

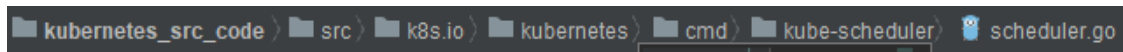
scheduler

开始前，我们做一部分记录，由于目前是在win下学习和调试代码，在win下需要注意如下事项：

- windows的git for windows在clone代码的时候，是默认不支持软连接的，如果直接使用git clone下载，会把软连接下载成为包含软连接路径的一个文件，go是无法识别的。git clone的时候加入-c core.symlinks=true这个参数，并且软连接的建立需要管理员的特定权限。
- 下载的路径应该放在\$GOPATH/src/k8s.io下，因为kubernetes的源码中引包路径就是k8s.io/.....
- 关于scheduler的调试，因为scheduler相对是独立的k8s组件，所以可以通过IDE，比如goland在个人PC上debug，访问虚拟机里搭建好的k8s集群，k8s集群disable掉自身的scheduler组件，在个人PC通过master节点ip从集群外访问apiserver，可以直接采用集群内部的scheduler.conf配置文件。由于我是采用kubeadm搭建的单节点集群，从外部通过外网ip访问apiserver时，会出现x509，可参考如下链接处理。另外切记调试的scheduler版本和其他组件版本应该匹配，否则可能会有版本不兼容问题。
<https://stackoverflow.com/questions/46360361/invalid-x509-certificate-for-kubernetes-master>

生成scheduler配置

接下来正式开始源码的学习，第一部分是scheduler，首先我们找到这部分的入口，



```

func main() {
    rand.Seed(time.Now().UnixNano())

    command := app.NewSchedulerCommand()

    // TODO: once we switch everything over to Cobra commands, we can go back to calling
    // utilflag.InitFlags() (by removing its pflag.Parse() call). For now, we have to set the
    // normalize func and add the go flag set by hand.
    pflag.CommandLine.SetNormalizeFunc(cliflag.WordSepNormalizeFunc)
    // utilflag.InitFlags()
    logs.InitLogs()
    defer logs.FlushLogs()

    if err := command.Execute(); err != nil {
        os.Exit(1)
    }
}

func NewSchedulerCommand(registryOptions ...Option) *cobra.Command {
    opts, err := options.NewOptions()
    if err != nil {
        klog.Fatalf("unable to initialize command options: %v", err)
    }

    cmd := &cobra.Command{
        Use: "kube-scheduler",
        Long: `The Kubernetes scheduler is a policy-rich, topology-aware,
workload-specific function that significantly impacts availability, performance,
and capacity. The scheduler needs to take into account individual and collective
resource requirements, quality of service requirements, hardware/software/policy
constraints, affinity and anti-affinity specifications, data locality, inter-workload
interference, deadlines, and so on. Workload-specific requirements will be exposed
through the API as necessary. See [scheduling](https://kubernetes.io/docs/concepts/scheduling/)
for more information about scheduling and the kube-scheduler component.`,
        Run: func(cmd *cobra.Command, args []string) {
            if err := runCommand(cmd, args, opts, registryOptions...); err != nil {
                fmt.Fprintf(os.Stderr, format: "%v\n", err)
                os.Exit(1)
            }
        },
    }
}

```

首先关于NewSchedulerCommand()方法,

- NewOptions()主要用于对scheduler进行默认配置的获取, 包括默认监听端口、默认调度器的名称、认证和鉴权配置
- flag相关主要是读取文件配置, 对对应flag的默认配置进行重写。

```

fs := cmd.Flags()
namedFlagSets := opts.Flags()
verflag.AddFlags(namedFlagSets.FlagSet("global"))
globalflag.AddGlobalFlags(namedFlagSets.FlagSet("global"), cmd.Name())
for _, f := range namedFlagSets.FlagSets {
    fs.AddFlagSet(f)
}

```

对于cobra，执行根命令kube-scheduler，会对应执行runCommand方法。同时在下方面也添加了Usage Help相关的命令行。

```
if err := command.Execute(); err != nil {
    fmt.Fprintf(os.Stderr, format: "%v\n", err)
    os.Exit( code: 1)
}
```

