云资源自动化 for Terraform

Provider 共建

产品文档

腾讯云

文档目录

Provider 共建

欢迎

最近更新时间：2023-03-07 10:35:48

Terraform Tencent Cloud Provider 由腾讯云的 IaC 团队维护，我们欢迎更多的开发者参与进来。

> 说明：
>
> 本文档适用于 Terraform Tencent Cloud Provider 代码开发人员。

贡献

请按照以下步骤确保您的 Contribute 能够顺利进行。当然您可以先对 Provider 的实现原理做一个大致的了解，同时也可以查看我们的要求与建议。

1. 配置开发环境

在开始开发之前，您需要安装 Terraform 和 Go 拉取代码库，进行程序编译并设置测试。请参阅开发环境配置。

2. 编写代码

根据您贡献代码的类型，从下面的表格中，参阅相应的指引文档。

贡献类型	描述
Resource and Data Sources	通过向 Terraform Tencent Cloud Provider 添加新 Resource 和 DataSource 实现，从而支持管理腾讯云产品功能或者查询腾讯云的远程数据
Bug Fix or Enhancement	大部分的请求，应该是对已有功能的增强和修复
Tagging Support	当前资源需要支持 Tag 标签，需要使用统一的 Tag 服务接口
Documentation Changes	文档的新增与变更

3. 编写测试

我们要求所有的代码贡献，必须有相对应的测试覆盖。请参阅用例编写。

4. 创建拉取请求

当您的贡献准备就绪时，在提供商存储库中创建一个拉取请求。请参阅发起PR。

5. 更新变更日志

开源项目需要维护一个用户友好、可读的 CHANGELOG.md，它允许用户一眼就知道发布是否应该对他们产生任何影响，并评估升级的风险。请参阅提交变更日志。

Modules

除了贡献 Provider 源码之外，我们还欢迎提交 Modules。请参与贡献 Modules。

贡献

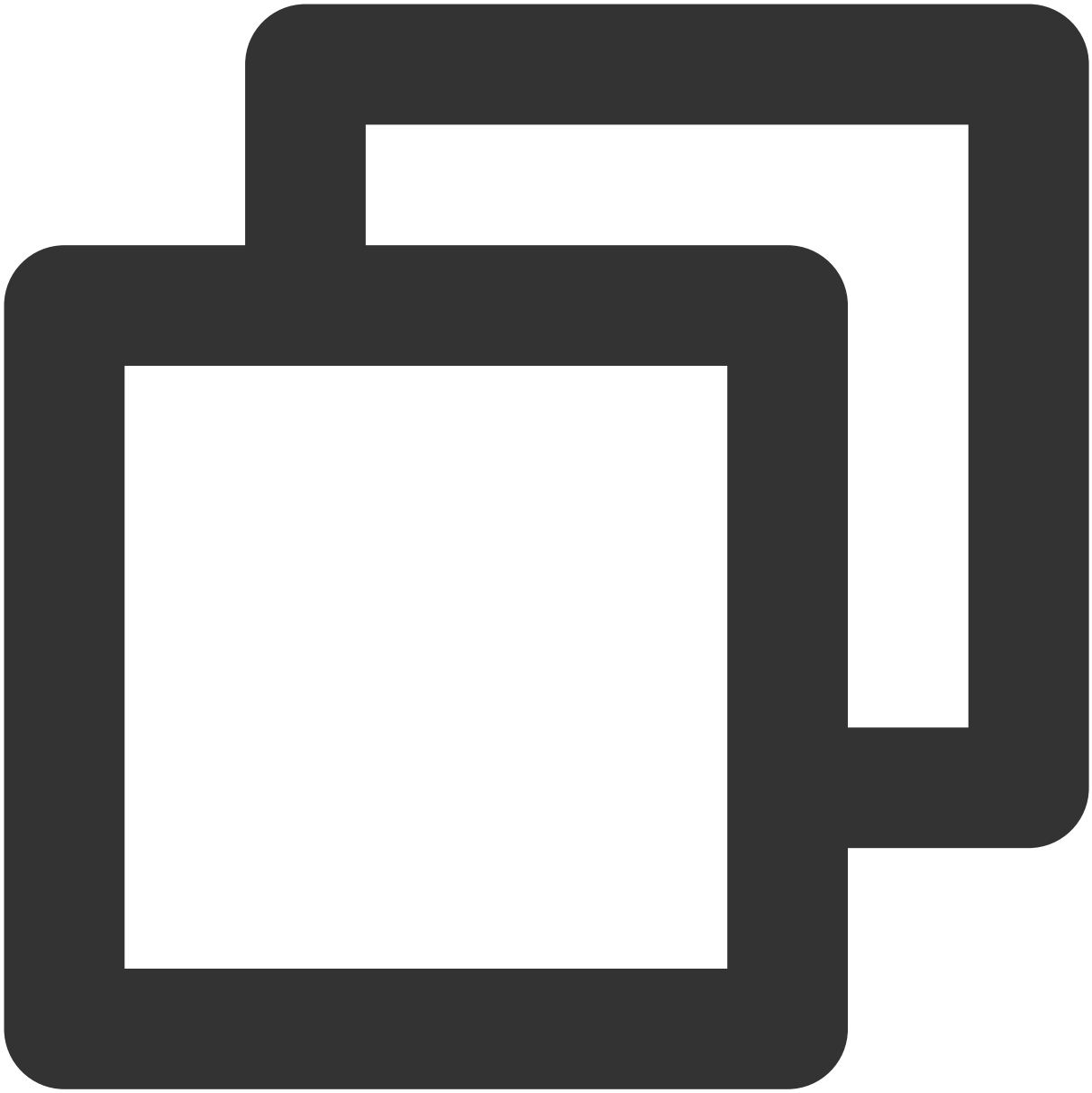
编写 Resource 和 DataSource

最近更新时间：2023-05-29 17:49:06

Terraform 中 Resource 和 DataSource 是最基本的可拓展数据集合。您可以通过这两种集合管理资源。

编写 Resource

Resource 用来描述一类资源实体，这类实体可以被添加、查询、修改和移除，例如云服务器 CVM、硬盘、数据库、TKE 集群、COS 桶等可以被增删改查的实体都是典型的 Resource。
本文以 CVM 为例。一台 CVM 实例的简单配置如下：
名称： my-cvm
可用区： 广州四区
机型： SA2.MEDUIM2
镜像： TencentOS Server 3.2 (Final)
所属网络： vpc-xxxxxxxx
所属子网： subnet-xxxxyyyy
计费类型： 按量计费
是否分配公网IP： 是
公网带宽： 10M
您可以使用高级配置语法 HCL 进行描述，代码示例如下：

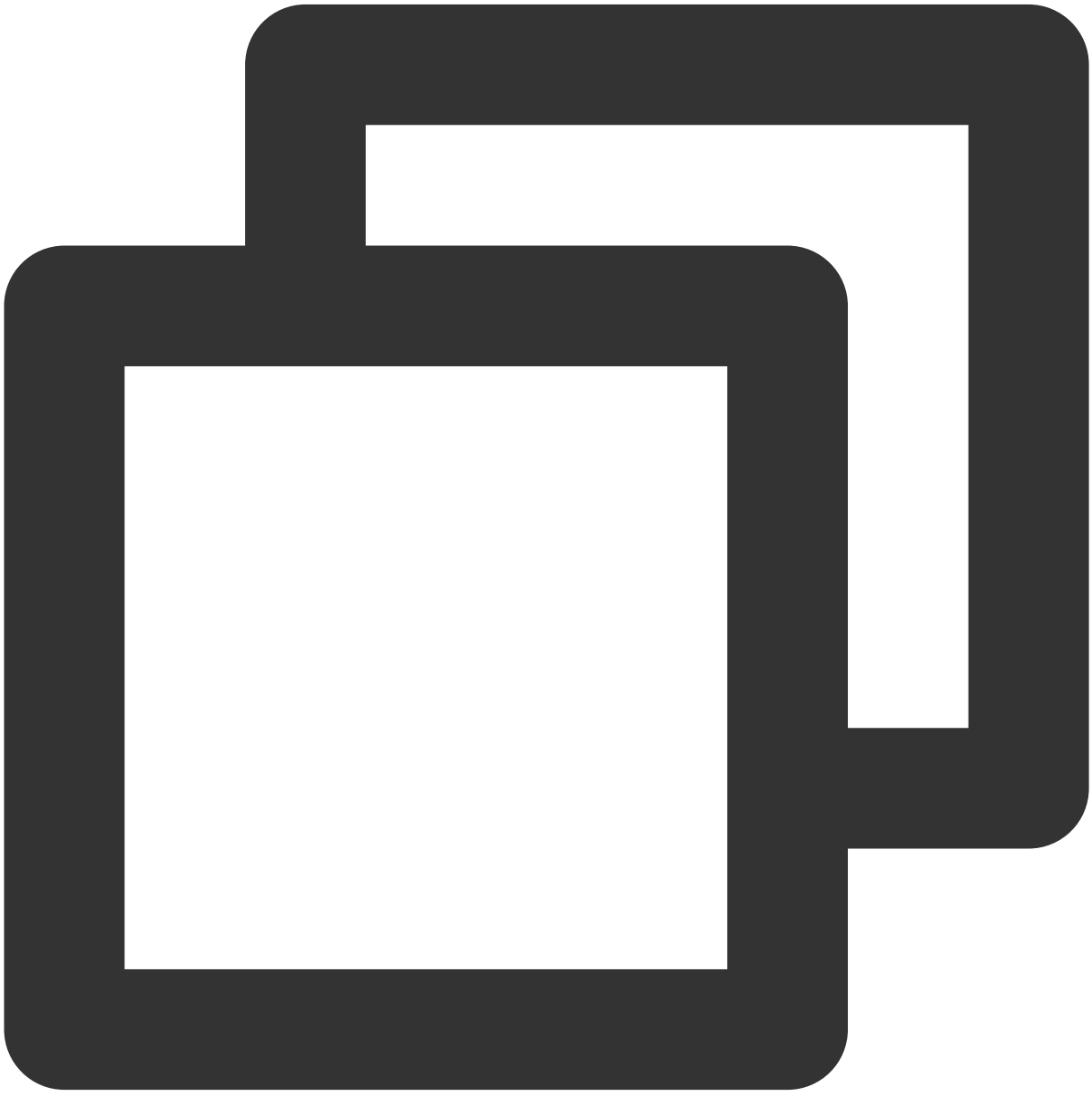


```
resource "tencentcloud_instance" "cvm1" {
  instance_name               = "my-cvm"
  availability_zone           = "ap-guangzhou-4"
  instance_type               = "SA2.MEDIUM2"
  instance_charge_type        = "POSTPAID_BY_HOUR"
  image_id                    = "img-9qrfy1xt"
  allocate_public_ip          = true
  internet_max_bandwidth_out = 10
}
```

当您第一次执行命令 `terraform apply` 时，已声明的资源会发起创建流程。在创建完毕后，若您改动这段代码中的配置，并再次执行命令 `terraform apply` ，将会发起更新流程。执行命令 `terraform destroy` 则会进入销毁流程。

编写 Resource 代码

若要使上面这段 HCL 生效，您需要注册腾讯云 Provider，在 tencentcloud/provider.go 中找到 `Provider/ResourceMap` 字段，添加名为 `tencentcloud_instance` 的资源结构体，按照 HCL 定义参数样板 (Schema)，代码如下：



```
package tencentcloud
```

```
import (
  "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
  "github.com/hashicorp/terraform-plugin-sdk/terraform"
)

func Provider() *schema.Provider {
  return &schema.Provider{
    ResourcesMap: map[string]*schema.Resource{
      "tencentcloud_xxx": { /* 其他声明好的资源 */ },
      "tencentcloud_yyy": { /* 其他声明好的资源 */ },
      "tencentcloud_instance": {
        Schema: map[string]*schema.Schema{
          "instance_name": {
            Optional: true, // 可选字段
            Type:     schema.TypeString, // 字段类型
            Description: "Instance Name.",
          },
          "availability_zone": {
            Required: true, // 必填字段
            Type:     schema.TypeString,
            ForceNew: true, // 当这个字段更改，提交变更后资源销毁重建，因为服务器实例无法切换
            Description: "Instance available zone.",
          },
          "instance_type": {
            Optional: true,
            Type:     schema.TypeString,
            Description: "Instance Type.",
          },
          "instance_charge_type": {
            Optional: true,
            Type:     schema.TypeString,
            Description: "Instance charge type.",
          },
          "image_id": {
            Required: true,
            Type:     schema.TypeString,
            Description: "Instance OS image Id.",
          },
          "allocate_public_ip": {
            Optional: true,
            Type:     schema.TypeBool, // 布尔类型
            Description: "Specify whether to allocate public IP.",
          },
          "internet_max_bandwidth_out": {
            Optional: true,
            Type:     schema.TypeInt, // 整数类型
```

```
            Description: "Specify maximum bandwith.",
          },
        },
      },
    },
  }
}
```

声明资源和参数样板后，Terraform 执行 `apply` / `plan` / `destroy` 时触发资源的增、删、改和同步逻辑也需要您自行定义。Terraform SDK (v1) 中定义了资源的 CRUD 四种方法：

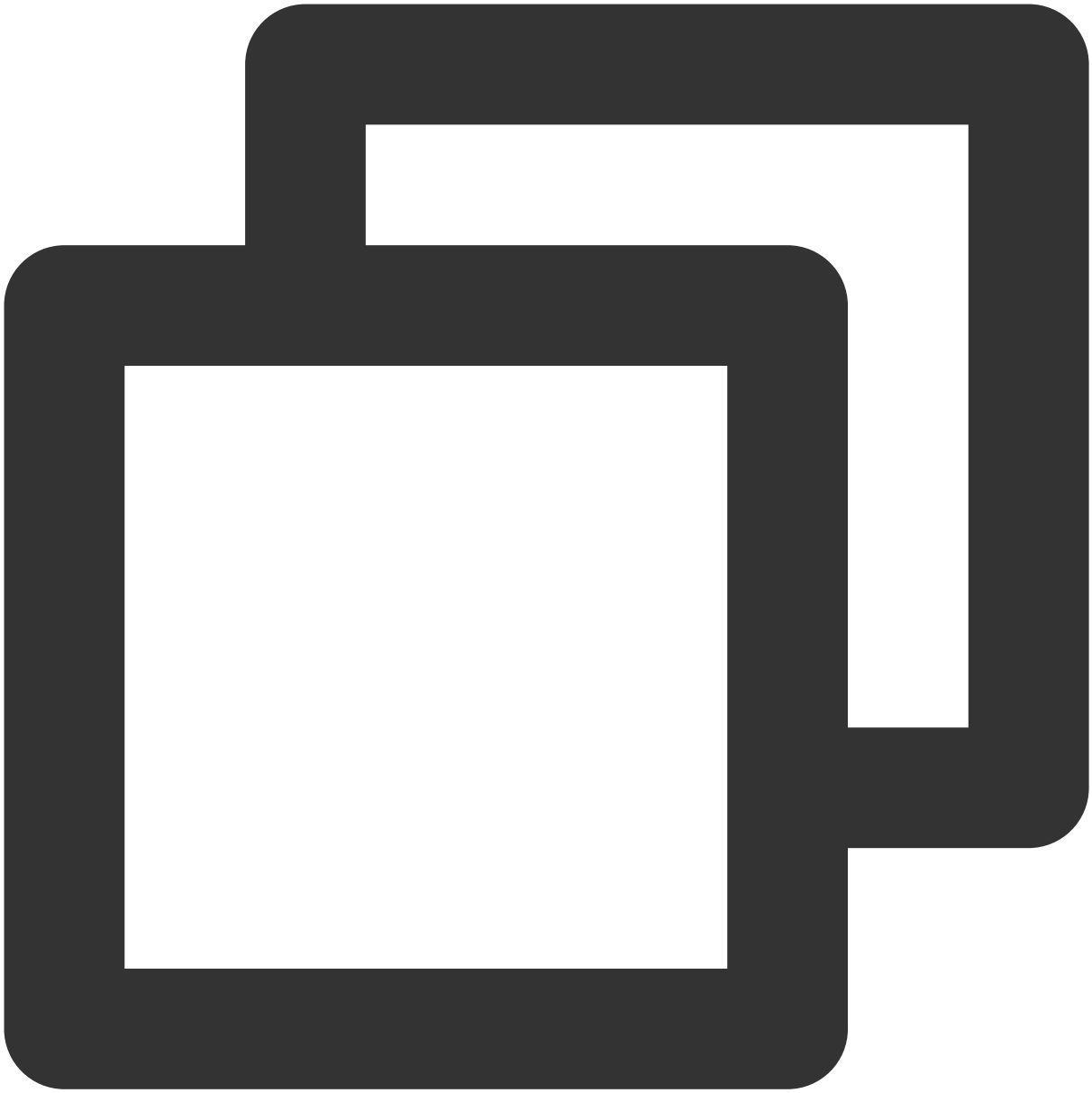
`Create` 执行 `terraform apply` 且为新建时调用。

`Read` 执行 `terraform plan / import` 时调用，用来同步远端状态。

`Update` 执行 `terraform apply` 且为更新时调用。

`Delete` 执行 `terraform destroy` 时调用。

接下来向 `schema.Resource` 结构体中添加这四个方法，代码如下：



```
package tencentcloud

import (
  "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
  "github.com/hashicorp/terraform-plugin-sdk/terraform"
)

func Provider() *schema.Provider {
  return &schema.Provider{
    ResourcesMap: map[string]*schema.Resource{
      "tencentcloud_xxx": { /* 其他声明好的资源 */ },
```

```
      "tencentcloud_yyy": { /* 其他声明好的资源 */ },
      "tencentcloud_instance": {
        Create: resourceTencentCloudInstanceCreate,
        Read:   resourceTencentCloudInstanceRead,
        Update: resourceTencentCloudInstanceUpdate,
        Delete: resourceTencentCloudInstanceDelete,
        Schema: map[string]*schema.Schema{
          // 上文写好的声明，略
        },
      },
    },
  }
}

func resourceTencentCloudInstanceCreate (d *schema.ResouceData, m interface{}) erro
  instanceName := d.Get("instance_name").(string)
  instanceType := d.Get("instance_type").(string)
  // ...

  // terraform apply 新建资源的时候执行
  instanceId := createInstance(&param{
    Name: &instanceName,
    Type: &instanceType,
    // ...
  })

  // 创建完成后，给资源设置唯一 Id
  d.SetId(instanceId)

  // 资源创建完毕后，需要再次同步远端状态，验证是否一致
  return resourceTencentCloudInstanceRead(d, m)
}

func resourceTencentCloudInstanceRead (d *schema.ResouceData, m interface{}) error
  // terraform plan 时和 terraform 创建/更新资源完毕时执行

  instanceId := d.Id() // Create 中设置的 Id
  instanceInfo := getInstance(instanceId)

  d.Set("instance_name", instanceInfo.Name)
  d.Set("instance_type", instanceInfo.Type)

  return nil
}

func resourceTencentCloudInstanceUpdate (d *schema.ResouceData, m interface{}) erro
  // terraform apply 更新已有资源时执行
```

```
  updateParam := &param{}

  // 逐项检查字段是否更新，组合更新参数
  if d.HasChange("instance_name") {
    name := d.Get("instance_name").(string)
    updateParam.Name = &name
  }

  if d.HasChange("instance_type") {
    insType := d.Get("instance_type").(string)
    updateParam.Type = &insType
  }
  // ...
  updateInstance(d.Id(), updateParam)

  // 资源创建完毕后，需要再次同步远端状态，验证是否一致
  return resourceTencentCloudInstanceRead(d, m)
}

func resourceTencentCloudInstanceDelete (d *schema.ResouceData, m interface{}) erro
  // terraform destroy 时执行
  destroyInstance(d.Id())

  return nil
}
```

可以看到在函数中，引用 `d.Get` 参数可以获取 HCL 字段的值， `d.Set` 可以回写（同步）这些值，此外每个资源创建完毕后还要调用 `d.SetId` 设置资源的唯一索引，以保证后续操作和远程真实资源的映射关系一致。

