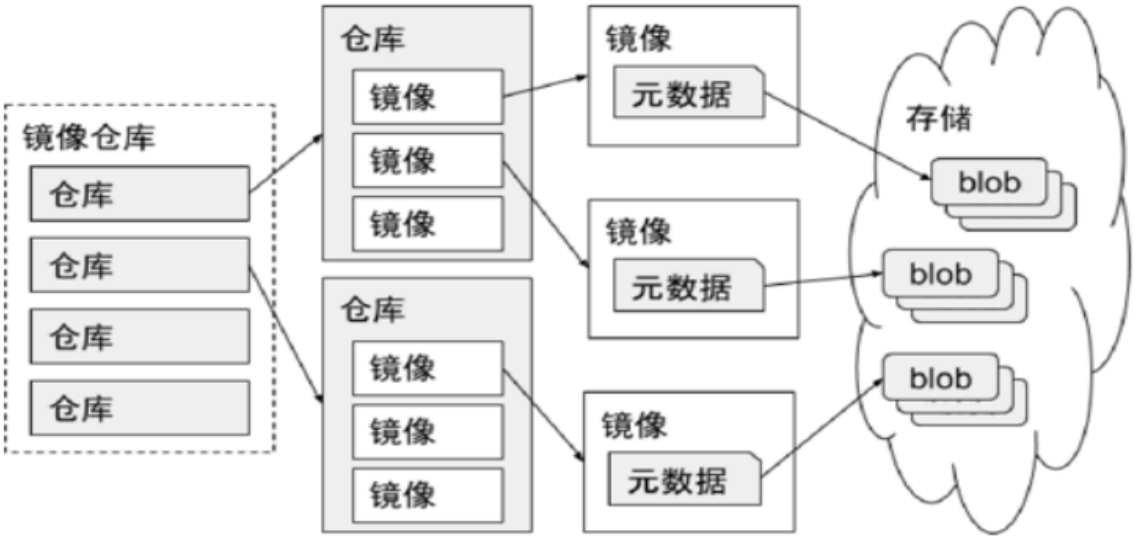
Kubernetes的生产化运维

镜像仓库

负责存储、管理和分发镜像。



有如下几个层级的对象

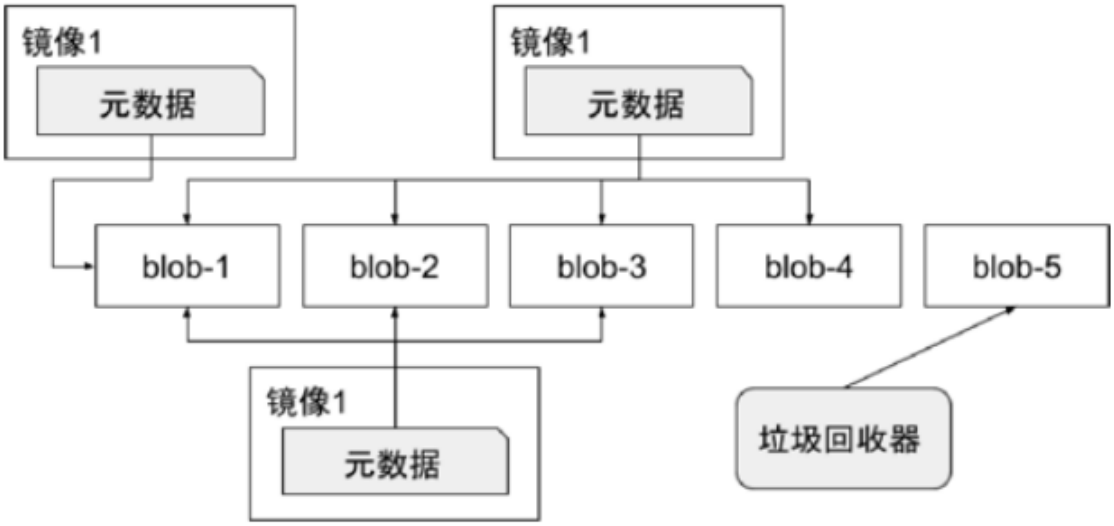
- 镜像仓库，通常通过DNS或IP地址来确定一个镜像仓库，如hub.docker.com。一个镜像仓库管理多个Repository；
- Repository，组织名，如cncamp，每个Repository包含一个或多个镜像
- 进行通过镜像名称+标签（Tag）来区分，如nginx:latest

镜像仓库遵循OCI的Distribution Spec

一个RestFUL的HTTP接口集合。

HTTP Verb	URL	功能
GET	/v2/	检查镜像仓库实现的规范、版本
GET	/v2/_catalog	获取仓库列表
GET	/v2/<name>/tags/list	获取一个仓库下所有的标签
PUT	/v2/<name>/manifests/<reference>	上传镜像manifest信息
DELETE	/v2/<name>/manifests/<reference>	删除镜像
GET	/v2/<name>/manifests/<reference>	获取一个镜像的manifest信息
GET	/v2/<name>/blobs/<digest>	获取一个镜像的文件层
POST	/v2/<name>/blobs/uploads/	启动一个镜像的上传
PUT	/v2/<name>/blobs/uploads/<session_id>	结束文件层上传

