47 | 故障容错：如何在Worker崩溃时进行重新调度？

2023-01-26 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 08:32　大小 11.70M

你好，我是郑建勋。

上一节课，我们用随机的方式为资源分配了它所属的 Worker。这一节课，让我们更进一步优化资源的分配。

对资源进行分配不仅发生在正常的事件内，也可能发生在 Worker 节点崩溃等特殊时期。这时，我们需要将崩溃的 Worker 节点中的任务转移到其他节点。

Master 调度的时机

具体来说，分配资源的时机可能有下面三种情况。

- 当 Master 成为 Leader 时。
- 当客户端调用 Master API 进行资源的增删查改时。

- 当 Master 监听到 Worker 节点发生变化时。

其中，第二点“调用 Master API 进行资源的增删查改”我们会在这节课的最后完成，下面让我们实战一下剩下两点是如何实现资源的调度的。

Master 成为 Leader 时的资源调度

在日常实践中，Leader 的频繁切换并不常见。不管是 Master 在初始化时选举成为了 Leader，还是在中途由于其他 Master 异常退出导致 Leader 发生了切换，我们都要全量地更新一下当前 Worker 的节点状态以及资源的状态。

在 Master 成为 Leader 节点时，我们首先要利用 m.updateWorkNodes 方法全量加载当前的 Worker 节点，同时利用 m.loadResource 方法全量加载当前的爬虫资源。

复制代码

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

  m.reAssign()

  atomic.StoreInt32(&m.ready, 1)
  return nil
}
```

接下来，调用 reAssign 方法完成一次资源的分配。m.reAssign 会遍历资源，当发现有资源还没有分配节点时，将再次尝试将资源分配到 Worker 中。如果发现资源都已经分配给了对应的 Worker，它就会查看当前节点是否存活。如果当前节点已经不存在了，就将该资源分配给其他的节点。

复制代码

```
func (m *Master) reAssign() {
  rs := make([]*ResourceSpec, 0, len(m.resources))

  for _, r := range m.resources {
    if r.AssignedNode == "" {
      rs = append(rs, r)
      continue
    }
```

```
    id, err := getNodeID(r.AssignedNode)

    if err != nil {
      m.logger.Error("get nodeid failed", zap.Error(err))
    }

    if _, ok := m.workNodes[id]; !ok {
      rs = append(rs, r)
    }
  }
  m.AddResources(rs)
}

func (m *Master) AddResources(rs []*ResourceSpec) {
  for _, r := range rs {
    m.addResources(r)
  }
}
```


之前我们已经维护了资源的 ID、事件以及分配的 Worker 节点等信息。 在这里让我们更进一步，当资源分配到节点上时，更新节点的状态。

为此我抽象出了一个新的结构 NodeSpec，我们用它来描述 Worker 节点的状态。NodeSpec 封装了 Worker 注册到 etcd 中的节点信息 registry.Node。同时，我们额外增加了一个 Payload 字段，用于标识当前 Worker 节点的负载。当资源分配到对应的 Worker 节点上时，则更新 Worker 节点的状态，让 Payload 负载加 1。


```
type NodeSpec struct {
  Node    *registry.Node
  Payload int
}

func (m *Master) addResources(r *ResourceSpec) (*NodeSpec, error) {
  ns, err := m.Assign(r)
  ...
  r.AssignedNode = ns.Node.Id + "|" + ns.Node.Address
  _, err = m.etcdCli.Put(context.Background(), getResourcePath(r.Name), encode(
  m.resources[r.Name] = r
  ns.Payload++
  return ns, nil
}
```

Worker 节点发生变化时的资源更新

当我们发现 Worker 节点发生变化时，也需要全量完成一次更新。这是为了及时发现当前已经崩溃的 Worker 节点，并将这些崩溃的 Worker 节点下的任务转移给其他 Worker 节点运行。

如下所示，当 Master 监听 workerNodeChange 通道，发现 Worker 节点产生了变化之后，就会像成为 Leader 一样，更新当前节点与资源的状态，然后调用 m.reAssign 方法重新调度资源。

