Job 与 CronJob 控制器

接下来给大家介绍另外一类资源对象：Job，我们在日常的工作中经常都会遇到一些需要进行批量数据处理和分析的需求，当然也会有按时间来进行调度的工作，在我们的 Kubernetes 集群中为我们提供了 Job 和 CronJob 两种资源对象来应对我们的这种需求。

Job 负责处理任务，即仅执行一次的任务，它保证批处理任务的一个或多个 Pod 成功结束。而 CronJob 则就是在 Job 上加上了时间调度。

Job

我们用 Job 这个资源对象来创建一个如下所示的任务，该任务负责计算 π 到小数点后 2000 位，并将结果打印出来，此计算大约需要 10 秒钟完成。对应的资源清单如下所示：

```yaml
# job-pi.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
        - name: pi
          image: perl:5.34.0
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

我们可以看到 Job 中也是一个 Pod 模板，和之前的 Deployment、StatefulSet 之类的是一致的，只是 Pod 中的容器要求是一个任务，而不是一个常驻前台的进程了，因为需要退出，另外值得注意的是 Job 的 RestartPolicy 仅支持 Never 和 OnFailure 两种，不支持 Always，我们知道 Job 就相当于来执行一个批处理任务，执行完就结束了，如果支持 Always 的话是不是就陷入了死循环了？

直接创建这个 Job 对象：

```
☸ ➜ kubectl apply -f job-pi.yaml
job.batch/pi created
☸ ➜  kubectl get job
NAME   COMPLETIONS   DURATION   AGE
pi     0/1           6s         6s
☸ ➜ kubectl get pods
NAME             READY   STATUS              RESTARTS      AGE
pi-4krgt         0/1     ContainerCreating   0             14s
```

Job 对象创建成功后，我们可以查看下对象的详细描述信息：

```
☸ ➜ kubectl describe job pi
Name:             pi
Namespace:        default
```

```
Selector:         controller-uid=951b4719-8f6b-4481-b15a-99d865c81840
Labels:           controller-uid=951b4719-8f6b-4481-b15a-99d865c81840
                  job-name=pi
Annotations:      batch.kubernetes.io/job-tracking:
Parallelism:      1
Completions:      1
Completion Mode:  NonIndexed
Start Time:       Thu, 15 Dec 2022 15:33:09 +0800
Pods Statuses:    1 Active (0 Ready) / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=951b4719-8f6b-4481-b15a-99d865c81840
           job-name=pi
  Containers:
   pi:
    Image:      perl:5.34.0
    Port:       <none>
    Host Port:  <none>
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  50s   job-controller  Created pod: pi-4krgt
```

可以看到，Job 对象在创建后，它的 Pod 模板，被自动加上了一个 controller-uid=< 一个随机字符串 > 这样的 Label 标签，而这个 Job 对象本身，则被自动加上了这个 Label 对应的 Selector，从而保证了 Job 与它所管理的 Pod 之间的匹配关系。而 Job 控制器之所以要使用这种携带了 UID 的 Label，就是为了避免不同 Job 对象所管理的 Pod 发生重合。

我们可以看到隔一会儿后 Pod 变成了 Completed 状态，这是因为容器的任务执行完成正常退出了，我们可以查看对应的日志:

