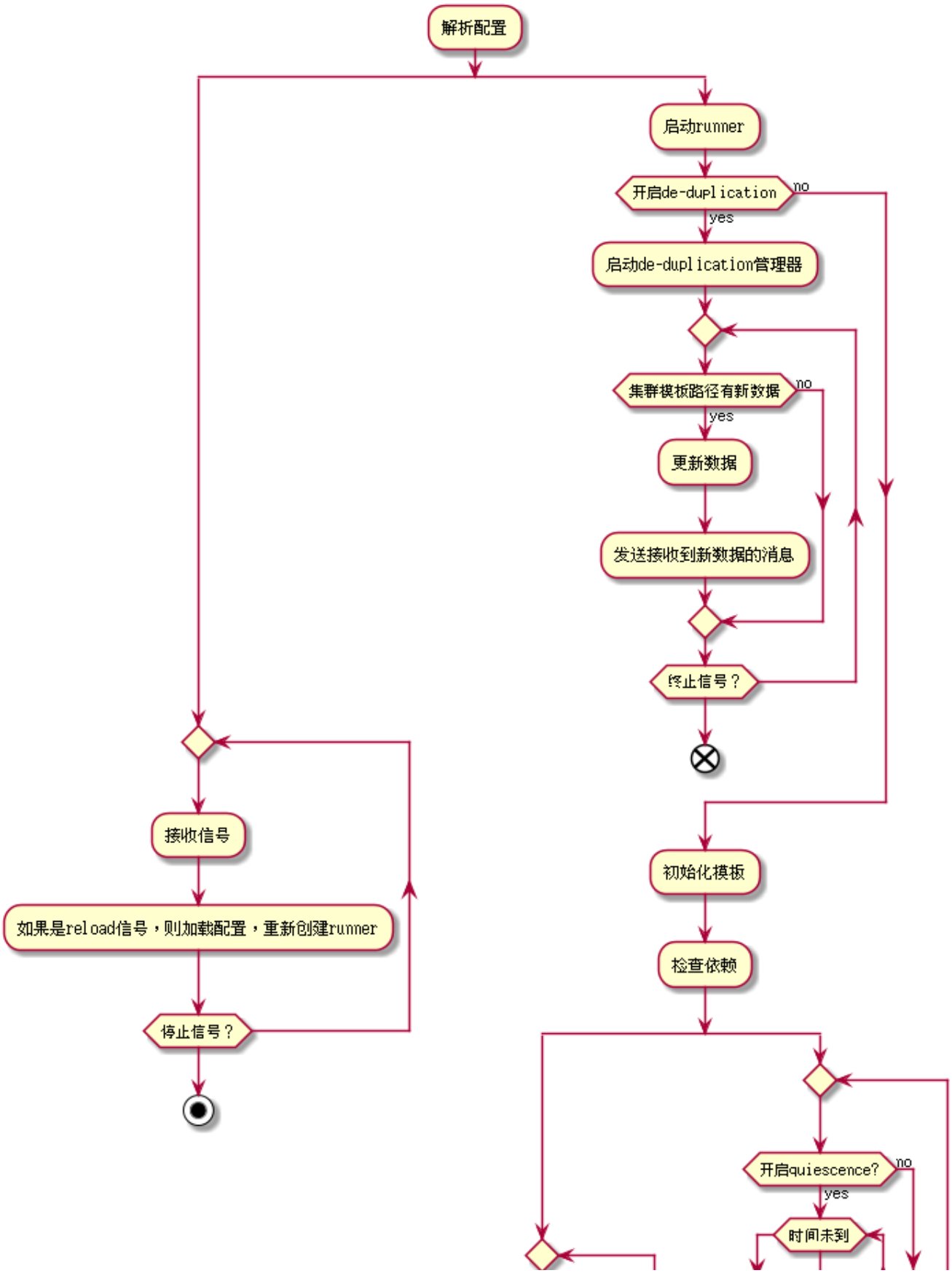
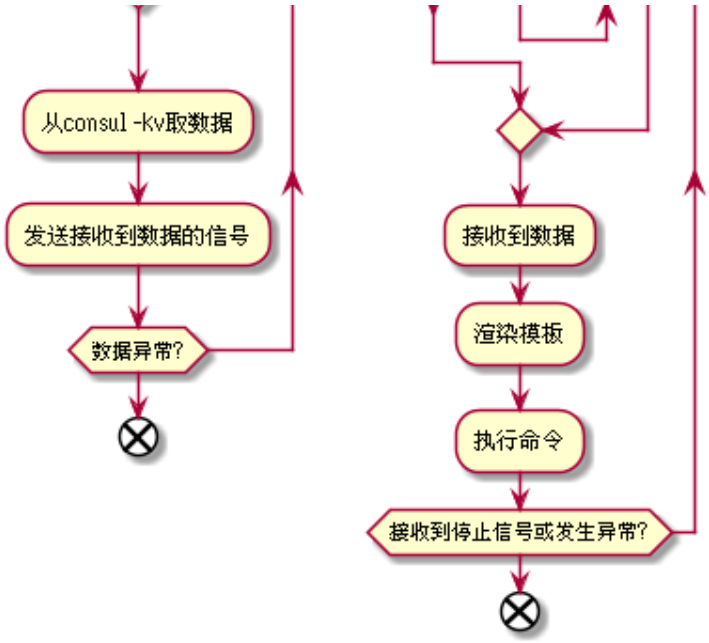
上一篇文章简单介绍了一下Consul-template基本用法，本篇主要深入看一下consul-template的源码。

