Artigo
Vinicius Maranh... · Jul. 6, 2021 11min de leitura

Escalando horizontalmente o IRIS no Kubernetes de acordo com o tamanho da fila - Parte 2

Introdução

Suponha que você desenvolveu uma nova aplicação utilizando a parte de Interoperabilidade do InterSystems IRIS e você tem certeza de que será um sucesso! No entanto, você ainda não tem um número concreto de quantas pessoas irão utilizá-la. Além disso, pode haver dias específicos em que há mais pessoas utilizando sua aplicação e dias em que quase ninguém irá acessar. Deste modo, você necessita de que sua aplicação seja escalável!

O Kubernetes já nos ajuda bastante nesta tarefa com um componente chamado Horizontal Pod Autoscaler, que permite escalar horizontalmente uma aplicação baseado em uma métrica específica. Através do componente " Metrics Server " do próprio Kubernetes, é possível obter métricas como utilização de CPU ou de memória. Mas, e se você necessitar escalar sua aplicação utilizando outra métrica, como por exemplo o tamanho da fila de todos os " Business Hosts " presentes em uma determinada Produção da Interoperabilidade do IRIS? Por se tratar de uma métrica customizada, será necessário desenvolver um serviço que irá expor essa métrica customizada para o Kubernetes.

Nesta série de artigos, veremos em mais detalhes como funciona o processo de escalonamento horizontal do Kubernetes. Na parte 1, será abordado o funcionamento do componente responsável pelo escalonamento horizontal do Kubernetes e a criação e exposição de uma métrica customizada do IRIS no formato esperado pelo Prometheus. Já na parte 2, abordaremos como configurar o Prometheus para ler essa métrica e como configurar o prometheus-adapter para expor essa métrica ao Kubernetes.

O projeto utilizado para este artigo está totalmente disponível no GitHub em:

https://github.com/vmrcastro/iris-autoscale

Prometheus – Configurando a leitura destas métricas

Na parte 1 deste artigo, criamos e expusemos uma métrica customizada no IRIS. Agora, necessitamos configurar o Prometheus para ler essa métrica e armazenar em seu banco de dados. No repositório " iris-autoscale " do GitHub que estamos utilizando, já há um arquivo yaml pronto para fazer o deploy do Prometheus já com as configurações necessárias, no seguinte diretório:
prometheus/manifests/00-prometheus-deploy.yml

A parte mais importante deste arquivo yaml é o ConfigMap. Nele é onde definimos o arquivo prometheus.yml que contém as configurações do Prometheus. Este é o conteúdo do arquivo prometheus.yml:

```
    # my global config

    global:

      scrape_interval:     15s # Set the scrape interval to every 15 seconds. Default
 is every 1 minute.

      evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is ever
y 1 minute.
```

```
      # scrape_timeout is set to the global default (10s).


    # Alertmanager configuration

    alerting:

      alertmanagers:

      - static_configs:

        - targets:

          # - alertmanager:9093


    # Load rules once and periodically evaluate them according to the global 'evaluat
ion_interval'.

    rule_files:

      # - "first_rules.yml"

      # - "second_rules.yml"


    # A scrape configuration containing exactly one endpoint to scrape:

    # Here it's Prometheus itself.

    scrape_configs:

      # The job name is added as a label `job=<job_name>` to any timeseries scraped f
rom this config.

      - job_name: 'prometheus'


        # metrics_path defaults to '/metrics'

        metrics_path: '/api/monitor/metrics'

        # scheme defaults to 'http'.


        kubernetes_sd_configs:

        - role: pod

        relabel_configs:

        # Example relabel to scrape only pods that have

        # "example.io/should_be_scraped = true" annotation.

        #  - source_labels: [__meta_kubernetes_pod_annotation_example_io_should_be_sc
raped]
```

```
        #    action: keep

        #    regex: true

        #

        # Example relabel to customize metric path based on pod

        # "example.io/metric_path = <metric path>" annotation.

        #  - source_labels: [__meta_kubernetes_pod_annotation_example_io_metric_path]

        #    action: replace

        #    target_label: __metrics_path__

        #    regex: (.+)

        #

        # Example relabel to scrape only single, desired port for the pod

        # based on pod "example.io/scrape_port = <port>" annotation.

        #  - source_labels: [__address__, __meta_kubernetes_pod_annotation_example_io
_scrape_port]

        #    action: replace

        #    regex: ([^:]+)(?::\d+)?;(\d+)

        #    replacement: $1:$2

        #    target_label: __address__

        - action: labelmap

          regex: __meta_kubernetes_pod_label_(.+)

        - source_labels: [__meta_kubernetes_namespace]

          action: replace

          target_label: kubernetes_namespace

        - source_labels: [__meta_kubernetes_pod_name]

          action: replace

          target_label: kubernetes_pod_name
```