```
func (m *Master) Campaign() {
  ...
  for {
    select {
    case resp := <-workerNodeChange:
      m.logger.Info("watch worker change", zap.Any("worker:", resp))
      m.updateWorkNodes()
      if err := m.loadResource(); err != nil {
        m.logger.Error("loadResource failed:%w", zap.Error(err))
      }
      m.reAssign()
    }
  }
}
```

负载均衡的资源分配算法

接下来，我们再重新看看资源的分配。上节课我们都是将资源随机分配到某一个 Worker 上的，但是在实践中很可能会有多个 Worker，而为了对资源进行合理的分配，需要实现负载均衡，让 Worker 节点分摊工作量。

负载均衡分配资源的算法有很多，例如轮询法、加权轮询法、随机法、最小负载法等等，而根据实际场景，还可能需要有特殊的调度逻辑。这里我们实现一种简单的调度算法：最小负载法。在我们当前的场景中，最小负载法能够比较均衡地将爬虫任务分摊到 Worker 节点中。它每一次都将资源分配给具有最低负载的 Worker 节点，这依赖于我们维护的节点的状态。

如下所示，第一步我们遍历所有的 Worker 节点，找到合适的 Worker 节点。其实这一步可以完成一些简单的筛选，过滤掉一些不匹配的 Worker。举一个例子，有些任务比较特殊，在计算时需要使用到 GPU，那么我们就只能将它调度到有 GPU 的 Worker 节点中。这里我们没有

实现更复杂的筛选逻辑，把当前全量的 Worker 节点都作为候选节点，放入到了 candidates 队列中。

复制代码

```
func (m *Master) Assign(r *ResourceSpec) (*NodeSpec, error) {
  candidates := make([]*NodeSpec, 0, len(m.workNodes))

  for _, node := range m.workNodes {
    candidates = append(candidates, node)
  }

  //  找到最低的负载
  sort.Slice(candidates, func(i, j int) bool {
    return candidates[i].Payload < candidates[j].Payload
  })

  if len(candidates) > 0 {
    return candidates[0], nil
  }

  return nil, errors.New("no worker nodes")
}
```

第二步，根据负载对 Worker 队列进行排序。这里我使用了标准库 sort 中的 Slice 函数。Slice 函数的第一个参数为 candidates 队列；第二个参数是一个函数，它可以指定排序的优先级条件，这里我们指定负载越小的 Worker 节点优先级越高。所以在排序之后，负载最小的 Worker 节点会排在前面。

第三步，取排序之后的第一个节点作为目标 Worker 节点。

现在，让我们来验证一下资源分配是否成功实现了负载均衡。首先，启动两个 Worker 节点。

复制代码

```
» go run main.go worker --id=1 --http=:8080  --grpc=:9090
» go run main.go worker --id=2 --http=:8079  --grpc=:9089
```

接着，我们在配置文件中加入 5 个任务，并启动一个 Master 节点。

复制代码

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

Master 在初始化时就会完成任务的分配，我们可以在 etcd 中查看资源的分配情况，结果如下所示。


```
» docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl get --prefix /
/resources/douban_book_list
{"ID":"16040658100100833328","Name":"douban_book_list","AssignedNode":"go.micro.
/resources/task-test-1
{"ID":"16040666770188574 72","Name":"task-test-1","AssignedNode":"go.micro.serve
/resources/task-test-2
{"ID":"16040666997561794 56","Name":"task-test-2","AssignedNode":"go.micro.serve
/resources/task-test-3
{"ID":"16040667162062397 44","Name":"task-test-3","AssignedNode":"go.micro.serve
/resources/xxx
{"ID":"16040658100268605 44","Name":"xxx","AssignedNode":"go.micro.server.worker
```

观察资源分配的 Worker 节点，会发现当前有 3 个任务分配到了 go.micro.server.worker-2，有 2 个节点分配到了 go.micro.server.worker-1，说明我们现在的负载均衡策略符合预期。

接下来，让我们删除 worker-1 节点，验证一下 worker-1 中的资源是否会自动迁移到 worker-2 中。输入 `Ctrl+C` 退出 worker-1 节点，然后回到 etcd 中查看资源分配的情况，发现所有的资源都已经迁移到了 worker-2 中。这说明当 Worker 节点崩溃后，重新调度任务的策略是符合预期的。