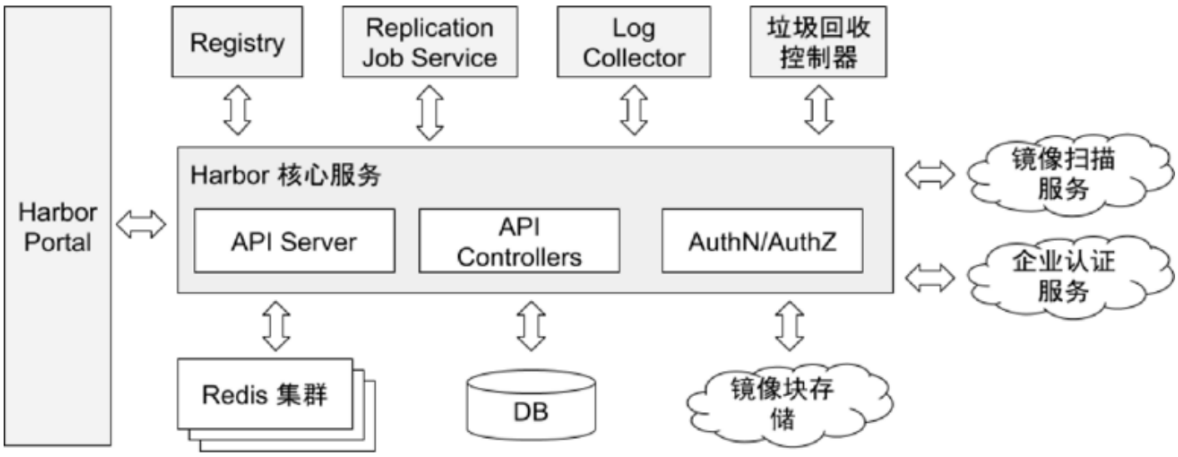
镜像的组成

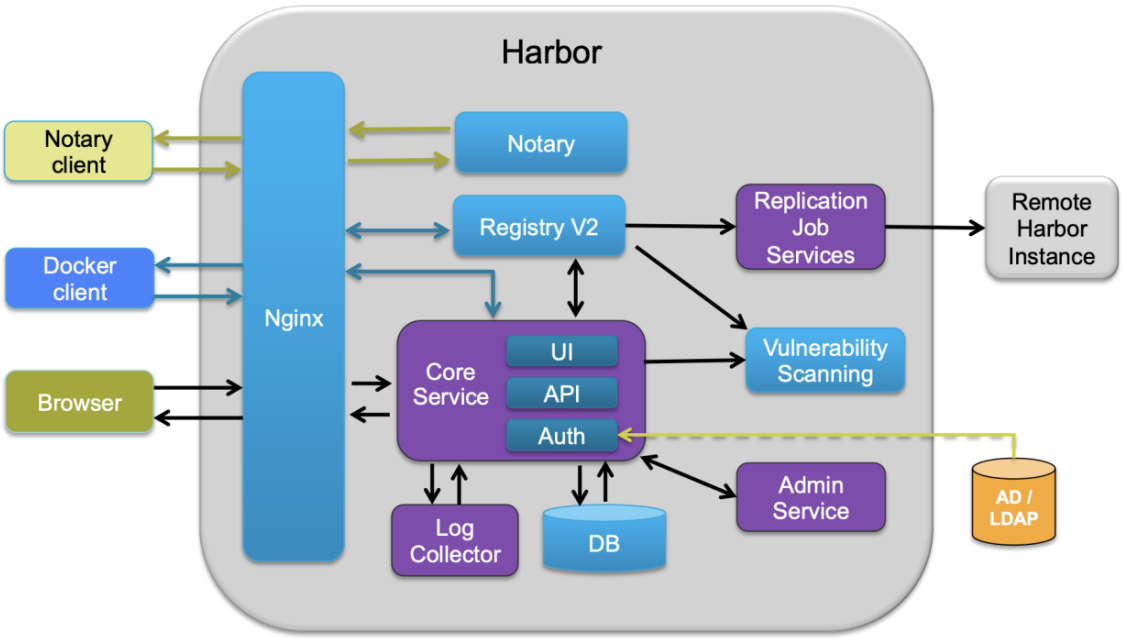


- 元数据
  - 元数据用于描述一个镜像的核心信息，包含镜像的镜像仓库、仓库、标签、校验码、文件层、镜像构建描述等信息。
  - 通过这些信息，可以从抽象层面完整地描述一个镜像：它是如何构建出来的、运行过什么构建命令、构建的每一个文件层的校验码、打的标签、镜像的校验码等。
- 块文件(blob)

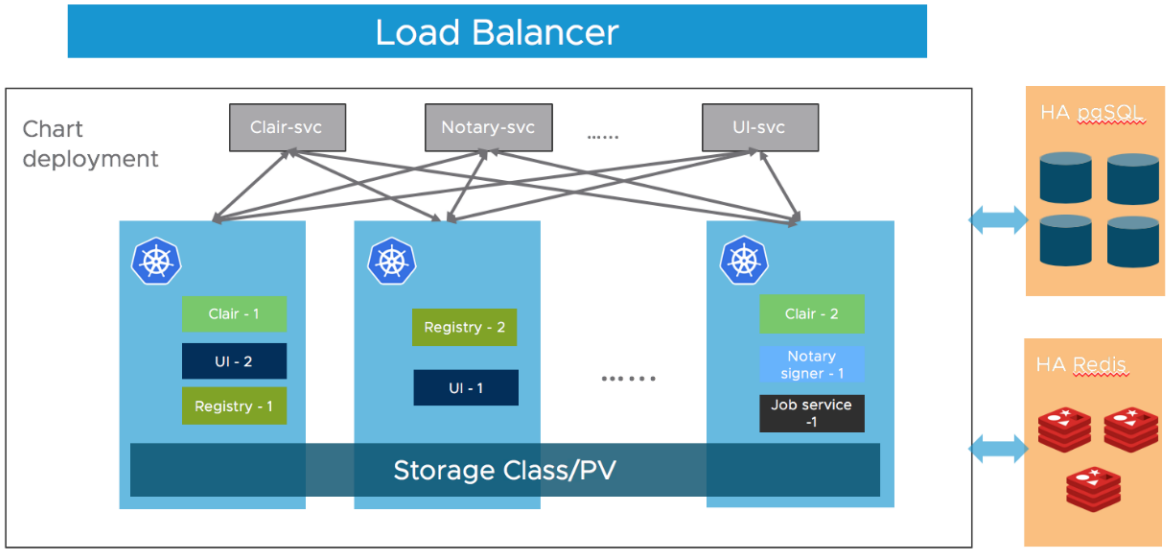
  块文件是组成镜像的联合文件层的实体，每一个块文件是一个文件层，内部包含对应文件层的变更

