34 | 服务注册与监听：Worker节点与etcd交互

2022-12-27 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

讲述：郑建勋

时长 08:48　大小 8.04M

你好，我是郑建勋。

这节课，让我们将 Worker 节点变为一个支持 GRPC 与 HTTP 协议访问的服务，让它最终可以被 Master 服务和外部服务直接访问。在 Worker 节点上线之后，我们还要将 Worker 节点注册到服务注册中心。

GRPC 与 Protocol buffers

一般要在微服务中进行远程通信，会选择 GRPC 或 RESTful 风格的协议。我们之前就提到过，GRPC 的好处包括：

- 使用了 HTTP/2 传输协议来传输序列化后的二进制信息，让传输速度更快；
- 可以为不同的语言生成对应的 Client 库，让外部访问非常便利；

- 使用 Protocol Buffers 定义 API 的行为，提供了强大的序列化与反序列化能力；
- 支持双向的流式传输（Bi-directional streaming）。

GRPC 默认使用 Protocol buffers 协议来定义接口，它有如下特点：

- 它提供了与语言、框架无关的序列化与反序列化的能力；
- 它序列化生成的字节数组比 JSON 更小，同时序列化与反序列化的速度也比 JSON 更快；
- 有良好的向后和向前兼容性。

Protocol buffers 将接口语言定义在以 .proto 为后缀的文件中，之后 proto 编译器结合特定语言的运行库生成特定的 SDK。这个 SDK 文件有助于我们在 Client 端访问，也有助于我们生成 GRPC Server。

现在让我们来实战一下 Protocol buffers 协议。

第一步，书写一个简单的文件 hello.proto：

复制代码

```
syntax = "proto3";
option go_package = "proto/greeter";

service Greeter {
  rpc Hello(Request) returns (Response) {}
}

message Request {
  string name = 1;
}

message Response {
  string greeting = 2;
}
```

proto 协议很容易理解：

- `syntax = "proto3";` 标识我们协议的版本，每个版本的语言可能会有所不同，目前最新的使用最多的版本是 proto3，它的语法你可以查看官方文档；

- option go_package 定义生成的 Go 的 package 名；
- service Greeter 定义了一个服务 Greeter，它的远程方法为 Hello，Hello 参数为结构体 Request，返回值为结构体 Response。

要根据这个 proto 文件生成 Go 对应的协议文件，我们需要做一下前置的工作：下载 proto 的编译器 protoc，安装 protoc 指定版本的方式可以查看官方的安装文档。

同时，我们还需要安装 protoc 的 Go 语言的插件。

```
go install google.golang.org/protobuf/cmd/protoc-gen-go@latest
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
```

第二步，输入命令 protoc 进行编译，编译完成后生成了 hello.pb.go 与 hello_grpc.pb.go 两个协议文件。

```
protoc -I $GOPATH/src  -I . --go_out=.  --go-grpc_out=.  hello.proto
```

在 hello_grpc.pb.go 中，我们会看到生成的文件为我们自动生成了 GreeterServer 接口，接口中有 Hello 方法。

```
type GreeterServer interface {
  Hello(context.Context, *Request) (*Response, error)
}
```

第三步，在我们的 main 函数中生成结构体 Greeter，实现 GreeterServer 接口，然后调用生成协议文件中的 pb.RegisterGreeterServer，将 Greeter 注册到 GRPC server 中。代码如下所示。要注意的是，xxx/proto/greeter 需要替换为你自己的项目中协议文件的位置。