Consul-template的整个流程还是比较清晰的，不过代码中大量运用了goroutine、channel和goto等高级特性，如果不仔细看的话，有些地方可能理不清楚。

下图即是consul-template的整个执行流程。整个流程开有多个单独的goroutine，分别是监听信号重新加载配置或者停止、监听consul kv存储的变化发送更新通知、监听数据的更新通知以及渲染模板并执行命令等





初始化资源

consul-template启动后首先解析配置，并初始化相关资源

```
//cli.go
//dry：dry模式下会将渲染内容展示在stdout中，不会改变生成的文件，方便验证模板内容是否正确
//once:是否只渲染一下，可用于调试
runner, err := manager.NewRunner(config, dry, once)
if err != nil {
    return logError(err, ExitCodeRunnerError)
}
go runner.Start()
//省略信号监控goroutine
```

```
//runner.go
//NewRunner中调用init方法
func (r *Runner) init() error {
        //...省略配置解析，异常捕获等代码
        //根据配置文件创建链接consul的客户端
        clients, err := newClientSet(r.config)

        //创建监听器
        watcher, err := newWatcher(r.config, clients, r.once)

        //解析配置的模板
        for _, ctmpl := range *r.config.Templates {
                tmpl, err :=
template.NewTemplate(&template.NewTemplateInput{
            //模板源文件的路径
            Source:        config.StringVal(ctmpl.Source),
            //模板内容  和上面的路径必须保证有一个存在
```

```
            Contents:      config.StringVal(ctmpl.Contents),
                })
        }

        //省略部分初始化代码

        if *r.config.Dedup.Enabled {
                if r.once {
                } else {
                        //如果开启了de-dup属性的话，这里会创建de-dup管理器
                        r.dedup, err = NewDedupManager(r.config.Dedup,
clients, r.brain, r.templates)
                        if err != nil {
                                return err
                        }
                }
        }

        return nil
}
```

启动de-dup管理器和数据监控goroutine

de-dup主要是为了优化性能，具体可参考上一篇基本用法。

```
func (r *Runner) Start() {

        //启动 de-duplication 管理器
        var dedupCh <-chan struct{}
        if r.dedup != nil {
                if err := r.dedup.Start(); err != nil {
                        r.ErrCh <- err
                        return
                }
                dedupCh = r.dedup.UpdateCh()
        }

        if err := r.Run(); err != nil {
                r.ErrCh <- err
                return
        }

        for {

        NEXT_Q:
                for _, t := range r.templates {
                        if _, ok := r.quiescenceMap[t.ID()]; ok {
                                continue NEXT_Q
                        }
```

```
                        for _, c := range r.templateConfigsFor(t) {
                                if *c.Wait.Enabled {
                                        r.quiescenceMap[t.ID()] =
newQuiescence(
                                                r.quiescenceCh,
*c.Wait.Min, *c.Wait.Max, t)
                                        continue NEXT_Q
                                }
                        }

                        if *r.config.Wait.Enabled {
                                r.quiescenceMap[t.ID()] = newQuiescence(
                                        r.quiescenceCh, *r.config.Wait.Min,
*r.config.Wait.Max, t)
                                continue NEXT_Q
                        }
                }



        OUTER:
                select {
                case view := <-r.watcher.DataCh():
                        r.Receive(view.Dependency(), view.Data())
                        //循环读取数据
                        for {
                                select {
                                case view := <-r.watcher.DataCh():
                                        r.Receive(view.Dependency(),
view.Data())
                                default:
                                        break OUTER
                                }
                        }

                case <-dedupCh:
                        //接收到de-dup消息
                        log.Printf("[INFO] (runner) watcher triggered by
de-duplication manager")
                        break OUTER

                }
                //开始渲染数据
                if err := r.Run(); err != nil {
                        r.ErrCh <- err
                        return
                }
        }
}
```