在main方法中，command.Execute(),接下来重点进入到runCommand方法，

- 进行Validate(), 对scheduler的相关配置进行校验，包括监听端口有效性、调度器名称、scheduler高可用的配置等
- 另外一个就是Config()方法，该方法基于opt返回scheduler的config，其中applyTo()方法用于将opt相关的配置生效到KubeSchedulerConfiguration这个结构体中，而KubeSchedulerConfiguration又是schedulerappconfig结构体中的一部分，scheduler实例config配置结构体如下，还包含了请求apiserver的client、处理event的client、高可用相关的配置以及informer相关的封装。

```
// Config has all the context to run a Scheduler
type Config struct {
    // config is the scheduler server's configuration object.
    ComponentConfig kubeschedulerconfig.KubeSchedulerConfiguration

    // LoopbackClientConfig is a config for a privileged loopback connection
    LoopbackClientConfig *restclient.Config

    InsecureServing      *apiserver.DeprecatedInsecureServingInfo // nil will disable serving on an insecure
    InsecureMetricsServing *apiserver.DeprecatedInsecureServingInfo // non-nil if metrics should be served in
    Authentication      apiserver.AuthenticationInfo
    Authorization        apiserver.AuthorizationInfo
    SecureServing        *apiserver.SecureServingInfo

    Client      clientset.Interface
    InformerFactory informers.SharedInformerFactory
    PodInformer  coreinformers.PodInformer
    EventClient  vlcore.EventsGetter
    Recorder     record.EventRecorder
    Broadcaster  record.EventBroadcaster

    // LeaderElection is optional.
    LeaderElection *leaderelection.LeaderElectionConfig
}
```

生成scheduler对象

如上就搞定了scheduler运行相关的配置，接下来就会带着上面的配置进入到Scheduler 运行相关的逻辑，执行Run(cc, stopCh)这个方法。

*** 首先看下scheduler结构体，**

scheduler其实就是一个属性

```
// Scheduler watches for new unscheduled pods. It attempts to find
// nodes that they fit on and writes bindings back to the api server.
type Scheduler struct {
    config *factory.Config
}
```

我们后面看下这个factory.Config这个结构体

```

type configFactory struct {
    // 与api-server通信的客户端
    client clientset.Interface
    // queue for pods that need scheduling
    // 存着那些需要调度的pod
    podQueue internalqueue.SchedulingQueue
    // a means to list all known scheduled pods.
    // 可以获得所有已经调度的pod
    scheduledPodLister corelisters.PodLister
    // a means to list all known scheduled pods and pods assumed to have been scheduled.
    // 可以获得所有已经调度的pod和那些assumed pod
    podLister algorithm.PodLister
    // a means to list all nodes
    nodeLister corelisters.NodeLister
    // a means to list all PersistentVolumes
    pvLister corelisters.PersistentVolumeLister
    // a means to list all PersistentVolumeClaims
    pvCLister corelisters.PersistentVolumeClaimLister
    // a means to list all services
    serviceLister corelisters.ServiceLister
    // a means to list all controllers
    controllerLister corelisters.ReplicationControllerLister
    // a means to list all replicaset
    replicaSetLister appslisters.ReplicaSetLister
    // a means to list all statefulsets
    statefulSetLister appslisters.StatefulSetLister
    // a means to list all PodDisruptionBudgets
    pdbLister policylisters.PodDisruptionBudgetLister
    // a means to list all StorageClasses
    storageClassLister storagelisters.StorageClassLister
    // Close this to stop all reflectors
    StopEverything <-chan struct{}
    scheduledPodsHasSynced cache.InformerSynced
    schedulerCache schedulerinternalcache.Cache
    // SchedulerName of a scheduler is used to select which pods will be
    // processed by this scheduler, based on pods's "spec.schedulerName".
    // 调度器的名字 默认为default-scheduler
    schedulerName string

    // RequiredDuringScheduling affinity is not symmetric, but there is an implicit PreferredDuringScheduling
    // corresponding to every RequiredDuringScheduling affinity rule.
    // HardPodAffinitySymmetricWeight represents the weight of implicit PreferredDuringScheduling
    hardPodAffinitySymmetricWeight int32
    // Equivalence class cache
    // 加速predicate阶段的equivalence class cache
    equivalencePodCache *equivalence.Cache
    // Enable equivalence class cache
    enableEquivalenceClassCache bool
    // Handles volume binding decisions
    volumeBinder *volumebinder.VolumeBinder
    // Always check all predicates even if the middle of one predicate fails.
    alwaysCheckAllPredicates bool
    // Disable pod preemption or not.
    // 是否禁止抢占
    disablePreemption bool
    // percentageOfNodesToScore specifies percentage of all nodes to score in each scheduling cycle
    percentageOfNodesToScore int32
}

