待回调的结果：

```

results, err := chainAsyncResult.Get(time.Duration(time.Millisecond * 5))
if err != nil {
    // getting result of a chain failed
    // do something with the error
}
for _, result := range results {
    fmt.Println(result.Interface())
}

```

定时任务和工作流 (Periodic Tasks & Workflows)

Machinery 现在支持调度周期性任务和工作流。看到下面的示例代码：

定时任务(Periodic Tasks)

```

import (
    "github.com/RichardKnop/machinery/v1/tasks"
)

signature := &tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 1,
        },
        {
            Type: "int64",
            Value: 1,
        },
    },
}

err := server.RegisterPeriodTask("0 6 * * ?", "periodic-task", signature)
if err != nil {
    // failed to register periodic task
}

```

定时任务组(Periodic Groups)

```

import (
    "github.com/RichardKnop/machinery/v1/tasks"
    "github.com/RichardKnop/machinery/v1"
)

```

```

signature1 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 1,
        },
        {
            Type: "int64",
            Value: 1,
        },
    },
}

signature2 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 5,
        },
        {
            Type: "int64",
            Value: 5,
        },
    },
}

group, _ := tasks.NewGroup(&signature1, &signature2)
err := server.RegisterPeriodGroup("0 6 * * ?", "periodic-group", group)
if err != nil {
    // failed to register periodic group
}

```

定时调用链(Periodic Chains)

```

import (
    "github.com/RichardKnop/machinery/v1/tasks"
    "github.com/RichardKnop/machinery/v1"
)

signature1 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",

```

```

        Value: 1,
    },
    {
        Type: "int64",
        Value: 1,
    },
},
}

signature2 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 5,
        },
        {
            Type: "int64",
            Value: 5,
        },
    },
}

signature3 := tasks.Signature{
    Name: "multiply",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 4,
        },
    },
}

chain, _ := tasks.NewChain(&signature1, &signature2, &signature3)
err := server.RegisterPeriodChain("0 6 * * ?", "periodic-chain", chain)
if err != nil {
    // failed to register periodic chain
}

```

Periodic Chord

```

import (
    "github.com/RichardKnop/machinery/v1/tasks"
    "github.com/RichardKnop/machinery/v1"
)

signature1 := tasks.Signature{

```



```

    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 1,
        },
        {
            Type: "int64",
            Value: 1,
        },
    },
}

signature2 := tasks.Signature{
    Name: "add",
    Args: []tasks.Arg{
        {
            Type: "int64",
            Value: 5,
        },
        {
            Type: "int64",
            Value: 5,
        },
    },
}

signature3 := tasks.Signature{
    Name: "multiply",
}

group := tasks.NewGroup(&signature1, &signature2)
chord, _ := tasks.NewChord(group, &signature3)
err := server.RegisterPeriodChord("0 6 * * ?", "periodic-chord", chord)
if err != nil {
    // failed to register periodic chord
}

```

开发(Development)

环境(Requirements)

- Go
- RabbitMQ (可选)
- Redis
- Memcached (可选)
- MongoDB (可选)

在OS X系统上, 您可以使用安装环境使用 [Homebrew](#)

```
brew install go
brew install rabbitmq
brew install redis
brew install memcached
brew install mongodb
```

也可以选择使用 `Docker` 容器:

```
docker run -d -p 5672:5672 rabbitmq
docker run -d -p 6379:6379 redis
docker run -d -p 11211:11211 memcached
docker run -d -p 27017:27017 mongo
docker run -d -p 6831:6831/udp -p 16686:16686 jaegertracing/all-in-one:latest
```

依赖(Dependencies)

自从Go 1.11以来, 推荐使用[modules](#)进行依赖管理。

这是Go的一个小弱点, 因为依赖管理还没有解决。以前Go官方推荐使用[dep工具](#), 但现在已经放弃了, 取而代之的是modules。

Testing

最简单的(和平台无关的)运行测试的方式是通过docker-compose:

```
make ci
```

运行docker-compose基本命令如下:

```
(docker-compose -f docker-compose.test.yml -p machinery_ci up --build -d) &&
(docker logs -f machinery_sut &) && (docker wait machinery_sut)
```

另一种方法是在您的机器上设置一个开发环境。

为了启用集成测试, 您将需要安装所有必需的服务(RabbitMQ, Redis, Memcache, MongoDB)和配置环境变量:

```
export AMQP_URL=amqp://guest:guest@localhost:5672/
export REDIS_URL=localhost:6379
export MEMCACHE_URL=localhost:11211
export MONGODB_URL=localhost:27017
```

要对SQS实例运行集成测试，您需要在SQS中创建一个“test_queue”，并配置以下环境变量：

```
export SQS_URL=https://YOUR_SQS_URL
export AWS_ACCESS_KEY_ID=YOUR_AWS_ACCESS_KEY_ID
export AWS_SECRET_ACCESS_KEY=YOUR_AWS_SECRET_ACCESS_KEY
export AWS_DEFAULT_REGION=YOUR_AWS_DEFAULT_REGION
```

然后运行：

```
make test
```

如果没有配置环境变量，那么make test将只运行单元测试。