```
☸ ➜ kubectl logs pi-4krgt
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803
48253421170679821480865132823066470938446095505822317253594081284811174502841027019385211
10555964462294895493038196442881097566593344612847564823378678316527120190914564856692346
60348610454326648213393607260249141273724587006606315588174881520920962829254091715364367
89259036001133053054882046652138414695194151160943305727036575959195309218611738193261179
31051185480744623799627495673518857527248912279381830119491298336733624406566430860213949
46395224737190702179860943702770539217176293176752384674818467669405132000568127145263560
82778577134275778960917363717872146844090122495343014654958537105079227968925892354201995
61121290219608640344181598136297747713099605187072113499999983729780499510597317328160963
18595024459455346908302642522308253344685035261931188171010003137838752886587533208381420
61717766914730359825349042875546873115956286388235378759375195778185778053217122668066130
01927876611195909216420198938095257201065485863278865936153381827968230301952035301852968
99577362259941389124972177528347913151557485724245415069595082953311686172785588907509838
17546374649393192550604009277016711390098488240128583616035637076601047101819429555961989
46767837449448255379774726847104047534646208046684259069491293313677028989152104752162056
96602405803815019351125338243003558764024749647326391419927260426992279678235478163600934
17216412199245863150302861829745557067498385054945885869269956909272107975093029553211653
44987202755960236480665499119881834797753566369807426542527862551818417574672890977772793
80008164706001614524919217321721477235014144197356854816136115735255213347574184946843852
33239073941433345477624168625189835694855620992192221842725502542568876717904946016534668
04988627232791786085784383827967976681454100953883786360950680064225125205117392984896084
12848862694560424196528502221066118630674427862203919494504712371378696095636437191728746
77646575739624138908658326459958133904780275901
☸ ➜ kubectl get pod
NAME       READY   STATUS      RESTARTS   AGE
pi-4krgt   0/1     Completed   0          11m
```

上面我们这里的 Job 任务对应的 Pod 在运行结束后，会变成 Completed 状态，但是如果执行任务的 Pod 因为某种原因一直没有结束怎么办呢？同样我们可以在 Job 对象中通过设置字段 spec.activeDeadlineSeconds 来限制任务运行的最长时间，比如：

```yaml
spec:
  activeDeadlineSeconds: 100
```

那么当我们的任务 Pod 运行超过了 100s 后，这个 Job 的所有 Pod 都会被终止，并且的终止原因会变成 DeadlineExceeded 。

如果的任务执行失败了，会怎么处理呢，这个和定义的 restartPolicy 有关系，比如定义如下所示的 Job 任务，定义 restartPolicy: Never 的重启策略：

```yaml
# job-failed-demo.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: job-failed-demo
spec:
  template:
    spec:
      containers:
        - name: test-job
          image: busybox
          command: ["echo123", "test failed job!"]
      restartPolicy: Never
```

直接创建上面的资源对象：

```
☸ ➜ kubectl apply -f job-failed-demo.yaml
job.batch/job-failed-demo created
☸ ➜ kubectl get pod
NAME                    READY   STATUS       RESTARTS   AGE
job-failed-demo-5hrdj   0/1     StartError   0          55s
job-failed-demo-bqxsb   0/1     StartError   0          49s
job-failed-demo-m29nr   0/1     StartError   0          39s
job-failed-demo-r8cfz   0/1     StartError   0          44s
job-failed-demo-v6228   0/1     StartError   0          35s
job-failed-demo-v7q2c   0/1     StartError   0          30s
job-failed-demo-zqk68   0/1     StartError   0          25s
```

可以看到当我们设置成 `Never` 重启策略的时候，Job 任务执行失败后会不断创建新的 Pod，但是不会一直创建下去，会根据 `spec.backoffLimit` 参数进行限制，默认为 6，通过该字段可以定义重建 Pod 的次数。

但是如果我们设置的 `restartPolicy: OnFailure` 重启策略，则当 Job 任务执行失败后不会创建新的 Pod 出来，只会不断重启 Pod，比如将上面的 Job 任务 `restartPolicy` 更改为 `OnFailure` 后查看 Pod：

```
☸ ➜ kubectl get pods
NAME                    READY   STATUS             RESTARTS      AGE
job-failed-demo-6l8vn   0/1     CrashLoopBackOff   3 (22s ago)   77s
```

除此之外，我们还可以通过设置 `spec.parallelism` 参数来进行并行控制，该参数定义了一个 Job 在任意时间最多可以有多少个 Pod 同时运行。并行性请求（`.spec.parallelism`）可以设置为任何非负整数，如果未设置，则默认为 1，如果设置为 0，则 Job 相当于启动之后便被暂停，直到此值被增加。

`spec.completions` 参数可以定义 Job 至少要完成的 Pod 数目。如下所示创建一个新的 Job 任务，设置允许并行数为 2，至少要完成的 Pod 数为 8：

```yaml
# job-para-demo.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: job-para-test
spec:
```

```yaml
  parallelism: 2
  completions: 8
  template:
    spec:
      containers:
        - name: test-job
          image: busybox
          command: ["echo", "test paralle job!"]
      restartPolicy: Never
```

创建完成后查看任务状态：