Harbor





高可用架构：



安装

```
# install download harbor helm chart
helm repo add harbor https://helm.goharbor.io
helm fetch harbor/harbor --untar  #这一条执行完毕后，就会在当前目录下有harbor目录
kubectl create ns harbor
```

```
# update values.yaml, vi .harbor/values.yaml and change
expose:
  type: nodePort
tls:
  commonName: "core.harbor.domain"

persistence: false

# install helm chart
helm install harbor ./harbor -n harbor

# wait for all pod being ready and access harbor portal
# 192.168.34.2:30002
# admin/Harbor12345
```

使用

- **add insecure registry to docker client and restart docker**

  ```
  {
    "features": {
      "buildkit": true
    },
    "experimental": false,
    "builder": {
      "gc": {
        "enabled": true,
        "defaultKeepStorage": "20GB"
      }
    },
    "insecure-registries": [
      "core.harbor.domain:32177"
    ]
  }
  ```

  上面的ENDPOINT "core.harbor.domain:32177"来自于使用helm安装时的配置

  ```
  expose:
    type: nodePort
  tls:
    commonName: "core.harbor.domain"

  persistence: false
  ```

- 然后就可以让docker指向这个地址了

  ```
  docker login -u harbor_registry_user -p harbor_registry_password
  core.harbor.domain:32083
  ```

- **check repositories and blobs**

```
kubectl exec -it harbor-registry-7d686859d7-xs5nv -n harbor bash
ls -la /storage/docker/registry/v2/repositories/
ls -la /storage/docker/registry/v2/blobs
```

- **db operator**

```
kubectl exec -it harbor-database-0 -n harbor bash
psql -U postgres -d postgres -h 127.0.0.1 -p 5432
\c registry
select * from harbor_user;
\dt
```

Harbor Demo Server

演示用，2天清理一次数据，不能push超过100M的镜像，不能使用管理功能。

地址：https://demo.goharbor.io/harbor/projects

相关命令(就是DOCKER把地址指向demo.goharbor.io而已)

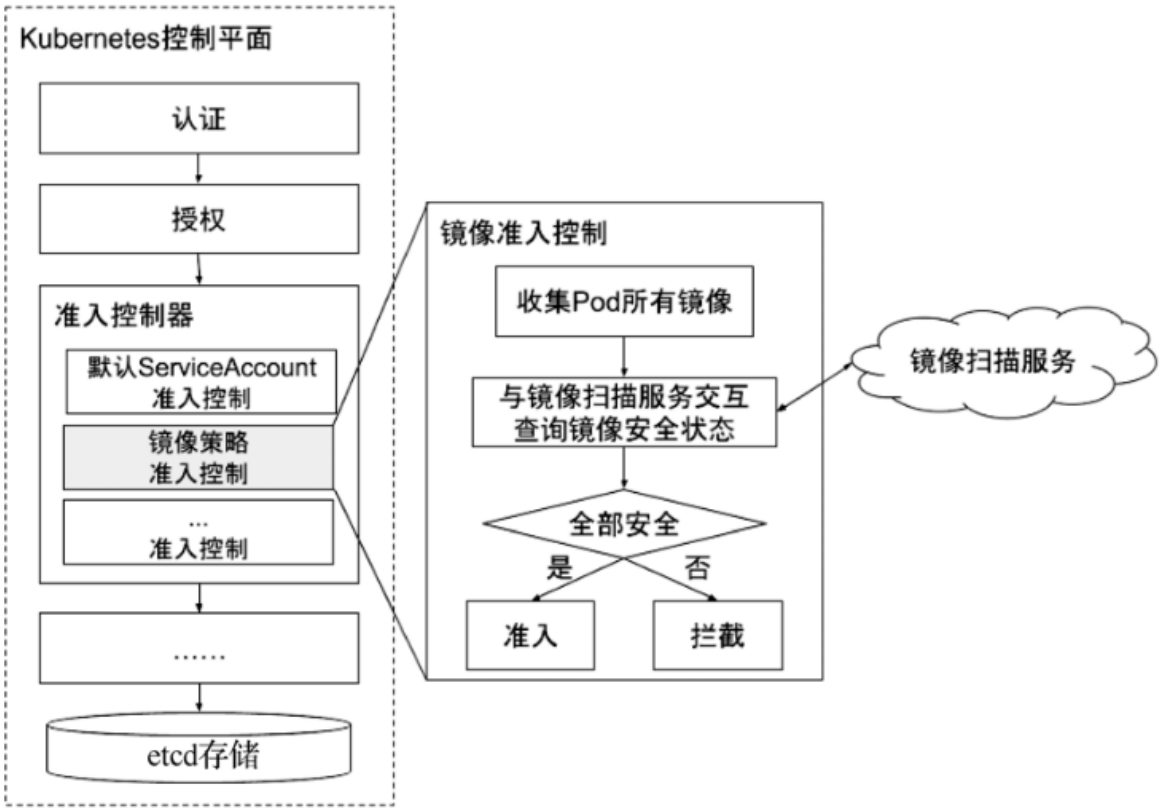
```
docker login demo.goharbor.io
docker build -t demo.goharbor.io/your-project/test-image
docker push demo.goharbor.io/your-project/test-image
```

镜像安全

最佳实践

- 构建指令问题
    - 避免在构建竟像时，添加密钥，Token等敏感信息（配置与代码应分离）
- 应用依赖问题
    - 应尽量避免安装不必要的依赖
    - 确保依赖无安全风险，一些长时间不更新的基础镜像的可能面临安全风险，比如基于openssl1.0，只支持tls1.0等
- 文件问题
    - 在构建镜像时，除应用本身外，还会添加应用需要的配置文件、模板等，在添加这些文件时，会无意间添加一些包含敏感信息或不符合安全策略的文件到镜像中。
    - 当镜像中存在文件问题时，需要通过引入该文件的构建指令行进行修复，而不是通过追加一条删除指令来修复。

镜像扫描和镜像策略准入控制



镜像扫描服务

供应商	Anchore	Aqua	Twistlock	Clair	Qualys
镜像文件扫描	支持	支持	支持	支持	支持
多CVE库支持	支持	支持	支持	支持	支持
是否开源	是	部分		是	
商业支持	支持	支持	支持		支持
可定制安全策略	支持		支持		
镜像仓库支持	Harbor			Quay、Harbor	