至此，我们就生成了一个 GRPC 服务了，该服务提供了 Hello 方法。

```go
package main

import (
  pb "xxx/proto/greeter"
  "log"
  "net"

  "google.golang.org/grpc"
)
type Greeter struct {
  pb.UnimplementedGreeterServer
}

func (g *Greeter) Hello(ctx context.Context, req *pb.Request) (rsp *pb.Response
  rsp.Greeting = "Hello " + req.Name
  return rsp, nil
}

func main() {
  println("gRPC server tutorial in Go")

  listener, err := net.Listen("tcp", ":9000")
  if err != nil {
    panic(err)
  }

  s := grpc.NewServer()
  pb.RegisterGreeterServer(s, &Greeter{})
  if err := s.Serve(listener); err != nil {
    log.Fatalf("failed to serve: %v", err)
  }
}
```


go-micro 与 GRPC-gateway

刚才我们看到了原生的生成 GRPC 服务器的方法。不过在我们的项目中，我打算用另一个目前微服务领域比较流行的框架 go-micro 来实现我们的 GRPC 服务器。

相比原生的方式，go-micro 拥有更丰富的生态和功能，更方便的工具和 API。例如，在 go-micro 中，服务注册可以方便地切换到 etcd、ZooKeeper、Gossip、NATS 等注册中心，方便我们实现服务注册功能。Server 端也同时支持 GRPC、HTTP 等多种协议。

要在 go-micro 中实现 GRPC 服务器，我们同样需要利用前面的 proto 文件生成的协议文件。不过，go-micro 在此基础上进行了扩展，我们需要下载 protoc-gen-micro 插件来生成 micro 适用的协议文件。这个插件的版本需要和我们使用的 go-micro 版本相同。目前，最新的 go-

micro 版本为 v4，我们这个项目就用最新的版本来开发。所以，我们需要先下载 protoc-gen-micro v4 版本：


```
go install github.com/asim/go-micro/cmd/protoc-gen-micro/v4@latest
```

接着输入如下命名，生成一个新的文件 hello.pb.micro.go：

```
protoc -I $GOPATH/src  -I .  --micro_out=. --go_out=.  --go-grpc_out=.  hello.p
```

在 hello.pb.micro.go 中，micro 生成了一个接口 GreeterHandler，所以我们需要在代码中实现这个新的接口：

```
type GreeterHandler interface {
  Hello(context.Context, *Request, *Response) error
}
```

用 go-micro 生成 GRPC 服务器的代码如下，Greeter 结构体实现了 GreeterHandler 接口。代码调用 pb.RegisterGreeterHandler 将 Greeter 注册到 micro 生成的 GRPC server 中。另外如果要查看使用 go-micro 的样例，可以查看 example 库。

```
package main

import (
  pb "xxx/proto/greeter"
  "log"
  "context"
   "go-micro.dev/v4"
  "google.golang.org/grpc"
)

type Greeter struct{}

func (g *Greeter) Hello(ctx context.Context, req *pb.Request, rsp *pb.Response)
  rsp.Greeting = "Hello " + req.Name
  return nil
```

```
}

func main() {
  service := micro.NewService(
    micro.Name("helloworld"),
  )

  service.Init()

  pb.RegisterGreeterHandler(service.Server(), new(Greeter))

  if err := service.Run(); err != nil {
    log.Fatal(err)
  }
}
```


但到这里我们还不满足。GRPC 在调试的时候比 HTTP 协议要繁琐，而且有些外部服务可能不支持使用 GRPC 协议，为了解决这些问题，我们可以让服务同时具备 GRPC 与 HTTP 的能力。

要实现这一目的，我们需要借助一个第三方库：grpc-gateway。grpc-gateway 的功能就是生成一个 HTTP 的代理服务，然后这个 HTTP 代理服务会将 HTTP 请求转换为 GRPC 的协议，并转发到 GRPC 服务器中。从而实现了服务同时暴露 HTTP 接口与 GRPC 接口的目的。

要实现 grpc-gateway 的能力，我们需要对 proto 文件进行改造：

复制代码

```
syntax = "proto3";
option go_package = "proto/greeter";
import "google/api/annotations.proto";

service Greeter {
  rpc Hello(Request) returns (Response) {
      option (google.api.http) = {
            post: "/greeter/hello"
            body: "*"
        };
  }
}

message Request {
  string name = 1;
}

message Response {
```

```
19    string greeting = 2;
20 }
```


