46 | Master任务调度：服务发现与资源管理

2023-01-24 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

课程介绍 >

讲述：郑建勋

时长 05:35　大小 5.10M

你好，我是郑建勋。

在上一节课程中，我们实现了 Master 的选主，这一节课，我们继续深入 Master 的开发，实现一下 Master 的服务发现与资源的管理。

Master 服务发现

首先我们来实现一下 Master 对 Worker 的服务发现。

Master 需要监听 Worker 节点的信息，感知到 Worker 节点的注册与销毁。和服务的注册一样，我们的服务发现也使用 micro 提供的 registry 功能，代码如下所示。

m.WatchWorker 方法调用 registry.Watch 监听 Worker 节点的变化，watch.Next() 会堵塞等待节点的下一个事件，当 Master 收到节点变化事件时，将事件发送到 workerNodeChange 通

道。m.Campaign 方法接收到变化事件后，会用日志打印出变化的信息。


```
func (m *Master) Campaign() {
  ...
  workerNodeChange := m.WatchWorker()

  for {
    select {
    ...
    case resp := <-workerNodeChange:
      m.logger.Info("watch worker change", zap.Any("worker:", resp))
    }
  }
}

func (m *Master) WatchWorker() chan *registry.Result {
  watch, err := m.registry.Watch(registry.WatchService(worker.ServiceName))
  if err != nil {
    panic(err)
  }
  ch := make(chan *registry.Result)
  go func() {
    for {
      res, err := watch.Next()
      if err != nil {
        m.logger.Info("watch worker service failed", zap.Error(err))
        continue
      }
      ch <- res
    }
  }()
  return ch

}
```

Master 中的 etcd registry 对象是我们在初始化时注册到 go-micro 中的。

复制代码

```
// cmd/master/master.go
reg := etcd.NewRegistry(registry.Addrs(sconfig.RegistryAddress))
master.New(
  masterID,
  master.WithLogger(logger.Named("master")),
  master.WithGRPCAddress(GRPCListenAddress),
  master.WithregistryURL(sconfig.RegistryAddress),
```

```
  master.WithRegistry(reg),
  master.WithSeeds(seeds),
)
```


深入 go-micro registry 接口

go-micro 提供的 registry 接口提供了诸多 API，其结构如下所示。

复制代码

```
type Registry interface {
  Init(...Option) error
  Options() Options
  Register(*Service, ...RegisterOption) error
  Deregister(*Service, ...DeregisterOption) error
  GetService(string, ...GetOption) ([]*Service, error)
  ListServices(...ListOption) ([]*Service, error)
  Watch(...WatchOption) (Watcher, error)
  String() string
}
```

对于 Master 的服务发现，我们借助了 registry.Watch 方法。Watch 方法借助 client.Watch 实现了对特定 Key 的监听，并封装了 client.Watch 返回的结果。

复制代码

```
func (e *etcdRegistry) Watch(opts ...registry.WatchOption) (registry.Watcher, e
   return newEtcdWatcher(e, e.options.Timeout, opts...)
}

func newEtcdWatcher(r *etcdRegistry, timeout time.Duration, opts ...registry.Wa
   var wo registry.WatchOptions
   for _, o := range opts {
      o(&wo)
   }
   watchPath := prefix
   if len(wo.Service) > 0 {
      watchPath = servicePath(wo.Service) + "/"
   }
   return &etcdWatcher{
      stop:    stop,
      w:       r.client.Watch(ctx, watchPath, clientv3.WithPrefix(), clientv3.W
      client:  r.client,
      timeout: timeout,
   }, nil
}
```

registry.Watch 方法返回了 Watcher 接口，Watcher 接口中有 Next 方法用于完成事件的迭代。

复制代码

```
type Watcher interface {
    // Next 堵塞调用
    Next() (*Result, error)
    Stop()
}
```

go-micro 的 etcd 插件库实现的 Next 方法也比较简单，只要监听 client.Watch 返回的通道，并将事件信息封装后返回即可。

复制代码

```
func (ew *etcdWatcher) Next() (*registry.Result, error) {
    for wresp := range ew.w {
        if wresp.Err() != nil {
            return nil, wresp.Err()
        }
        if wresp.Canceled {
            return nil, errors.New("could not get next")
        }
        for _, ev := range wresp.Events {
            service := decode(ev.Kv.Value)
            var action string
            switch ev.Type {
            case clientv3.EventTypePut:
                if ev.IsCreate() {
                    action = "create"
                } else if ev.IsModify() {
                    action = "update"
                }
            case clientv3.EventTypeDelete:
                action = "delete"
                // get service from prevKv
                service = decode(ev.PrevKv.Value)
            }
            if service == nil {
                continue
            }
            return &registry.Result{
                Action:  action,
                Service: service,
            }, nil
        }
    }
```

```
    return nil, errors.New("could not get next")
}
```