```
☸ ➜ kubectl get pods
NAME                   READY   STATUS              RESTARTS   AGE
job-para-test-qcmms    0/1     ContainerCreating   0          3s
job-para-test-rsdbv    0/1     ContainerCreating   0          3s
☸ ➜ kubectl get job
NAME            COMPLETIONS   DURATION   AGE
job-para-test   8/8           15s        15s
☸ ➜ kubectl get pod
NAME                   READY   STATUS      RESTARTS   AGE
job-para-test-bzpj7    0/1     Completed   0          31s
job-para-test-c2zll    0/1     Completed   0          23s
job-para-test-c47fg    0/1     Completed   0          17s
job-para-test-hsf5p    0/1     Completed   0          25s
job-para-test-q9j6j    0/1     Completed   0          29s
job-para-test-qcmms    0/1     Completed   0          36s
job-para-test-qfmq7    0/1     Completed   0          19s
job-para-test-rsdbv    0/1     Completed   0          36s
```

可以看到一次可以有 2 个 Pod 同时运行，需要 8 个 Pod 执行成功，如果不是 8 个成功，那么会根据 `restartPolicy` 的策略进行处理，可以认为是一种检查机制。

此外带有确定完成计数的 Job，即 `.spec.completions` 不为 null 的 Job， 都可以在其 `.spec.completionMode` 中设置完成模式：

- `NonIndexed（默认值）`：当成功完成的 Pod 个数达到 `.spec.completions` 所设值时认为 Job 已经完成。换言之，每个 Job 完成事件都是独立无关且同质的。要注意的是，当 `.spec.completions` 取值为 null 时，Job 被默认处理为 `NonIndexed` 模式。
- `Indexed`：Job 的 Pod 会获得对应的完成索引，取值为 0 到 `.spec.completions-1`，该索引可以通过三种方式获取：
  - Pod 的注解 `batch.kubernetes.io/job-completion-index`。
  - 作为 Pod 主机名的一部分，遵循模式 `$(job-name)-$(index)`。当你同时使用带索引的 Job（Indexed Job）与服务（Service），Job 中的 Pod 可以通过 DNS 使用确切的主机名互相寻址。
  - 对于容器化的任务，在环境变量 `JOB_COMPLETION_INDEX` 中可以获得。

当每个索引都对应一个成功完成的 Pod 时，Job 被认为是已完成的。

索引完成模式

下面我们将运行一个使用多个并行工作进程的 Kubernetes Job，每个 worker 都是在自己的 Pod 中运行的不同容器。Pod 具有控制平面自动设置的索引编号（index number），这些编号使得每个 Pod 能识别出要处理整个任务的哪个部分。

Pod 索引在注解 batch.kubernetes.io/job-completion-index 中呈现，具体表示为一个十进制值字符串。为了让容器化的任务进程获得此索引，我们可以使用 downward API 机制来获取注解的值。而且控制平面会自动设置 Downward API 在 JOB_COMPLETION_INDEX 环境变量中暴露索引。

我们这里创建一个如下所示的 Job 资源清单文件：

```yaml
# job-indexed.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: indexed-job
spec:
  completions: 5
  parallelism: 3
  completionMode: Indexed
  template:
    spec:
      restartPolicy: Never
      initContainers:
        - name: "input"
          image: "bash"
          command:
            - "bash"
            - "-c"
            - |
              items=(foo bar baz qux xyz)
              echo ${items[$JOB_COMPLETION_INDEX]} > /input/data.txt
          volumeMounts:
            - mountPath: /input
              name: input
      containers:
        - name: "worker"
          image: "busybox"
          command:
            - "rev"
            - "/input/data.txt"
          volumeMounts:
            - mountPath: /input
              name: input
      volumes:
        - name: input
          emptyDir: {}
```

在该资源清单中我们设置了 completionMode: Indexed，表示这是一个 Indexed 完成模式的 Job 任务。这里我们还使用了 Job 控制器为所有容器设置的内置 JOB_COMPLETION_INDEX 环境变量。 Init 容器将索引映射到一个静态值，并将其写入一个文件，该文件通过 emptyDir 卷 与运行 worker 的容器共享。

> rev（reverse）命令用于将文件中的每行内容以字符为单位反序输出，即第一个字符最后输出，最后一个字符最先输出，以此类推。

直接创建该资源对象即可：