这里我们引入了一个依赖 google/api/annotations.proto，并且加入了自定义的 option 选项，grpc-gateway 的插件会识别到这个自定义选项，并为我们生成 HTTP 代理服务。

要生成指定的协议文件，我们需要先安装 grpc-gateway 的插件：

复制代码

```
go install github.com/grpc-ecosystem/grpc-gateway/v2/protoc-gen-grpc-gateway@la
go install github.com/grpc-ecosystem/grpc-gateway/v2/protoc-gen-openapiv2@lates
```

同时，提前下载依赖文件：google/api/annotations.proto。在这里，我手动下载了依赖文件并放入到了 GOPATH 中：

复制代码

```
git clone git@github.com:googleapis/googleapis.git
mv googleapis/google  $(go env GOPATH)/src/google
```

最后，利用下面的指令将 proto 文件生成协议文件。要注意的是，这里我们同时加入了 go-micro 的插件和 grpc-gateway 的插件，两个插件之间可能存在命名冲突。所以我指定了 grpc-gateway 的选项 register_func_suffix 为 Gw，它能够让生成的函数名包含该 Gw 前缀，这就解决了命名冲突问题。

复制代码

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

这样我们就生成了 4 个文件，分别是 hello.pb.go、hello.pb.gw.go、hello.pb.micro.go 和 hello_grpc.pb.go。 其中，hello.pb.gw.go 就是 grpc-gateway 插件生成的文件。

接下来我们借助 go-micro 与 grpc-gateway 为项目生成具备 GRPC 与 HTTP 能力的服务器，代码如下所示。


```go
package main

import (
  "context"
  "fmt"
  pb "xxx/proto/greeter"
  gs "github.com/go-micro/plugins/v4/server/grpc"
  "github.com/grpc-ecosystem/grpc-gateway/v2/runtime"
  "go-micro.dev/v4"
  "go-micro.dev/v4/registry"
  "go-micro.dev/v4/server"
  "google.golang.org/grpc"
  "log"
  "net/http"
)

type Greeter struct{}

func (g *Greeter) Hello(ctx context.Context, req *pb.Request, rsp *pb.Response)
  rsp.Greeting = "Hello " + req.Name
  return nil
}

func main() {
  // http proxy
  go HandleHTTP()

  // grpc server
  service := micro.NewService(
    micro.Server(gs.NewServer()),
    micro.Address(":9090"),
    micro.Name("go.micro.server.worker"),
  )

  service.Init()

  pb.RegisterGreeterHandler(service.Server(), new(Greeter))

  if err := service.Run(); err != nil {
    log.Fatal(err)
  }
}

func HandleHTTP() {
  ctx := context.Background()
  ctx, cancel := context.WithCancel(ctx)
  defer cancel()

  mux := runtime.NewServeMux()
  opts := []grpc.DialOption{grpc.WithInsecure()}

  err := pb.RegisterGreeterGwFromEndpoint(ctx, mux, "localhost:9090", opts)
```

```
53    if err != nil {
54      fmt.Println(err)
55    }
56
57    http.ListenAndServe(":8080", mux)
58  }
```


其中，HandleHTTP 函数生成 HTTP 服务器，监听 8080 端口。同时，我们利用了 grpc-gateway 生成的 RegisterGreeterGwFromEndpoint 方法，指定了要转发到哪一个 GRPC 服务器。当访问该 HTTP 接口后，该代理服务器会将请求转发到 GRPC 服务器。

现在让我们来验证一下功能，我们使用 HTTP 协议去访问服务：

复制代码

```
curl -H "content-type: application/json" -d '{"name": "john"}' http://localhost
```