另外，Worker 节点也利用了 registry 接口的 Register 方法实现了服务的注册。如下所示，Register 方法最终调用了 clientv3 的 Put 方法，将包含节点信息的键值对写入了 etcd 中。

复制代码

```
func (e *etcdRegistry) Register(s *registry.Service, opts ...registry.RegisterC
  // register each node individually
  for _, node := range s.Nodes {
    err := e.registerNode(s, node, opts...)
    if err != nil {
      gerr = err
    }
  }
  return gerr
}

func (e *etcdRegistry) registerNode(s *registry.Service, node *registry.Node, c
  service := &registry.Service{
    Name:      s.Name,
    Version:   s.Version,
    Metadata:  s.Metadata,
    Endpoints: s.Endpoints,
    Nodes:     []*registry.Node{node},
  }
  ...
  // create an entry for the node
  if lgr != nil {
    _, err = e.client.Put(ctx, nodePath(service.Name, node.Id), encode(service)
  } else {
    _, err = e.client.Put(ctx, nodePath(service.Name, node.Id), encode(service)
  }
  if err != nil {
    return err
  }
}
```

现在让我们来看一看服务发现的效果。首先，启动 Master 服务。

复制代码

```
» go run main.go master --id=2 --http=:8081  --grpc=:9091
```

接着启动 Worker 服务。

```
» go run main.go worker --id=2 --http=:8079  --grpc=:9089
```


Worker 启动后，在 Master 的日志中会看到变化的事件。其中，"Action":"create" 表明当前的事件为节点的注册。

```
{"level":"INFO","ts":"2022-12-12T16:55:42.798+0800","logger":"master","caller":
```

中止 Worker 节点后，我们还会看到 Master 的信息。其中，"Action":"delete" 表明当前的事件为节点的删除。

```
{"level":"INFO","ts":"2022-12-12T16:58:31.985+0800","logger":"master","caller":
```

维护 Worker 节点信息

完成服务发现之后，让我们更进一步，维护 Worker 节点的信息。在 updateWorkNodes 函数中，我们利用 registry.GetService 方法获取当前集群中全量的 Worker 节点，并将它最新的状态保存起来。

```
func (m *Master) Campaign() {
  ...
  workerNodeChange := m.WatchWorker()

  for {
    select {
    ...
    case resp := <-workerNodeChange:
      m.logger.Info("watch worker change", zap.Any("worker:", resp))
    }
  }
}

type Master struct {
  ...
  workNodes map[string]*registry.Node
}
```

```
func (m *Master) updateWorkNodes() {
  services, err := m.registry.GetService(worker.ServiceName)
  if err != nil {
    m.logger.Error("get service", zap.Error(err))
  }

  nodes := make(map[string]*registry.Node)
  if len(services) > 0 {
    for _, spec := range services[0].Nodes {
      nodes[spec.Id] = spec
    }
  }

  added, deleted, changed := workNodeDiff(m.workNodes, nodes)
  m.logger.Sugar().Info("worker joined: ", added, ", leaved: ", deleted, ", cha

  m.workNodes = nodes

}

```


我们还可以使用 workNodeDiff 函数比较集群中新旧节点的变化。

复制代码

```
func workNodeDiff(old map[string]*registry.Node, new map[string]*registry.Node)
  added := make([]string, 0)
  deleted := make([]string, 0)
  changed := make([]string, 0)
  for k, v := range new {
    if ov, ok := old[k]; ok {
      if !reflect.DeepEqual(v, ov) {
        changed = append(changed, k)
      }
    } else {
      added = append(added, k)
    }
  }
  for k := range old {
    if _, ok := new[k]; !ok {
      deleted = append(deleted, k)
    }
  }
  return added, deleted, changed
}
```

当节点发生变化时，可以打印出日志。

复制代码

```
{"level":"INFO","ts":"2022-12-12T16:55:42.810+0800","logger":"master","caller":
{"level":"INFO","ts":"2022-12-12T16:58:32.026+0800","logger":"master","caller":
```