渲染模板

consul-template的模板语法其实是采用的golang模板的模板语法，通过自定义函数，来进行数据注入

```
//runner.go
//go runner.Start()
func (r *Runner) Run() error {

        var newRenderEvent, wouldRenderAny, renderedAny bool
        runCtx := &templateRunCtx{
                depsMap: make(map[string]dep.Dependency),
        }
    //渲染模板
        for _, tmpl := range r.templates {
                //渲染单个模板
                event, err := r.runTemplate(tmpl, runCtx)
                if err != nil {
                        return err
                }

        }
    //渲染模板完毕执行命令
        var errs []error
        for _, t := range runCtx.commands {
                command := config.StringVal(t.Exec.Command)
                env := t.Exec.Env.Copy()
                env.Custom = append(r.childEnv(), env.Custom...)
                if _, err := spawnChild(&spawnChildInput{
                        //省略
                }); err != nil {
                        s := fmt.Sprintf("failed to execute command %q from
%s", command, t.Display())
                        errs = append(errs, errors.Wrap(err, s))
                }
        }


        return nil
}
```

```
//runner.go
func (r *Runner) runTemplate(tmpl *template.Template, runCtx
*templateRunCtx) (*RenderEvent, error) {

        // 检查本示例节点是否是leader节点
        isLeader := true
        if r.dedup != nil {
                isLeader = r.dedup.IsLeader(tmpl)
        }

    //尝试渲染模板
```

```
        result, err := tmpl.Execute(&template.ExecuteInput{
                Brain: r.brain,
                Env:   r.childEnv(),
        })
        if err != nil {
                return nil, errors.Wrap(err, tmpl.Source())
        }

        //检查模板渲染所需要的数据是否都满足了，如果不满足，则加入监控列表
        missing, used := result.Missing, result.Used
        for _, d := range used.List() {
                if isLeader && !r.watcher.Watching(d) {
                        missing.Add(d)
                }
                if _, ok := runCtx.depsMap[d.String()]; !ok {
                        runCtx.depsMap[d.String()] = d
                }
        }

        if l := unwatched.Len(); l > 0 {
                for _, d := range unwatched.List() {
                        if isLeader || !d.CanShare() {
                                //注意此处将调用goroutine监控consul kv的变化
                                r.watcher.Add(d)
                        }
                }
                return event, nil
        }

    //如果开启了de-duplication模式，并且本示例为leader节点，则更新consul中模板渲染
的结果，便于其他节点使用
        if r.dedup != nil && isLeader {
                if err := r.dedup.UpdateDeps(tmpl, used.List()); err != nil
{
                        log.Printf("[ERR] (runner) failed to update
dependency data for de-duplication: %v", err)
                }
        }

        //如果开启了quiescence特性，则检查一定时间内是否已经更新过了，如果更新过了，不
再更新，直接返回
        if q, ok := r.quiescenceMap[tmpl.ID()]; ok {
                q.tick()
                event.ForQuiescence = true
                return event, nil
        }

        // 对于每一个模板，将其渲染后的数据写入文件，并存储需要后续执行的命令
        for _, templateConfig := range r.templateConfigsFor(tmpl) {

                result, err := renderer.Render(&renderer.RenderInput{
                        Backup:
config.BoolVal(templateConfig.Backup),
                        Contents:       result.Output,
```

```
                        CreateDestDirs:
config.BoolVal(templateConfig.CreateDestDirs),
                        Dry:            r.dry,
                        DryStream:      r.outStream,
                        Path:
config.StringVal(templateConfig.Destination),
                        Perms:
config.FileModeVal(templateConfig.Perms),
                })

                if result.DidRender {
                        //省略模板渲染后执行后续命令的代码
                }
        }

        return event, nil
}
```

```
//renderer/renderer.go
func Render(i *RenderInput) (*RenderResult, error) {
        existing, err := ioutil.ReadFile(i.Path)
        if err != nil && !os.IsNotExist(err) {
                return nil, errors.Wrap(err, "failed reading file")
        }
    //读取上次渲染后的结果，比较和本次结果是否一致，一致的话就不再重复写入
        if bytes.Equal(existing, i.Contents) {
                return &RenderResult{
                        DidRender:   false,
                        WouldRender: true,
                        Contents:    existing,
                }, nil
        }

        if i.Dry {
        //开启dry模式的话，不写入文件，只打印结果
                fmt.Fprintf(i.DryStream, "> %s\n%s", i.Path, i.Contents)
        } else {
        //否则，确保原子写入文件
                if err := AtomicWrite(i.Path, i.CreateDestDirs, i.Contents,
i.Perms, i.Backup); err != nil {
                        return nil, errors.Wrap(err, "failed writing file")
                }
        }
        return &RenderResult{
                DidRender:   true,
                WouldRender: true,
                Contents:    i.Contents,
        }, nil
}
```

```
//template/template.go
type ExecuteInput struct {
        // Brain 存储模板渲染所需要的数据
        Brain *Brain
        Env []string
}

type Brain struct {
    sync.RWMutex
    //判断是否收到数据更新
        receivedData map[string]struct{}
    //存储具体的数据
    data map[string]interface{}
}


//解析渲染模板：一、需要模板  二需要数据
func (t *Template) Execute(i *ExecuteInput) (*ExecuteResult, error) {
        if i == nil {
                i = &ExecuteInput{}
        }

        var used, missing dep.Set
    //consul-template使用的go自带的模板渲染引擎
    tmpl := template.New("")
    //自定义分隔符
    tmpl.Delims(t.leftDelim, t.rightDelim)
    //自定义函数，包括需要渲染的数据都在自定义函数里面
        tmpl.Funcs(funcMap(&funcMapInput{
                t:       tmpl,
                brain:   i.Brain,
                env:     i.Env,
                used:    &used,
                missing: &missing,
        }))

    //解析模板
        tmpl, err := tmpl.Parse(t.contents)

        //开始渲染，返回结果放在b中，这里需要传的数据为nil，是因为数据都在自定义函数里
面
        var b bytes.Buffer
        if err := tmpl.Execute(&b, nil); err != nil {
                return nil, errors.Wrap(err, "execute")
        }

        return &ExecuteResult{
                Used:    &used,
                Missing: &missing,
                Output:  b.Bytes(),
        }, nil
}
```

```
func funcMap(i *funcMapInput) template.FuncMap {
        return template.FuncMap{
                "datacenters":  datacentersFunc(i.brain, i.used,
i.missing),
        "file":         fileFunc(i.brain, i.used, i.missing),
        //所有的值都放在这里面，因此在模板中定义时需要添加上 {{ key xxx}}
        //keyFunc这个函数主要就是从brain中取出数据
        "key":          keyFunc(i.brain, i.used, i.missing),
        //省略其它自定义函数
        }
}
```