完成四个函数的主逻辑骨架后，terraform 执行 `plan apply destroy` 即可正常调用，接下来您可以发起云 API 调用，真实地操作资源。

云服务器的 CVM 的 CRUD 接口分别为：

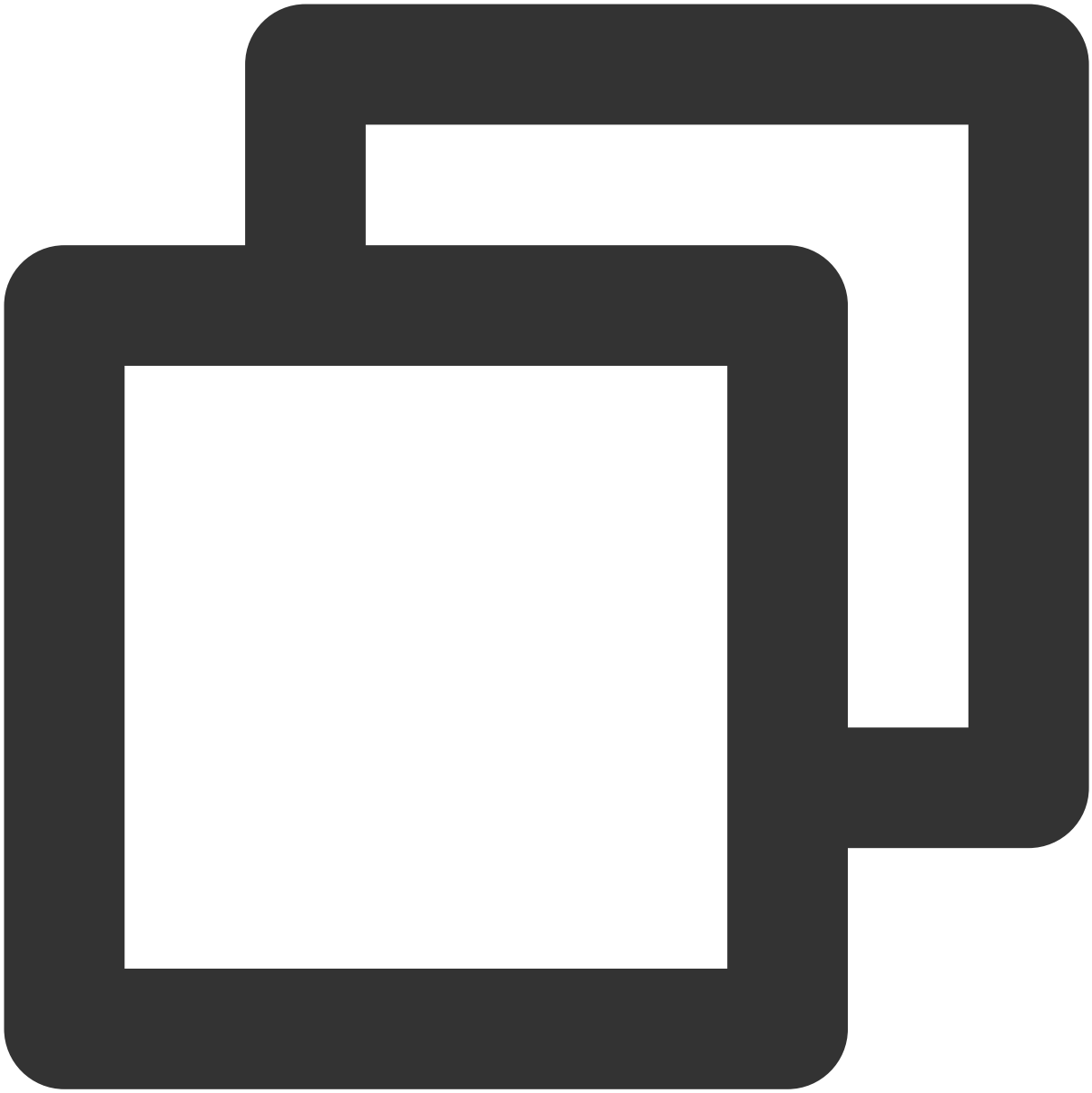
RunInstances ：购买新实例

DescribeInstances ：查询实例列表

ModifyInstancesAttribute ：修改实例属性

TerminateInstances ：退还实例

您可以将从 HCL 获取的参数传给 CVM API 所需参数发起调用，代码如下：



```
package main

import (
  cvm "github.com/tencentcloud/tencentcloud-sdk-go/tencentcloud/cvm/v20170312"
)

func resourceTencentCloudInstanceCreate(d *schema.ResourceData, meta interface{}) e
  // 从 HCL 中读取字段的值
  instanceName        := d.Get("instance_name").(string)
  instanceType        := d.Get("instance_type").(string)
  imageId             := d.Get("image_id").(string)
```

```
    instanceChargeType := d.Get("instance_charge_type").(string)
    zone               := d.Get("availability_zone").(string)
    allocatePublicIp   := d.Get("allocate_public_ip").(bool)
    internetBandWith   := int64(d.Get("internet_max_bandwidth_out").(int))

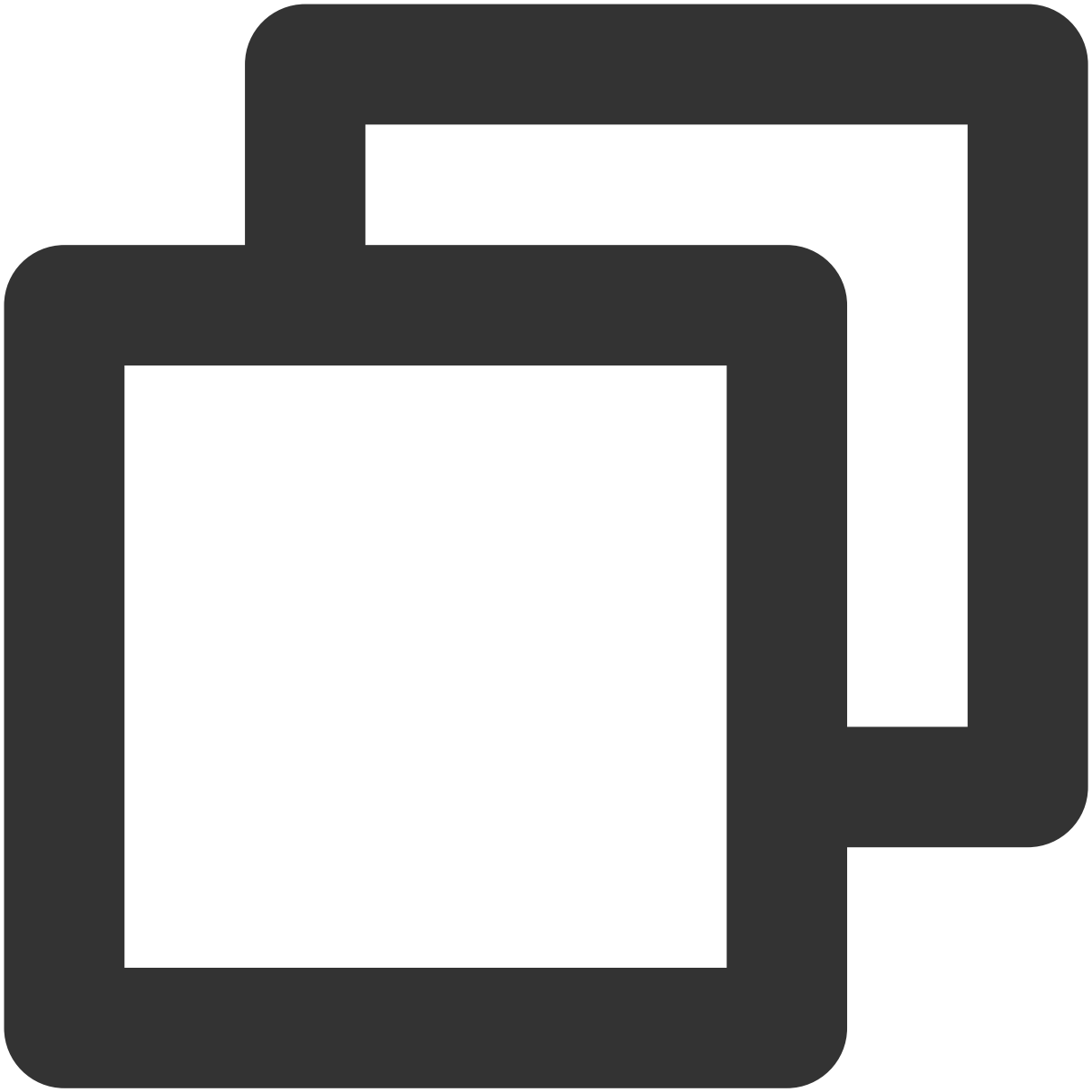
    client, _ := cvm.NewClient(credential, "ap-guangzhou")
    request := cvm.NewRunInstancesRequest()
    //  组合创建 CVM 所需的参数
    request.InstanceName       = &instanceName
    request.InstanceType       = &instanceType
    request.ImageId            = &imageId
    request.InstanceChargeType = &instanceChargeType
    request.Placement          = &cvm.Placement {
      Zone: &zone,
    }
    request.InternetAccessible = &cvm.InternetAccessible {
      InternetMaxBandwidthOut: &internetBandWith,
      PublicIpAssigned:        &allocatePublicIp,
    }

    // 发起调用
    id, err := client.RunInstances(request)
    if err != nil {
      return err
    }

    // 将创建返回的 ID 设置为资源 ID
    d.SetId(id)

    // 回写状态
    return resourceTencentCloudInstanceRead(d, meta)
  }
```

实现 Create 后，再接着实现 Read / Update / Delete ：



```
package main

import (
  cvm "github.com/tencentcloud/tencentcloud-sdk-go/tencentcloud/cvm/v20170312"
)

func resourceTencentCloudInstanceRead(d *schema.ResourceData, meta interface{}) err
  // Create 中设置或导入时拿到的 ID
  id := d.Id()

  client, _ := cvm.NewClient(credential, "ap-guangzhou")
```

```
  request, _ := cvm.NewDescribeInstancesRequest()
  request.InstanceIds = []*string{&id}
  response, err := client.DescribeInstances(request)
  if err != nil {
    return err
  }
  if len(response.Response.InstanceSet) == 0 {
    return fmt.Errorf("instance %s not exists.", id)
  }
  instance := response.Response.InstanceSet[0]

  d.Set("instance_name", instance.InstanceName)
  d.Set("instance_type", instance.InstanceType)
  // d.Set 回写其他如 `image_id` `instance_charge_type` `availability_zone` 等字段，略

  return nil
}

func resourceTencentCloudInstanceUpdate(d *schema.ResourceData, meta interface{}) e
  id := d.Id()
  client, _ := cvm.NewClient(credential, "ap-guangzhou")
  request, _ := cvm.NewModifyInstancesAttributeRequest()
  request.InstanceIds = []*string{&id}

  // 检查字段是否变更且添加参数
  if d.HasChange("instance_name") {
    name := d.Get("instance_name").(string)
    request.InstanceName = &name
  }

  // 处理其他 HasChange 字段，略

  _, err := client.ModifyInstanceAttribute(request)
  if err != nil {
    return err
  }

  return resourceTencentCloudInstanceRead(d, meta)
}

func resourceTencentCloudInstanceDelete(d *schema.ResourceData, meta interface{}) e
  id := d.Id()

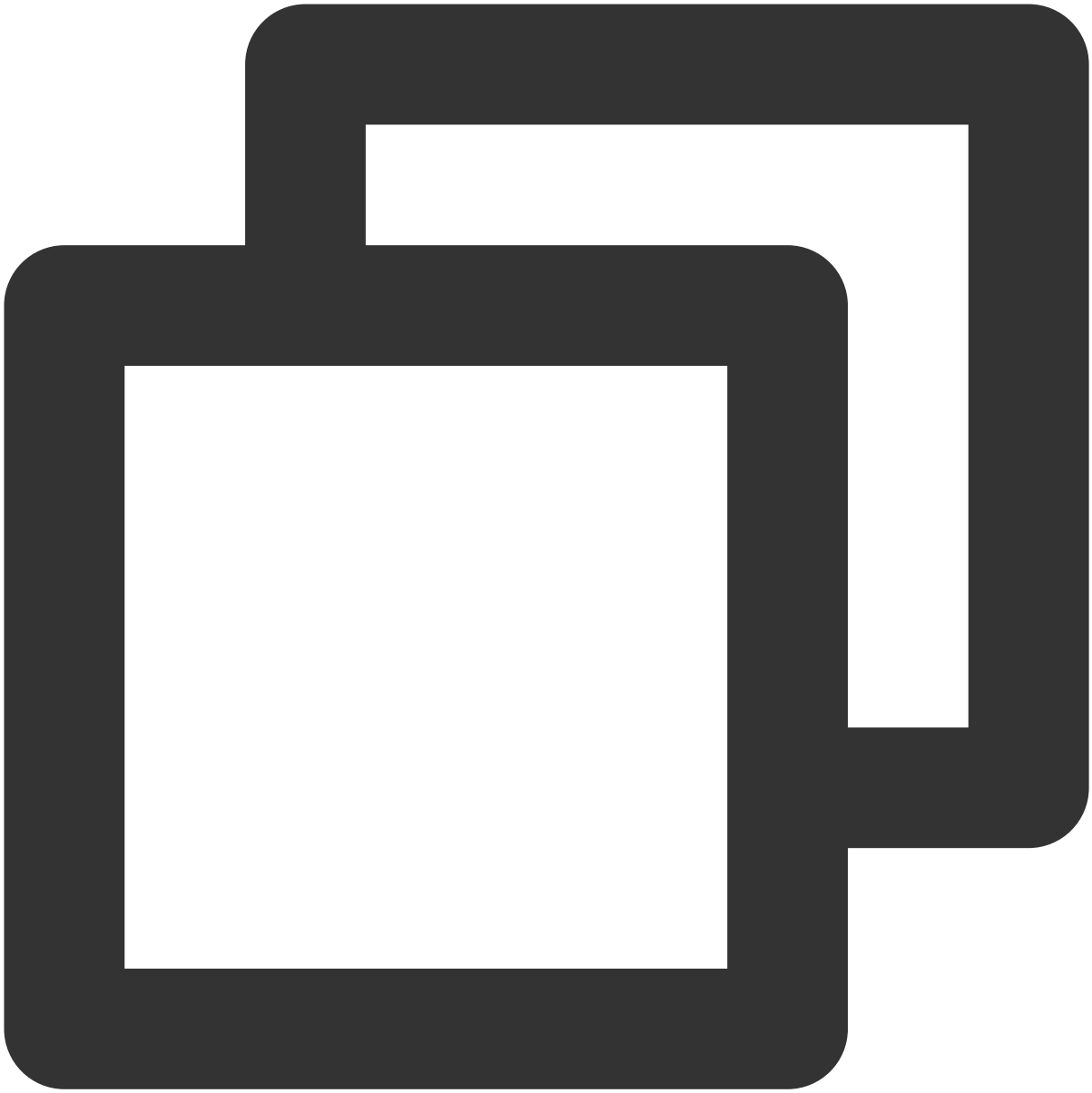
  client, _ := cvm.NewClient(credential, "ap-guangzhou")
  request, _ := cvm.NewTerminateInstancesRequest()
  request.InstanceIds = []*string{&id}
  _, err := client.TerminateInstances(request)
```

```
    if err != nil {
      return err
    }
    return nil
  }
```

至此，在腾讯云 Provider 下一个包含名称、字段声明和赠删改查逻辑的资源全部实现。

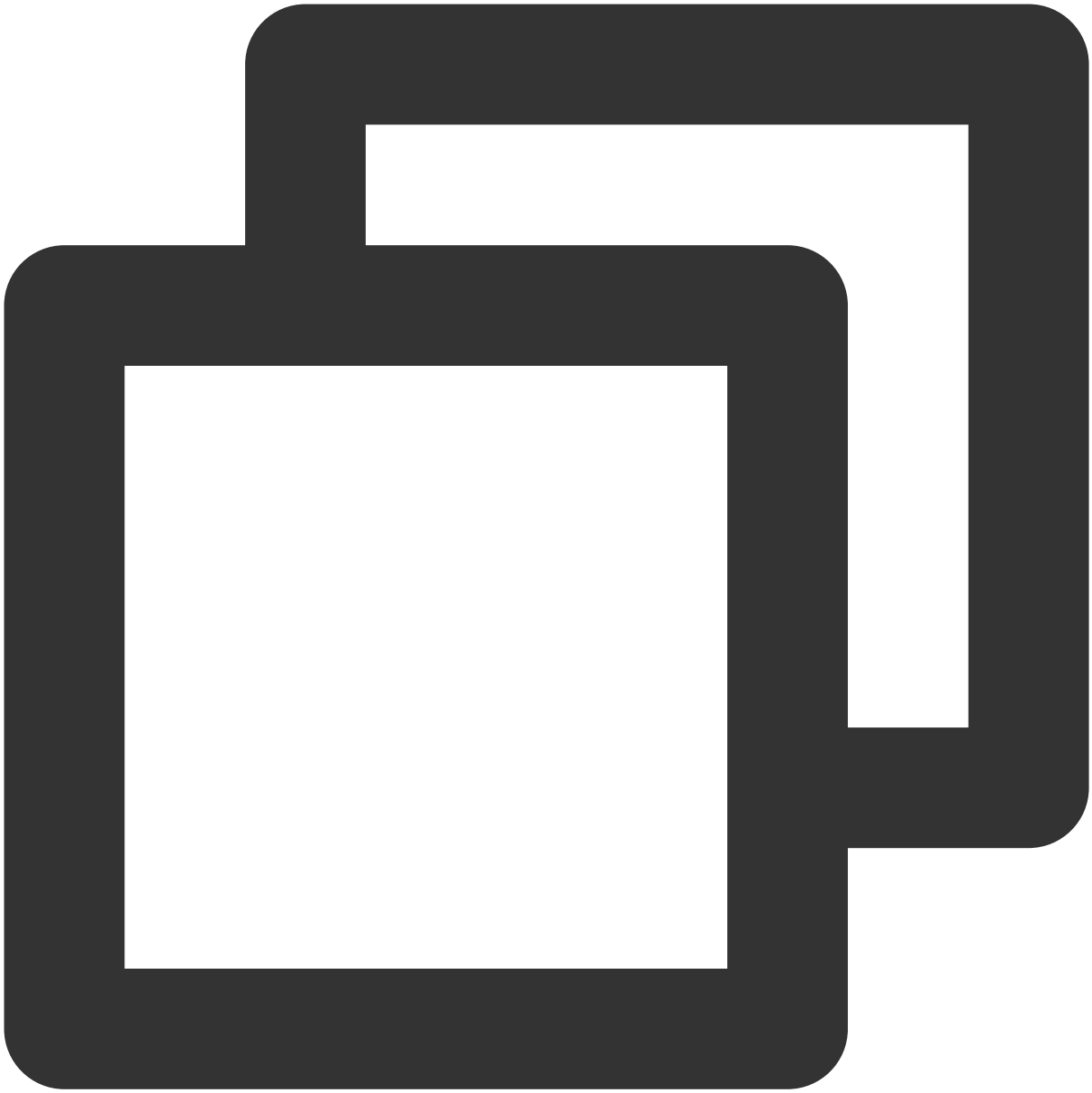
实现 Import 逻辑

成功实现了资源的增删改查后，可以实现资源的导入逻辑。导入单个资源的命令格式为 `terraform import <type>.<index> <id>` ，例如：



```
terraform import tencentcloud_instance.cvm1 ins-abcd1234
```

可以看到，资源导入只有一个 `ins-abcd1234` 输入，回想我们已实现的 Read 方法，可以得知：当 Id 确定时，可以根据 Id 查询远端资源的详细配置，自动将配置同步到本地。所以我们只需要在 schema.Resource 中声明 `Importer` 表示该资源可以被导入，后续同步操作由 Read 接管：



```
package tencentcloud

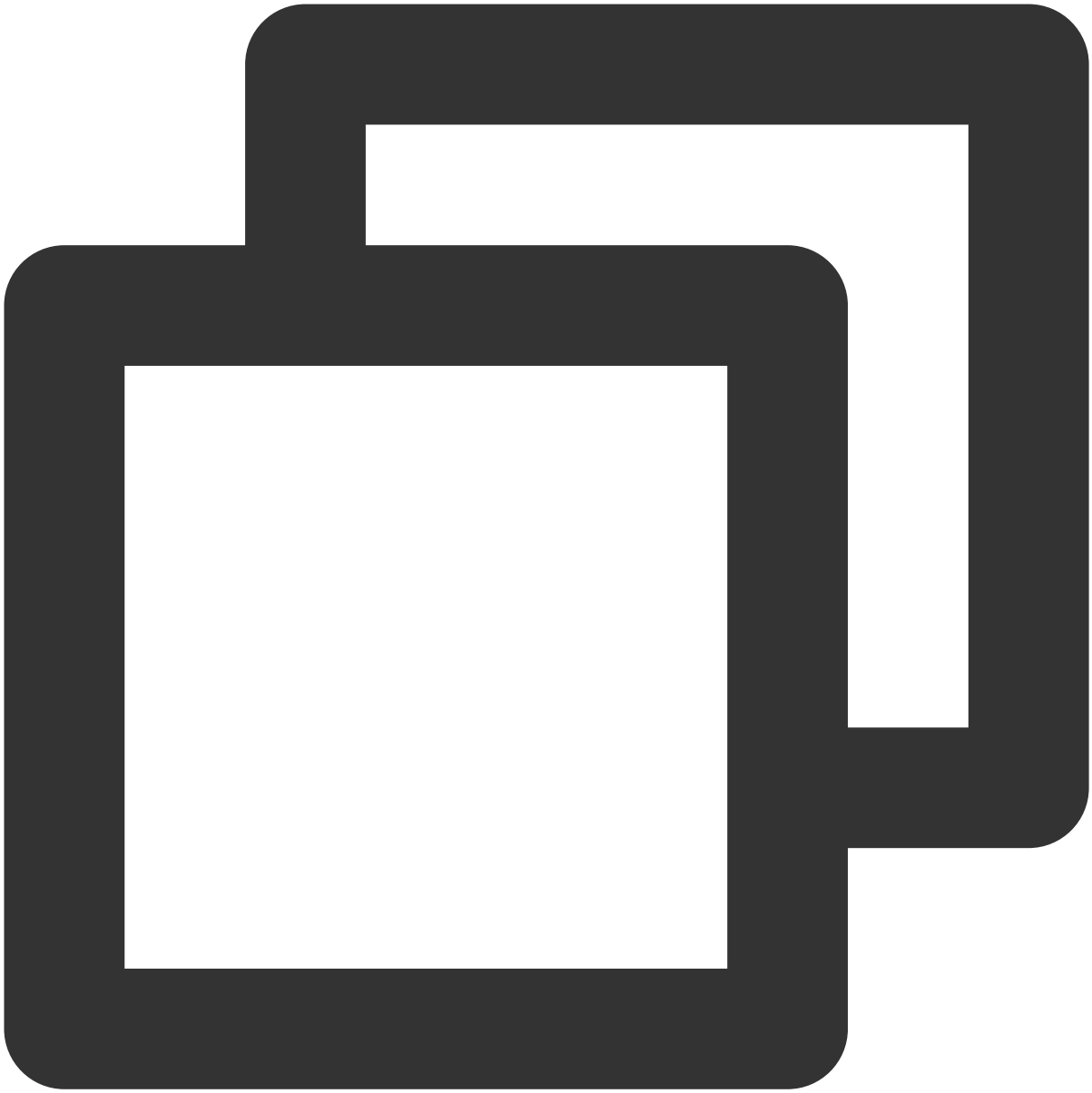
import (
  "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
  "github.com/hashicorp/terraform-plugin-sdk/terraform"
)

func Provider() *schema.Provider {
  return &schema.Provider{
    ResourcesMap: map[string]*schema.Resource{
      "tencentcloud_xxx": { /* 其他声明好的资源 */ },
```

```
        "tencentcloud_yyy": { /* 其他声明好的资源 */ },
        "tencentcloud_instance": {
          Create: resourceTencentCloudInstanceCreate,
          Read:   resourceTencentCloudInstanceRead,
          Update: resourceTencentCloudInstanceUpdate,
          Delete: resourceTencentCloudInstanceDelete,
          Schema: map[string]*schema.Schema{
            // 上文写好的声明，略
          },
          // 绝大部分情况，只需要添加 schema.ImportStatePassthrough 即可
          Importer: &schema.ResourceImporter{
            State: schema.ImportStatePassthrough,
          },
        },
      },
    }
}
```

DataSource

DataSource 代表只读实体，仅用做查询，无论调用多少次都不影响现有资源，如云服务器列表、镜像列表、可用区列表、DB 列表和参数模板、审计日志等。理论上只要能够通过 API 查询出来的数据，都可看作 DataSource，不知您是否注意到：上文的 CVM 镜像对外显示为 `TencentOS Server 3.2 (Final)`，但是实际传递给 API 的参数却是它的 ID `img-9qrfy1xt`，如何通过镜像名称获取对应的 ID 呢？这时候就可以借助 DataSource 来查询并引用了：



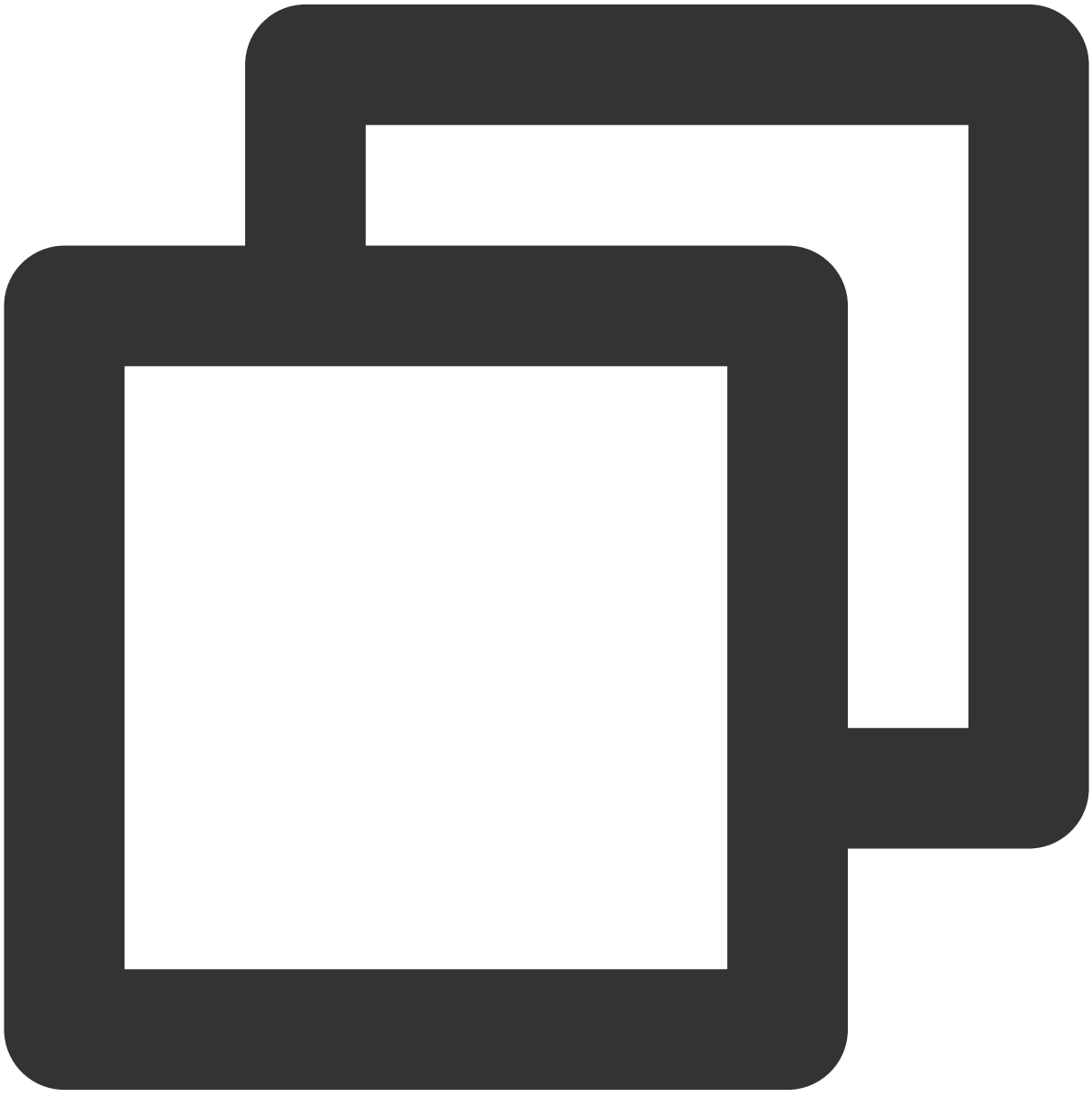
```
data "tencentcloud_images" "fav_os" {
  filters = {
    image_name: "TencentOS Server 3.2",
    image_type: "PUBLIC_IMAGE"
  }
}

resource "tencentcloud_instance" "cvm1" {
  image_id = data.tencentcloud_images.fav_os.images.0.image_id
}
```