Não iremos abordar aqui todos os parâmetros definidos, até mesmo porque muitos já possuem um comentário autoexplicativo. No entanto, se restar alguma dúvida, é possível consultar a documentação oficial do Prometheus disponível em:

https://prometheus.io/docs/prometheus/latest/configuration/configuration/

Vale ressaltar, no entanto, alguns parâmetros. Em " metricspath " , vamos indicar o caminho da URL que o Prometheus irá enviar a requisição para buscar as métricas. Portanto, definimos " /api/monitor/metrics " . Desta forma, só nos resta indicar ao Prometheus qual é, ou quais são os endereços e portas do IRIS. No entanto, não podemos especificar um valor fixo para o hostname do(s) POD(s) do IRIS, já que eles não serão fixos! Lembrem-se que os PODs serão adicionados automaticamente mediante um aumento da fila e serão removidos também automaticamente mediante uma diminuição da fila. Por sorte, o Prometheus tem a capacidade de chamar as APIs do Kubernetes para buscar quais são os PODs presentes no cluster naquele momento, ficando sempre sincronizado com o estado do cluster de Kubernetes. Por isso que utilizamos a seção " kubernetessdconfigs " na configuração do Prometheus. Como o Prometheus rodará no mesmo cluster de Kubernetes que estarão rodando os PODs que desejamos monitorar, não precisamos especificar nenhuma informação para o Prometheus de como se comunicar com o Kubernetes, tudo isso já será detectado automaticamente.

Outro ponto importante: quando tivermos com várias instâncias do IRIS rodando, cada uma delas terá sua métrica " queuesize " . Como saberemos no Prometheus qual instancia determinada métrica " queuesize " está se referindo? Podemos solucionar isso fazendo com que o Prometheus adicione em cada métrica um " label " para identificar de qual POD a métrica está se referindo. Por padrão, para cada POD que o Prometheus descobriu no cluster Kubernetes, há uma série de " meta labels " que podem ser incorporados nas métricas, através de configurações de " relabeling " . É isso que as seguintes linhas do arquivo de configuração do Prometheus fazem:

```
        - source_labels: [__meta_kubernetes_namespace]

          action: replace

          target_label: kubernetes_namespace

        - source_labels: [__meta_kubernetes_pod_name]

          action: replace

          target_label: kubernetes_pod_name
```