老师推荐Clair和harbor配合使用。

基于kubernetesde DevOps

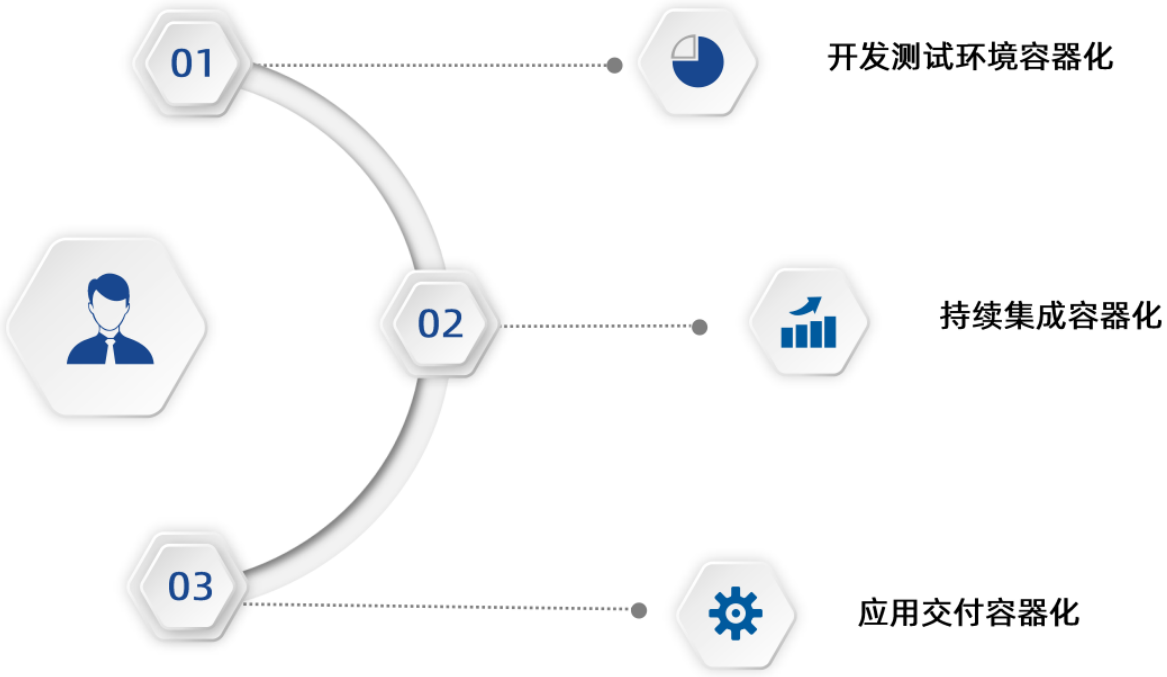
- 传统运维模式有如下问题：

- 缺乏一致性环境
- 平台与应用部署相互割裂
- 缺乏工具链支持
- 缺乏统一的灰度发布管理
- 缺乏统一监控能力和持续运维能力

针对上面的第2点，k8s将平台和应用融合在了一起：k8s的众多对像都是面对应用的——镜像的entry point就是一个应用。

- 传统交付模式下
  - 业务方在漫长的开发期内参与度极低；
  - 没有更多考虑运维，导致后续运维部署困难；
  - 开发各自为政，烟囱式开发，开发资产无法有效积累