Master 资源管理

下一步，让我们来看看对爬虫任务的管理。

爬虫任务也可以理解为一种资源。和 Worker 一样，Master 中可以有一些初始化的爬虫任务存储在配置文件中。初始化时，程序通过读取配置文件将爬虫任务注入到 Master 中。这节课我们先将任务放置到配置文件中，下节课我们还会构建 Master 的 API 来完成任务的增删查改。

复制代码

```
seeds := worker.ParseTaskConfig(logger, nil, nil, tcfg)
  master.New(
    masterID,
    master.WithLogger(logger.Named("master")),
    master.WithGRPCAddress(GRPCListenAddress),
    master.WithregistryURL(sconfig.RegistryAddress),
    master.WithRegistry(reg),
    master.WithSeeds(seeds),
  )
```

在初始化 Master 时，调用 m.AddSeed 函数完成资源的添加。m.AddSeed 会首先调用 etcdCli.Get 方法，查看当前任务是否已经写入到了 etcd 中。如果没有，则调用 m.AddResource 将任务存储到 etcd，存储在 etcd 中的任务的 Key 为 `/resources/xxxx`。

复制代码

```
func (m *Master) AddSeed() {
  rs := make([]*ResourceSpec, 0, len(m.Seeds))
  for _, seed := range m.Seeds {
    resp, err := m.etcdCli.Get(context.Background(), getResourcePath(seed.Name)
    if err != nil {
      m.logger.Error("etcd get faiiled", zap.Error(err))
      continue
    }
    if len(resp.Kvs) == 0 {
      r := &ResourceSpec{
        Name: seed.Name,
      }
      rs = append(rs, r)
    }
```

```
  }

  m.AddResource(rs)
}

const (
  RESOURCEPATH = "/resources"
)

func getResourcePath(name string) string {
  return fmt.Sprintf("%s/%s", RESOURCEPATH, name)
}
```


在添加资源的时候，我们可以设置资源的 ID、创建时间等。在这里我借助了第三方库 Snowflake，使用雪花算法来为资源生成了一个单调递增的分布式 ID。

复制代码

```
func (m *Master) AddResource(rs []*ResourceSpec) {
  for _, r := range rs {
    r.ID = m.IDGen.Generate().String()
    ns, err := m.Assign(r)
    if err != nil {
      m.logger.Error("assign failed", zap.Error(err))
      continue
    }
    r.AssignedNode = ns.Id + "|" + ns.Address
    r.CreationTime = time.Now().UnixNano()
    m.logger.Debug("add resource", zap.Any("specs", r))

    _, err = m.etcdCli.Put(context.Background(), getResourcePath(r.Name), encod
    if err != nil {
      m.logger.Error("put etcd failed", zap.Error(err))
      continue
    }
    m.resources[r.Name] = r
  }
}
```

Snowflake 利用雪花算法生成了一个 64 位的唯一 ID，其结构如下。

复制代码

```

+--------------------------------------------------------------------------+
| 1 Bit Unused | 41 Bit Timestamp |  10 Bit NodeID  |   12 Bit Sequence ID |
+--------------------------------------------------------------------------+
```

其中，41 位用于存储时间戳；10 位用于存储 NodeID，在这里就是我们的 Master ID；最后 12 位为序列号。如果我们的程序打算在同一个毫秒内生成多个 ID，那么每生成一个新的 ID，序列号会递增 1，这意味着每个节点每毫秒最多能够产生 4096 个不同的 ID，这已经能满足我们当前的场景了。雪花算法确保了我们生成的资源 ID 是全局唯一的。

添加资源时，还有一步很重要，那就是调用 m.Assign 计算出当前的资源应该被分配到哪一个节点上。在这里，我们先用随机的方式选择一个节点，后面还会再优化调度逻辑。

复制代码

```
func (m *Master) Assign(r *ResourceSpec) (*registry.Node, error) {
  for _, n := range m.workNodes {
    return n, nil
  }
  return nil, errors.New("no worker nodes")
}
```

设置好资源的 ID 信息、分配信息之后，调用 etcdCli.Put，将资源的 KV 信息存储到 etcd 中。其中，存储到 etcd 中的 Value 需要是 string 类型，所以我们书写了 JSON 的序列化与反序列化函数，用于存储信息的序列化和反序列化。