如上所示：将 `image_id` 写成对 `data.tencentcloud_images.fav_os` 的引用，Terraform 就会先读取 DataSource 的信息，再将结果计算求值。相比于静态的写法，编写 DataSource 能有效地将带有依赖的资源组织起来。

编写 DataSource

实现 DataSource 的方法和 Resource 差不多，同样需要在腾讯云 Provider 中注册，回到 tencentcloud/provider.go 源码，这次我们找到 `Provider/DataSourceMap` 字段，添加名为 `tencentcloud_images` 的结构体（所有 DataSource 名字都应该以 `s` 结尾），按照 HCL 定义参数样板 (Schema)：



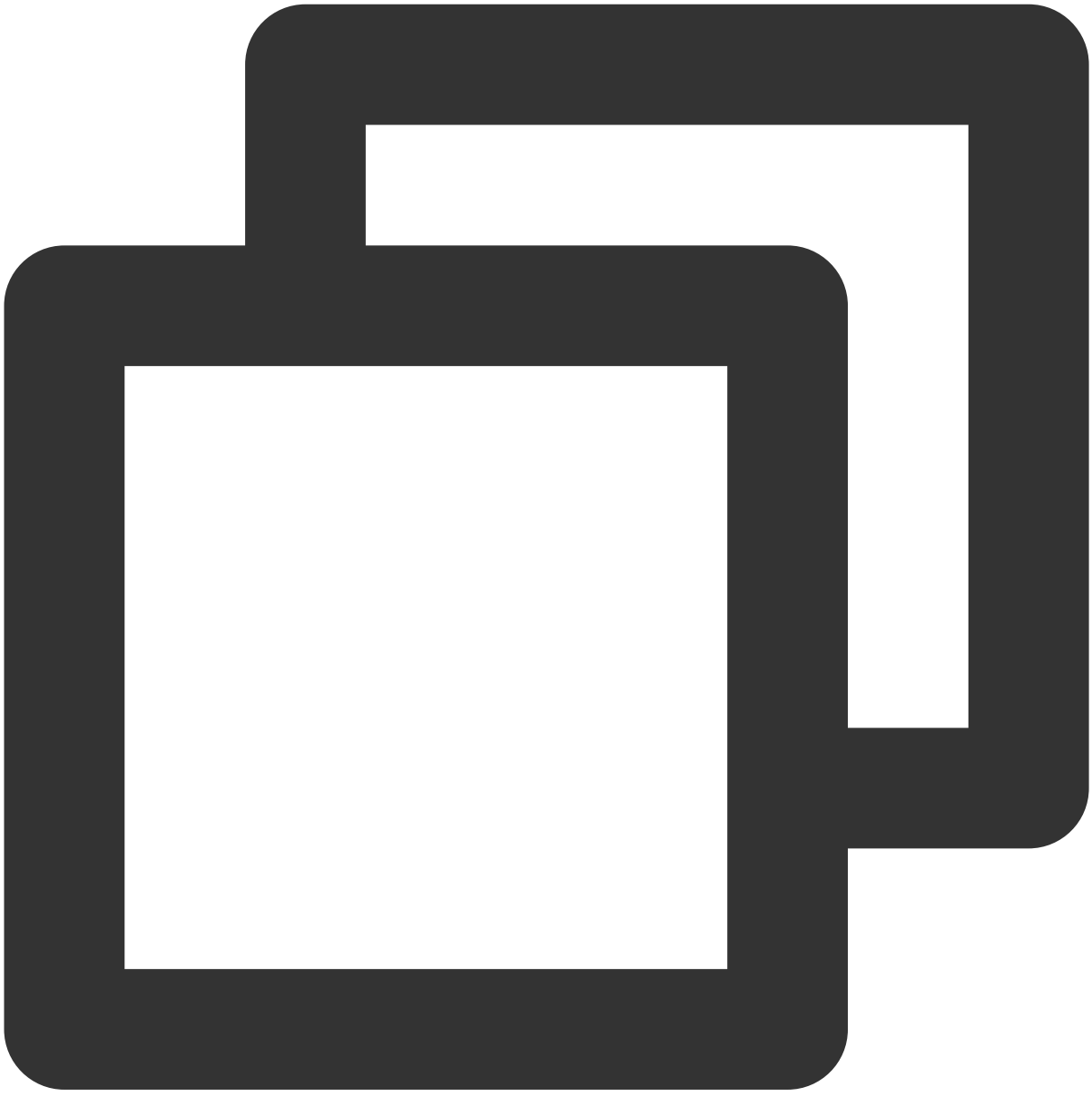
```
package tencentcloud
```

```
import (
  "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
  "github.com/hashicorp/terraform-plugin-sdk/terraform"
)

func Provider() *schema.Provider {
  return &schema.Provider{
    DataSourceMap: map[string]*schema.Resource{
      "tencentcloud_xxxs": { /* 其他声明好的数据源 */ },
      "tencentcloud_yyys": { /* 其他声明好的数据源 */ },
      "tencentcloud_images": {
        Schema: map[string]*schema.Schema{
          "filters": {
            Optional: true,
            Type:     schema.TypeMap, // 定义为 Map 类型
            Description: "Query filter",
          },
          // 用来将结果以 JSON 的形式保存在本地。
          // TencentCloud Provider 约定每个 DataSource 都应带有 `result_output_file`
          "result_output_file": {
            Optional: true,
            Type: schema.TypeString,
            Description: "Used for store as local file.",
          },
          "result": {
            Computed: true, // Computed 可以表示该字段会进行被动更新
            Type:    schema.TypeList,
            Description: "",
            Elem: &schema.Resource{
              Schema: map[string]*schema.Schema{
                "id": {
                  Type: schema.TypeString,
                  Computed: true,
                  Description: "Image Id.",
                },
                "name": {
                  Type: schema.,
                  Computed: true,
                  Description: "Image name.",
                },
              },
            },
          },
        },
      },
    },
```

```
    }
}
```

与 Resource 相比， DataSource 只需要实现 Read 方法即可：



```
package tencentcloud

import (
  "encoding/json"
  "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
  "github.com/hashicorp/terraform-plugin-sdk/terraform"
```

```
)

const ImgNameFilterKey = "image-name"

func Provider() *schema.Provider {
  return &schema.Provider{
    DataSourceMap: map[string]*schema.Resource{
      "tencentcloud_xxxs": { /* 其他声明好的数据源 */ },
      "tencentcloud_yyys": { /* 其他声明好的数据源 */ },
      "tencentcloud_images": {
        Schema: map[string]*schema.Schema{ /* 略 */},
        Read: func(d *schema.ResourceData, meta interface{}) {
          // 声明 Client 和 request
          client, _ := cvm.NewClient(credential, "ap-guangzhou")
          request := cvm.NewDescribeImagesRequest()

          // 获取 Filters , 为 Map 类型
          filters := d.Get("filters").(map[string]interface{})

          // 检查 filters["name"] 并添加参数
          if name, ok := filters["name"].(string); ok {
            request.Filters = append(request.Filters, &cvm.Filter{
              Name: &ImgNameFilterKey,
              Values: []*string{&name}
            })
          }

          // 发起调用
          response, err := client.DescribeImages(request)
          if err != nil {
            return err
          }

          // 组装 `result` 字段并写入
          imageSet := response.Response.ImageSet
          result := make([]map[string]interface{}, 0, len(imageSet))
          for i := range imageSet {
            item := imageSet[i]
            result = append(result, map[string]interface{}{
              "id": *item.Id,
              "name": *item.Name,
            })
          }
          d.Set("result", result)

          // 如果 `result_output_file` 有定义，则往指定文件写入结果
          if v, ok := d.GetOk("result_output_file"); ok {
```

```
            writeToFile(v.(string), result)
          }
          return nil
        }
      },
    },
  }
}
```

这样我们变完成了 `tencentcloud_images` 的编写，后续可以使用这个 DataSource ，查询特定的 OS 镜像。

编写验收测试

为了确保您新增的 Resource 或 DataSource 能正常运作，您需要编写验收测试，详情请参考编写测试用例。

问题修复或功能增强

最近更新时间：2023-03-07 10:35:48

Provider 需要不断完善和迭代，您可以主动往仓库贡献代码。代码贡献遵循最小改动原则，每一次拉取请求都只实现一种增强或问题修复，避免大面积或跨功能的代码改动，增加单次测试和 Review 成本。

代码检查

您的提交合并请求会触发合并检查相关的 Github Actions 包括格式化、文档、验收测试等基本检查，当这些检查通过，我们会主动 Review 您的代码，给出反馈或者直接合入。在分支推送到远端之前您也可以在本地执行这些步骤。

设置提交钩子

本地仓库下目录执行 `make hooks` ，该命令会安装格式化检查的相关依赖，以及将 pre-commit.sh 添加到提交钩子中。

代码格式化

可以主动执行 `make fmt` 格式化 go 代码和 import 顺序。

文档同步

执行 `make doc` ，只要您在代码中更改文档相关的内容， `./gendoc` 脚本就能解析并将改动同步到 `./website` 下。

编写验收测试

要确保您的变更能如期工作，您需要编写验收测试用例，或者在已有的用例中添加参数断言，以 cover 您的变更。详情参考 编写测试用例一文。测试付费资源会真实地发起计费流程，您也可以先写好测试用例，之后由我们来运行您的测试用例。

文档更新

最近更新时间：2023-03-07 10:35:48

腾讯云 Provider 的 文档，原文件位于项目根目录 `website` 下，且全部由代码中的注释和 Schema 的描述中提取并自动生成，下文将介绍文档页面中的侧边栏、示例代码和参数描述的生成机制。

文档生成

观察文档页面，可以看见页面由以下几个部分组成。

目录

在 `tencentcloud/provider.go` 文件顶部注释，以 `Resources List` 开始解析，格式如下：

```
/*
Resources List

产品名称
Data Source
tencentcloud_foos
tencentcloud_bars
Resource
tencentcloud_baz

*/

package tencentcloud
```

以上文本会解析成 `产品名称 -> DataSource / Resource -> tencentcloud_*` 的树形结构，在文档左侧边栏展示。

示例

文档的主题页由简介、用法示例和参数说明组成，所有示例都由每个模块的 `.go` 文件头部注释生成，模板如下：

```
/*
Provides a resource/datasource to create/query something.

~> **NOTE:** This is an optional TIPS, add it if needed.

Example Usage
```

```
Basic usage

hcl
resource "tencentcloud_foo" "foo" {
name = "baz"
}

Another usage

hcl
resource "tencentcloud_foo" "foo" {
name = "baz"
another = 12345
}

Import
This resource can be imported, e.g.

bash
$ terraform import tencentcloud_foo.foo foo_id

*/
package tencentcloud
// ...
```

首先简单介绍 Resource / DataSource 的用法和注意事项（如果有），接着以 `Example Usage` 起头，附上示例代码块。最后，如果是 Resource 类型且提供了导入方法，就写上导入命令。

参数说明

每个资源下的参数说明都由 `Description` 解析并且紧跟在当前资源的示例代码下：

```
map[string]*schema.Schema{
"name": {
Type: schema.TypeString,
Description: "This is what will generated.",
}
}
```

脚本会通过读取 Provider 中 Schema，输出格式为 `名称 - （约束，类型） 介绍文字` 的条目。

文档更新

主动更新

生成文档的脚本位于 `./gendoc` 下。当上述位置的代码有变更时， cd 到 `./gendoc` 并运行 `go run .../.` 可以执行文档生成操作，也可以直接在项目目录下执行 `make doc` ，二者等效。

提交检查

如果您配置了项目中的提交 hook，即使没有运行 `gendoc` ，提交 hook 也会自动帮您检查是否同步。如果提交 hook 执行后，文档出现了变更，说明您尚未把改动的文档同步到暂存区，本次提交操作将会中止。即便您绕过 hook，合并检查也会执行这一步骤，取保您涉及到文档的变更能准确同步。

支持标签功能

最近更新时间：2023-03-07 10:35:48

标签（Tag）是腾讯云提供的云资源管理工具，以 `key:values` 的形式存在，用来关联您的绝大部分云资源，对于资源的分类、搜索和聚合十分有用。
在 Terraform 中，通过 Map 来定义一个资源的 Tag ：

```
resource "tencentcloud_instance" "cvm" {
tags = {
key1: "val1",
key2: "val2"
}
}
```

Tag 代码实现

通过云 API 对资源添加 Tag ，有两种实现方式，一种是通过 CreateAPI 参数传入：

```
rsType := "instance"

request := cvm.NewRunInstanceRequest()
request.TagSpecification = append(request.TagSpecification, &cvm.TagSpecification
{
ResourceType: &rsType,
Tags: []*cvm.Tag{
{
Key: &key,
Value: &value,
},
},
})
```

目前仅有部分资源的创建 API 支持传入 Tag ，且数据结构也有所出入。为了统一管理 Tag 代码，我们采用第二种方式，即创建后单独调用 Tag API 关联，入参为 资源六段式 ，格式为：

```
qcs:<project_id，此处置空>:<模块>:<地域>:<账号/uin>:<资源/ID>
```

以 VPC 举例，如要修改 VPC 实例关联的 Tag ，入参如下：

```
qcs::vpc:ap-singapore:uin/:vpc/vpc-xxxxxxxx
```

代码中则调用封装好的 ModifyTags 即可：

```
package main

func ResourceTencentCloudVPCUpdate(d *schema.ResourceData, meta interface{}) {
ctx := context.TODO()
region := meta.(*TencentCloudClient).apiV3Conn.Region
id := d.Id()

resourceName := fmt.Sprintf("qcs::vpc:%s:uin/:vpc/%s", region, id)
replaceTags, deleteTags := diffTags(oldTags.(map[string]interface{}), newTags.(ma
p[string]interface{}))

if err := tagService.ModifyTags(ctx, resourceName, replaceTags, deleteTags); err
!= nil {
return err
}
}
```

编写测试用例

最近更新时间：2023-03-07 10:35:48

对于构建一套健全的程序系统，完善的测试用例不可或缺，Terraform 的 SDK 集成了一套测试套件，用来验证您编写的 Terraform Resource 和 DataSource ， 下文将介绍如何对腾讯云 Provider 编写测试用例。

验收测试

验收测试 (Acceptance Test) 覆盖一个资源的完整生命周期：创建、查询、更新、导入和删除，每一个资源都需要编写至少一个测试用例。运行验收测试将 **真实地发起 API 调用，影响云资源** 。运行测试之前请留意您的账号成本消耗。编写验收测试的步骤如下：

设置环境变量，打开执行验收测试开关：

```
export TF_ACC=true
```

设置凭证相关环境变量，指定账户运行测试：

```
export TENCENTCLOUD_SECRET_ID=xxxx
export TENCENTCLOUD_SECRET_KEY=yyyy
```

设置日志输出级别和目录，方便调试：

```
export TF_LOG=DEBUG
export TF_LOG_PATH=./terraform.log
```

以私有网络 VPC 举例，测试该资源传入的参数，创建和更新后是否全部符合预期。

`tencentcloud` 目录下创建 `resource_tc_vpc_test.go` 文件，指定一个 VPC 资源，编写其初始配置和更新配置：

```
// filename: tencentcloud/resource_tc_vpc_test.go
const testAccVpcConfig = `
resource "tencentcloud_vpc" "foo" {
name = "test-vpc"
cidr_block = "172.16.0.0/16"
}
`

const testAccVpcConfigUpdate = `
resource "tencentcloud_vpc" "foo" {
name = "test-vpc__update"
```

```
cidr_block = "172.16.0.0/22"
is_multicast = true
}
`
```

编写用例函数，函数名以 `TestAccTencentCloud` 起头表示验收测试，函数名规则为 `TestAccTencentCloud${模块名}${资源类型}_${子名称}`，正则表达式: `TestAccTencentCloud[a-zA-Z]+(Resource|DataSource)_[a-zA-Z]+`， 调用 `resource.Test` 中引入这两个配置并添加断言：

```
package tencentcloud

import (
"testing"

"github.com/hashicorp/terraform-plugin-sdk/helper/resource"
"github.com/hashicorp/terraform-plugin-sdk/terraform"
)

func TestAccTencentCloudVpcResource_Basic(t *testing.T) {
resource.Test(t, resource.TestCase{
Providers: testAccProviders,
Steps: []resource.TestStep{
{
// 上文声明的初始配置
Config: testAccVpcConfig,
Check: resource.ComposeTestCheckFunc(
testAccCheckVpcExists("tencentcloud_vpc.foo"),
resource.TestCheckResourceAttr("tencentcloud_vpc.foo", "cidr_block", "172.16.0.0/
16"),
resource.TestCheckResourceAttr("tencentcloud_vpc.foo", "name", "test-vpc"),
),
},
{
// 上文声明的更新配置
Config: testAccVpcConfig,
Check: resource.ComposeTestCheckFunc(
testAccCheckVpcExists("tencentcloud_vpc.foo"),
resource.TestCheckResourceAttr("tencentcloud_vpc.foo", "cidr_block", "172.16.0.0/
22"),
resource.TestCheckResourceAttr("tencentcloud_vpc.foo", "name", "test-vpc__update"
),
resource.TestCheckResourceAttr("tencentcloud_vpc.foo", "is_multicast", "true"),
),
},
},
```

```
})
}
```

编写完毕，项目根目录执行 `go test -v -run TestAccTencentCloudVpcResource ./tencentcloud` 验证测试结果：

```
TestAccTencentCloudVpcResource_Basic
=== RUN TestAccTencentCloudVpcResource_Basic
=== PAUSE TestAccTencentCloudVpcResource_Basic
=== CONT TestAccTencentCloudVpcResource_Basic
--- PASS: TestAccTencentCloudVpcResource_Basic (26.30s)
PASS
ok github.com/tencentcloudstack/terraform-provider-tencentcloud/tencentcloud 27.1
53s
```

如果设置了日志相关的环境变量，日志详情将会在 `tencentcloud/terraform.log` 中写入。

单元测试

相比上文的验收测试，单元测试的粒度更细，测试成本更低，当您对于某些复杂逻辑代码难以确认是否能正确执行，编写单元测试是一个很好的验证方式。
例如，项目中有函数 `isExpectError(err)` 用来检查云 API 的错误码决定程序应该重试（比如客户端网络不稳定）还是异常退出，可以编写以下测试用例。以 `Test*` 开始：

```
package tencentcloud

import (
"testing"
sdkErrors "github.com/tencentcloud/tencentcloud-sdk-go/tencentcloud/common/error
s"
"github.com/stretchr/testify/assert"
)

func TestIsExpectError(t *testing.T) {

err := sdkErrors.NewTencentCloudSDKError("ClientError.NetworkError", "", "")

// Expected
expectedFull := []string{"ClientError.NetworkError"}
expectedShort := []string{"ClientError"}
assert.Equalf(t, isExpectError(err, expectedFull), true, "")
assert.Equalf(t, isExpectError(err, expectedShort), true, "")
```

```
// Unexpected
unExpectedMatchHead := []string{"ClientError.HttpStatusCodeError"}
unExpectedShort := []string{"SystemError"}
assert.Equalf(t, isExpectError(err, unExpectedMatchHead), false, "")
assert.Equalf(t, isExpectError(err, unExpectedShort), false, "")
}
```

项目根目录下执行 `go test -v -run TestIsExpectError ./tencentcloud` ，查看结果：

```
=== RUN TestIsExpectError
--- PASS: TestIsExpectError (0.00s)
PASS
ok github.com/tencentcloudstack/terraform-provider-tencentcloud/tencentcloud 1.31
2s
```

清理测试资源

Terraform 提供主动清理测试资源的机制 Sweeper ， 当某些资源测试过程中出现异常导致资源没有进行最后的回收步骤，需要我们进行主动清理。
Sweeper 是一组可以通过参数过滤选择性执行的函数，开发者需要主动声明并编写具体的删除逻辑。以清理 Vpc 为例，在 `tencentcloud/resource_tc_vpc_test.go` 中加入 init 函数，注册一个名为 `tencentcloud_vpc` 的 Sweeper：

```
func init() {
resource.AddTestSweepers("tencentcloud_vpc", &resource.Sweeper{
Name: "tencentcloud_vpc",
F: testSweepVpcInstance,
})
}

// 伪代码逻辑，实现指定地域下 test 开头的 VPC 清理
func testSweepVpcInstance(region string) {
vpcs := getAllVpc(region)
for _, vpc in range vpcs {
if vpc.name == "test" {
deleteVpc(vpc.id)
}
}
}
```

如果要清理特定地域（如广州）的 VPC 实例，则执行 `go test -v ./tencentcloud -sweep=ap-guangzhou -sweep-run=tencentcloud_vpc` ， Terraform SDK 会匹配名称为 `tencentcloud_vpc` 的

Sweeper 并传入 `-sweep` 指定的地域给函数并执行，完成该地域的资源清理。

创建拉取请求

最近更新时间：2023-03-07 10:35:48

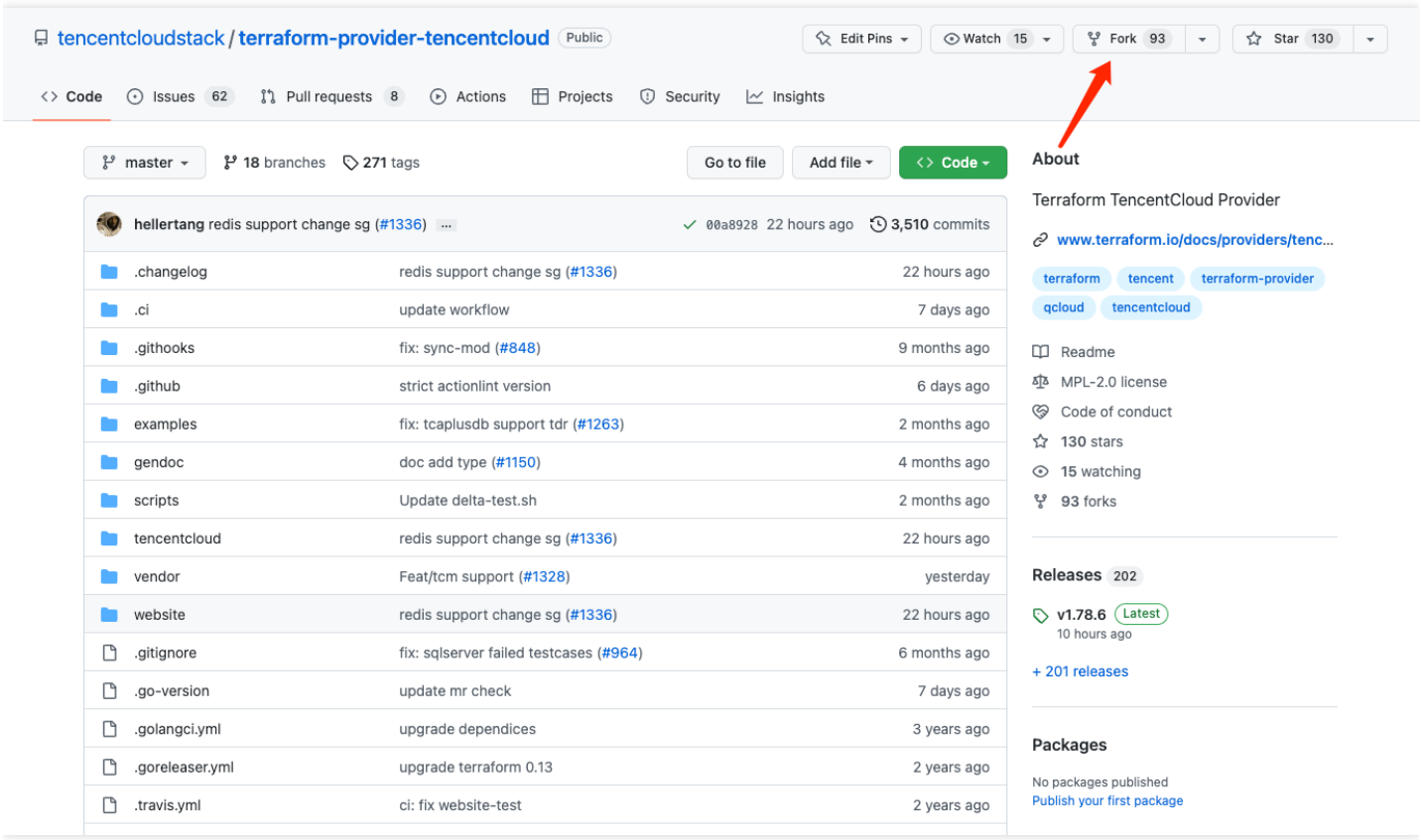
我们欢迎任何人和团队向腾讯云 Provider 贡献代码，向 Provider 发起拉取请求的流程如下：