基于Docker的开发模式驱动持续集成

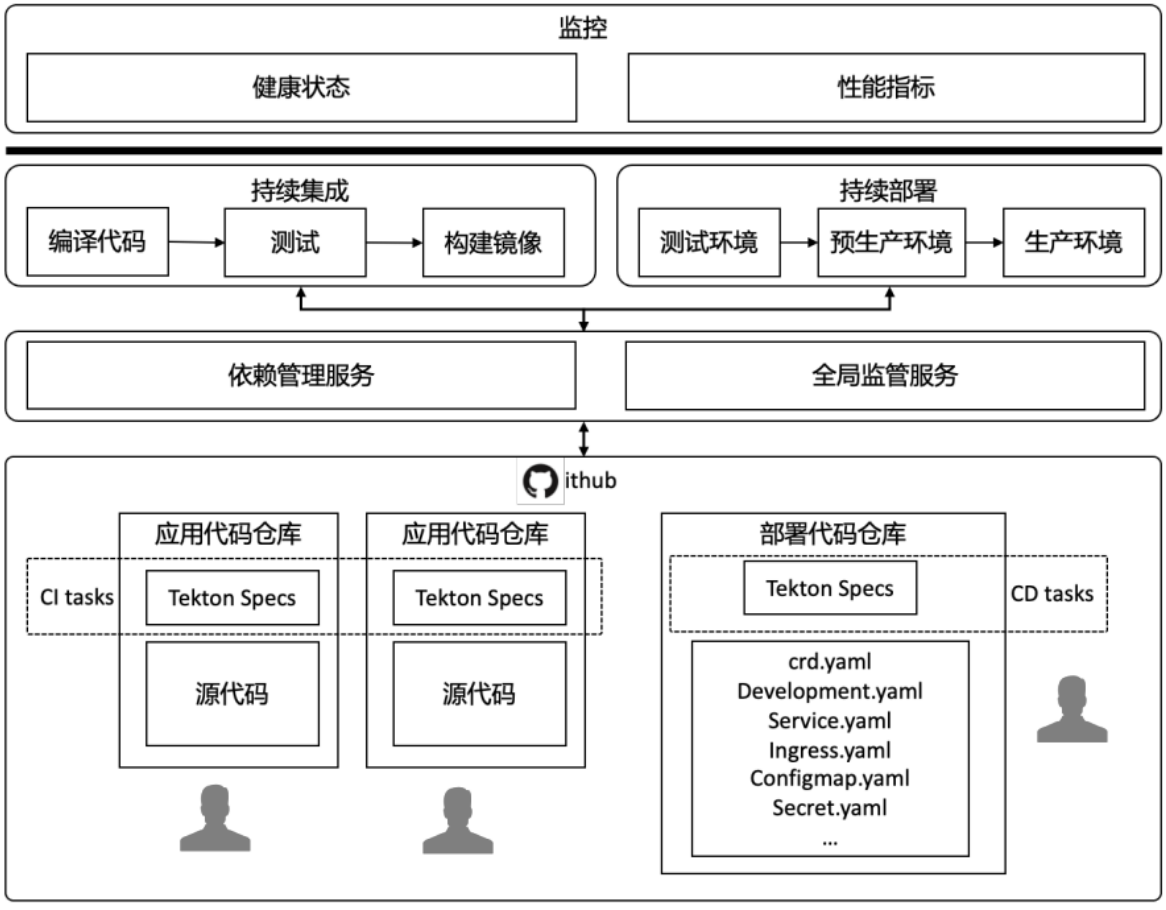


production readiness

Function ready vs. production ready

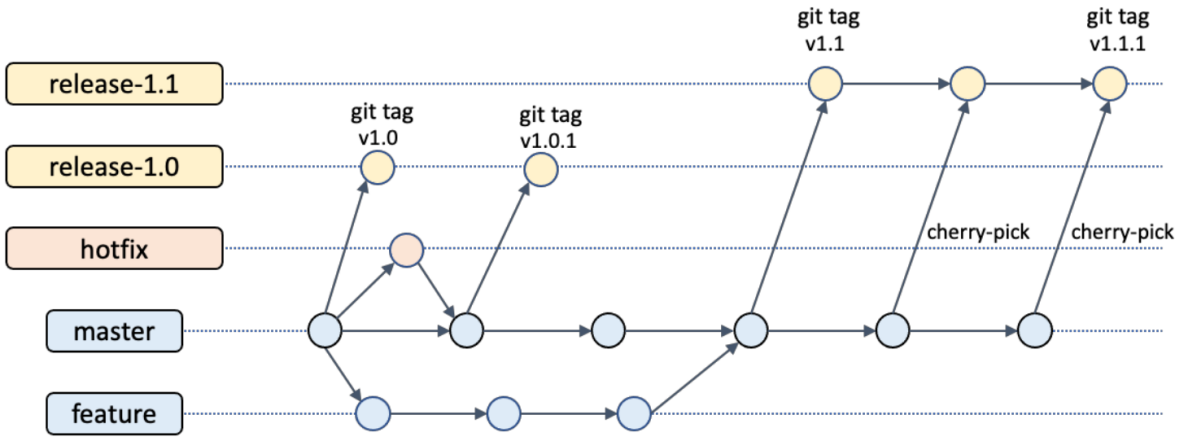
- Function Ready只是交付的软件产品从功能层面满足需求定义
- Production Ready除了功能就绪还包含
  - LnP测试通过，满足性能需求
  - 用户手册完成，用户可按照用户手册使用既定功能
  - 管理手册完成，运维人员可以依照管理手册部署，升级产品并解决现网问题
  - 监控，包括
    - 组件健康状态检查（UP）
    - 性能指标（Metrics）
    - 基于性能指标，定义alert rule，在系统故障或缓慢时，发送告警信息给运维人员
    - Assertion，定期测试某功能并检查结果，比如每小时创建service，测试vip连通性

DevOps流程概览



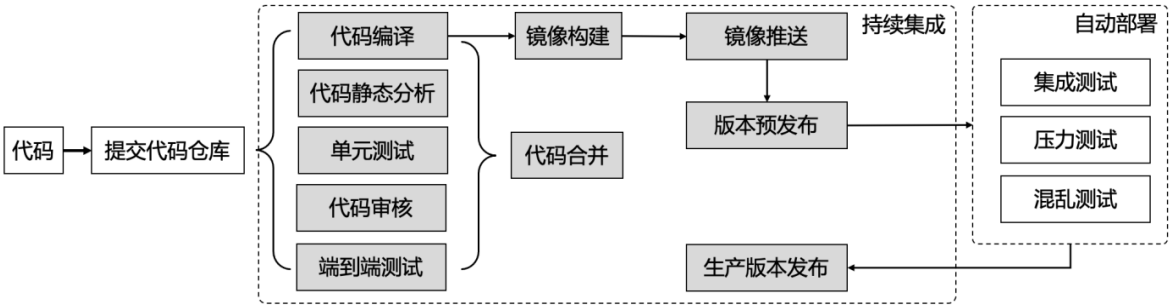
在K8S下，部署由一堆yaml文件代表，也被代码化了，所以也可以由GIT驱动了。

代码分支管理

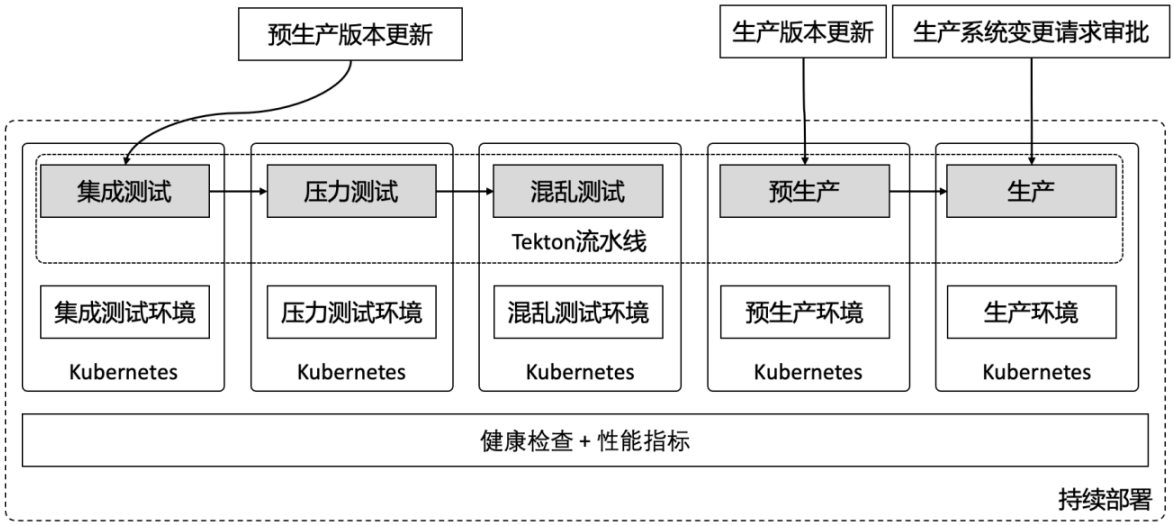


- release分支对应每个版本，比如1.0、1.1等
- master分支是不稳定的，一直在变化，这些变化会合并到release分支，这叫cherry-pick，形成小版本，比如1.0.1、1.1.1等。
- feature分支是开发新的功能，演进后合并到主分支。