复制代码

```
func encode(s *ResourceSpec) string {
  b, _ := json.Marshal(s)
  return string(b)
}

func decode(ds []byte) (*ResourceSpec, error) {
  var s *ResourceSpec
  err := json.Unmarshal(ds, &s)
  return s, err
}
```

最后一步，当 Master 成为新的 Leader 后，我们还要全量地获取一次 etcd 中当前最新的资源信息，并把它保存到内存中，核心逻辑位于 loadResource 函数中。

复制代码

```

```

```go
func (m *Master) BecomeLeader() error {
  if err := m.loadResource(); err != nil {
    return fmt.Errorf("loadResource failed:%w", err)
  }
  atomic.StoreInt32(&m.ready, 1)
  return nil
}

func (m *Master) loadResource() error {
  resp, err := m.etcdCli.Get(context.Background(), RESOURCEPATH, clientv3.WithS
  if err != nil {
    return fmt.Errorf("etcd get failed")
  }

  resources := make(map[string]*ResourceSpec)
  for _, kv := range resp.Kvs {
    r, err := decode(kv.Value)
    if err == nil && r != nil {
      resources[r.Name] = r
    }
  }
  m.logger.Info("leader init load resource", zap.Int("lenth", len(m.resources))
  m.resources = resources
  return nil
}
```


验证 Master 资源分配结果

最后让我们实战验证一下 Master 的资源分配结果。

首先我们需要启动 Worker。要注意的是，如果先启动了 Master，初始的任务将会由于没有对应的 Worker 节点而添加失败。


```
» go run main.go worker --id=2 --http=:8079  --grpc=:9089
```

接着启动 Master 服务。


```
» go run main.go master --id=2 --http=:8081  --grpc=:9091
```

现在查看 etcd 的信息会发现，当前两个爬虫任务都已经设置到 etcd 中，并且 Master 为他们分配的 Worker 节点为"go.micro.server.worker-2|192.168.0.107:9089"，说明 Master 的资源分配成功了。


```
» docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl get --prefix /
/micro/registry/go.micro.server.master/go.micro.server.master-2
{"name":"go.micro.server.master","version":"latest","metadata":null,"endpoints"

/micro/registry/go.micro.server.worker/go.micro.server.worker-2
{"name":"go.micro.server.worker","version":"latest","metadata":null,"endpoints"

/resources/douban_book_list
{"ID":"1602250527540776960","Name":"douban_book_list","AssignedNode":"go.micro.

/resources/election/3f3584fc571ae9d0
master2-192.168.0.107:9091

/resources/xxx
{"ID":"1602250527570137088","Name":"xxx","AssignedNode":"go.micro.server.worker
```

总结

这节课。我们实现了 Master 的两个重要功能：服务发现与资源管理。

对于服务发现，我们借助了 micro registry 提供的接口，实现了节点的注册、发现和状态获取。micro 的 registry 接口是一个插件，这意味着我们可以轻松使用不同插件与不同的注册中心交互。在这里我们使用的仍然是 go-micro 的 etcd 插件，借助 etcd clientv3 的 API 实现了服务发现与注册的相关功能。

而对于资源管理，这节课我们为资源加上了必要的 ID 信息，我们使用了分布式的雪花算法来保证生成 ID 全局唯一。同时，我们用随机的方式为资源分配了其所属的 Worker 节点并验证了分配的效果。在下一节课程中，我们还会继续实现负载均衡的资源分配。

课后题

学完这节课，给你留两道思考题。

1. 我们什么时候需要全量拉取资源？什么时候需要使用事件监听机制？你认为监听机制是可靠的吗？

2. 我们前面提到的 Snowflake 雪花算法生成的分布式 ID，在什么场景下是不适用的？

欢迎你在留言区与我交流讨论，我们下节课见。

上一篇　45丨Master高可用：怎样借助etcd实现服务选主？

下一篇　47丨故障容错：如何在Worker崩溃时进行重新调度？

精选留言 (1)

写留言

Realm

2023-02-01 来自浙江

1 “当 Master 成为新的 Leader 后，我们还要全量地获取一次 etcd 中当前最新的资源信息，并把它保存到内存中”，
当添加单个task任务时，使用事件监听；
事件监听机制在服务异常时可能丢信息？
望勋哥指点.

2 雪花算法生成的id有可能会出现重复。
如一个节点时，把服务器时钟回拨的情况；
多个节点时候，假如服务器的标志位一样，同一毫秒不同的节点可能产生的id相同；