返回结果如下：

复制代码

```
{
    "greeting": "Hello "
}
```

这就表明我们已经成功地使用 HTTP 请求访问到了 GRPC 服务器。

注册中心与 etcd

刚才，我们将 Worker 变成了 GRPC 服务器，也看到了 go-micro 的使用方式，接下来让我们看看如何用 go-micro 完成服务的注册。

在 go-micro 中使用 micro.NewService 生成一个 service。其中，service 可以用 option 的模式注入参数。而 micro.NewService 有许多默认的 option，默认情况下生成的服务器并不是 GRPC 类型的。为了生成 GRPC 服务器，我们需要导入 go-micro 的 GRPC 插件库，生成一个 GRPC server 注入到 micro.NewService 中。同时，micro.Address 指定了服务器监听的地址，而 micro.Name 表示服务器的名字。

```
import (
  "go-micro.dev/v4"
  "github.com/go-micro/plugins/v4/server/grpc"
)

func main() {
  ...
  // grpc server
  service := micro.NewService(
    micro.Server(gs.NewServer()),
    micro.Address(":9090"),
    micro.Name("go.micro.server.worker"),
  )
}
```


在 micro.NewService 中还可以注入 register 模块，用于指定使用哪一个注册中心。我们的项目中将使用 etcd 作为注册中心。为了在 go-micro v4 中使用 etcd 作为注册中心，我们需要导入 etcd 插件库，如下所示。这里的 etcd 注册模块仍然使用了 option 模式，registry.Addrs 指定了当前 etcd 的地址。


```
import (
  etcdReg "github.com/go-micro/plugins/v4/registry/etcd"
)
func main() {
  ...
  reg := etcdReg.NewRegistry(
      registry.Addrs(":2379"),
    )

  service := micro.NewService(
      micro.Server(gs.NewServer()),
      micro.Address(":9090"),
      micro.Registry(reg),
      micro.Name("go.micro.server"),
    )
}
```

接下来，让我们首先启动 etcd 服务器。启动服务器的方式有很多种，你可以参考官方文档。这里我利用 Docker 来启动一个 etcd 的服务器（关于 Docker，我们在之后的章节会详细介绍）：

```
rm -rf /tmp/etcd-data.tmp && mkdir -p /tmp/etcd-data.tmp && \\
  docker rmi gcr.io/etcd-development/etcd:v3.5.6 || true && \\
  docker run \\
  -p 2379:2379 \\
  -p 2380:2380 \\
  --mount type=bind,source=/tmp/etcd-data.tmp,destination=/etcd-data \\
  --name etcd-gcr-v3.5.6 \\
  gcr.io/etcd-development/etcd:v3.5.6 \\
  /usr/local/bin/etcd \\
  --name s1 \\
  --data-dir /etcd-data \\
  --listen-client-urls <http://0.0.0.0:2379> \\
  --advertise-client-urls <http://0.0.0.0:2379> \\
  --listen-peer-urls <http://0.0.0.0:2380> \\
  --initial-advertise-peer-urls <http://0.0.0.0:2380> \\
  --initial-cluster s1=http://0.0.0.0:2380 \\
  --initial-cluster-token tkn \\
  --initial-cluster-state new \\
  --log-level info \\
  --logger zap \\
  --log-outputs stderr
```


要验证用 Dokcer 启动 etcd 服务器是否成功，功能是否正常，我们可以使用下面几条命令。这些命令会打印 etcd 的版本，并用一个简单的 Key-Value 值验证出 put 与 get 功能是正常的。


```
docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcd --version"
docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl version"
docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl endpoint health"
docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl put foo bar"
docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl get foo"
```

接下来，让我们启动 go-micro 构建的的 GRPC 服务器，服务的信息会注册到 etcd 中，并且会定时发送自己的健康状况用于保活。

下面让我们验证一下：


```
» docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl get --prefix /
/micro/registry/go.micro.server/go.micro.server-707c1d61-2c20-42b4-95a0-6d3e847
{"name":"go.micro.server","version":"latest","metadata":null,"endpoints":[{"nam
```