持续集成——从代码到版本

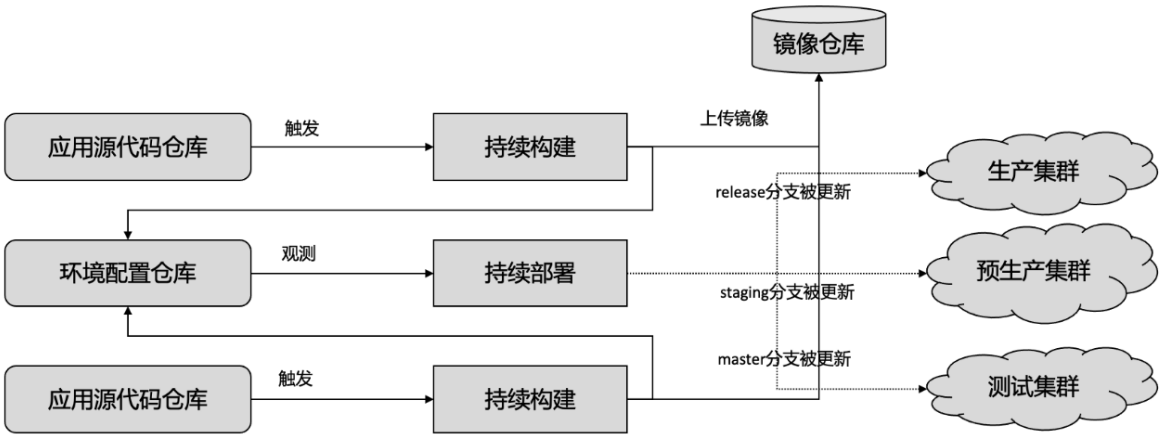


持续部署——版本在不同环境的穿梭



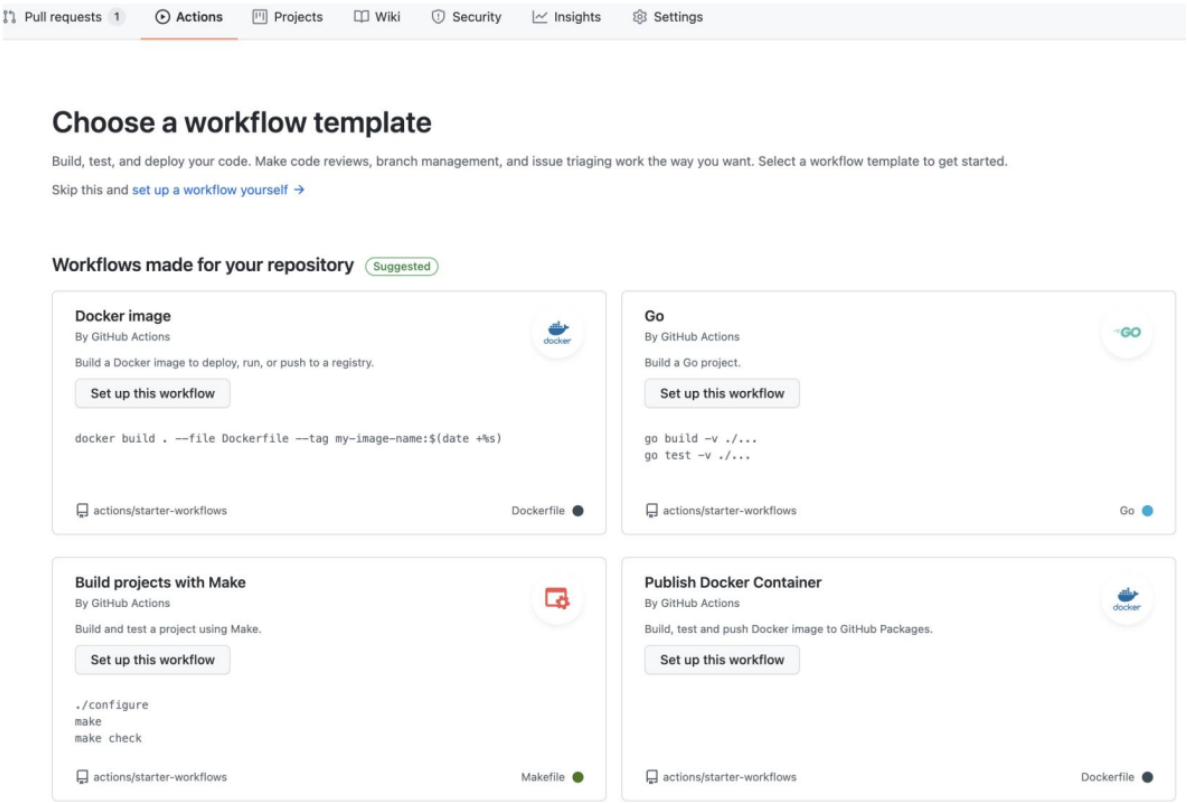
持续集成、持续部署的统一：GitOps

因为在K8S下，部署过程也可以用yaml文件来描述，所以它们都可以由git驱动



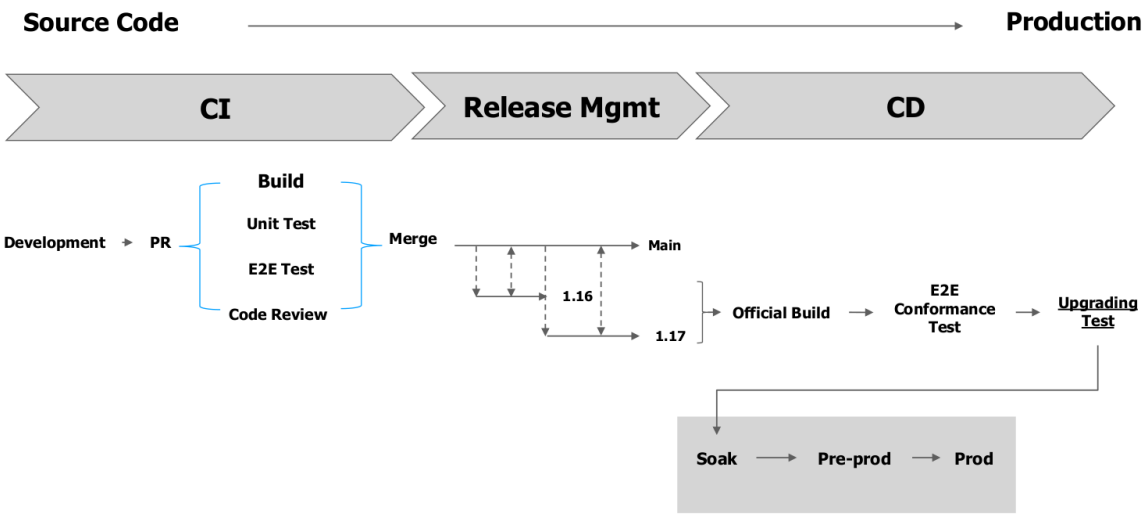
基于GitHub Action的自动化流水线

在GitHub上点击Action按钮进行创建



基于Jenkins的自动化流水线

Kubernetes CI&CD完整流程



持续集成容器化

解决了CI构建环境和开发构建环境的统一问题，使用容器作为标准的构建环境，将代码库作为Volume挂载进构建容器。

> 在构建容器中构建的目标也是一个镜像，这就是Docke in Docker(DIND)问题。
>
> 后来的方法是把Docker Daemon用的Unix Socket挂载进容器，这样容器就可以操作宿主机了：
>
> ```
> docker run -v /var/run/docker.sock:/var/run/docker.sock
> ```
>
> 还有谷歌的解决方案Kaniko。

其余内容略，因为目前没有使用Jenkins。

声明式流水线——Tekton