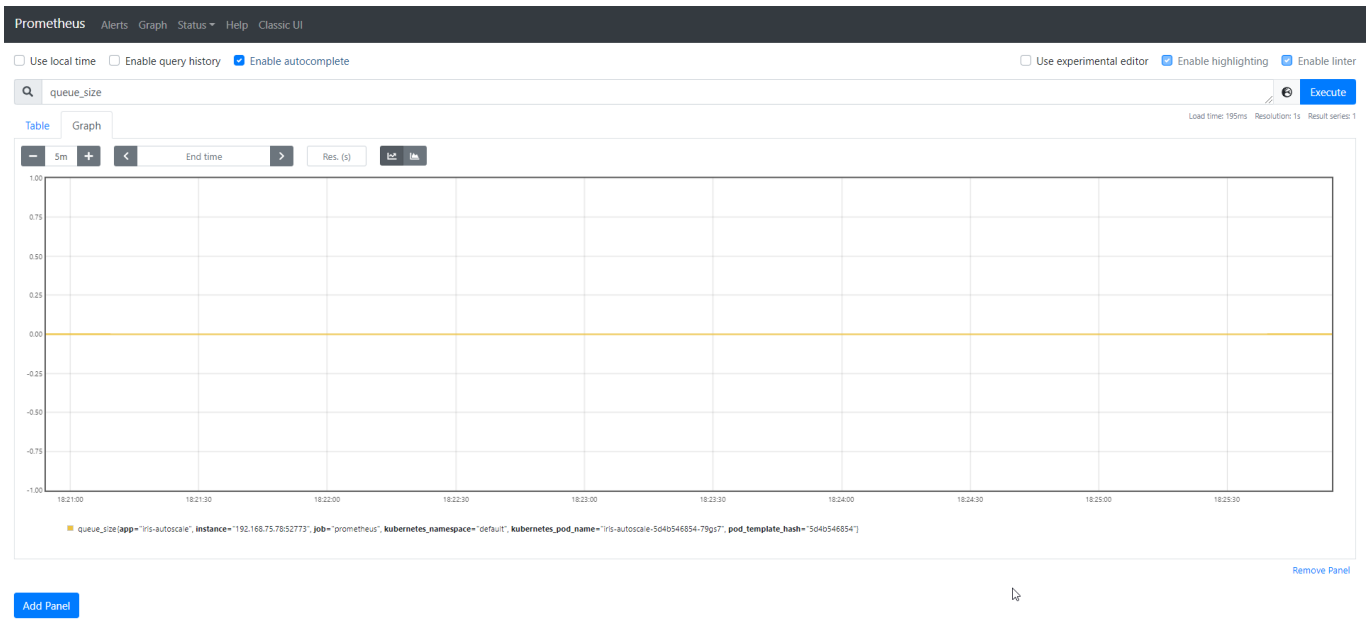
Neste caso, estamos adicionando para cada métrica, um label que se chama " kubernetesnamespace " que contém o nome do namespace do Kubernetes em que o POD está rodando e um label denominado " kubernetespodname " que contém o nome do POD.

Com o Prometheus rodando, é possível verificar se as configurações estão corretas. Basta acessar a seguinte URL no seu browser:
http://<IPdoPrometheus>:<PortadoPrometheus>/graph

Obs: Por padrão, é criado um " service " do Kubernetes do tipo Load Balancer para acessar o Prometheus. Você pode descobrir o IP do Prometheus verificando o campo " External IP " do service " prometheus-svc " .

É possível então, escrever o nome da métrica " queuesize " no campo de busca e clicar no botão " Execute " .

Ao clicar na aba " Graph " , se estiver tudo certo, você deverá visualizar uma linha amarela no valor 0, como mostra a figura abaixo.



Prometheus-Adapter - expondo métricas ao Kubernetes

Com o Prometheus configurado e coletando as métricas dos PODs do IRIS, agora só nos resta configurar nosso " metric API server " , no nosso caso o prometheus-adapter, para expor essas métricas de uma maneira que o Kubernetes entenda. Assim como no deploy do Prometheus, o deploy do prometheus-adapter no Kubernetes já está pronto, sendo realizado através dos arquivos yaml no diretório prometheus/manifests/ no repositório " iris-autoscale " . No entanto, vale a pena comentarmos alguns detalhes.

Primeiramente, para o prometheus adapter funcionar, é necessário especificar qual é o endereço e porta do Prometheus. Isso é especificado no arquivo " custom-metrics-apiserver-deployment.yaml " , no parâmetro " prometheus-url " na seção " args " do container " custom-metrics-apiserver " .

Assim como no deploy do Prometheus, as configurações do prometheus-adapter se encontram em um " configMap " , definido dentro do arquivo " custom-metrics-config-map.yaml " . Dentro do " configMap " , é definido o arquivo " config.yaml " que contém as seguintes configurações:

```
rules:

- seriesQuery: 'queue_size{kubernetes_namespace!="",kubernetes_pod_name!=""}'

  resources:

    overrides:

      kubernetes_namespace: {resource: "namespace"}

      kubernetes_pod_name: {resource: "pod"}

  metricsQuery: '<<.Series>>{<<.LabelMatchers>>}'
```

Como é possível observar, em " seriesQuery " é definido a consulta que será enviada ao Prometheus para descobrir quais são as métricas disponíveis. Como neste exemplo só vamos trabalhar com a métrica " queuesize " , vamos especificá-la diretamente. Portanto, nesta query estamos buscando todas as métricas denominadas " queuesize "

disponíveis, desde que " kubernetesnamespace " e " kubernetespodname " não sejam nulos. Na seção " overrides " , vamos especificar ao prometheus-adapter de qual label deve ser extraído o namespace do Kubernetes e o nome do POD. Finalmente, em " metricsQuery " , vamos especificar qual é a consulta que desejamos enviar ao Prometheus para obter o valor da métrica a ser exposta para o Kubernetes. É possível, por exemplo, utilizar alguma função de agregação para transformar o valor antes de expô-la ao Kubernetes. No entanto, no nosso caso, queremos expor a métrica exatamente como ela se encontra no Prometheus, por isso especificamos apenas metricsQuery: '<<.Series>>{<<.LabelMatchers>>}'