官方仓库

访问 官方仓库。

Fork 代码

您需从主仓库中 Fork 一份代码到子集的仓库，并对 Fork 出来的仓库进行代码变更。



分支命名约束

分支命名需要约遵循语义化的命名，一般以 `type/scope-content` 的格式，能让他人快速定位您改动的范围和内容，常用的分支前缀如下：

- `fix/*` 修复问题。
- `feat/*` 新增功能。
- `doc/*` 文档变更。
- `style/*` 格式、拼写等不影响逻辑的代码改动。
- `chore/*` 杂项提交，不涉及代码逻辑。

后缀内容尽可能概括改动模块和内容，如：

- `fix/tke-auth-retry` 表示修复 TKE 模块鉴权重试的问题。
- `feat/new-free-ssl-resource` 表示增加新的 SSL 资源。
- `doc/cvm-field-misspell` 表示修改 CVM 文档某处文字错误。

避免出现如下命名，如：

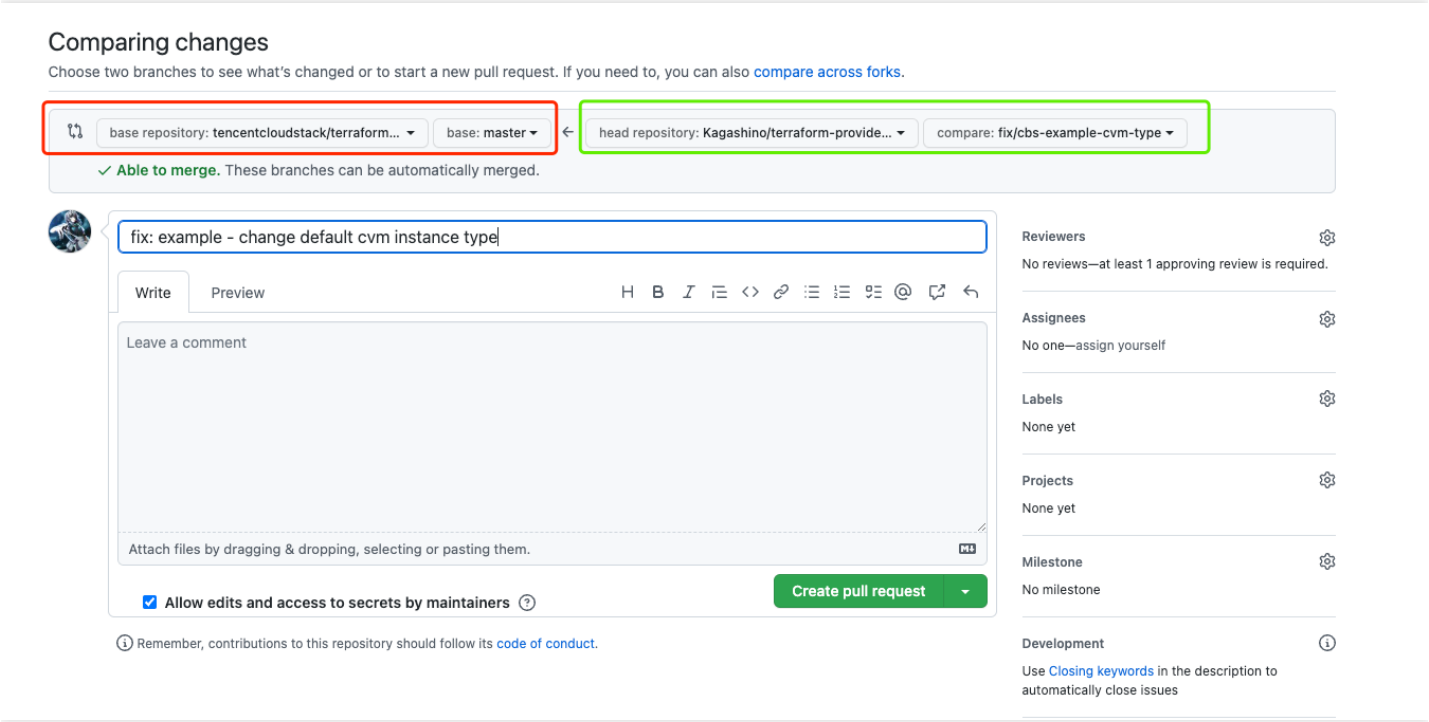
- `john-test` 直接以某位开发者的名字命名 。
- `fix/20221027` 无法体现改动了什么范围和内容。
- `fix/bug` 以及其他带有不适当内容的名称。

验收测试

为了确保您的改动符合预期，涉及到逻辑的变更需要编写并执行验收测试。请参考 编写测试用例。

发起拉取请求

当您的改动完成，请创建一个合并请求到 主仓库 。图中红框选择主仓库，绿框选择您的仓库。



提交 changelog 清单

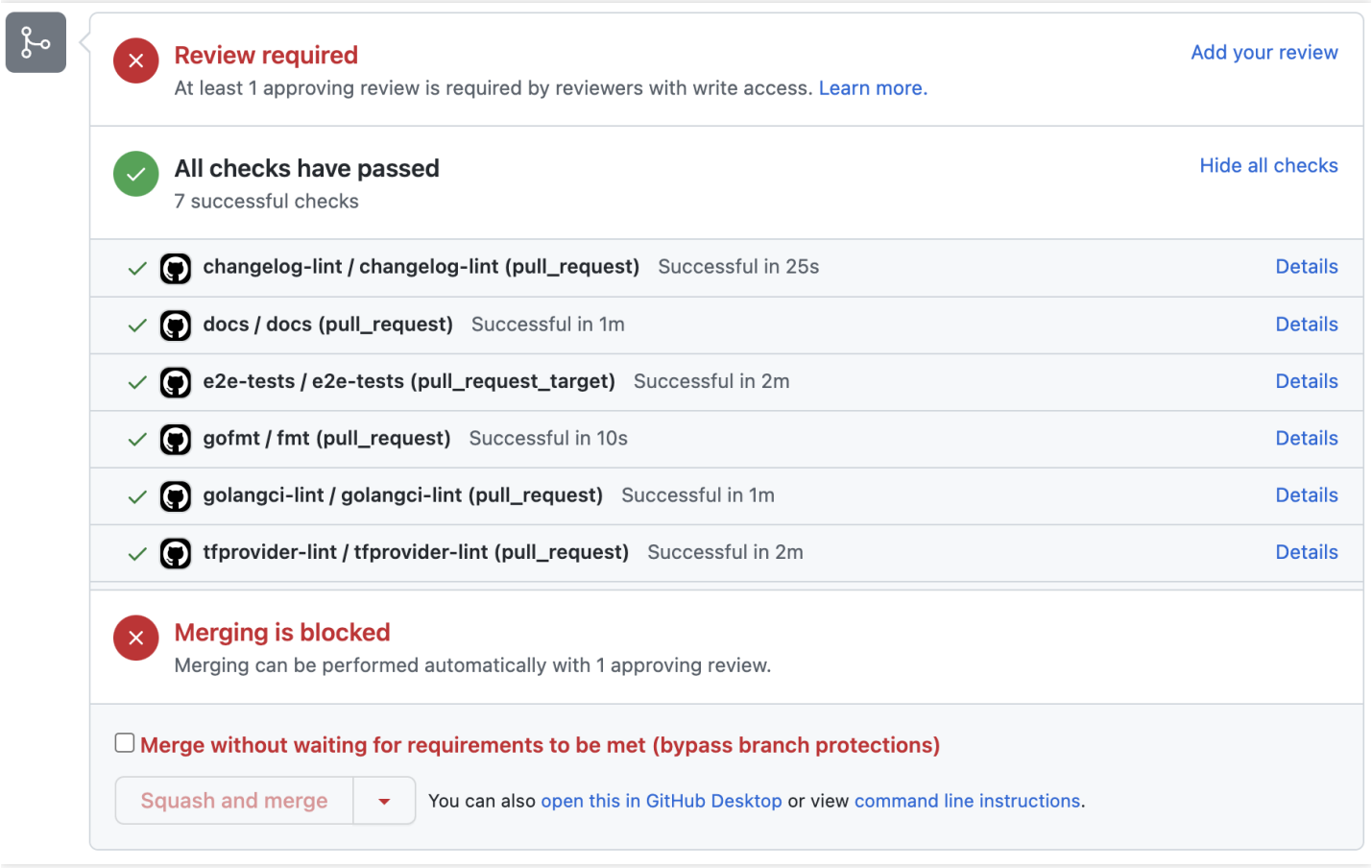
当拉取请求发起后，您还需要再追加提交一条 `.changelog/<pr号码>.txt` 文件，按格式描述本次拉取请求的类型、模块和改动内容，模板如下：

```
resource/<module>: something has done
```

具体内容请参考 提交变更日志。

拉取请求检查

拉取请求发起后，Action 会运行一些基本的合并检查。



如果您的代码需要验收测试，则由代码仓库成员打上 `run-check` 标记，触发执行能覆盖您变更的测试用例。

代码合并

当合并检查通过，仓库成员 Review 并确认分支可以合并后，我们会帮您把分支合入主干，后续会根据合入情况，进行版本发布。至此，整个代码贡献流程已走完，您的贡献将会帮到更多的用户！</pr号码>

提交变更日志

最近更新时间：2023-05-29 15:30:35

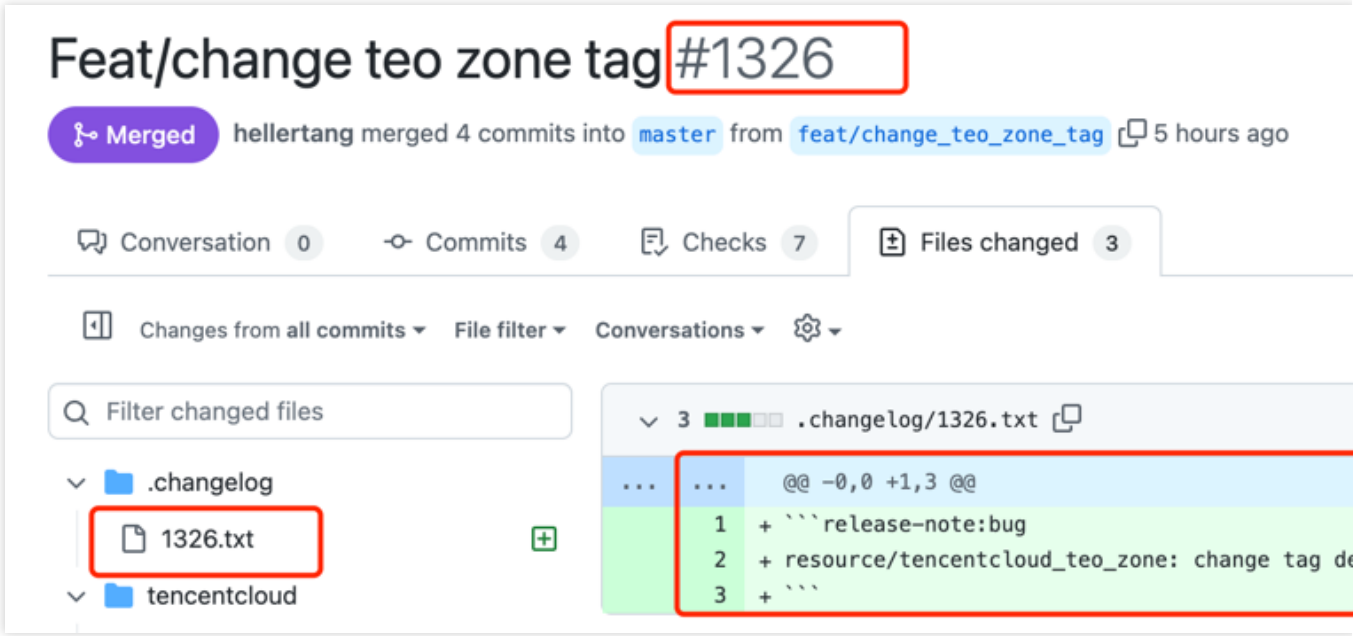
开源项目始终保持用户友好、可读的 CHANGELOG.md，可以让用户快速判断出一个版本是否会对他们产生任何影响，并评估升级的风险。

按照规范，当您发起 PR 时，要求有个 `.txt` 变更文件来描述您这次更改的内容。之后我们会通过 go-changelog 工具，解析 `.changelog` 目录中的 txt 文件，生成 CHANGELOG.md。CHANGELOG.md 的更新规则如下：

变更日志格式

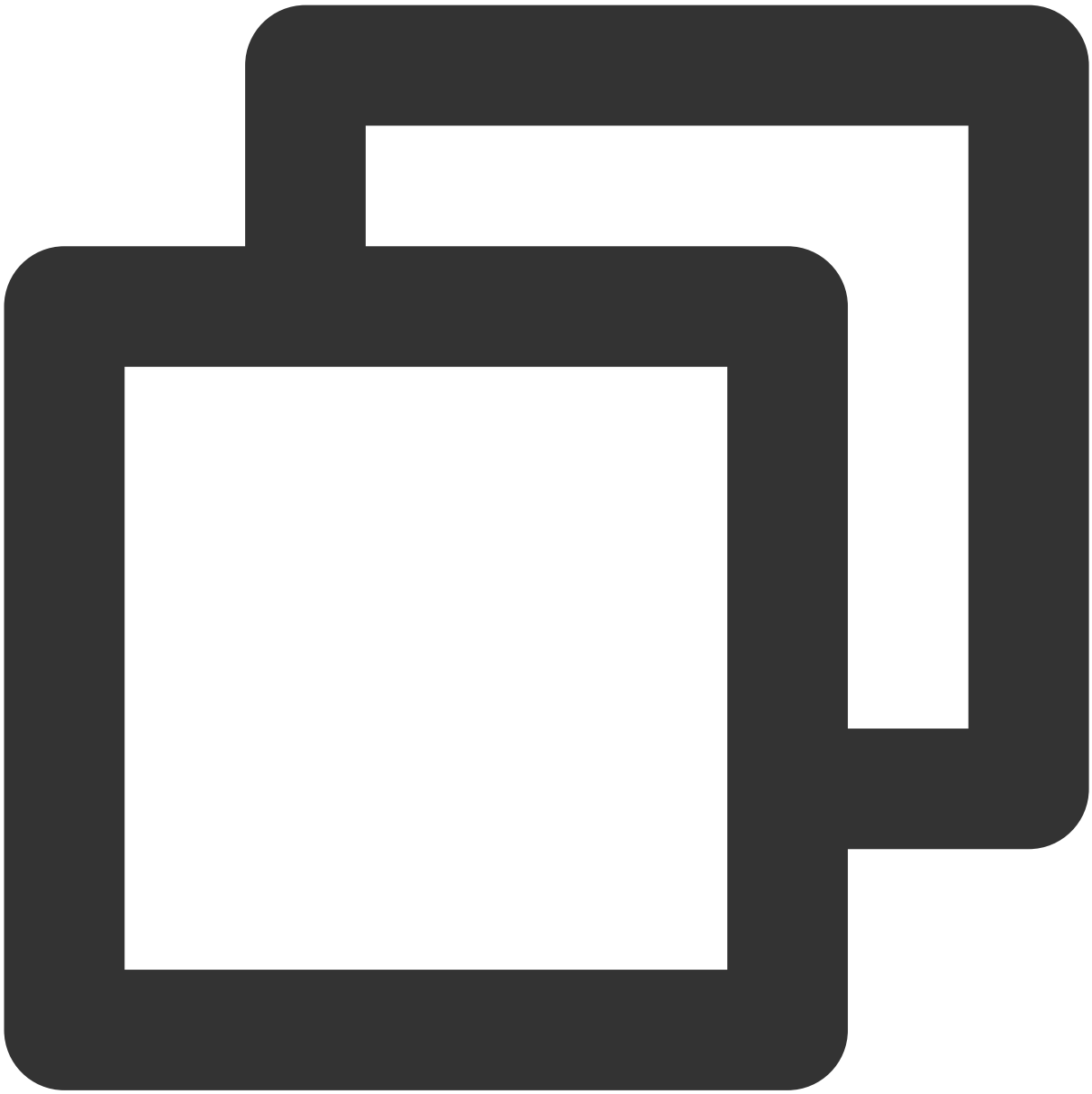
命名

当前 PR 的涉及的 changelog 日志，应该提交在 `.changelog` 目录下，名称为 `{PR-NUMBER}.txt` 。例如，PR 的 Number 是1326，则应该有一个名为 `.changelog/1326.txt` 的变更日志。



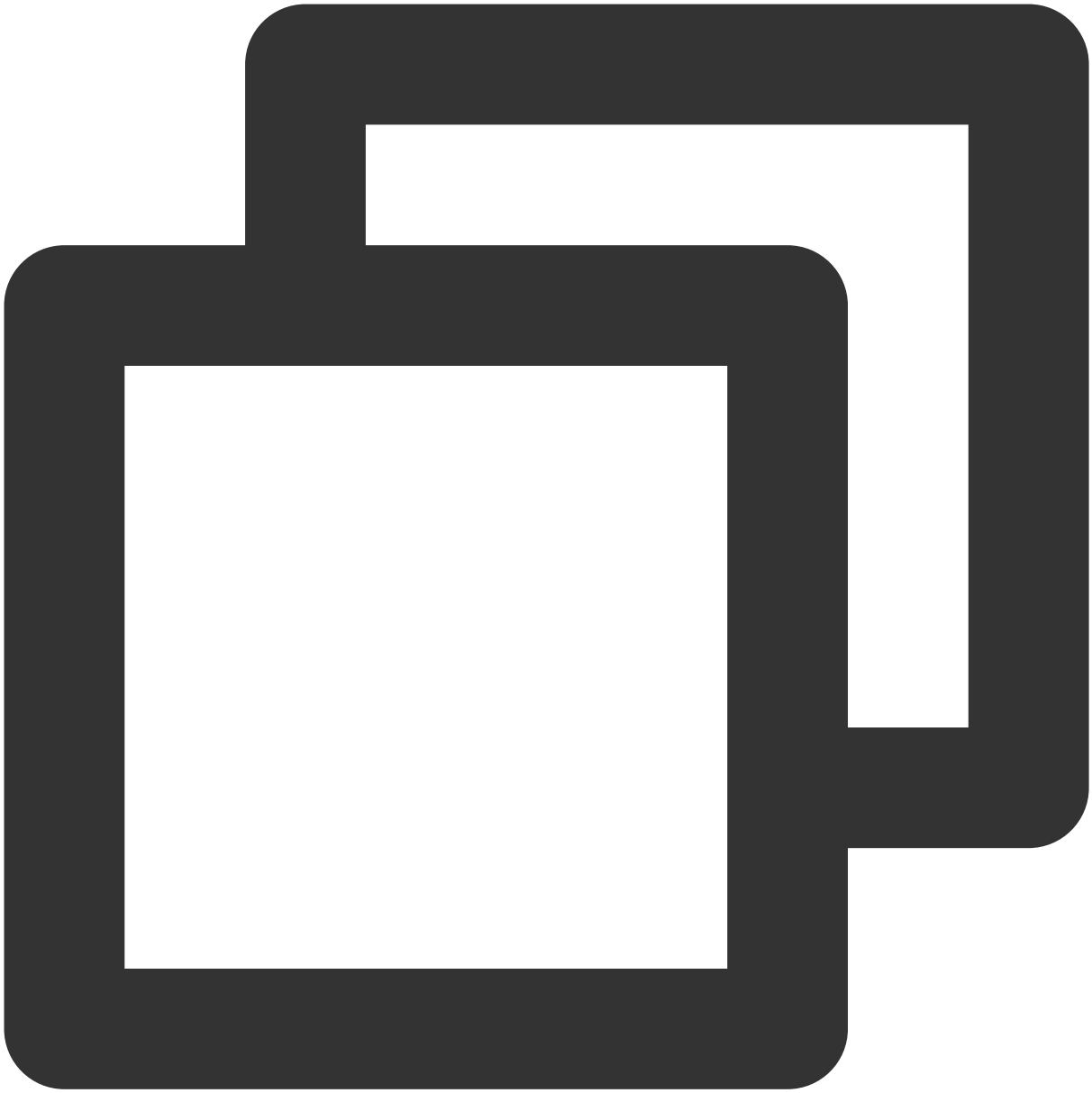
格式

`changelog.txt` 格式如下，其中 `{HEADER}` 为对应 changelog 类别， `{ENTRY}` 为对应 changelog 的详细内容。



```
```release-note:{HEADER}
{ENTRY}
```
```

如果一个拉取请求应该包含多个变更日志条目，那么可以将多个块添加到同一个变更日志文件中。例如：



```release-note:bug
resource/tencentcloud_teo_zone: change tag description from zoneName to zoneId.
```