```
» docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl get --prefix /
/resources/douban_book_list
{"ID":"16040658100100833328","Name":"douban_book_list","AssignedNode":"go.micro.
/resources/task-test-1
{"ID":"16040692652357754 88","Name":"task-test-1","AssignedNode":"go.micro.serve
/resources/task-test-2
{"ID":"16040666997561794 56","Name":"task-test-2","AssignedNode":"go.micro.serve
/resources/task-test-3
{"ID":"16040692652525527 04","Name":"task-test-3","AssignedNode":"go.micro.serve
/resources/xxx
{"ID":"16040692652693299 20","Name":"xxx","AssignedNode":"go.micro.server.worker
```

最后我们来看看 Master Leader 切换时的情况。我们新建一个 Master，它的 ID 为 3。输入 Ctrl+C 中断之前的 Master 节点。

```
» go run main.go master --id=3 --http=:8082  --grpc=:9092
```

这时再次查看 etcd 中的资源分配情况，会发现资源的信息没有任何变化。这是符合预期的，因为当前的资源在之前都已经分配给了 Worker，不需要再重新分配了。

实战 Master 资源处理 API

接下来，让我们为 Master 实现对外暴露的 API，方便外部客户端进行访问，实现资源的增删查改。按照惯例，我们仍然会为 API 实现 GRPC 协议和 HTTP 协议。

首先，我们要在 crawler.proto 中书写 Master 服务的 Protocol Buffer 协议。

我们先为 Master 加入两个 RPC 接口。其中，AddResource 接口用于增加资源，参数为结构体 ResourceSpec，表示添加资源的信息。其中最重要的参数是 name，它标识了具体启动哪一个爬虫任务。返回值为结构体 NodeSpec，NodeSpec 描述了资源分配到 Worker 节点的信息。 DeleteResource 接口用于删除资源，请求参数为资源信息，而不需要有任务返回值信息，因此这里定义为空结构体 Empty。为了引用 Empty，在这里我们导入了 google/protobuf/empty.proto 库。

```
syntax = "proto3";
option go_package = "proto/crawler";
import "google/api/annotations.proto";
import "google/protobuf/empty.proto";

service CrawlerMaster {
  rpc AddResource(ResourceSpec) returns (NodeSpec) {
      option (google.api.http) = {
            post: "/crawler/resource"
            body: "*"
        };
  }
  rpc DeleteResource(ResourceSpec) returns (google.protobuf.Empty){
      option (google.api.http) = {
            delete: "/crawler/resource"
            body: "*"
        };
  }
}

message ResourceSpec {
```

```
      string id = 1;
    string name = 2;
    string assigned_node = 3;
    int64 creation_time = 4;
}

message NodeSpec {
    string id = 1;
    string Address = 2;
}
```


代码中的 option 是 GRPC-gateway 使用的信息，用于生成与 GRPC 方法对应的 HTTP 代理请求。在 option 中，AddResource 对应的 HTTP 方法为 POST，URL 为 /crawler/resource。

DeleteResource 对应的 URL 仍然为 /crawler/resource，不过 HTTP 方法为 DELETE。body: "*" 表示 GRPC-gateway 将接受 HTTP Body 中的信息，并会将其解析为对应的请求。

下一步，执行 protoc 命令，生成对应的 micro GRPC 文件和 HTTP 代理文件。

复制代码

```
» protoc -I $GOPATH/src  -I .  --micro_out=. --go_out=.  --go-grpc_out=.  --grp
```

这里的 allow_delete_body 表示对于 HTTP DELETE 方法，HTTP 代理服务也可以解析 Body 中的信息，并将其转换为请求参数。

接下来，我们需要为 Master 书写对应的方法，让 Master 实现 micro 生成的 CrawlerMasterHandler 接口。

复制代码

```
type CrawlerMasterHandler interface {
  AddResource(context.Context, *ResourceSpec, *NodeSpec) error
  DeleteResource(context.Context, *ResourceSpec, *empty.Empty) error
}
```

实现 DeleteResource 和 AddResource 这两个方法比较简单。其中，DeleteResource 负责判断当前的任务名是否存在，如果存在则调用 etcd delete 方法删除资源 Key，并更新节点的负载。而 AddResource 方法可以调用我们之前就写好的 m.addResources 方法来添加资源，返回资源分配的节点信息。

复制代码