自定义：Tekton对象是高度自定义的，可扩展性极强。平台工程师可预定义可重用模块以详细的模块目录提供，开发人员可在其他项目中直接引用。

可重用：Tekton对象的可重用性强，组件只需一次定义，即可被组织内的任何人在任何流水线都可重用。使得开发人员无需重复造轮子即可构建复杂流水线。

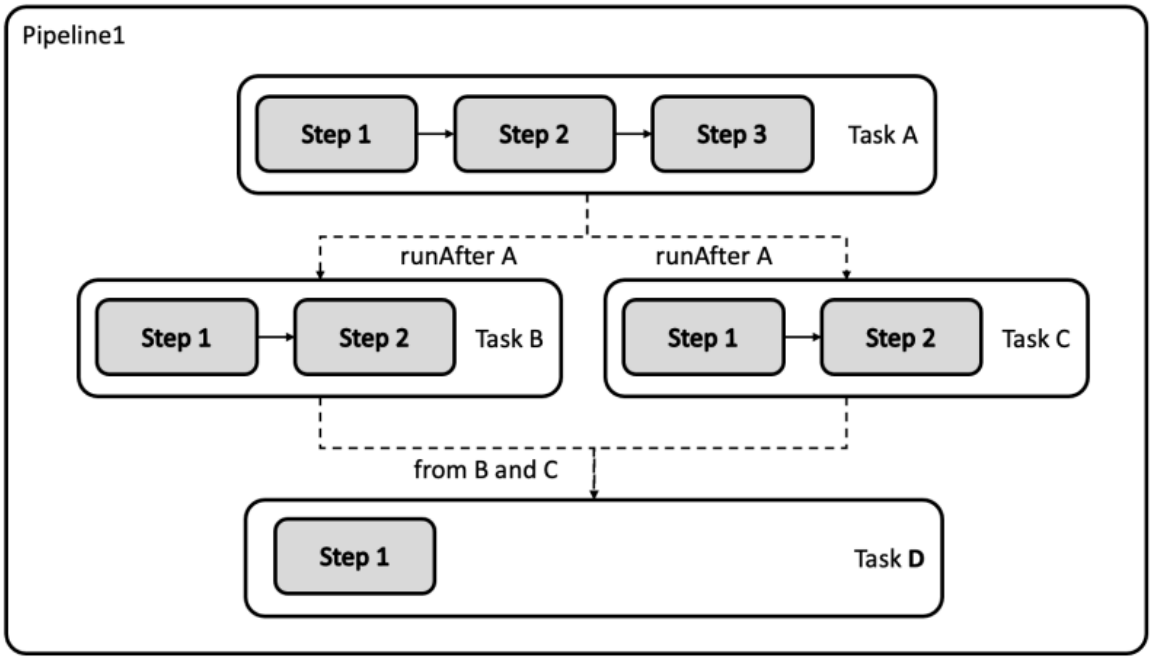
可扩展性：Tekton组件目录（Tekton Catalog）是一个社区驱动的Tekton组件的存储仓库。任何用户可以直接从社区获取成熟的组件并在此之上构建复杂流水线，也就是当你要构建一个流水线时，很可能你需要的所有代码和配置都可以从Tekton Catalog直接拿下来复用，而无需重复开发。

标准化：Tekton作为Kubernetes集群的扩展安装和运行，并使用业界公认的Kubernetes资源模型；Tekton作业以Kubernetes容器形态执行。

规模化支持：只需增加Kubernetes节点，即可增加作业处理能力。Tekton的能力可依照集群规模随意扩充，无需重新定义资源分配需求或者重新定义流水线。

模型

- Pipeline: 定义流水线作业，由一个或数个Task对象组成
- Task：一个可独立运行的任务，如获取代码、编译等。流水线运行时，K8S为每个Task创建一个POD。每个task由多个step组成
- Step：Task的组成单位，每个Step为Task对应Pod中的一个容器