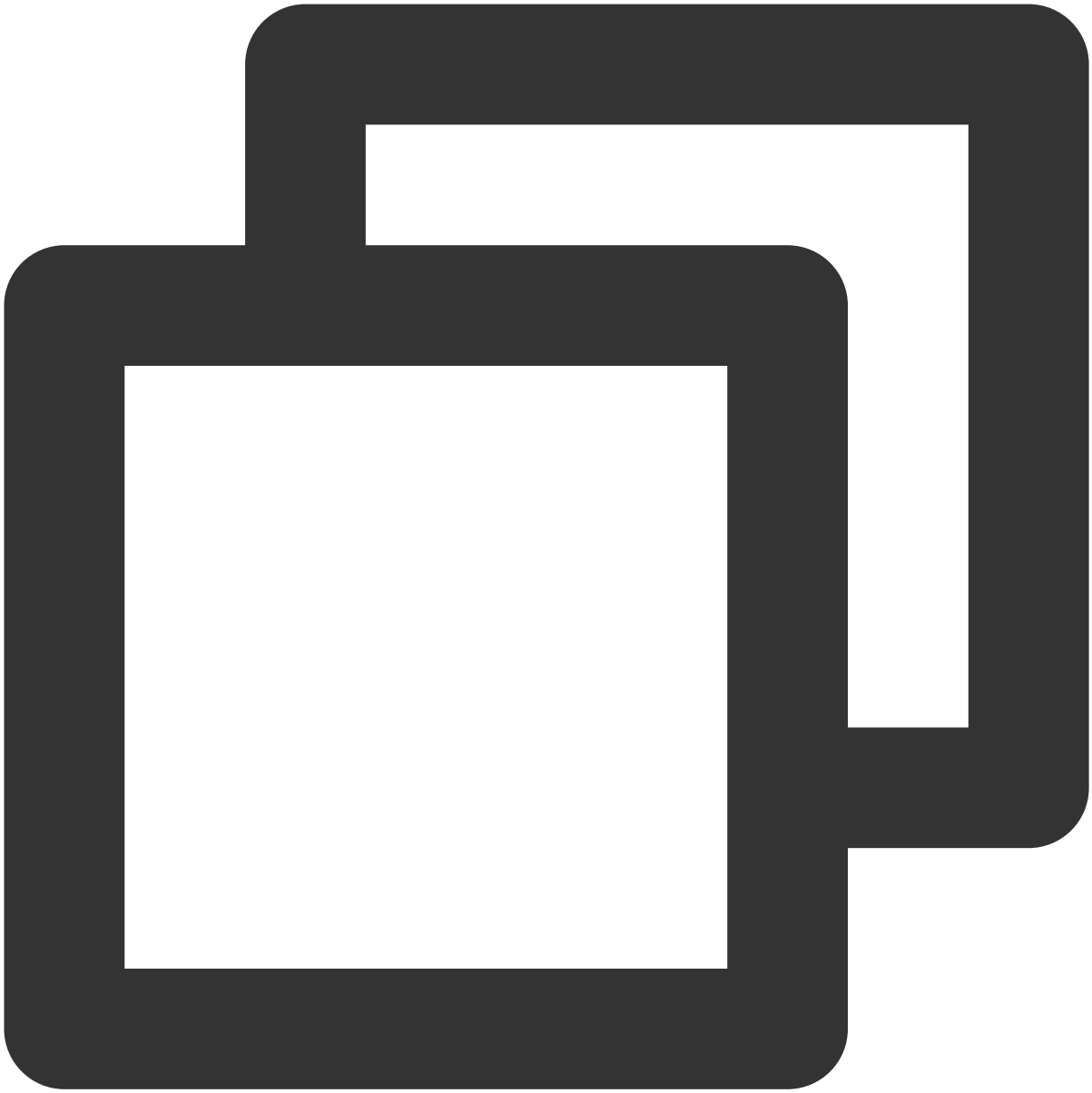
```release-note:enhancement
resource/tencentcloud_redis_instance: support update `security_groups`.
```

变更日志规则分类

CHANGELOG 旨在显示对特定版本的代码库影响操作员的更改。如果对代码的每一次更改或提交都会导致一个条目，那么 CHANGELOG 对操作员的用处就会降低。以下列表是需要做出决定以决定更改是否应具有条目的一般准则和示例。

新资源

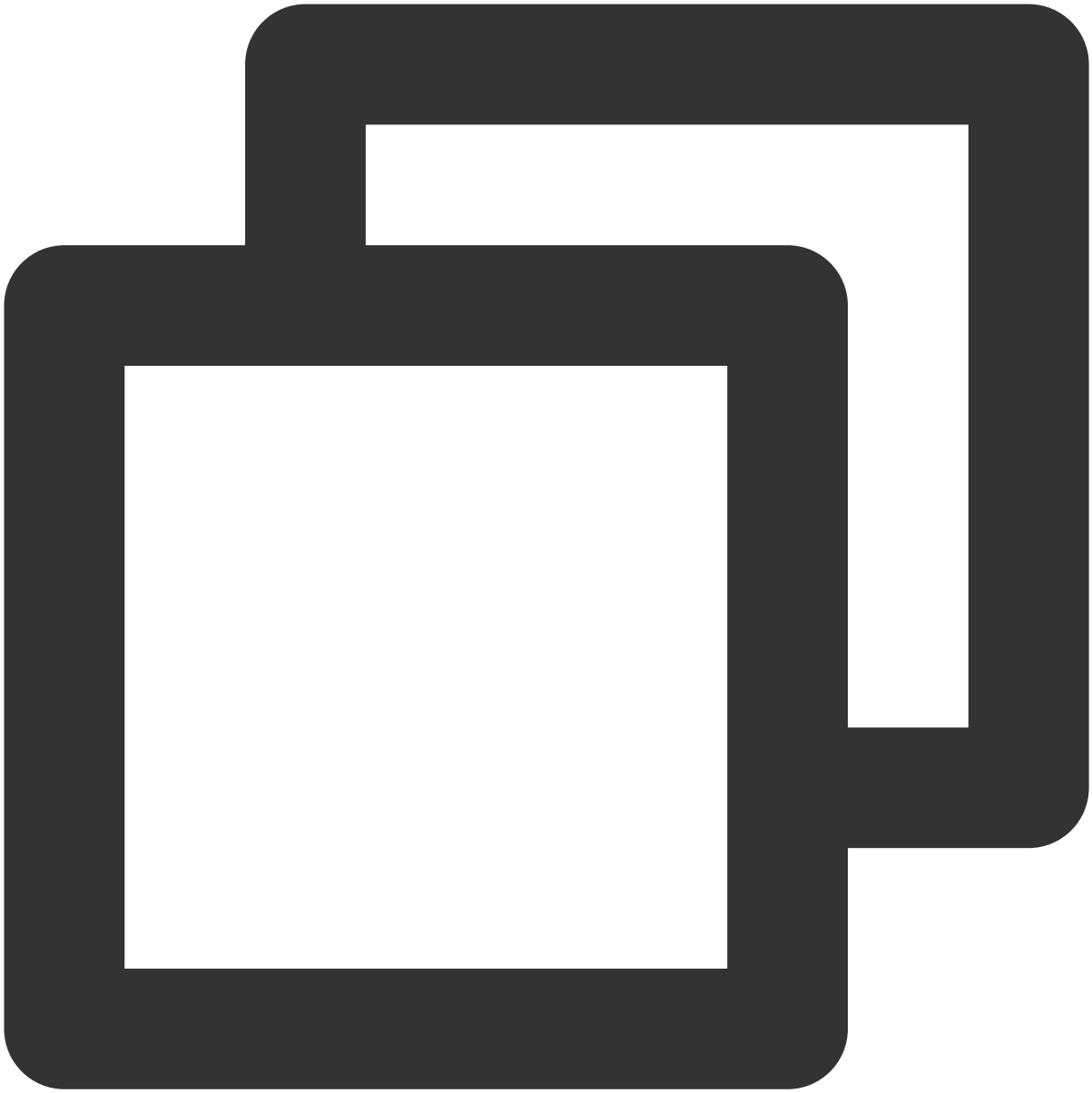
一个新的资源，它的变更日志应该只包含资源的名称，并使用release-note:new-resource标题。



```release-note:new-resource
tencentcloud_postgresql_instance
```

新数据源

一个新的数据源，它的变更应该只包含数据源的名称，并使用release-note:new-data-source标题。



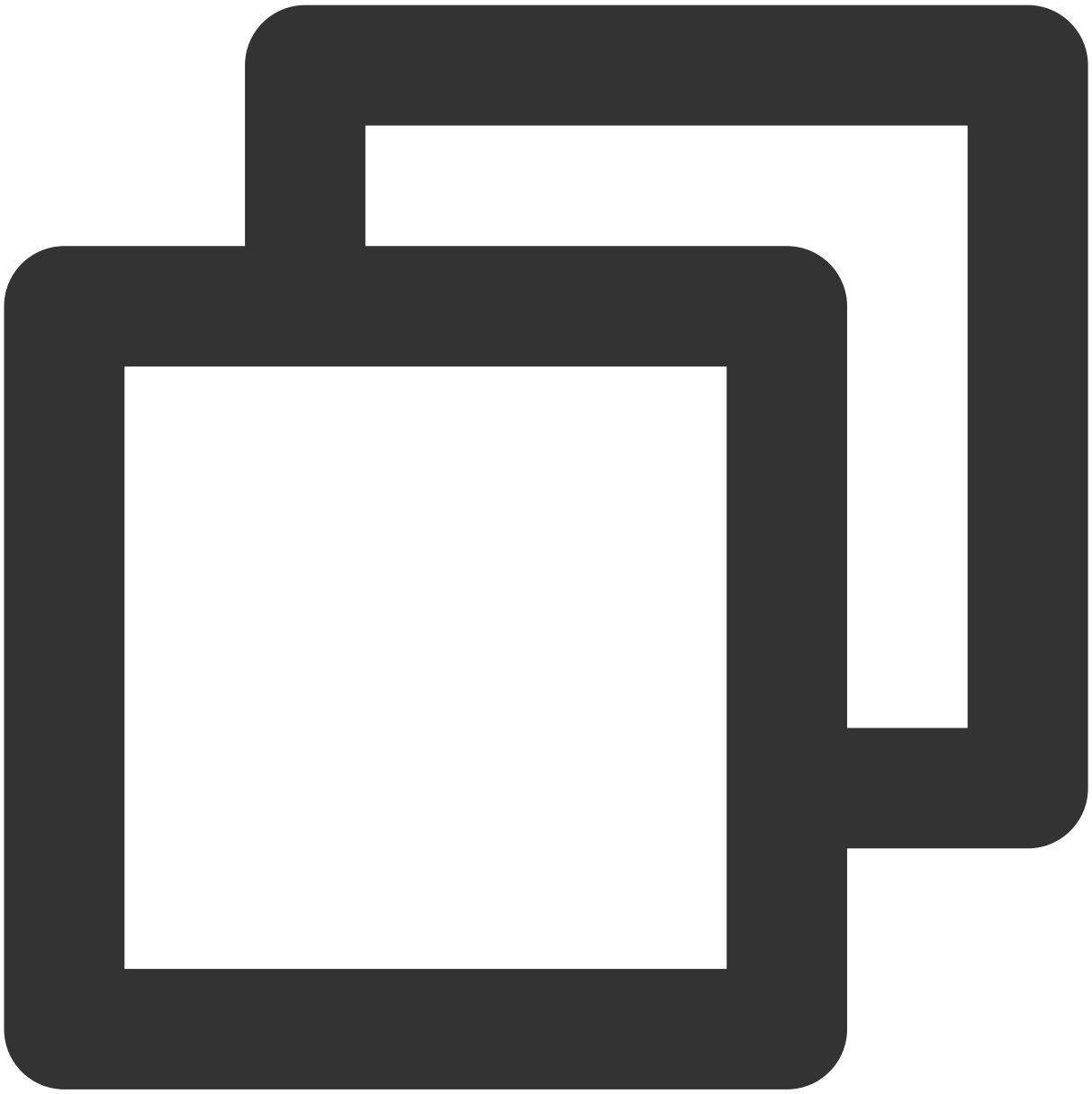
```release-note:new-data-source
tencentcloud_sqlserver_zone_config
```

问题修复

一个新的问题修复，它的变更应该使用release-note:bug标题，并有一个前缀指示它对应的资源或数据源，一个冒号，然后是一个简短的摘要。

说明：

如果是Provider级别的修复，使用provider前缀



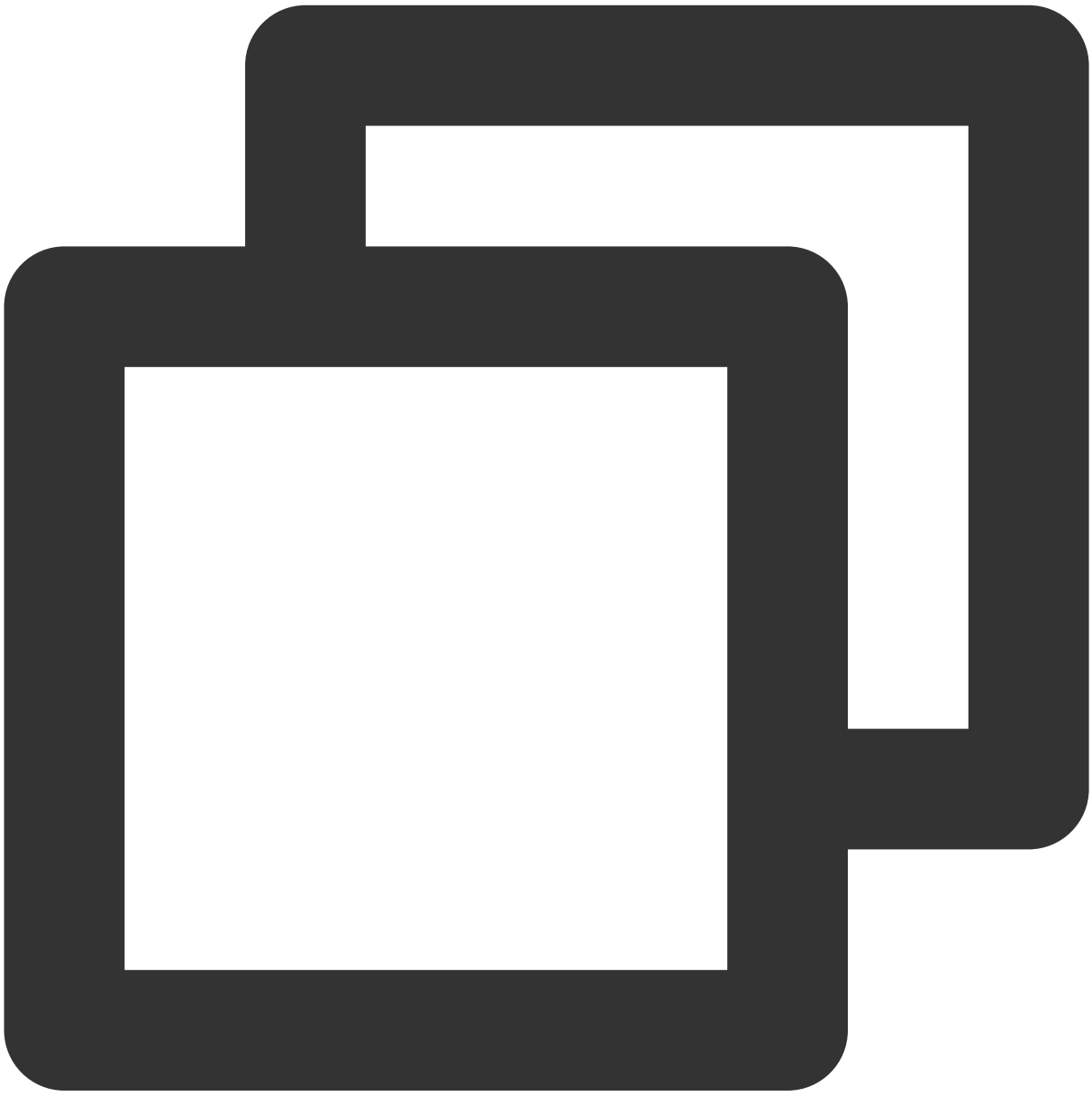
```release-note:bug
resource/tencentcloud_cdn_domain: Fix `https_config` inconsistency after apply
```

功能增强

一个新的功能增强，它的变更应该使用release-note:enhancement标题，并有一个前缀指示它对应的资源或数据源，一个冒号，然后是一个简短的摘要。

说明：

如果是Provider级别的功能增强，使用provider前缀



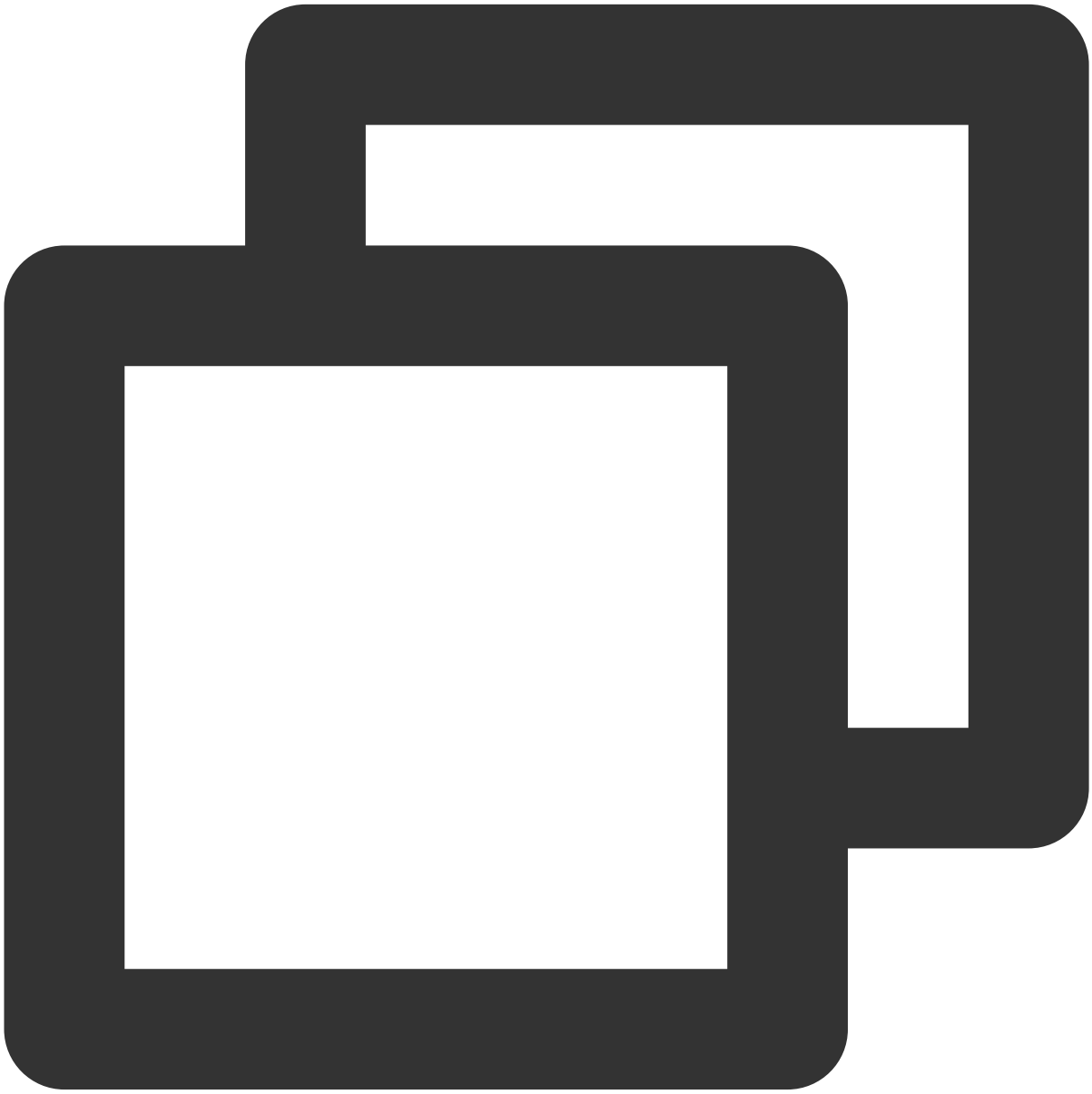
```
```release-note:enhancement
resource/tencentcloud_cdn_domain: Support follow redirect and authentication
```
```

功能弃用

一个功能弃用，它的变更应该使用release-note:deprecation标题，并有一个前缀指示它对应的资源或数据源，一个冒号，然后是一个简短的摘要。

说明：

如果是 Provider 级别的功能弃用，需要使用 provider 前缀。



```
```release-note:deprecation
resource/tencentcloud_kubernetes_cluster: The `as_enabled` attribute is being depre
```
```

不需要提交变更日志的情况

资源和提供者文档更新

测试更新

代码重构

Module 发布

最近更新时间：2023-03-07 10:35:48

操作场景

Module 是 Terraform 组合多种资源的配置形态。在部分多资源场景下，使用 Module 能够更好地抽象业务，减少配置成本。您可将 Github 中的 Modules 发布到 terraform 仓库。本文介绍如何创建及发布 Terraform TencentCloud Module。

操作步骤

创建公共 Module

在 GitHub 中新建代码仓库。

- 命名格式为 `terraform-<provider>-<name>`，例如 terraform-tencentcloud-vpc。
- 一个基本的 Module 需包含以下文件：

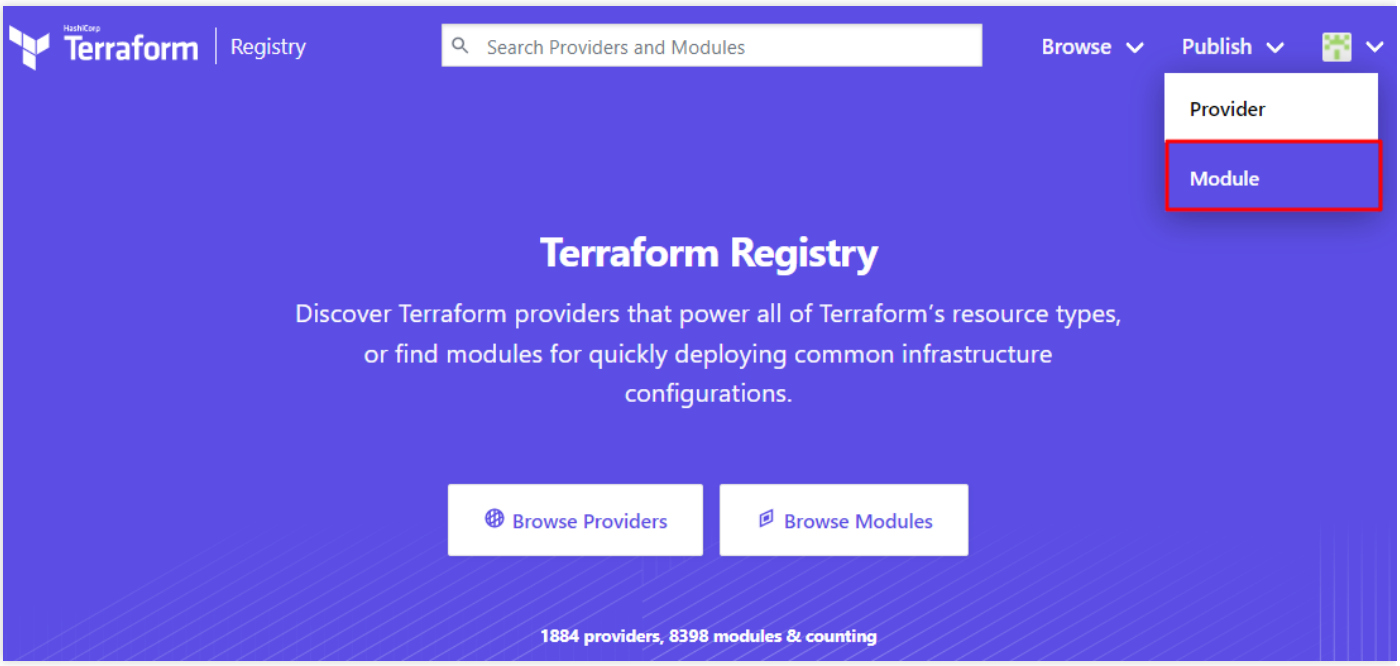
```
.
├── main.tf # 编写模块资源
├── variables.tf # 声明模块变量
├── outputs.tf # 声明模块输出
├── LICENCE # 声明许可
└── README.md # 自述文件
```