```
☸ ➜ kubectl apply -f job-indexed.yaml
☸ ➜ kubectl get job
NAME          COMPLETIONS   DURATION   AGE
indexed-job   0/5           4s         4s
```

当你创建此 Job 时，控制平面会创建一系列 Pod，你指定的每个索引都会运行一个 Pod。.spec.parallelism 的值决定了一次可以运行多少个 Pod， 而 .spec.completions 决定了 Job 总共创建了多少个 Pod。

因为 .spec.parallelism 小于 .spec.completions， 所以控制平面在启动更多 Pod 之前，将等待第一批的某些 Pod 完成。

```
☸ ➜ kubectl get pods
NAME                  READY   STATUS     RESTARTS   AGE
indexed-job-0-th2ks   0/1     Init:0/1   0          6s
indexed-job-1-8jbrp   0/1     Init:0/1   0          6s
indexed-job-2-zpv67   0/1     Init:0/1   0          6s
☸ ➜ kubectl get pods
NAME                  READY   STATUS      RESTARTS   AGE
indexed-job-0-th2ks   0/1     Completed   0          63s
indexed-job-1-8jbrp   0/1     Completed   0          63s
indexed-job-2-zpv67   0/1     Completed   0          63s
indexed-job-3-khcq8   0/1     Completed   0          26s
indexed-job-4-nzdhr   0/1     Completed   0          24s
```

创建 Job 后，稍等片刻，就能检查进度：

```
☸ ➜ kubectl describe jobs/indexed-job
Name:                indexed-job
Namespace:           default
Selector:            controller-uid=06708c09-0432-4fac-bd5c-f0b3041bf9b7
Labels:              controller-uid=06708c09-0432-4fac-bd5c-f0b3041bf9b7
                     job-name=indexed-job
Annotations:         batch.kubernetes.io/job-tracking:
Parallelism:         3
Completions:         5
Completion Mode:     Indexed
Start Time:          Thu, 15 Dec 2022 16:12:38 +0800
Completed At:        Thu, 15 Dec 2022 16:13:28 +0800
Duration:            50s
Pods Statuses:       0 Active (0 Ready) / 5 Succeeded / 0 Failed
Completed Indexes:   0-4
# ......
Events:
  Type    Reason            Age                From            Message
  ----    ------            ----               ----            -------
```

```
  Normal  SuccessfulCreate  100s               job-controller  Created pod: indexed-job-
0-th2ks
  Normal  SuccessfulCreate  100s               job-controller  Created pod: indexed-job-
2-zpv67
  Normal  SuccessfulCreate  100s               job-controller  Created pod: indexed-job-
1-8jbrp
  Normal  SuccessfulCreate  63s                job-controller  Created pod: indexed-job-
3-khcq8
  Normal  SuccessfulCreate  61s                job-controller  Created pod: indexed-job-
4-nzdhr
  Normal  Completed         50s (x2 over 50s)  job-controller  Job completed
```

这里我们使用的每个索引的自定义值运行 Job，我们可以检查其中一个 Pod 的输出：

```
☸ ➜ kubectl logs indexed-job-3-khcq8
xuq
```

我们在初始化容器中执行了如下所示的命令：

```
items=(foo bar baz qux xyz)
echo ${items[$JOB_COMPLETION_INDEX]} > /input/data.txt
```

当 JOB_COMPLETION_INDEX=3 的时候表示我们将 items[3] 的 qux 值写入到了 /input/data.txt 文件中，然后通过 volume 共享，在主容器中我们通过 rev 命令将其反转，所以输出结果就位 xuq 了。

上面我们这个示例中每个 Pod 只做一小部分工作（反转一个字符串）。在实际工作中肯定比这复杂，比如你可能会创建一个基于场景数据制作 60 秒视频任务的 Job，此视频渲染 Job 中的每个工作项都将渲染该视频剪辑的特定帧，索引完成模式意味着 Job 中的每个 Pod 都知道通过从剪辑开始计算帧数，来确定渲染和发布哪一帧，这样就可以大大提高工作任务的效率。

CronJob

CronJob 其实就是在 Job 的基础上加上了时间调度，我们可以在给定的时间点运行一个任务，也可以周期性地在给定时间点运行。这个实际上和我们 Linux 中的 crontab 就非常类似了。

一个 CronJob 对象其实就对应中 crontab 文件中的一行，它根据配置的时间格式周期性地运行一个 Job，格式和 crontab 也是一样的。

crontab 的格式为：分 时 日 月 星期 要运行的命令 。