```

包括如下定义，

各种资源的Lister，Lister是列出各种资源的方法，包括pod、node、PV、PVC、svc、rc、rs、ss、pdb、storage、已经调度的pod和assumed pod（为了考虑性能，在调度环

境不会直接调用apiserver完成bind，而是假设调度的结果可以bind的，此类型pod就是 assumed pod) Lister

与apiserver通信的client、需要调度的pod队列、调度器的名称、是否禁止抢占

* 接下来进入scscheduler的New()方法，

New方法入参是前面scheduler config的各种参数，返回前面这个结构体。

1. schedulerCache := internalcache.New(30*time.Second, stopEverything)
初始化了一个schedulerCache，schedulerCache会不断的更新Pod和Node信息
2. c.scheduledPodLister = assignedPodLister{args.PodInformer.Lister()}
args.PodInformer.Lister()可以获得所有的pod，过滤可以得到已经被调度的pod
c.scheduledPodLister
3. go cc.PodInformer.Informer().Run(stopCh)
cc.InformerFactory.Start(stopCh)
开启podinformer
4. 等LeaderElection 完成后，默认支持高可用，运行run，执行sched.Run()

执行调度器sched.Run

```
// Run begins watching and scheduling. It waits for cache to be synced, then starts a goroutine and returns immediately.
func (sched *Scheduler) Run() { sched: *k8s.io/kubernetes/pkg/scheduler.Scheduler / 0xc00045f5d0
    if !sched.config.WaitForCacheSync() {
        return
    }

    go wait.Until(sched.scheduleOne, period, sched.config.StopEverything)
}
```

时间间隔是0s，串行循环执行schedulerOne方法，scheduleOne方法是调度最核心的逻辑，调度一个pod的完整逻辑流程。

1. NextPod()从queue中获取待调度的下一个pod，debug时此时pod为pending状态

```
▼ pod = {k8s.io/kubernetes/vendor/k8s.io/api/core/v1.Pod | 0xc0009d0380}
  ▶ TypeMeta = {k8s.io/kubernetes/vendor/k8s.io/apimachinery/pkg/apis/meta/v1.TypeMeta}
  ▶ ObjectMeta = {k8s.io/kubernetes/vendor/k8s.io/apimachinery/pkg/apis/meta/v1.ObjectMeta}
  ▶ Spec = {k8s.io/kubernetes/vendor/k8s.io/api/core/v1.PodSpec}
  ▶ Status = {k8s.io/kubernetes/vendor/k8s.io/api/core/v1.PodStatus}
```

2. scheduleResult, err := sched.schedule(pod)
完成pod的调度，得到调度的节点信息

进入schedule(pod),

```
func (sched *Scheduler) schedule(pod *v1.Pod) (core.ScheduleResult, error) {
    result, err := sched.config.Algorithm.Schedule(pod, sched.config.NodeLister)
    if err != nil {
        pod = pod.DeepCopy()
        sched.recordSchedulingFailure(pod, err, v1.PodReasonUnschedulable, err.Error())
        return core.ScheduleResult{}, err
    }
    return result, err
}
```

继续进入该方法 result, err := sched.config.Algorithm.Schedule(pod, sched.config.NodeLister)

- * pod进行检测，是否有pv挂载。
- * 通过nodeLister方法获取所有节点
- * 更新node信息快照

进入预选逻辑

进入findNodesThatFit，该方法即是node预选的核心逻辑。

*** allNodes := int32(g.cache.NodeTree().NumNodes())**

获取所有node的数量

*** numNodesToFind := g.numFeasibleNodesToFind(allNodes)**

```
func (g *genericScheduler) numFeasibleNodesToFind(numAllNodes int32) (numNodes int32) {
    if numAllNodes < minFeasibleNodesToFind || g.percentageOfNodesToScore >= 100 {
        return numAllNodes
    }

    adaptivePercentage := g.percentageOfNodesToScore
    if adaptivePercentage <= 0 {
        adaptivePercentage = schedulerapi.DefaultPercentageOfNodesToScore - numAllNodes/125
        if adaptivePercentage < minFeasibleNodesPercentageToFind {
            adaptivePercentage = minFeasibleNodesPercentageToFind
        }
    }

    numNodes = numAllNodes * adaptivePercentage / 100
    if numNodes < minFeasibleNodesToFind {
        return minFeasibleNodesToFind
    }

    return numNodes
}
```