说明：

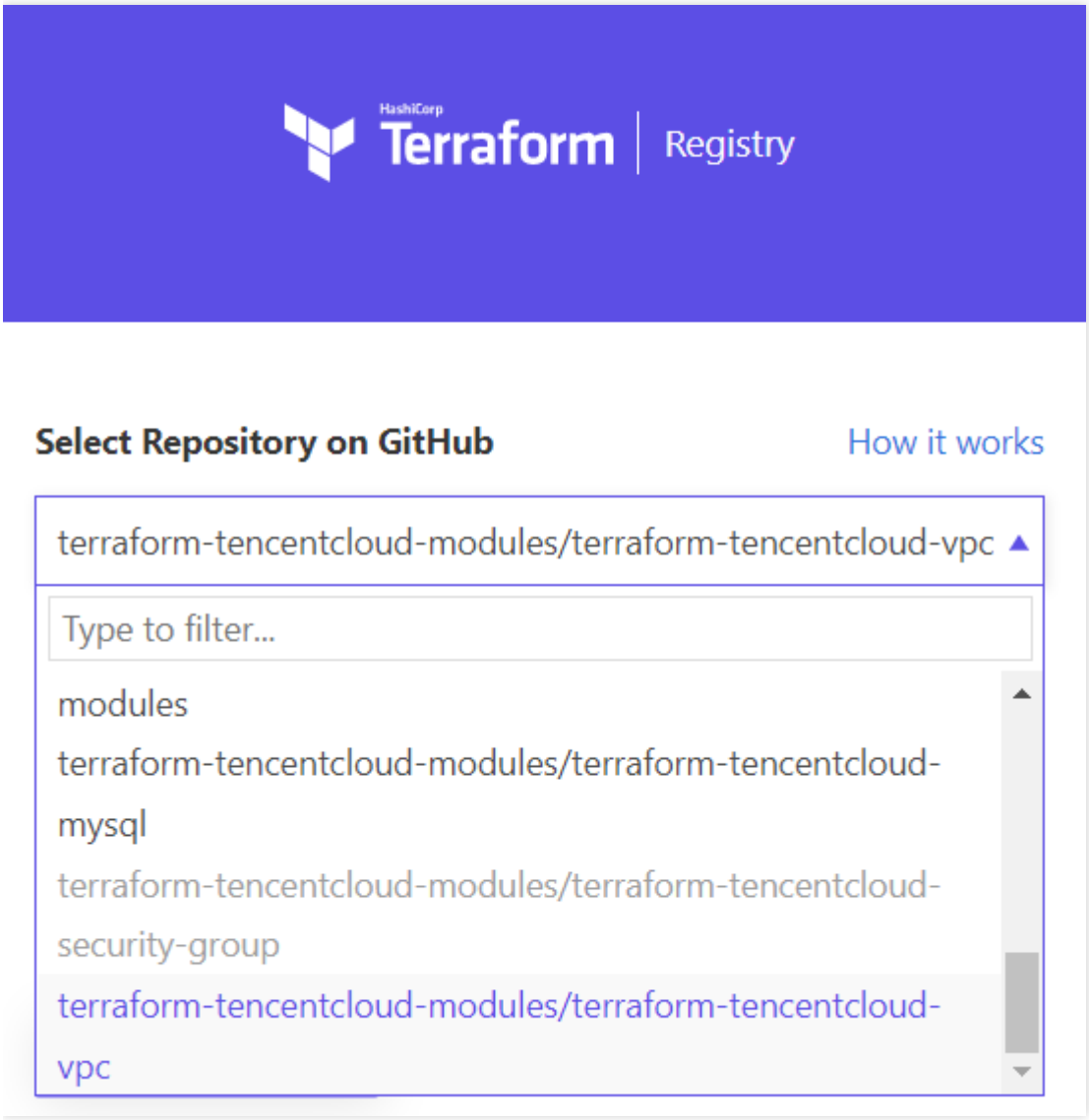
建议添加 example 目录，存放该模块引入和使用示例。您可参考 examples 进行添加。

发布 Module

1. 登录 registry.terraform.io ，选择页面右上角的 **Publish**，并在下拉列表中单击 **Module**。如下图所示：



2. 在页面中展开 “Select Repository on GitHub” 下拉列表，可在列表中查看个人账户下有管理权限的 Modules 仓库，选择需发布的 Module。如下图所示：

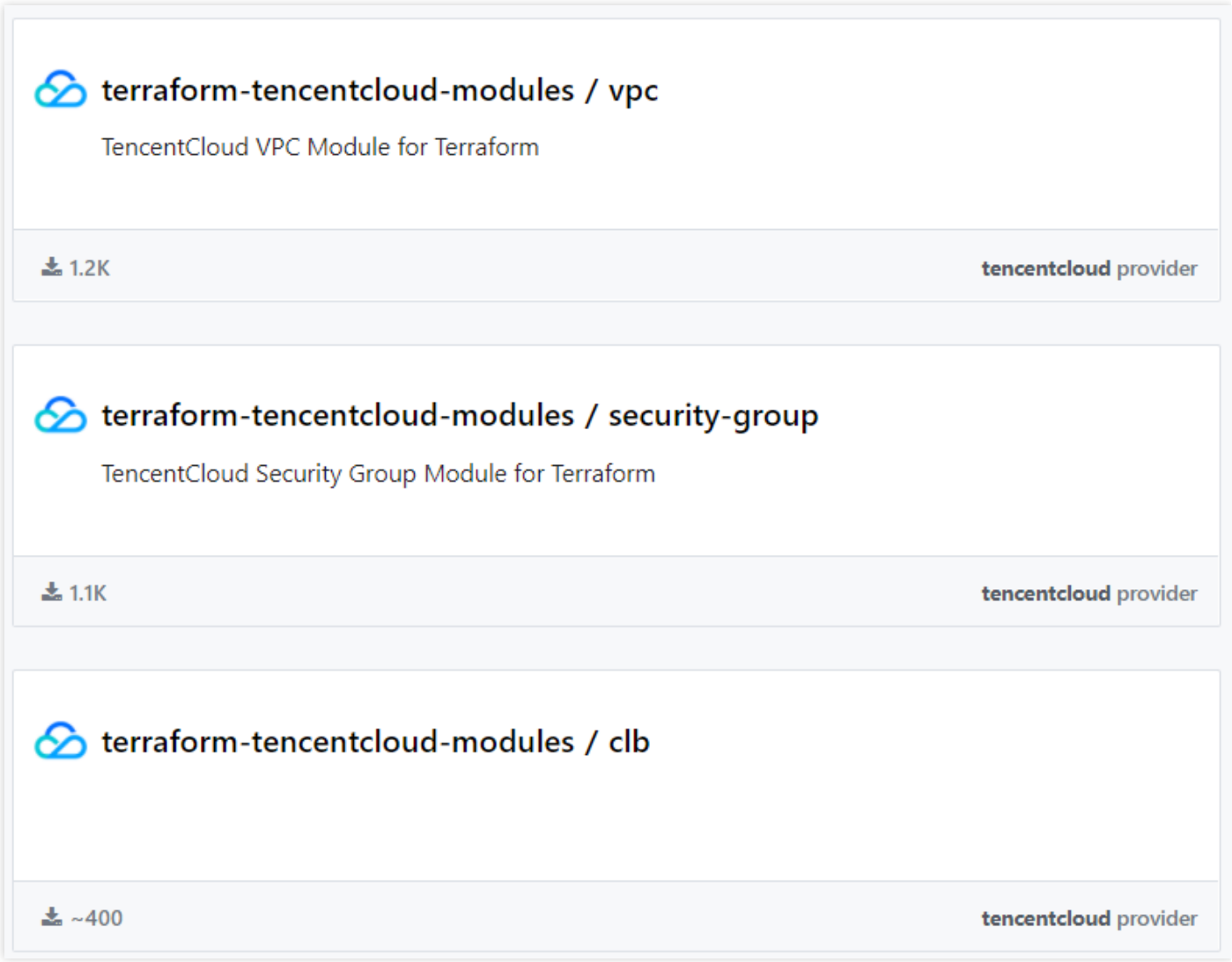


注意：

Module 可以使用个人 GitHub 仓库发布。若仓库名称符合 `terraform-tencentcloud-<name>`，则该 Modules 也会收录在 tencentcloud 的 Modules 中。

3. 勾选 “ I agree to the Terms of Use.”后，单击 **PUBLISH MODULE**。

4. 该仓库将会在几分钟后，自动同步到 terraform registry 中。如下图所示：



添加仓库合并检查（可选）

若您的 Module 涉及多人协作，则可以借助 GitHub Action 对请求合并的代码做初步检查。

本文以 terraform-tencentcloud-vpc 为例，在仓库根目录下新建 `.github/workflow` 目录，创建 `pull-request.yml` 文件。示例代码如下：

```
name: MR_CHECK

on:
pull_request:
branches: [ master ]
workflow_dispatch:

jobs:
```

```
build:
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v2
- uses: hashicorp/setup-terraform@v1
- name: Module Files Checking
run: |
files=(
LICENSE
main.tf
version.tf
variables.tf
outputs.tf
README.md
)

test -d examples || echo "[WARN] Missing ./examples in modules directory, we stro
ngly recommend you to provide example usage of this module."

for i in ${files[@]} ; do
fileCount=$(find ./ -name $i | wc -l)
if [[ $fileCount -gt 0 ]]; then
echo "[INFO] File: $i exist."
else
echo "[ERROR] Missing $i, a recommend module should include these files:\n ${file
s[@]}"
exit -1
fi
done
- name: Terraform Validate
run: |
terraform init
terraform validate

- name: Terraform Format Check
run: |
terraform fmt -diff -check -recursive
```

说明如下：

- `Module Files Checking`：检查该目录下是否包含上文中需要的文件。
- `Terraform Validate`：进行 Module 参数检查。

- `Terraform Format Check` ：校验 Module 中的 tf 代码格式。

开发者参考
实现原理

最近更新时间：2023-03-07 10:35:48

本文将介绍 Terraform Tencent Cloud Provider 的目录结构。

目录结构

```
├─terraform-provider-tencentcloud 根目录
| ├─main.go 程序入口文件
| ├─AUTHORS 作者信息
| ├─CHANGELOG.md 变更日志
| ├─LICENSE 授权信息
| ├─debug.tf.example 调试配置文件示例
| ├─examples 示例配置文件目录
| | ├─tencentcloud-eip EIP示例tf文件
| | ├─tencentcloud-instance CVM示例tf文件
| | ├─tencentcloud-nat NAT网关示例tf文件
| | ├─tencentcloud-vpc VPC示例tf文件
| | └─ ... 更多examples目录
| ├─tencentcloud Provider核心目录
| | ├─basic_test.go 基础单元测试
| | ├─config.go 公共配置文件
| | ├─data_source_tc_availability_zones.go 可用区查询
| | ├─data_source_tc_availability_zones_test.go
| | ├─data_source_tc_nats.go NAT网关列表查询
| | ├─data_source_tc_nats_test.go
| | ├─data_source_tc_vpc.go VPC查询
| | ├─data_source_tc_vpc_test.go
| | ├─... 更多Data Source
| | ├─helper.go 一些公共函数
| | ├─provider.go Provider核心文件
| | ├─provider_test.go
| | ├─resource_tc_eip.go EIP资源管理程序
| | ├─resource_tc_eip_test.go
| | ├─resource_tc_instance.go CVM实例资源管理程序
| | ├─resource_tc_instance_test.go
| | ├─resource_tc_nat_gateway.go NAT网关资源管理程序
| | ├─resource_tc_nat_gateway_test.go
| | ├─resource_tc_vpc.go VPC网关资源管理程序
| | ├─resource_tc_vpc_test.go
```

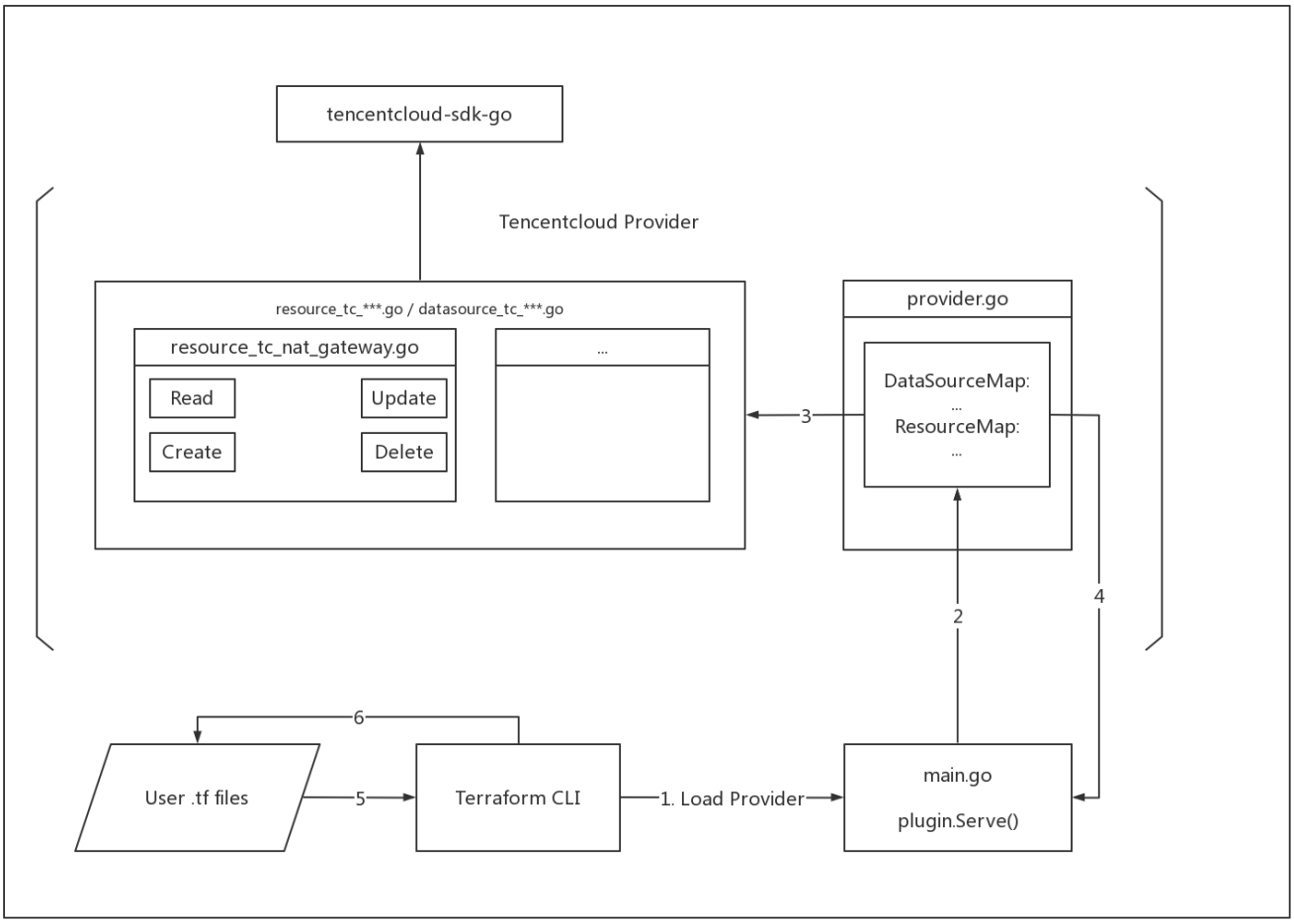
```
| | ├... 更多资源管理程序
| | ├service_eip.go 封装的EIP相关Service
| | ├service_instance.go 封装的CVM实例相关Service
| | ├service_vpc.go 封装的VPC相关Service
| | ├...
| | ├validators.go 公共的参数校验函数
| ├vendor 依赖的第三方库
| ├website Web相关文件
| | ├tencentcloud.erb 文档左侧菜单栏
| | ├docs 文档markdown源文件目录
| | | ├d data相关文档（data_source_*）
| | | | ├availability_zones.html.md
| | | | ├nats.html.markdown
| | | | ├vpc.html.markdown
| | | | ├...
| | | ├index.html.markdown
| | | ├r resource相关文档(resource_*)
| | | | ├instance.html.markdown
| | | | ├nat_gateway.html.markdown
| | | | ├vpc.html.markdown
| | | | └...
```

结构主要分五部分：

- **main.go**：插件入口。
- **examples**：示例目录，其中包含的示例可直接使用。
- **tencentcloud**：插件目录，存放的业务代码。其中：
  - provider.go：插件的根源，用于描述插件的属性。例如，配置的密钥、支持的资源列表及回调配置等。
  - data_source_*.go：定义一些用于读调用的资源，主要是查询接口。
  - resource_*.go：定义一些写调用的资源，包含资源增删改查接口。
  - service_*.go：按资源大类划分的一些公共方法。
- **vendor**：依赖的第三方库。
- **website**：文档，重要性同 examples。

Provider 生命周期

Terraform 执行过程如下图所示：



说明：

1 - 4：寻找 Provider，此时加载 tencentcloud 插件。

5：读取用户的配置文件，通过配置文件，可以获得分别属于哪种资源，以及每个资源的状态。

6：根据资源的状态，调用不同的函数（Create/Update/Delete/Update）。

- Create

当在 .tf 文件增加一个新的资源配置时，Terraform 判断为 Create。

- Update

当修改 .tf 文件中已经创建好的资源一个或多个参数时，Terraform 判断为 Update。

- Delete

当把 .tf 文件中已经创建好的资源配置删掉后，或执行 terraform destroy 命令时，Terraform 判断为 Delete。

- Read

Read 是一个查询资源的操作，实际作用就是检查资源是否存在，以及更新资源属性到本地。

- tencentcloud-sdk-go

tencentcloud-sdk-go 是基于 Tencent Cloud API 的 Go 版 SDK，其作用为调用 Tencent Cloud API 来实现资源管理。