Ao final desta etapa é possível testar se está tudo certo simulando a requisição que o Horizontal Pod Autoscaler faria para a Custom Metrics API através do seguinte comando:
$ kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1/namespaces/default/pods/*/queuesize"

A resposta esperada seria algo do parecido com o seguinte:

```
{
        "kind": "MetricValueList",
        "apiVersion": "custom.metrics.k8s.io/v1beta1",
        "metadata": {
                "selfLink": "/apis/custom.metrics.k8s.io/v1beta1/namespaces/default/po
ds/%2A/queue_size"
        },
        "items": [{
                "describedObject": {
                        "kind": "Pod",
                        "namespace": "default",
                        "name": "iris-autoscale-5d4b546854-79gs7",
                        "apiVersion": "/v1"
                },
                "metricName": "queue_size",
                "timestamp": "2021-05-03T18:31:03Z",
                "value": "0",
                "selector": null
        }]
}
```

InterSystems IRIS - Produção criada para testes

A produção criada neste projeto é uma produção bem simples, apenas para testes. A ideia geral é expor um

endpoint REST que ao receber uma requisição, envia uma mensagem assíncrona para qualquer um dos três Business Operations disponíveis (denominados " Teste 1 ", " Teste 2 " e " Teste 3 ") de maneira randômica. Esses " Business Operations " são bem simples e, a única coisa que eles fazem é um " Hang " de 0 a 30 segundos, também de maneira randômica. Obviamente, essa produção não faz nenhum sentido, a ideia aqui é apenas simular o enfileiramento de mensagens no IRIS.

A requisição que deve ser enviada ao IRIS é a seguinte:
$ curl 'http://<IPdoIRIS>:<PortaWebServerIRIS>/api/autoscale/request'

Se der tudo certo, a seguinte resposta retornará juntamente com um código HTTP status de " 200 – OK ":
{"responseText":"The request has been sended!"}

Fazendo o deploy e testando o projeto

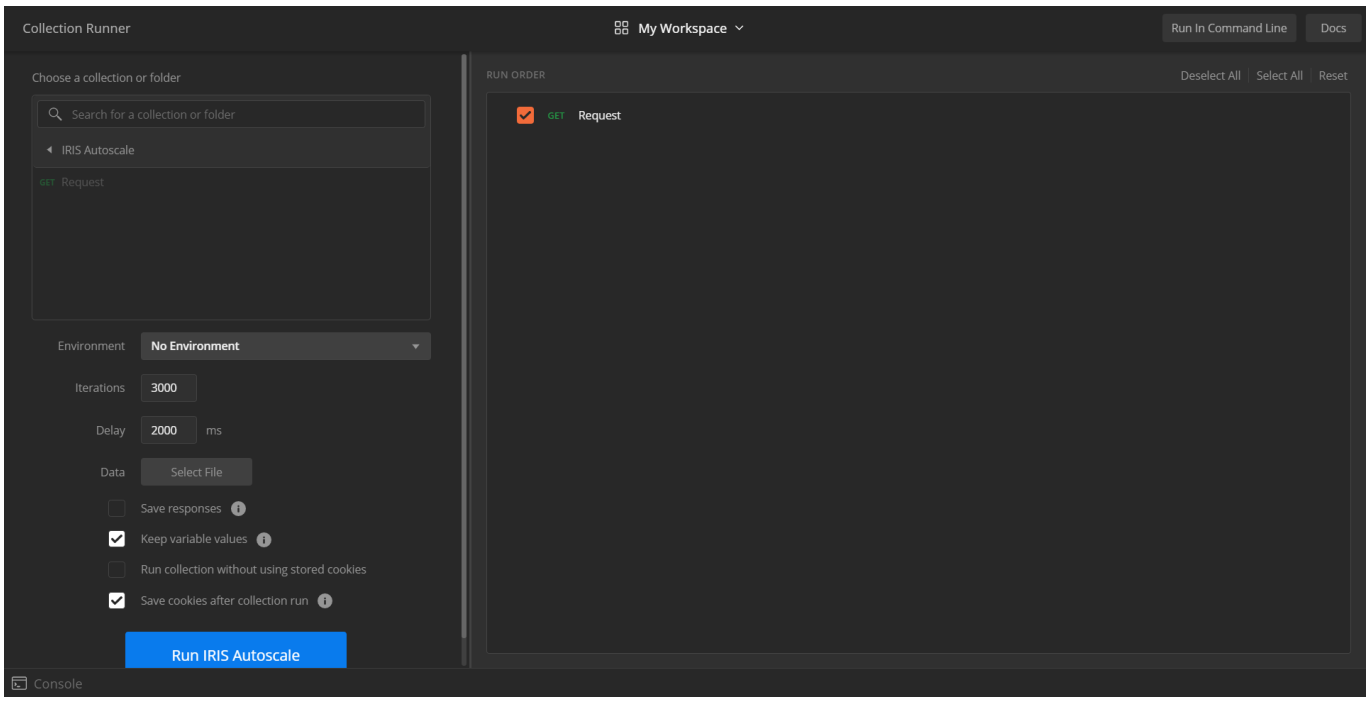
Para fazer o deploy do projeto em seu cluster de Kubernetes, basta clonar o repositório " iris-autoscale ", indicado na Introdução deste artigo e executar os seguintes comandos a partir da raiz do repositório clonado:

```
$ kubectl apply -f prometheus/manifests

$ kubectl apply -f kubernetes.yml
```