设定最多需要进行筛选的节点数量，避免node数太多影响调度效率。这部分的逻辑是

1. 如果节点数小于minFeasibleNodesToFind(默认为100)，或者设定的percentageOfNodesToScore(百分比，默认为50)大于100，则返回总节点数
2. 如果节点数大于100，则numNodes根据总节点数换算percentageOfNodesToScore与100的比例得到最终结果

*** 该方法启动16个workers完成int(allNodes)个checkNode任务，该方法实现可以参考用在平时类似场景。**

```
func ParallelizeUntil(ctx context.Context, workers, pieces int, doWorkPiece DoWorkPieceFunc) {
    var stop <-chan struct{}
    if ctx != nil {
        stop = ctx.Done()
    }

    toProcess := make(chan int, pieces)
    for i := 0; i < pieces; i++ {
        toProcess <- i
    }
    close(toProcess)

    if pieces < workers {
        workers = pieces
    }

    wg := sync.WaitGroup{}
    wg.Add(workers)
    for i := 0; i < workers; i++ {
        go func() {
            defer utilruntime.HandleCrash()
            defer wg.Done()
            for piece := range toProcess {
                select {
                case <-stop:
                    return
                default:
                    doWorkPiece(piece)
                }
            }
        }()
    }
    wg.Wait()
}
```

首先假定我们需要对100个node进行check，此时我们启动16个workers完成该任务

1. 定义一个长度为100的channel，依次写入0-99的整数进该channel
2. 如果任务数小于worker数，那么worker数置为与任务数相同
3. 循环16个worker，每个worker循环从前面channel中获取编号，然后去处理对应编号的任务
4. 最后wg.Wait()等待所有worker任务完成。

*** 接下来我们看上面方法实际执行的任务checkNode**


```

// 此处是定义了一个方法，判断node是否满足调度
checkNode := func(i int) {
    nodeName := g.cache.NodeTree().Next()
    fits, failedPredicates, err := podFitsOnNode(
        pod,
        meta,
        g.nodeInfoSnapshot.NodeInfoMap[nodeName],
        g.predicates,
        g.schedulingQueue,
        g.alwaysCheckAllPredicates,
    )
}

```

1. 首先从NodeTree()获取下一个node
2. 判断当前node和pod是否满足调度要求，即podFitsOnNode()方法，后面会详细解析该方法
3. 如果返回为true，表示当前pod满足调度要求，就将该节点加入filtered列表
4. 如果返回为false或报错，就将node加入failedPredicateMap[nodeName]中，并且记录下不满足的err

* 分析podFitsOnNode()

这个方法是根据pod和nodeinfo判断pod是否fit node。

从下面的逻辑中，我们可以看出通过两次for循环执行了两次预选

1. 关于两次执行预选的理解，在源码的注释里有详细的解释，在第一次时，可能会有其他 优先级大于等于 当前pod的其他pod已经bind到当前node，但是还未完成调度过程，此时我们需要将这些pods加入到meta和node info后再进行一次预选。
2. 如果加入更高优先级pods后进行预选，当前node满足调度条件，那么就需要进行第二次预选计算，就是不加入更高优先级pod情况下，进行计算，判断node是否满足调度。-----这部分的原因是可能会有pod亲和性相关的机制，此时就需要考虑更高有限级pods没有加入时能否满足亲和性条件。

```

// predicates.Ordering() 可以得到[]string, 包含所有predicates方法列表
for _, predicateKey := range predicates.Ordering() {
    var (
        fit      bool
        reasons []predicates.PredicateFailureReason
        err       error
    )
    // TODO (yastij) : compute average predicate restrictiveness to export it as Prometheus metric

    // predicateFuncs是一个map, key是方法名, value是对应的func
    // 如果存在对应的方法, 那么就进入下面的逻辑
    if predicate, exist := predicateFuncs[predicateKey]; exist {
        // 通过对应的predicate方法进行node能否调度的判断逻辑
        fit, reasons, err = predicate(pod, metaToUse, nodeInfoToUse)
        if err != nil {
            return false, []predicates.PredicateFailureReason{}, err
        }
    }
}