```
# ┌──────────── 分钟 (0 - 59)
# │ ┌────────── 小时 (0 - 23)
# │ │ ┌──────── 月的某天 (1 - 31)
# │ │ │ ┌────── 月份 (1 - 12)
# │ │ │ │ ┌──── 周的某天 (0 - 6) (周日到周一；在某些系统上，7 也是星期日)
# │ │ │ │ │                或者是 sun, mon, tue, web, thu, fri, sat
# │ │ │ │ │
# │ │ │ │ │
# * * * * *
```

- 第 1 列分钟 0 ~ 59

- 第 2 列小时 0 ~ 23
- 第 3 列日 1 ~ 31
- 第 4 列月 1 ~ 12
- 第 5 列星期 0 ~ 7（0 和 7 表示星期天）
- 第 6 列要运行的命令

所有 CronJob 的 schedule 时间都是基于 kube-controller-manager 的时区，如果你的控制平面在 Pod 中运行了 kube-controller-manager， 那么为该容器所设置的时区将会决定 CronJob 的控制器所使用的时区。官方并不支持设置如 CRON_TZ 或者 TZ 等变量，这两个变量是用于解析和计算下一个 Job 创建时间所使用的内部库中一个实现细节，不建议在生产集群中使用它。

但是如果在 kube-controller-manager 中启用了 CronJobTimeZone 这个 Feature Gates，那么我们就可以为 CronJob 指定一个时区（如果你没有启用该特性门控，或者你使用的是不支持试验性时区功能的 Kubernetes 版本，集群中所有 CronJob 的时区都是未指定的）。启用该特性后，你可以将 spec.timeZone 设置为有效时区名称。

现在，我们用 CronJob 来管理我们上面的 Job 任务，定义如下所示的资源清单：

```yaml
# cronjob-demo.yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: cronjob-demo
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
            - name: hello
              image: busybox
              args:
                - "bin/sh"
                - "-c"
                - "for i in 9 8 7 6 5 4 3 2 1; do echo $i; done"
```

这里的 Kind 变成了 CronJob 了，要注意的是 .spec.schedule 字段是必须填写的，用来指定任务运行的周期，格式就和 crontab 一样，另外一个字段是 .spec.jobTemplate，用来指定需要运行的任务，格式当然和 Job 是一致的。还有一些值得我们关注的字段 .spec.successfulJobsHistoryLimit（默认为 3）和 .spec.failedJobsHistoryLimit（默认为 1），表示历史限制，是可选的字段，指定可以保留多少完成和失败的 Job。然而，当运行一个 CronJob 时，Job 可以很快就堆积很多，所以一般推荐设置这两个字段的值，如果设置限制的值为 0，那么相关类型的 Job 完成后将不会被保留。

我们直接新建上面的资源对象：

```
☸ ➜ kubectl apply -f cronjob-demo.yaml
cronjob "cronjob-demo" created
```

然后可以查看对应的 Cronjob 资源对象：

```
☸ ➜ kubectl get cronjob
NAME           SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
cronjob-demo   */1 * * * *   False     1        2s              12s
```

稍微等一会儿查看可以发现多了几个 Job 资源对象，这个就是因为上面我们设置的 CronJob 资源对象，每 1 分钟执行一个新的 Job：

```
☸ ➜ kubectl get job
NAME                     COMPLETIONS   DURATION   AGE
cronjob-demo-27851555    1/1           7s         2m30s
cronjob-demo-27851556    1/1           7s         90s
cronjob-demo-27851557    1/1           6s         30s
☸ ➜ kubectl get pods
NAME                           READY   STATUS      RESTARTS   AGE
cronjob-demo-27851555-9x7hr    0/1     Completed   0          2m43s
cronjob-demo-27851556-9tdpl    0/1     Completed   0          103s
cronjob-demo-27851557-n6jdz    0/1     Completed   0          43s
```

这个就是 CronJob 的基本用法，一旦不再需要 CronJob，我们可以使用 kubectl 命令删除它：

```
☸ ➜ kubectl delete cronjob cronjob-demo
cronjob "cronjob-demo" deleted
```

不过需要注意的是这将会终止正在创建的 Job，但是运行中的 Job 将不会被终止，不会删除 Job 或 它们的 Pod。

> 思考：那如果我们想要在每个节点上去执行一个 Job 或者 Cronjob 又该怎么来实现呢？