Para verificar se tudo correu bem, vamos dar uma olhada nos pods. Verifique se os seguintes PODs estão presentes com status de " Running " e com " READY 1/1 ":

```
$ kubectl get pods

NAME                                          READY   STATUS    RESTARTS   AGE

iris-autoscale-5b9b77577f-jb2z4               1/1     Running    0         119m

prometheus-b974955c5-jwrhd                    1/1     Running    0         119m

$ kubectl get pods -n custom-metrics

NAME                                          READY   STATUS    RESTARTS   AGE

custom-metrics-apiserver-776cf7cb8b-4v5b5     1/1     Running   0          121m
```

Se tudo correu bem, só nos resta testar o escalonamento automático. Para isso, utilizaremos o Collection Runner do PostMan para ficar enviando várias requisições ao Service do Kubernetes " iris-autoscale-svc ". Vamos enviar um total de 3000 requisições com um espaçamento de 2000 ms entre elas:



Podemos acompanhar as filas crescendo através da interface do Prometheus do lado esquerdo da tela e novos PODs do IRIS sendo criados e deletados automaticamente, como mostra o vídeo a seguir:

Conclusão

Neste artigo, abordamos todos os componentes necessários para escalar horizontalmente o InterSystems IRIS utilizando uma métrica customizada, no nosso caso a métrica " queuesize " que nos informa qual é o tamanho total das filas de uma produção em um namespace do IRIS. Por se tratar de uma métrica customizada, foi necessário fazer o deploy e configurar o Prometheus, para coletar essa métrica de cada instância de IRIS em execução, e o prometheus-adapter, para expor essa métrica no formato esperado pelo Kubernetes. Com isso, foi possível definir essa métrica na configuração do Kubernetes Horizontal Pod Autoscaler para que o escalonamento de nossas instâncias de IRIS seja realizado automaticamente.

Referências

https://learnk8s.io/autoscaling-apps-kubernetes

https://prometheus.io/docs/introduction/firststeps/

https://prometheus.io/docs/prometheus/latest/configuration/configuration/

https://github.com/kubernetes-sigs/prometheus-adapter/blob/master/docs/config-walkthrough.md

https://github.com/kubernetes-sigs/prometheus-adapter/blob/master/docs/walkthrough.md

https://github.com/prometheus/prometheus/blob/release-2.26/documentation/examples/prometheus-kubernetes.yml

#Conteinerização #Entrega Contínua #Integração Contínua #Kubernetes #Nuvem #InterSystems IRIS

URL de origem:https://pt.community.intersystems.com/post/escalando-horizontalmente-o-iris-no-kubernetes-de-acordo-com-o-tamanho-da-fila-parte-2