```

3. 那么具体的预选方法有哪些呢？

```
var (
    predicatesOrdering = []string{CheckNodeConditionPred, CheckNodeUnschedulablePred,
        GeneralPred, HostNamePred, PodFitsHostPortsPred,
        MatchNodeSelectorPred, PodFitsResourcesPred, NoDiskConflictPred,
        PodToleratesNodeTaintsPred, PodToleratesNodeNoExecuteTaintsPred, CheckNodeLabelPresencePred,
        CheckServiceAffinityPred, MaxEBSVolumeCountPred, MaxGCEPDVolumeCountPred, MaxCSIVolumeCountPred,
        MaxAzureDiskVolumeCountPred, MaxCinderVolumeCountPred, CheckVolumeBindingPred, NoVolumeZoneConflictPred,
        CheckNodeMemoryPressurePred, CheckNodePIDPressurePred, CheckNodeDiskPressurePred, MatchInterPodAffinityPred,
    }
)
```

我们可以看到，默认的预选方法是按照上图的顺序进行的，当然这个是可以被配置文件覆盖的

4. 具体的func是在这个文件里，这部分各个方法的逻辑比较简单，基本可以通过看方法名确定方法的内容。

k8s_rsc_code_v1.5 > src > k8s.io > kubernetes > pkg > scheduler > algorithm > predicates > predicates.go

进入优选逻辑

经过预选逻辑，我们可以得到filterNodes
filteredNodes, failedPredicateMap, err :=
g.findNodesThatFit(pod, nodes)

*** 如果经过预选逻辑只剩一个node，那么就将该node与pod进行绑定**

*** priorityList, err :=**

**PrioritizeNodes(pod,g.nodeInfoSnapshot.NodeInfoMap,
metaPrioritiesInterface, g.prioritizers, filteredNodes, g.extenders)**

```
func PrioritizeNodes(
    pod *v1.Pod,
    nodeNameToInfo map[string]*scheduler/nodeinfo.NodeInfo,
    meta interface{},
    priorityConfigs []priorities.PriorityConfig,
    nodes []*v1.Node,
    extenders []algorithm.SchedulerExtender,
) (schedulerapi.HostPriorityList, error) {
```

该方法入参包括当前需要调度的pod、key为node name，value为node相关信息的map、包含优选算法的各种信息、node的集合、extender(后续单独分析)

出参设计为一个HostPriority的列表，HostPriority由node的host name和对应的score构成，即每个node会得到加权的分数和。

进入该方法，首先看该方法的注释，我们大概能知道源码中优选的基本逻辑：

1. k8s定义了很多优选的方法，对应每个方法传入的pod和nodes列表中每一个node都会有0-10分的score
2. 每个方法都有自己的权重
3. 最终会对每个node计算出一个所有方法加权后的总分数，总分数最高的node即为优选的结果

```
if len(priorityConfigs) == 0 && len(extenders) == 0 {
    result := make(schedulerapi.HostPriorityList, 0, len(nodes))
    for i := range nodes {
        hostPriority, err := EqualPriorityMap(pod, meta, nodeNameToInfo[nodes[i].Name])
        if err != nil {
            return nil, err
        }
        result = append(result, hostPriority)
    }
    return result, nil
}
```

当优选算法没有设置或extender也没有配置时，此时相当于给每个node安排一个score为1

```
var (
    mu    = sync.Mutex{}
    wg    = sync.WaitGroup{}
    errs []error
)

appendError := func(err error) {
    mu.Lock()
    defer mu.Unlock()
    errs = append(errs, err)
}
```

定义并发等待的mutex，一个lock和错误集合

定义了一个err收集的方法，为了保证并发场景下写入errs同步，用了锁机制

```
results := make([]schedulerapi.HostPriorityList, len(priorityConfigs), len(priorityConfigs))
```

定义了一个result，是一个HostPriorityList的列表，len和cap与PriorityConfig数量相同

```
// PriorityConfig is a config used for a priority function.
type PriorityConfig struct {
    Name      string
    Map       PriorityMapFunction
    Reduce     PriorityReduceFunction
    // TODO: Remove it after migrating all functions to
    // Map-Reduce pattern.
    Function   PriorityFunction
    Weight     int
}
```