```
func (m *Master) DeleteResource(ctx context.Context, spec *proto.ResourceSpec,
  r, ok := m.resources[spec.Name]

  if !ok {
    return errors.New("no such task")
  }

  if _, err := m.etcdCli.Delete(context.Background(), getResourcePath(spec.Name
    return err
  }

  if r.AssignedNode != "" {
    nodeID, err := getNodeID(r.AssignedNode)
    if err != nil {
      return err
    }

    if ns, ok := m.workNodes[nodeID]; ok {
      ns.Payload -= 1
    }
  }
  return nil
}

func (m *Master) AddResource(ctx context.Context, req *proto.ResourceSpec, resp
  nodeSpec, err := m.addResources(&ResourceSpec{Name: req.Name})
  if nodeSpec != nil {
    resp.Id = nodeSpec.Node.Id
    resp.Address = nodeSpec.Node.Address
  }
  return err
}
```

最后，我们还要调用 micro 生成的 crawler.RegisterCrawlerMasterHandler 函数，将 Master 注册为 GRPC 服务。这之后就可以正常处理客户端的访问了。

复制代码

```
func RunGRPCServer(MasterService *master.Master, logger *zap.Logger, reg regist
  ...
```

```
  if err := crawler.RegisterCrawlerMasterHandler(service.Server(), MasterServic
    logger.Fatal("register handler failed", zap.Error(err))
  }

  if err := service.Run(); err != nil {
    logger.Fatal("grpc server stop", zap.Error(err))
  }
}
```


让我们来验证一下 Master 是否正在正常对外提供服务。

首先，启动 Master。接着，通过 HTTP 访问 Master 提供的添加资源的接口。如下所示，添加资源“task-test-4”。

复制代码

```
» go run main.go master --id=2 --http=:8081  --grpc=:9091
» curl -H "content-type: application/json" -d '{"id":"zjx","name": "task-test-4
{"id":"go.micro.server.worker-2", "Address":"192.168.0.107:9089"}
```

通过返回值可以看到，当前资源分配到了 worker-2，worker-2 的 IP 地址为"192.168.0.107:9089"。

查看 etcd 中的资源，发现资源已经成功写入了 etcd，而且其中分配的 Worker 节点信息与 HTTP 接口返回的信息相同。

复制代码

```
» docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl get /resources
/resources/task-test-4
{"ID":"16041091256947875 84","Name":"task-test-4","AssignedNode":"go.micro.serve
```

接着，我们尝试调用 Master 服务的删除资源接口，删除我们刚刚生成添加的资源。

复制代码

```
» curl -X DELETE  -H "content-type: application/json" -d '{"name": "task-test-4
```

再次查看 etcd 中的资源"task-test-4"，发现资源已经被删除了。Master API 提供的添加和删除功能验证成功。

总结

这节课，我们对资源分配的时机和资源分配的算法进行了优化。我们模拟了 Master 和 Worker 节点崩溃的情况，并用简单的方式实现了节点的重新分配，让当前系统在分布式下具备了故障容错的特性。

在 Master 调度的时机上，当 Master 成为 Leader，Worker 节点崩溃，或者外部调用资源增删查改接口时，Leader 需要对资源进行重新调度。对于调度算法，为了实现负载的均衡，我选择了当前负载最低的 Worker 的节点作为了优先级最高的节点。在实践中，我们需要结合对应的业务场景设计出最合适的调度逻辑。

最后，我们还为 Master 实现了 GRPC 与 HTTP API，让 Masrer 具备了添加资源和删除资源的能力。

课后题

这节课，我们用简单的方式实现了负载均衡的调度算法，但在生产实践中，调度器可能会更复杂。例如有些资源的负载会更大，有些资源只能在某一个 Worker 上执行，有些资源需要具有亲和性等等。你认为应该如何处理这些情况呢？

如何让我们的程序轻松地切换到另一个调度算法上？

（提示：可以参考 Kubernetes 对资源的复杂调度。）

欢迎你在留言区与我交流讨论，我们下节课见。

下一篇 48 | 完善核心能力：Master请求转发与Worker资源管理

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。