监控consul-kv

这里监控consul的数据，采用的consul的阻塞get方法，默认会等待1分钟。如果有数据更新，立即返回；否则会超时返回。

```
//watch/watcher.go

func (w *Watcher) Add(d dep.Dependency) (bool, error) {
        //创建监听视图
        v, err := NewView(&NewViewInput{
                Dependency: d,
                Clients:    w.clients,
                MaxStale:   w.maxStale,
                Once:       w.once,
                RetryFunc:  retryFunc,
                VaultGrace: w.vaultGrace,
        })
        if err != nil {
                return false, errors.Wrap(err, "watcher")
        }

        w.depViewMap[d.String()] = v
        //开启单独的poll
        go v.poll(w.dataCh, w.errCh)

        return true, nil
}
```

```
// watch/view.go
func (v *View) poll(viewCh chan<- *View, errCh chan<- error) {
        var retries int

        for {
                doneCh := make(chan struct{}, 1)
                successCh := make(chan struct{}, 1)
                fetchErrCh := make(chan error, 1)
```

```
                //开启单独的goroutine从consul中获取数据
                go v.fetch(doneCh, successCh, fetchErrCh)

        WAIT:
                select {
                        //省略接收到数据后的一些充值操作
                }
        }
}
```

```
//watch/view.go
func (v *View) fetch(doneCh, successCh chan<- struct{}, errCh chan<- error)
{

        for {

                case <-v.stopCh:
                        return
                default:
                }

                data, rm, err := v.dependency.Fetch(v.clients,
&dep.QueryOptions{
                        AllowStale: allowStale, //此变量决定是否从consul的
flower节点拉取数据
                        WaitTime:   defaultWaitTime,//defaultWaitTime = 60
* time.Second 默认超时时间为1分钟，不可以修改
                        WaitIndex:  v.lastIndex,//consul通过此字段来判断客户端
和consul的数据是否同步
                        VaultGrace: v.vaultGrace,
                })

                select {
                        //通知收到了数据
                case successCh <- struct{}{}:
                default:
                }
                //省略内容
                return
        }
}
```

```
//dependency/kv_get.go
func (d *KVGetQuery) Fetch(clients *ClientSet, opts *QueryOptions)
(interface{}, *ResponseMetadata, error) {
        //这里会阻塞一分钟，在这一分钟内，如果数据有更新，会立即返回，否则会超时返回
        pair, qm, err := clients.Consul().KV().Get(d.key,
opts.ToConsulOpts())
        if err != nil {
```

```
                return nil, nil, errors.Wrap(err, d.String())
        }
        rm := &ResponseMetadata{
                LastIndex:   qm.LastIndex,
                LastContact: qm.LastContact,
                Block:       d.block,
        }
        value := string(pair.Value)
        return value, rm, nil
}
```