这是PriorityConfig的结构，包含部分旧的算法Function，也包含map-reduce模式的新的方法，Name和Weight分别代表该算法对应的名称和去权重。

```
// DEPRECATED: we can remove this when all priorityConfigs implement the
// Map-Reduce pattern.
for i := range priorityConfigs {
    if priorityConfigs[i].Function != nil {
        wg.Add(delta: 1)
        go func(index int) {
            defer wg.Done()
            var err error
            results[index], err = priorityConfigs[index].Function(pod, nodeNameToInfo, nodes)
            if err != nil {
                appendError(err)
            }
        }(i)
    } else {
        results[i] = make(schedulerapi.HostPriorityList, len(nodes))
    }
}
```

首先是旧的function的处理逻辑，我们可以看出传入的算法列表中，当旧的function存在时，采用旧的function进行优选计算，此处并发的计算每个旧的function对于node列表的score，每个function对应node列表计算出来一个score列表，存入results[index],index为传入的function序号。

接下来就是map-reduce的逻辑，其实map是映射本质上就是用传入的方法对列表中的每个元素进行f操作，而reduce就是归约，综合列表中所有元素，比如求和等得到最终结果。

```

workqueue.ParallelizeUntil(context.TODO(), workers: 16, len(nodes), func(index int) {
    nodeInfo := nodeNameToInfo[nodes[index].Name]
    for i := range priorityConfigs {
        if priorityConfigs[i].Function != nil {
            continue
        }

        var err error
        results[i][index], err = priorityConfigs[i].Map(pod, meta, nodeInfo)
        if err != nil {
            appendError(err)
            results[i][index].Host = nodes[index].Name
        }
    }
})

```

如上为map相关的逻辑，此处又延用了预选中的方法，启动16个worker从nodes列表中执行func任务，具体是什么func任务呢？我们继续看，进入循环首先判断是否有old的function，如果不为nil，则表明该算法已经用old func处理过了，此时进入map逻辑，map对每个node进行function结果的运算映射，结果存在results[i]中，i表示function序号，index表示node序号。所以此时得到的results对应每个算法每个node的score。

接下来就要对该结果进行reduce，

```

for i := range priorityConfigs {
    if priorityConfigs[i].Reduce == nil {
        continue
    }
    wg.Add(1)
    go func(index int) {
        defer wg.Done()
        if err := priorityConfigs[index].Reduce(pod, meta, nodeNameToInfo, results[index]); err != nil {
            appendError(err)
        }
        if klog.V(10) {
            for _, hostPriority := range results[index] {
                klog.Infof("pv -> %v: %v, Score: (%d)", util.GetPodFullName(pod), hostPriority.Host, priorityConfigs[index].Name, hostPriority.Score)
            }
        }
    }(i)
}
// Wait for all computations to be finished.
wg.Wait()

```

reduce进行归约，得到result中每个算法对应所有节点的score

```

for i := range nodes {
    result = append(result, schedulerapi.HostPriority{Host: nodes[i].Name, Score: 0})
    for j := range priorityConfigs {
        result[i].Score += results[j][i].Score * priorityConfigs[j].Weight
    }
}

```

外层循环遍历node，内层循环遍历算法，按照权重计算出每个node的score。

* 预选中func和map-reduce的逻辑

TODO。。。。

抢占机制

前面的优选计算完成后，我们一层层退出到schedOne的主逻辑，

```
scheduleResult, err := sched.schedule(pod)
if err != nil {
    // schedule() may have failed because the pod would not fit on any host, so we try to
    // preempt, with the expectation that the next time the pod is tried for scheduling it
    // will fit due to the preemption. It is also possible that a different pod will schedule
    // into the resources that were preempted, but this is harmless.
    if fitError, ok := err.(*core.FitError); ok {
        if !util.PodPriorityEnabled() || sched.config.DisablePreemption {
            klog.V(3).Infof("Pod priority feature is not enabled or preemption is disabled by scheduler. No preemption is performed.")
        } else {
            preemptionStartTime := time.Now()
            sched.preempt(pod, fitError)
            metrics.PreemptionAttempts.Inc()
            metrics.SchedulingAlgorithmPreemptionEvaluationDuration.Observe(metrics.SinceInSeconds(preemptionStartTime))
            metrics.DeprecatedSchedulingAlgorithmPreemptionEvaluationDuration.Observe(metrics.SinceInMicroseconds(preemptionStartTime))
            metrics.SchedulingLatency.WithLabelValues(metrics.PreemptionEvaluation).Observe(metrics.SinceInSeconds(preemptionStartTime))
            metrics.DeprecatedSchedulingLatency.WithLabelValues(metrics.PreemptionEvaluation).Observe(metrics.SinceInMicroseconds(preemptionStartTime))
        }
        // Pod did not fit anywhere, so it is counted as a failure. If preemption
        // succeeds, the pod should get counted as a success the next time we try to
        // schedule it. (hopefully)
        metrics.PodScheduleFailures.Inc()
    } else {