这里，命令 get --prefix / 表示获取前缀为 / 的 Key。 我们会发现，go-micro 注册到 etcd 中的 Key 为 /micro/registry/c/go.micro.server-707c1d61-2c20-42b4-95a0-6d3e8473727e，其中 go.micro.server 是服务的名字，最后的一串 ID 是随机字符。

我们可以通过在生成 server 时指定特殊的 ID 来替换掉随机的 ID，如下所示：

复制代码

```
func main(){
  ...
  service := micro.NewService(
    micro.Server(gs.NewServer(
      server.Id("1"),
    )),
    micro.Address(":9090"),
    micro.Registry(reg),
    micro.Name("go.micro.server.worker"),
  )
}
```

这时会发现注册到 etcd 服务中的 Key 已经发生了变化：

复制代码

```
» docker exec etcd-gcr-v3.5.6 /bin/sh -c "/usr/local/bin/etcdctl get --prefix /
/micro/registry/go.micro.server.worker/go.micro.server.worker-1
{"name":"go.micro.server.worker","version":"latest","metadata":null,"endpoints"
```

上述完整的代码位于v0.3.0中。

最后，我们也可以用一个 GRPC 的客户端去访问我们的服务器：

复制代码

```
import (
  grpccli "github.com/go-micro/plugins/v4/client/grpc"
  "go-micro.dev/v4"
  "go-micro.dev/v4/registry"
  pb "xxx/proto/greeter"
)

func main() {
  reg := etcdReg.NewRegistry(
    registry.Addrs(":2379"),
```

```
  )
  // create a new service
  service := micro.NewService(
    micro.Registry(reg),
    micro.Client(grpccli.NewClient()),
  )

  // parse command line flags
  service.Init()

  // Use the generated client stub
  cl := pb.NewGreeterService("go.micro.server.worker", service.Client())

  // Make request
  rsp, err := cl.Hello(context.Background(), &pb.Request{
    Name: "John",
  })
  if err != nil {
    fmt.Println(err)
    return
  }

  fmt.Println(rsp.Greeting)
}
```


这里 pb.NewGreeterService 的第一个参数代表服务器的注册名。如果运行后能够正常地返回结果，代表 GRPC 客户端访问 GRPC 服务器成功了。GRPC 返回的结果如下所示：


```
» go run main.go
Hello John
```

总结

好了，总结一下。这节课，我们为 Worker 服务构建了 GRPC 服务器和 HTTP 服务器。其中，HTTP 服务器是用 grpc-gateway 生成的一个代理，它最终也会访问 GRPC 服务器。构建 GRPC 服务器需要安装一些必要的依赖，还要书写定义接口行为的 proto 文件。

在这节课的例子中，我们使用了 go-micro 微服务框架实现了 GRPC 服务器，它为微服务提供了比较丰富的能力，然后我们使用 go-micro 的插件将服务注册到了 etcd 注册中心当中。客户端可以通过服务器注册的服务名找到该服务并完成调用。如果同一个服务名找到了多个服务器，go-micro 会默认使用负载均衡机制保障公平性。

课后题

这节课，我们用 HTTP POST 请求访问了 HTTP 代理服务服务器：


```
curl -H "content-type: application/json" -d '{"name": "john"}' http://localhost
```

返回结果如下：

```
{
    "greeting": "Hello "
}
```

但是不知道你注意到没有，我们预期返回的信息应该是：

```
{
    "greeting": "Hello john"
}
```

你知道是哪个地方出现了问题吗？

欢迎你跟我交流讨论，我们也会在后面修复这一问题。下节课见！

精选留言 (2)

写留言

viclilei
2023-01-11 来自广东

docker etcd error

共 1 条评论 >

shuff1e
2022-12-27 来自北京

mux := runtime.NewServeMux()
看起来是HandleHTTP没指定路由？name是从body取，还是从query参数取，也没指定？