开发与调试

最近更新时间：2023-05-29 15:53:34

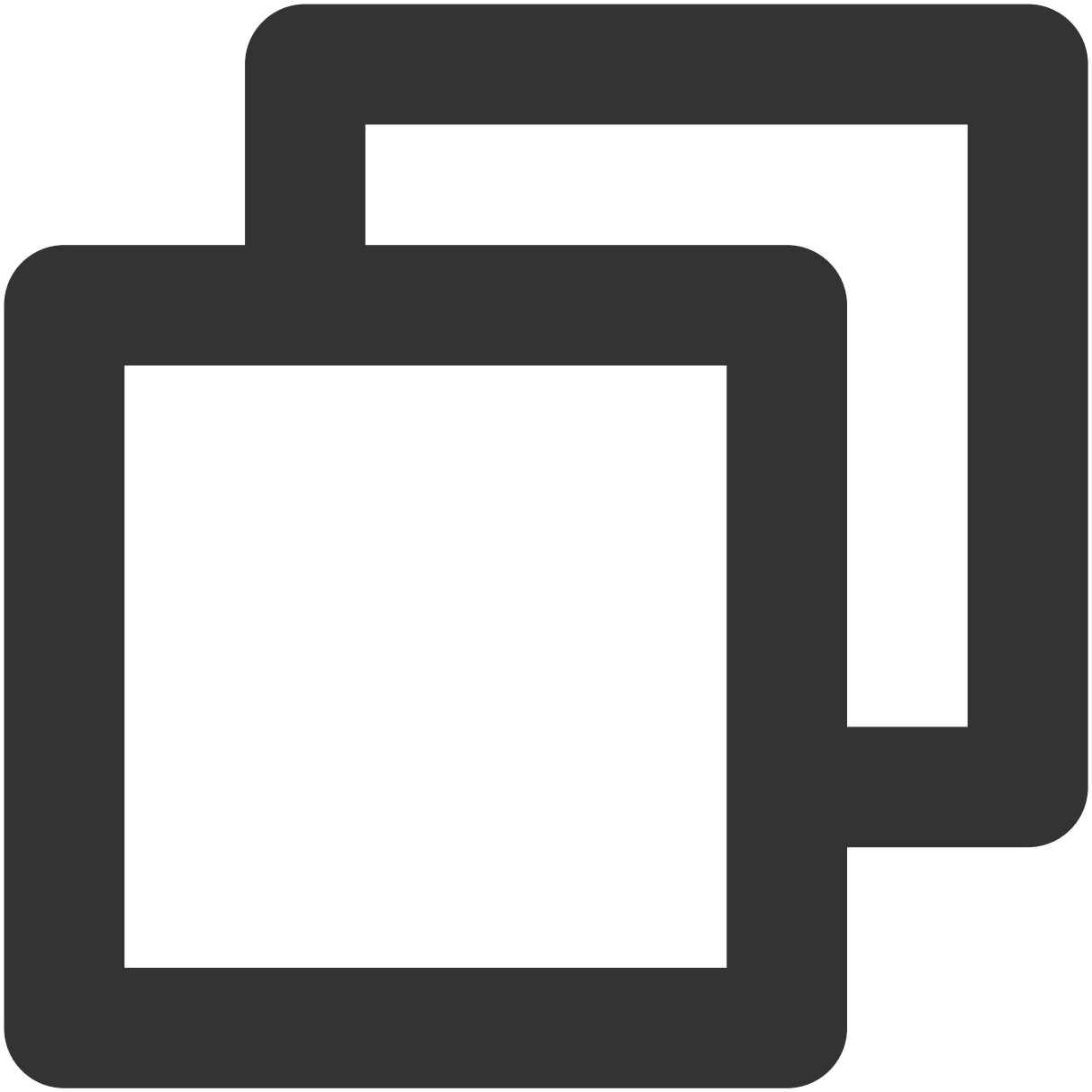
本文介绍如何在本地进行基本的 Terraform 开发与调试工作。

步骤1：安装 Terraform

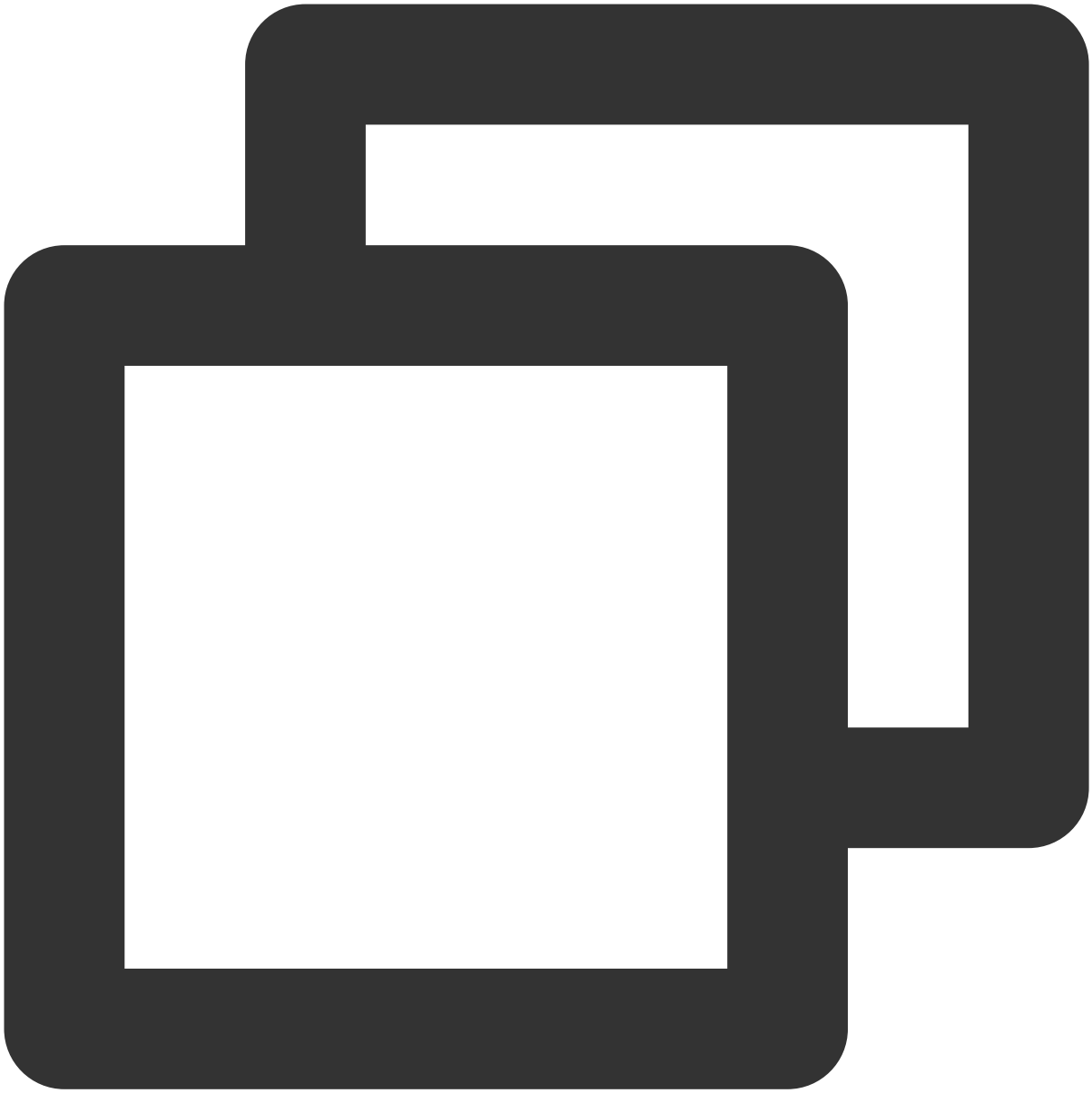
参考 安装 Terraform，完成 Terraform 安装及全局路径配置。

步骤2：Provider 拉取

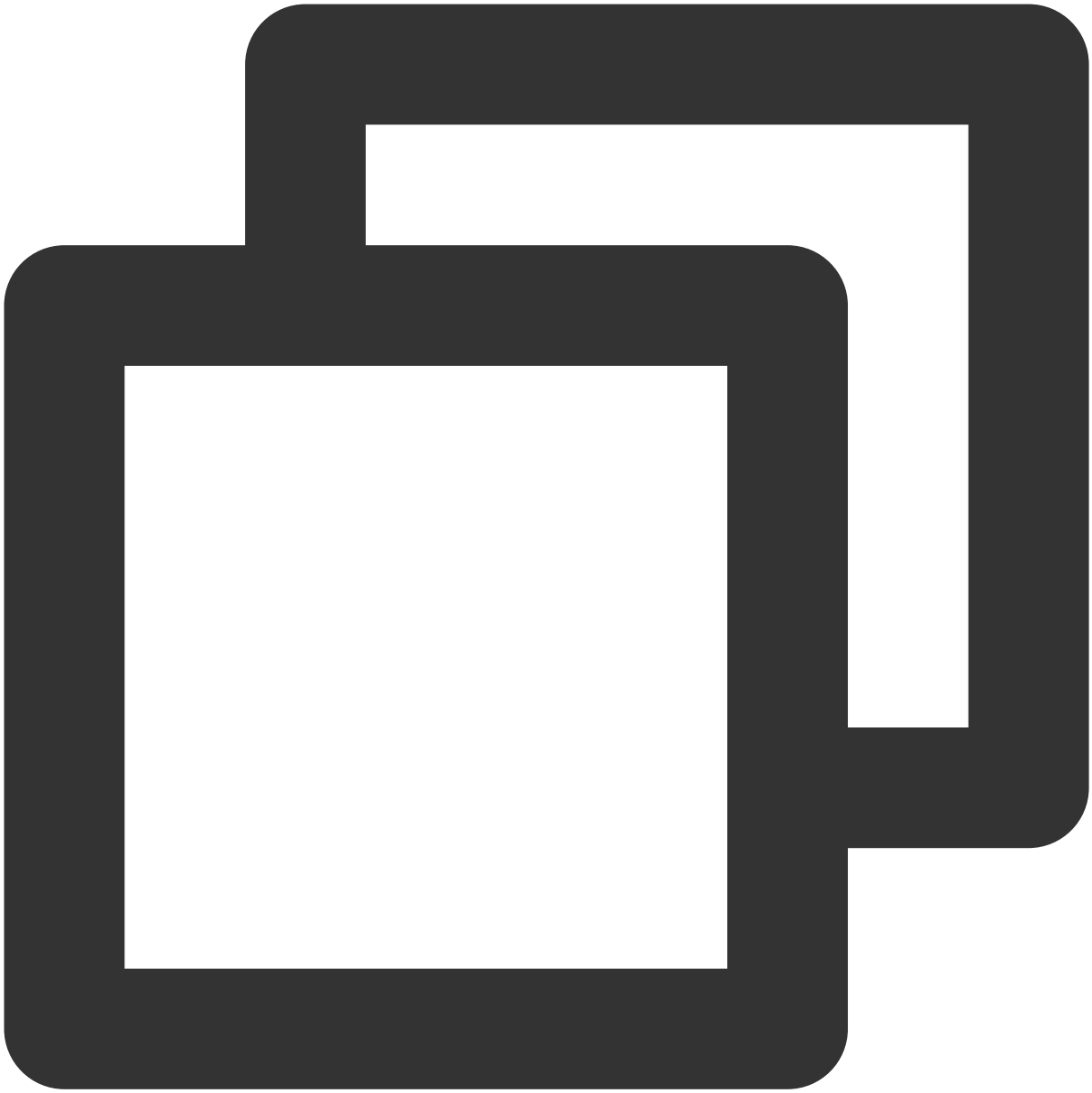
1. 前往 terraform-provider-tencentcloud，将 provider 代码 fork 到个人仓库。
2. 依次执行以下命令，本地拉取并设置上游远程仓库。



```
$ git clone https://github.com/{您的用户名}/terraform-provider-tencentcloud # 这里的代
```

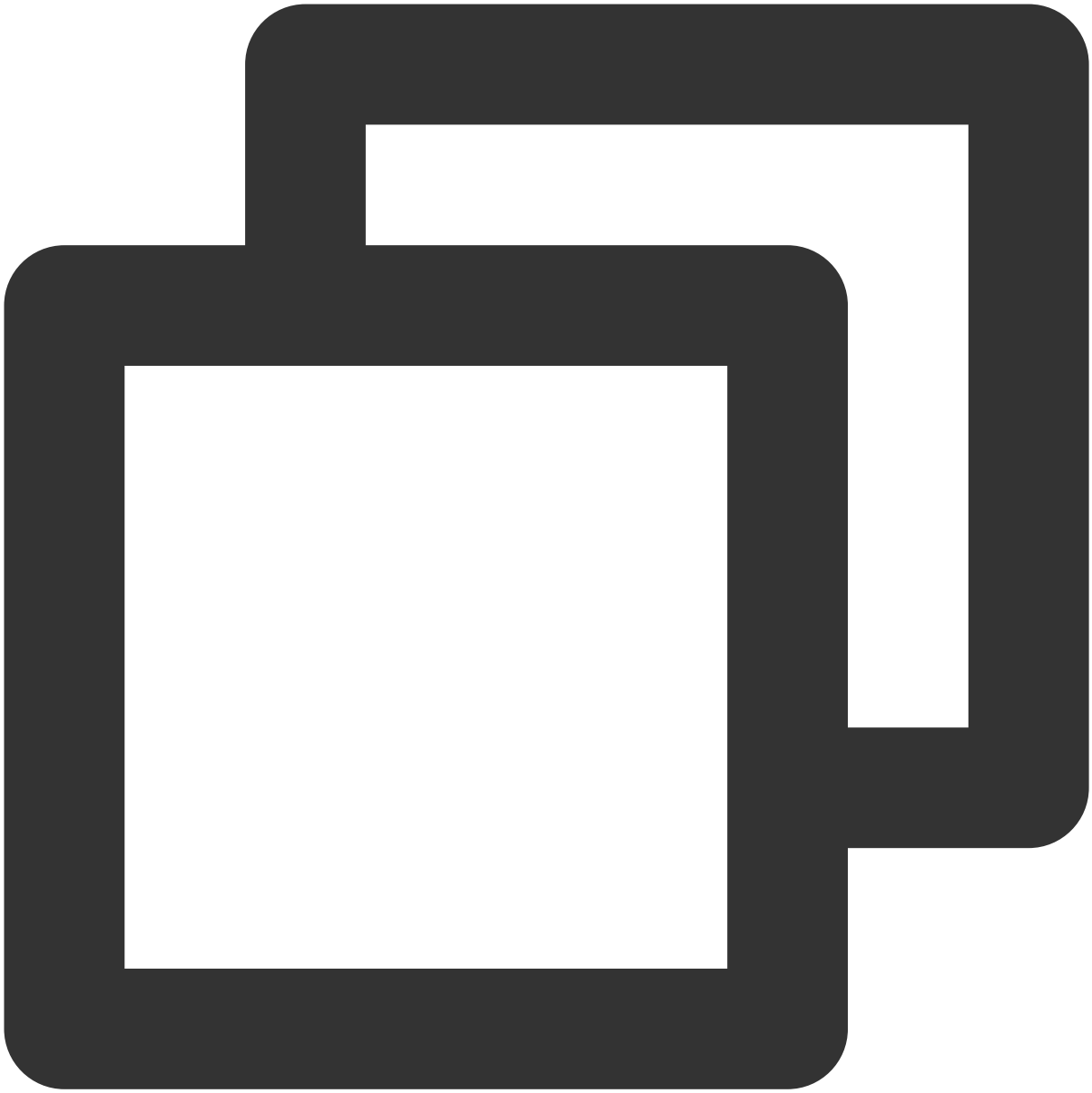


```
$ cd terraform-provider-tencentcloud
```



```
$ git remote add upstream https://github.com/tencentcloudstack/terraform-provider-t
```

拉取成功后，可查看代码结构如下：

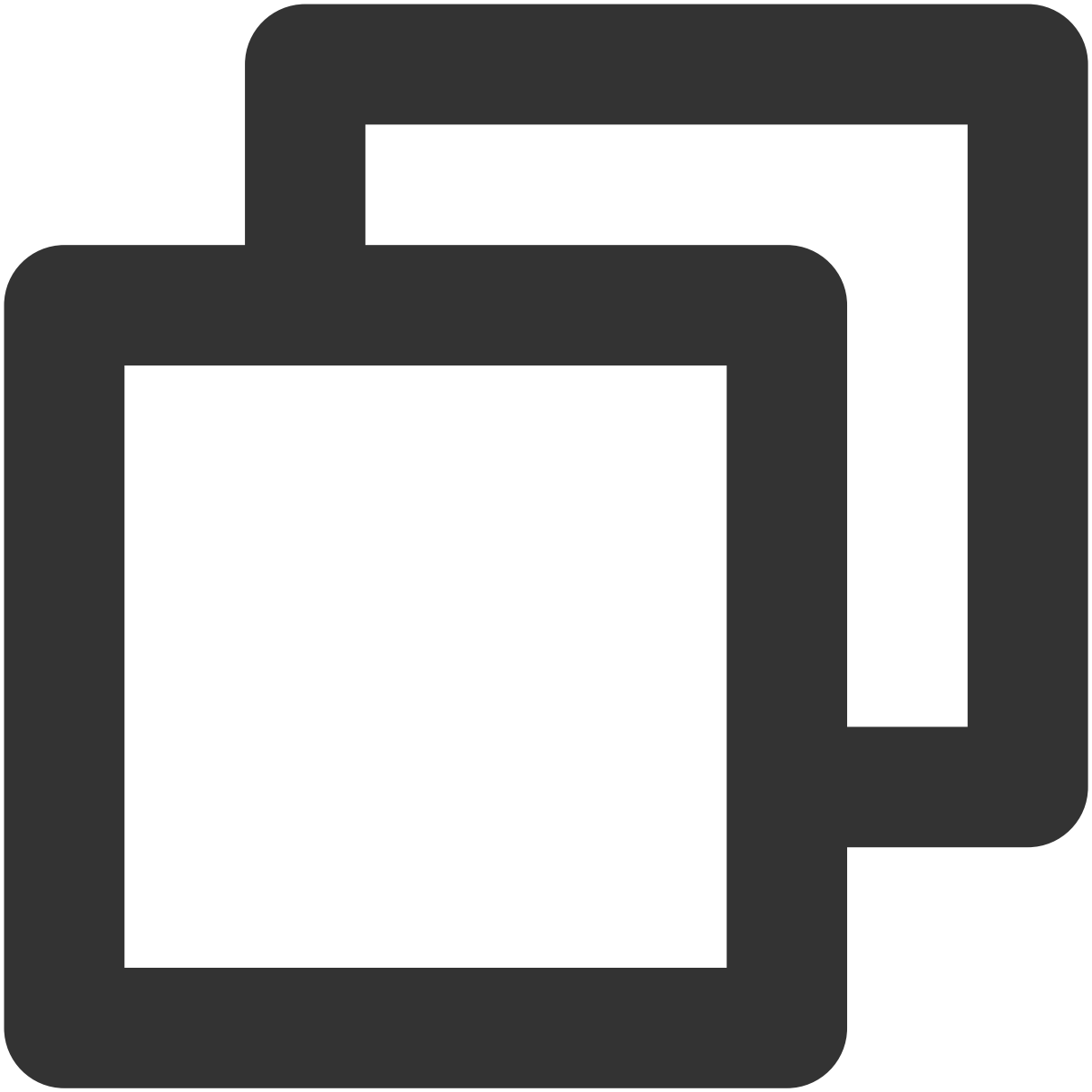


```
.
├── .githooks/
├── .github/
├── examples/ # 示例代码，原则上请确保产出的代码用户可以直接复制粘贴使用
├── gendoc/ # 文档生成工具
├── scripts/
├── tencentcloud/ # 产品逻辑
├── vendor/ # 本地依赖缓存
├── website/ # 生成的文档目录
├── .gitignore
├── .go-version
```

```
├── .golangci.yml
├── .goreleaser.yml
├── .travis.yml
├── AUTHORS
├── CHANGELOG.md
├── GNUmakefile
├── LICENSE
├── README.md
├── go.mod
├── go.sum
├── main.go
├── staticcheck.conf
└── tools.go
```

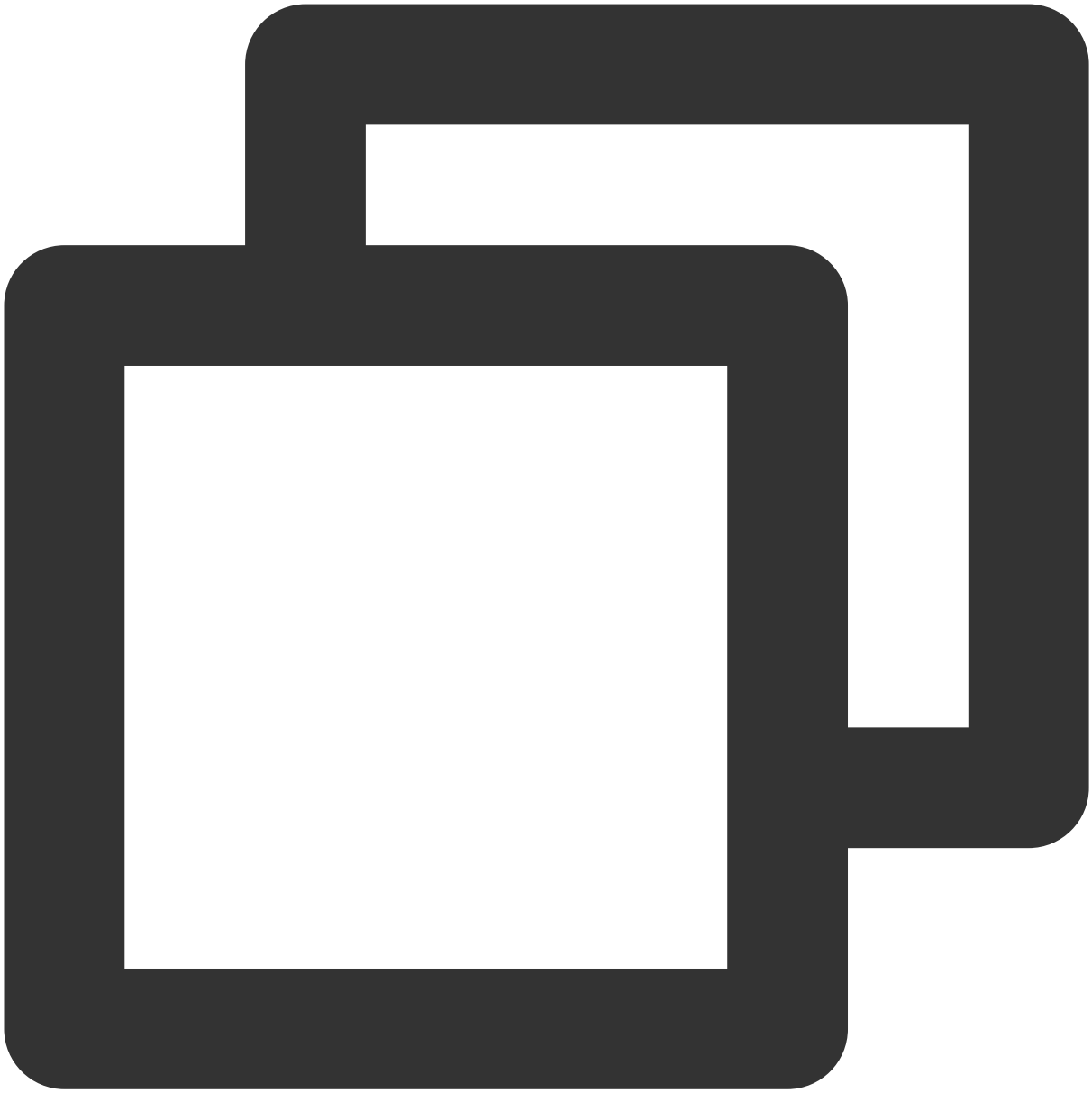
步骤3：本地调试

1. 在项目根目录执行以下命令，构建二进制文件 `terraform-provider-tencentcloud` 。



```
go build
```

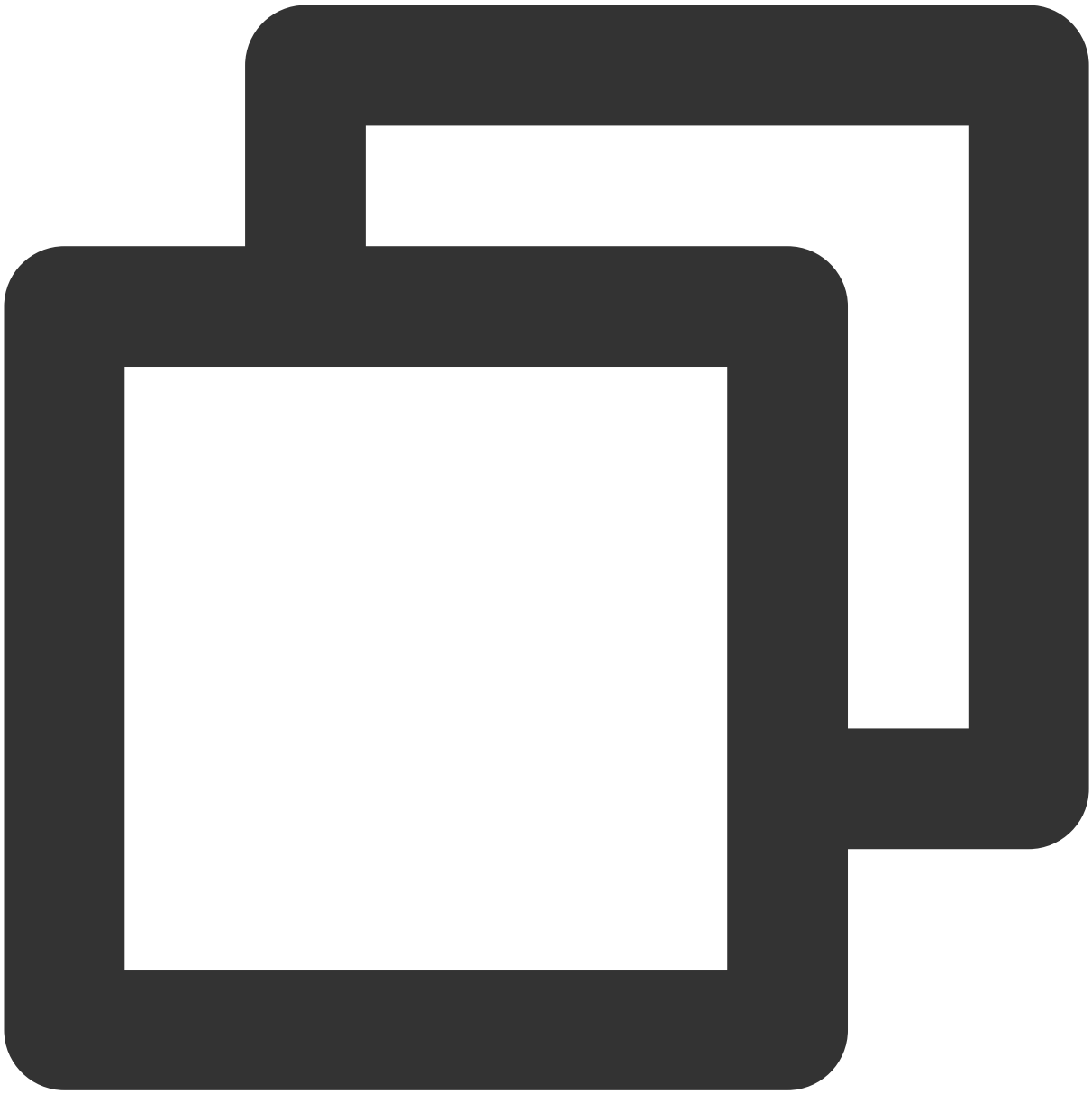
2. 创建 `dev.tfrc` 文件，并写入以下内容，设置 `tencentcloudstack/tencentcloud` 指向二进制文件的位置。



```
provider_installation {
 # Use /home/developer/tmp/terraform-null as an overridden package directory
 # for the hashicorp/null provider. This disables the version and checksum
 # verifications for this provider and forces Terraform to look for the
 # null provider plugin in the given directory.
 dev_overrides {
     "tencentcloudstack/tencentcloud" = "path/to/your/provider/terraform-provider-t
 }
}
```