```

可以看到如果没有调度到合适的node，且此时没有开启抢占机制，那么就会调度失败，该方法return，等待下一次调度。

如果没有调度到合适的node，此时开启了抢占机制，此时就会进入下面的方法。

```
func (sched *Scheduler) preempt(preemptor *v1.Pod, scheduleErr error) (string, error) {
    preemptor, err := sched.config.PodPreemptor.GetUpdatedPod(preemptor)
    if err != nil {
        klog.Errorf("Error getting the updated preemptor pod object: %v", err)
        return "", err
    }

    node, victims, nominatedPodsToClear, err := sched.config.Algorithm.Preempt(preemptor, sched.config.NodeLister, scheduleErr)
    if err != nil {
        klog.Errorf("Error preempting victims to make room for %v/%v.", preemptor.Namespace, preemptor.Name)
        return "", err
    }

    var nodeName = ""
    if node != nil {
        nodeName = node.Name
        // Update the scheduling queue with the nominated pod information. Without
        // this, there would be a race condition between the next scheduling cycle
        // and the time the scheduler receives a Pod Update for the nominated pod.
        sched.config.SchedulingQueue.UpdateNominatedPodForNode(preemptor, nodeName)

        // Make a call to update nominated node name of the pod on the API server.
        err = sched.config.PodPreemptor.SetNominatedNodeName(preemptor, nodeName)
        if err != nil {
            klog.Errorf("Error in preemption process. Cannot set 'NominatedPod' on pod %v/%v: %v", preemptor.Namespace, preemptor.Name, err)
            sched.config.SchedulingQueue.DeleteNominatedPodIfExists(preemptor)
            return "", err
        }
    }
}
```

1. 首先更新pod信息，获取到该pod最新的状态

```
preemptor, err := sched.config.PodPreemptor.GetUpdatedPod(preemptor)
```

2. 进行preempt过程

```
func (g *genericScheduler) Preempt(pod *v1.Pod, nodeLister
algorithm.NodeLister, scheduleErr error) (*v1.Node, []*v1.Pod,
[]*v1.Pod, error)
```

该方法入参是待调度的pod，集群的nodeLister和前面预选优选调度失败的err，出参是抢占获得的node、以及抢占过程从该node上需要驱逐的pod列表和

todo。。。。

3. 更新NominatedPod队列，这个队列表示的是对于每一个node已经分派的pod队列，所以如果抢占成功，会将抢占过程中得到的node的NominatedPod队列中加入该pod

```
sched.config.SchedulingQueue.UpdateNominatedPodForNode(preemptor, nodeName)
```

4. 调用apiserver更新该pod的Status.NominatedNodeName，如果该更新过程失败，则从前面的队列中删除该pod

```
if err != nil {  
    klog.Errorf(format: "Error in preemption process. Cannot set 'NominatedPod' on pod %v/%v: %v", p.Namespace, p.Name, err)  
    sched.config.SchedulingQueue.DeleteNominatedPodIfExists(preemptor)  
    return "", err  
}
```

5. 由于抢占成功，所以需要将该节点上的部分pod进行驱逐。

```
for _, victim := range victims {  
    if err := sched.config.PodPreemptor.DeletePod(victim); err != nil {  
        klog.Errorf(format: "Error preempting pod %v/%v: %v", victim.Namespace, victim.Name, err)  
        return "", err  
    }  
    sched.config.Recorder.Eventf(victim, v1.EventTypeNormal, reason, "Preempted", messageFmt "by %v/%v c")  
}
```