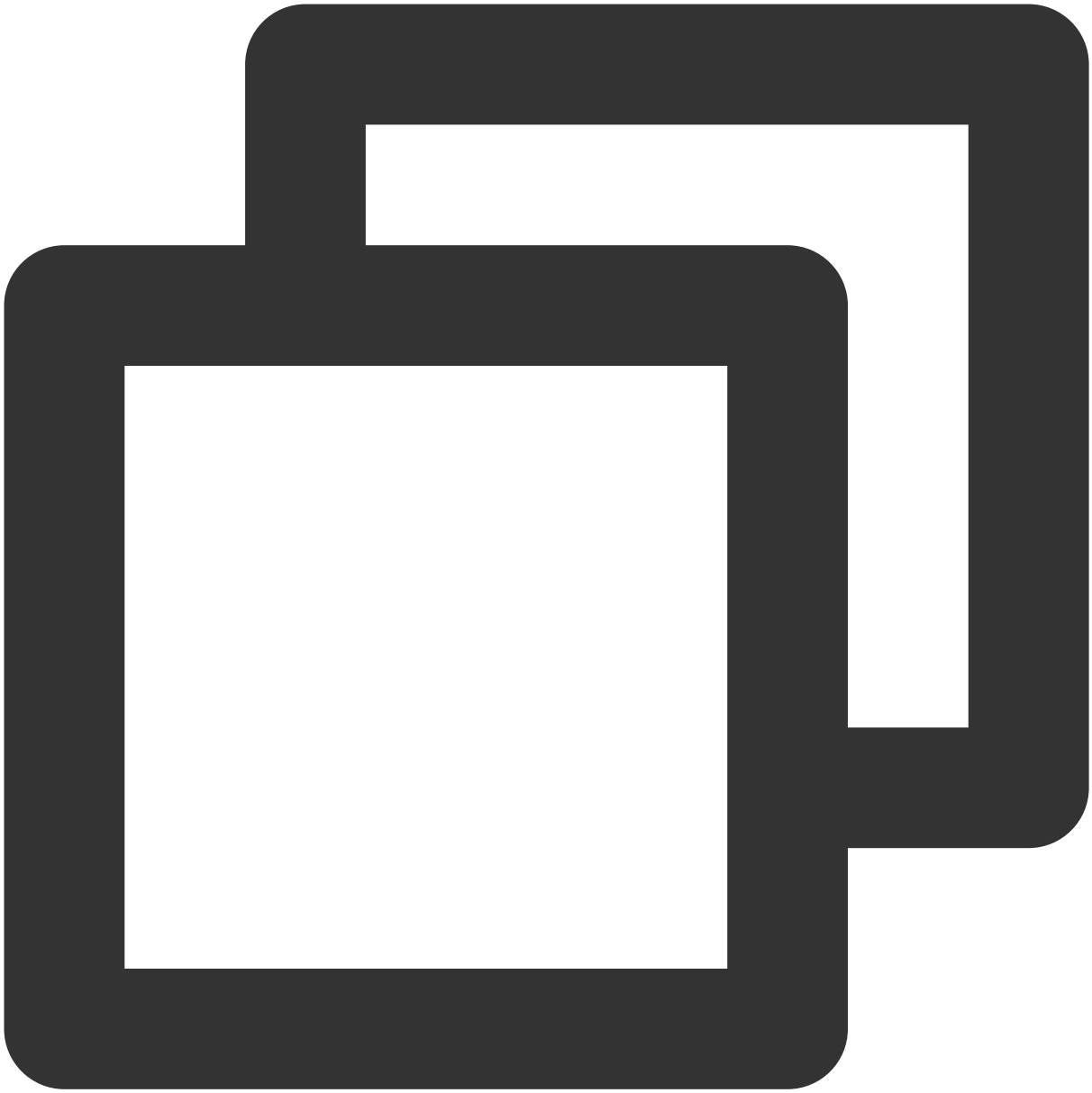
3. 设置以下环境变量。

设置 `TF_CLI_CONFIG_FILE` 环境变量，指向 `dev.tfrc` 所在位置。



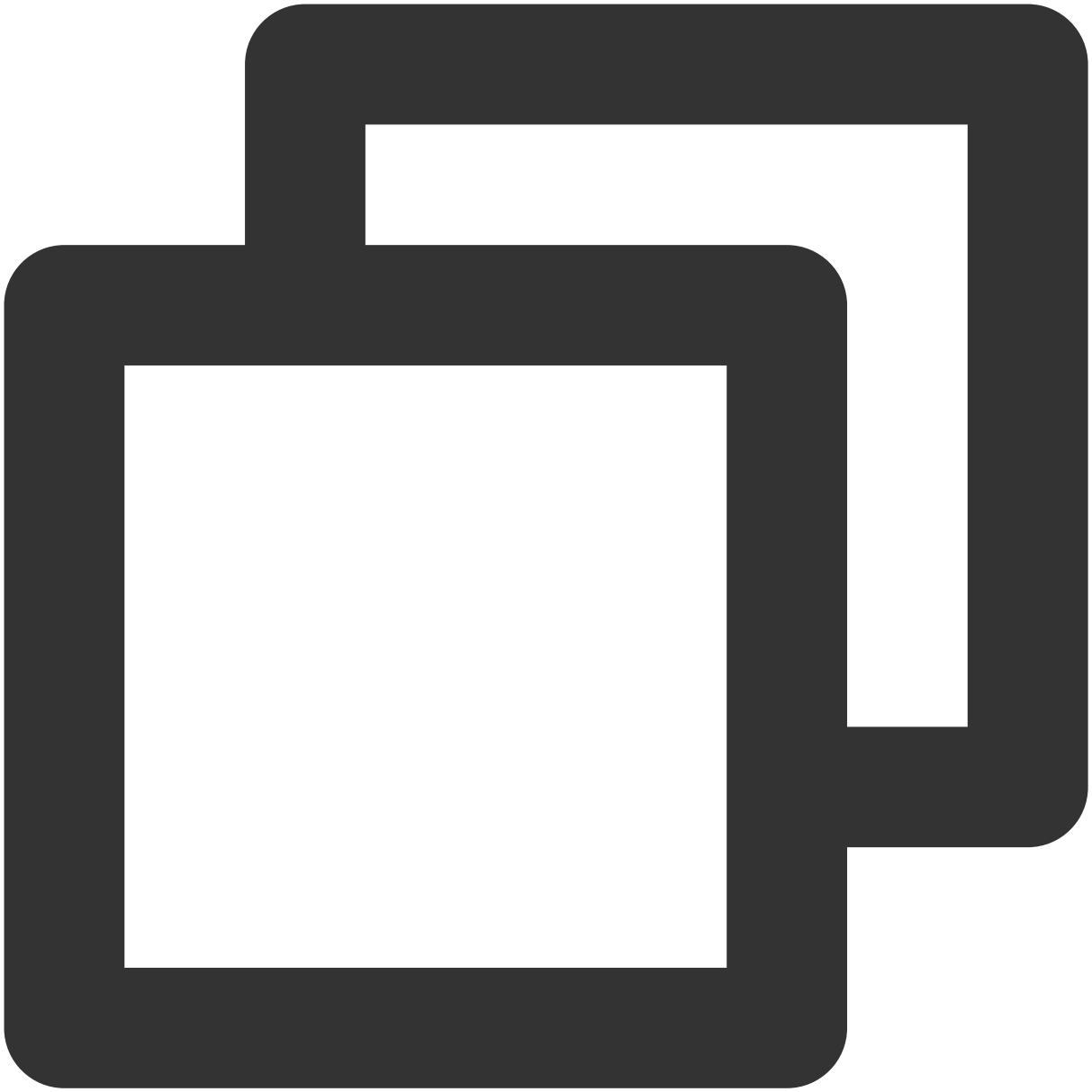
```
$ export TF_CLI_CONFIG_FILE=/Users/you/dev.tfrc
```

设置 `TF_LOG` 环境变量，以打开日志。



```
$ export TF_LOG=TRACE
```

设置个人的腾讯云凭证。可前往 API密钥管理 页面获取。



```
$ export TENCENTCLOUD_SECRET_ID=xxx
$ export TENCENTCLOUD_SECRET_KEY=xxx
```

4. 至此，provider 已完成本地替换，您可自行编写 `.tf` 文件，执行 `terraform plan/apply/destroy` 等命令进行调试。

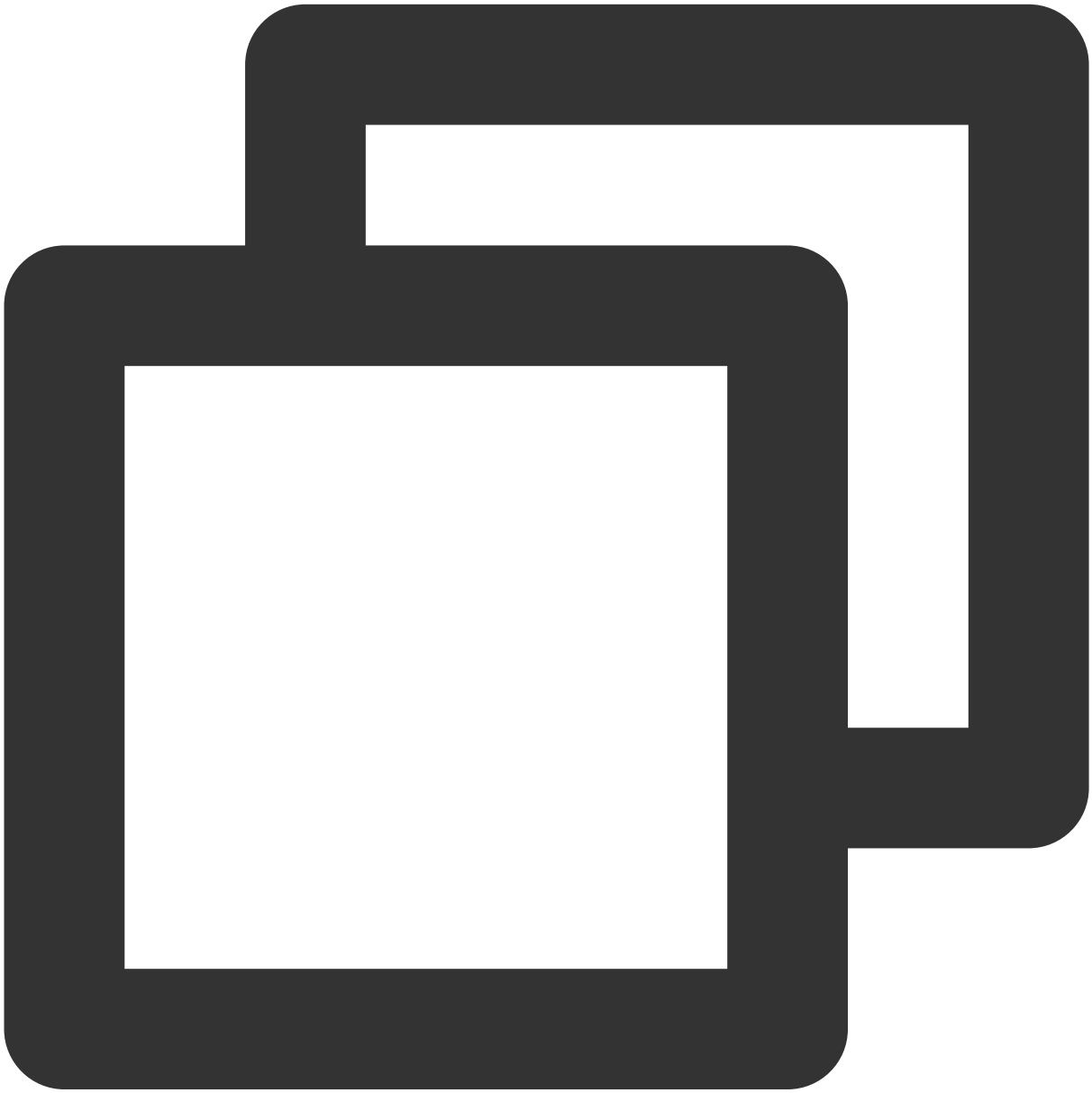
步骤4：单元测试

说明：

强烈建议您自行编写单元测试用例进行测试。您可在 `tencentcloud/` 下查看众多 `*_test.go` 单元测试用例。

如需成为 Terraform 官方认证的 provider，则必须具备单元测试用例。

1. 以 NAT 网关为例，代码如下：



```
package tencentcloud

import (
    "encoding/json"
```

```
    "fmt"
    "log"
    "testing"

    "github.com/hashicorp/terraform/helper/resource"
    "github.com/hashicorp/terraform/terraform"
    "github.com/zqfan/tencentcloud-sdk-go/common"
    vpc "github.com/zqfan/tencentcloud-sdk-go/services/vpc/unversioned"
)

func TestAccTencentCloudNatGateway_basic(t *testing.T) {
    resource.Test(t, resource.TestCase{
        PreCheck:     func() { testAccPreCheck(t) },
        Providers:    testAccProviders,
        // 配置 资源销毁结果检查函数
        CheckDestroy: testAccCheckNatGatewayDestroy,
        // 配置 测试步骤
        Steps: []resource.TestStep{
            {
                // 配置 配置内容
                Config: testAccNatGatewayConfig,
                // 配置 验证函数
                Check: resource.ComposeTestCheckFunc(
                    // 验证资源ID
                    testAccCheckTencentCloudDataSourceID("tencentcloud_nat_gateway.
                    // 验证资源属性，能匹配到，肯定就是创建成功了
                    resource.TestCheckResourceAttr("tencentcloud_nat_gateway.my_nat
                    resource.TestCheckResourceAttr("tencentcloud_nat_gateway.my_nat
                    resource.TestCheckResourceAttr("tencentcloud_nat_gateway.my_nat
                    resource.TestCheckResourceAttr("tencentcloud_nat_gateway.my_nat
                ),
            },
            {
                // 配置 配置内容
                Config: testAccNatGatewayConfigUpdate,
                Check: resource.ComposeTestCheckFunc(
                    testAccCheckTencentCloudDataSourceID("tencentcloud_nat_gateway.
                    // 验证修改后的属性值，如果能匹配到，肯定就是修改成功了
                    resource.TestCheckResourceAttr("tencentcloud_nat_gateway.my_nat
                    resource.TestCheckResourceAttr("tencentcloud_nat_gateway.my_nat
                    resource.TestCheckResourceAttr("tencentcloud_nat_gateway.my_nat
                    resource.TestCheckResourceAttr("tencentcloud_nat_gateway.my_nat
                ),
            },
        },
    })
}
```

```
// testAccProviders 在测试前会根据 Config 建立测试资源，测试结束后又会全部销毁
// 这个函数就是检查资源是否销毁用的，代码逻辑比较好理解，就是根据ID查询资源是否存在
func testAccCheckNatGatewayDestroy(s *terraform.State) error {

    conn := testAccProvider.Meta().(*TencentCloudClient).vpcConn

    // 这用到了 s.RootModule().Resources 数组
    // 这个数组的属性反应的就是资源状态文件 terraform.tfstate
    for _, rs := range s.RootModule().Resources {
        if rs.Type != "tencentcloud_nat_gateway" {
            continue
        }

        descReq := vpc.NewDescribeNatGatewayRequest()
        descReq.NatId = common.StringPtr(rs.Primary.ID)
        descResp, err := conn.DescribeNatGateway(descReq)

        b, _ := json.Marshal(descResp)

        log.Printf("[DEBUG] conn.DescribeNatGateway response: %s", b)

        if _, ok := err.(*common.APIError); ok {
            return fmt.Errorf("conn.DescribeNatGateway error: %v", err)
        } else if *descResp.TotalCount != 0 {
            return fmt.Errorf("NAT Gateway still exists.")
        }
    }
    return nil
}

// 基本用法配置文件，与debug的tf文件一致
const testAccNatGatewayConfig = `
resource "tencentcloud_vpc" "main" {
  name       = "terraform test"
  cidr_block = "10.6.0.0/16"
}
resource "tencentcloud_eip" "eip_dev_dnat" {
  name = "terraform_test"
}
resource "tencentcloud_eip" "eip_test_dnat" {
  name = "terraform_test"
}

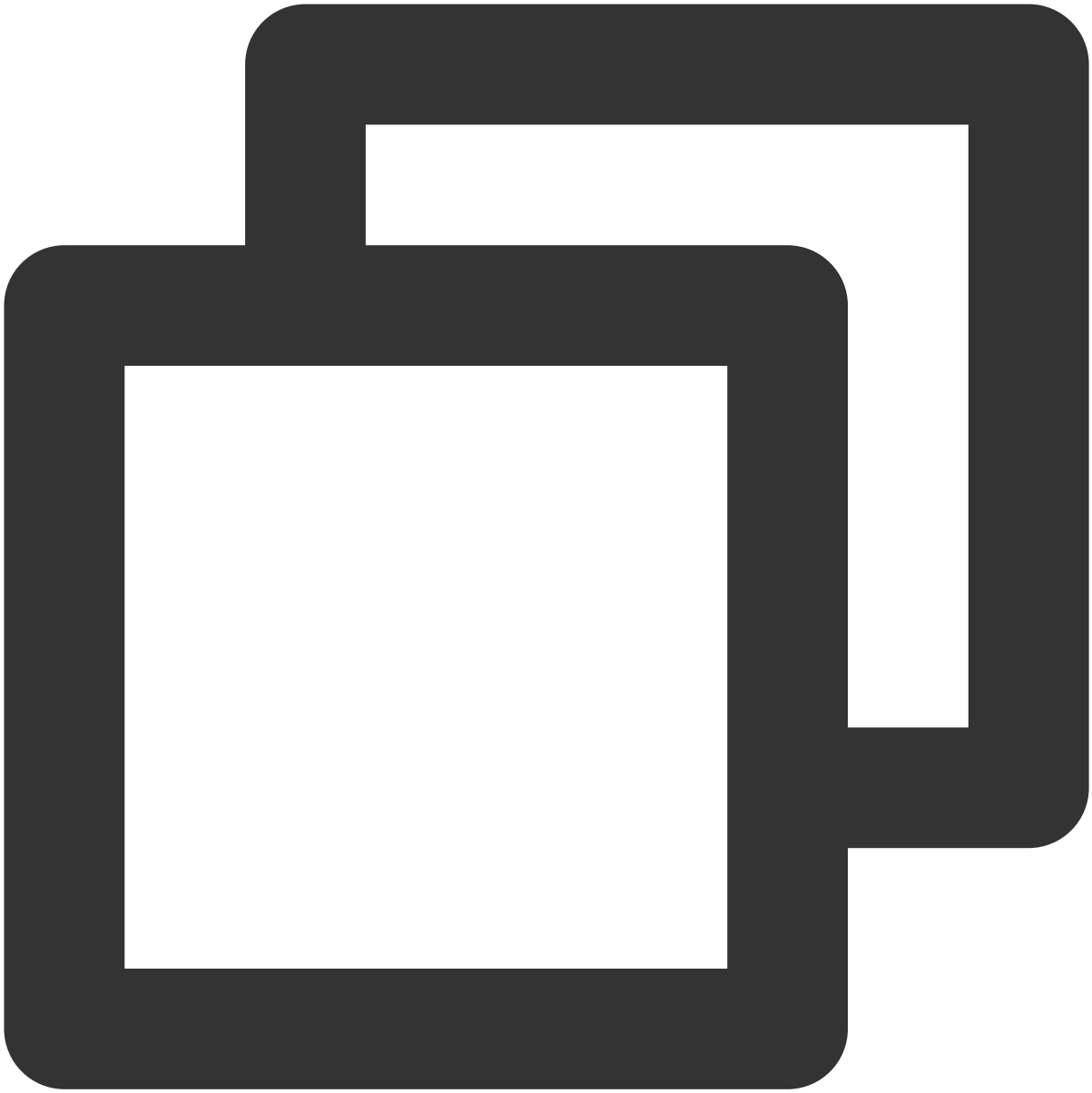
resource "tencentcloud_nat_gateway" "my_nat" {
  vpc_id           = "${tencentcloud_vpc.main.id}"
  name             = "terraform_test"
```

```
  max_concurrent    = 3000000
  bandwidth         = 500
  assigned_eip_set = [
    "${tencentcloud_eip.eip_dev_dnat.public_ip}",
    "${tencentcloud_eip.eip_test_dnat.public_ip}",
  ]
}
`

// 修改用法配置文件，与debug修改后的tf文件一致
const testAccNatGatewayConfigUpdate = `
resource "tencentcloud_vpc" "main" {
  name       = "terraform test"
  cidr_block = "10.6.0.0/16"
}
resource "tencentcloud_eip" "eip_dev_dnat" {
  name = "terraform_test"
}
resource "tencentcloud_eip" "eip_test_dnat" {
  name = "terraform_test"
}
resource "tencentcloud_eip" "new_eip" {
  name = "terraform_test"
}

resource "tencentcloud_nat_gateway" "my_nat" {
  vpc_id           = "${tencentcloud_vpc.main.id}"
  name             = "new_name"
  max_concurrent   = 10000000
  bandwidth        = 1000
  assigned_eip_set = [
    "${tencentcloud_eip.eip_dev_dnat.public_ip}",
    "${tencentcloud_eip.new_eip.public_ip}",
  ]
}
`
```

2. 执行 `TestAccTencentCloudNatGateway_basic` 函数，进行单元测试。



```
$ export TF_ACC=true
$ cd tencentcloud
$ go test -i; go test -test.run TestAccTencentCloudNatGateway_basic -v
```

通过该示例可得知官方的 testAccProviders 除自动编译外，测试流程也更加标准化，全面覆盖 Create/Read/Update/Delete。您可针对同一个资源管理程序，编写更多复杂的场景，并将其加入 Steps 或分为多个测试用例，使得测试更加全面。

要求与建议

最近更新时间：2023-03-07 10:35:48

资源约束

由于本项目是开源项目，欢迎个人或团队贡献代码。为了减少沟通成本，提升贡献者的开发效率和用户体验，请查阅并遵守以下原则：

- 输出产品详细说明、字段清单以及对应 API 接口。
- 由于产品的增删改查需通过调用云 API 实现，云 API 需要暴露增删改查接口，至少支持 API 添加和删除。
- Resource 资源创建后必须要返回唯一 ID，如没有 ID 可使用 Name、序号等唯一值代替。
- 输入参数必须能够查询，以保证配置和实际资源状态一致。
- 必须提供单元测试并保证测试通过。
- 职责单一原则：每次变更仅做一件事，避免依赖或影响其他变更。

慎用 ForceNew

设置了 ForceNew 的字段，如果检测出变更**会导致资源销毁重建**。如果需要设置，请确保本地的输入和远端状态始终一致。避免出现资源创建后，远端返回状态与用户输入不一致产生的 Diff 从而被判定为销毁重建。

默认值 Import

某些资源定义了默认值：

```
map[string]*schema.Schema{
"create_strategy": {
Type: schema.TypeBool,
Optional: true,
Default: "foo",
Description: "Available `foo`, `bar`.",
```

```
},
}
```

假设 `create_strategy` 仅作为创建时传入的只写参数，创建后无法获取，那么这个参数在导入的时候， Value 为空，会导致比对默认值 `foo` 时产生 `+diff` ，所以在实现导入的时候，应该主动设置为它的默认值：

```
&schema.Resource{
Importer: &schema.ResourceImporter{
// helper.ImportWithDefaultValue 在 provider 已封装，可以直接使用
State: helper.ImportWithDefaultValue(map[string]interface{}{
"create_strategy": "foo",
}),
},
```

避免自行约束入参

以 CVM 为例，购买实例 API 文档中，对数据盘的入参有如下说明：

参数名称	是否必填	数据类型	参数说明
DataDisks.N	否	Array of DataDisk	实例数据盘配置信息。若不指定该参数，则默认不购买数据盘。支持购买的时候指定21块数据盘，其中最多包含1块LOCAL_BASIC数据盘或者LOCAL_SSD数据盘，最多包含20块CLOUD_BASIC数据盘、CLOUD_PREMIUM数据盘或者CLOUD_SSD数据盘。

通过参数可知，数据盘最多支持21块数据盘。通常在 schema 中做如下约束：

```
map[string]*schema.Schema{
"data_disks": {
Type: schema.TypeList,
Optional: true,
Description: "数据盘配置",

MaxItems: 21,
},
}
```

但不推荐您使用该约束。因为参数限制会根据产品实际情况不定期地更新。假设因为业务拓展，CVM 支持硬盘数超过21块，那么 Terraform 也要被动地进行同步，这些校验滞后的问题积累起来会给开发者和用户造成影响。

您需要尽量避免此类参数设置输入限制，可以改为在 `Description` 中说明，因为异常输入会被云 API 拦截，Terraform 无需额外加这一层校验。

嵌套结构体设计

TypeList 块和 TypeMap

TypeList 子项支持数字、字符等基本类型。如下所示：

```
resource "foo" "bar" {
strs = ["a", "b", "c"]
nums = [1, 2, 3]
}
```

同时也支持复合 Object 以及嵌套。如下所示：

```
resource "foo" "bar" {
list_item {
name = "l1" # foo.bar.list_item.0.name
}
list_item {
name = "l2" # foo.bar.list_item.1.name
sub_item {
name = "l-2-1" # foo.bar.list_item.1.sub_item.0.name
}
}
}
```

代码中声明和获取的示例如下：

```
var s = map[string]*schema.Schema{
"list_item": {
Type: schema.TypeList,
Elem: &schema.Resource{
Schema: map[string]*schema.Schema{
"sub_item": {
Type: schema.TypeList,
},
},
},
},
}

func create(d *schema.ResourceData) {
```

```
listItems := d.Get("list_item").([]interface{})
listItem1 := listItems[1].(map[string]interface{})
listItem1Sub := listItem1["sub_item"].([]interface{})[0].(map[string]interface{})
}
```

TypeMap 与TypeList 的区别是 TypeMap 在代码 schema 中不支持嵌套，且参数和花括号之间要用等号 `=` 衔接。如下所示：

```
resource "foo" "bar" {
map_item = {
key1: "1", # foo.bar.map_item.key1
key2: "2" # foo.bar.map_item.key2
}
}
```

```
var s = map[string]*schema.Schema{
"map_item": {
Type: schema.TypeMap,
Optional: true,
// Elem: Map 不支持定义 Elem, sdk v1 中无效, v2 中会抛出异常
},
}

func create(d *schema.ResourceData) {
mapItem := d.Get("map_item").(map[string]interface{})
mapVal1 := mapItem["key1"].(string)
}
```

由于 TypeList 的类型约束更完整，如果您需要实现嵌套类型的参数，建议您优先使用 TypeList。而 Map 适合扁平的键值对，如 Tag 标签。

TypeSet

TypeSet 是一种特殊的 TypeList ，用法和 TypeList 完全相同，但是具有唯一性和无序性。

```
resource "foo" "bar" {
set_list = ["a", "b", "c", "c"] # 实际为 ["a", "b", "c"]
}

resource "foo" "bar_update" {
set_list = ["c", "b", "a"] # 等效于 ["a", "b", "c"] ，不会产生 Diff
}
```

代码中 schema 可以自定义 `Get` 函数，返回唯一索引，只要索引相同则视为元素相同。如下所示：

```
var s = map[string]*schema.Schema{
"set_list": {
Type: schema.TypeList,
Elem: &schema.Schema{Type: schema.TypeString},
Get: func(v interface) { return getValueHash(v.(string)) },
},
}

func create(d *schema.ResourceData) {
setList := d.Get("set_list").(*schema.Set).List()
setListItemHead := listItems[0].(string)
}
```