> 这才是字面意义上的“服务编排”呀。

- Task对象例子

  相当于程序而不是进程，不是运行态的东西

```yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: hello
spec:
  steps:
    - name: hello
      image: ubuntu  #每个step是一个容器
      command:
        - echo
      args:
        - "Hello $(params.username)!"
  params:       #此处只是定义了“形参”
  - name: username
    type: string
```

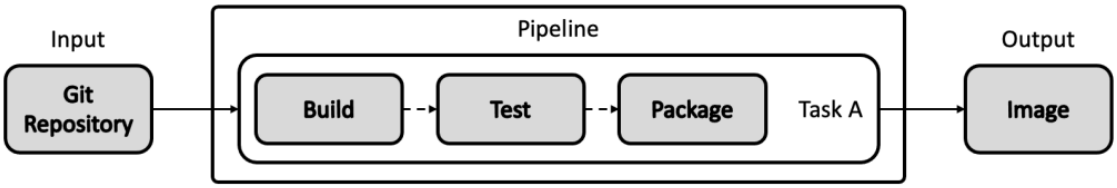
- TaskRun例子

  相对于Task，它才相当于进程

```yaml
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  generateName: hello-run-
spec:
  taskRef:
    name: hello     #引用已经定义好的Task
  params:           #给Task传递“实参“
  - name: "username"
    value: "jesse"
```

- 输入输出资源

  Pipline和Task对象可以接收git repository, pull request等资源作为输入，可以将Image、Kubernetes Cluster、Storage、CloudEvent等对象作为输出



比如下面的Task中就定义了一个git类型的资源

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: github-build
spec:
  params:
  - name: IMAGE
    description: Name (reference) of the image to build.
    default: cncamp/httpserver:v1.0
  - name: DOCKERFILE
    description: Path to the Dockerfile to build.
    default: ./httpserver/Dockerfile
  resources:
    inputs:          #定义输入资源，但也只是形参，实参在TaskRun中传入
    - name: repo
      type: git
  steps:
  - name: github-build
    image: cncamp/executor
    workingDir: /workspace/repo
    args:
    - "--dockerfile=./httpserver/Dockerfile"
    - "--context=./httpserver"
    - "--destination=cncamp/httpserver:v1.0"
  workspaces:
  - name: dockerconfig
    description: Includes a docker `config.json`
    optional: true
    mountPath: /kaniko/.docker
```

定义TaskRun传入实际的资源

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  name: github-build
spec:
  serviceAccountName: cncamp-sa
  taskRef:
    name: github-build
  resources:
    inputs:              #任务资源实参
    - name: repo
```

```
      resourceRef:       #还是一个"引用类型"的实参
        name: cncamp-golang
  workspaces:
  - name: dockerconfig
    secret:
      secretName: docker-auth
```

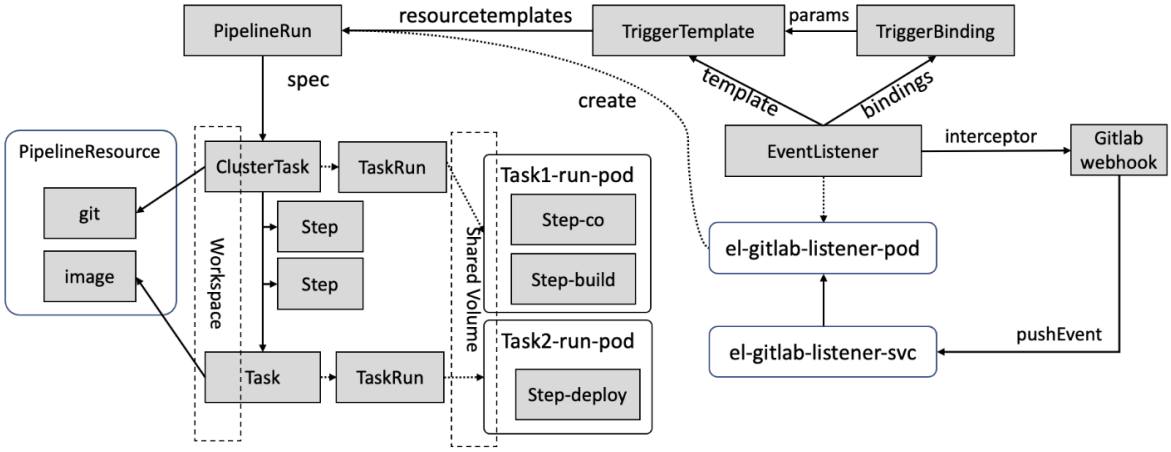
上面的resource实参是一个"引用"，实际定义在PipelineResource对象中

```
apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  name: cncamp-golang
spec:
  type: git
  params:
    - name: url
      value: https://github.com/cncamp/golang.git
    - name: revision
      value: master
  secrets:
    - fieldName: authToken
      secretName: github-secrets
      secretKey: token
```

> 上面定义中涉及到的secret等对象也需要单独定义，在下面
>
> ```
> apiVersion: v1
> kind: Secret
> metadata:
> name: github-secrets
> type: Opaque
> data:
> token: changeme # in base64 encoded form
> ---
> apiVersion: v1
> kind: Secret
> metadata:
> name: docker-auth
> type: Opaque
> stringData:
> config.json: |-
>  {
>    "auths": {
>      "https://index.docker.io/v1/": {
>        "auth": "changeme"
>      }
>    }
>  }
> ---
> apiVersion: v1
> kind: ServiceAccount
> metadata:
> name: cncamp-sa
> secrets:
> - name: docker-auth
> ```

事件触发

上面只是定义了任务（下图左半部分），这些任务的执行还需要事件去触发（下图右半部分）



- EventListener

  事件监听器，该对象核心属性是interceptors拦截器，该拦截器可监听多种类型的事件，比如监听来自GitLab的Push事件。

  当该EventListener对象被创建以后，Tekton 控制器会为该EventListener创建Kubernetes Pod和Service，并启动一个HTTP服务以监听Push事件。

  当用户在GitLab项目中设置webhook并填写该EventListener的服务地址以后，任何人针对被管理项目发起的Push操作，都会被EventListener捕获。

> TODO：研究课程实例代码

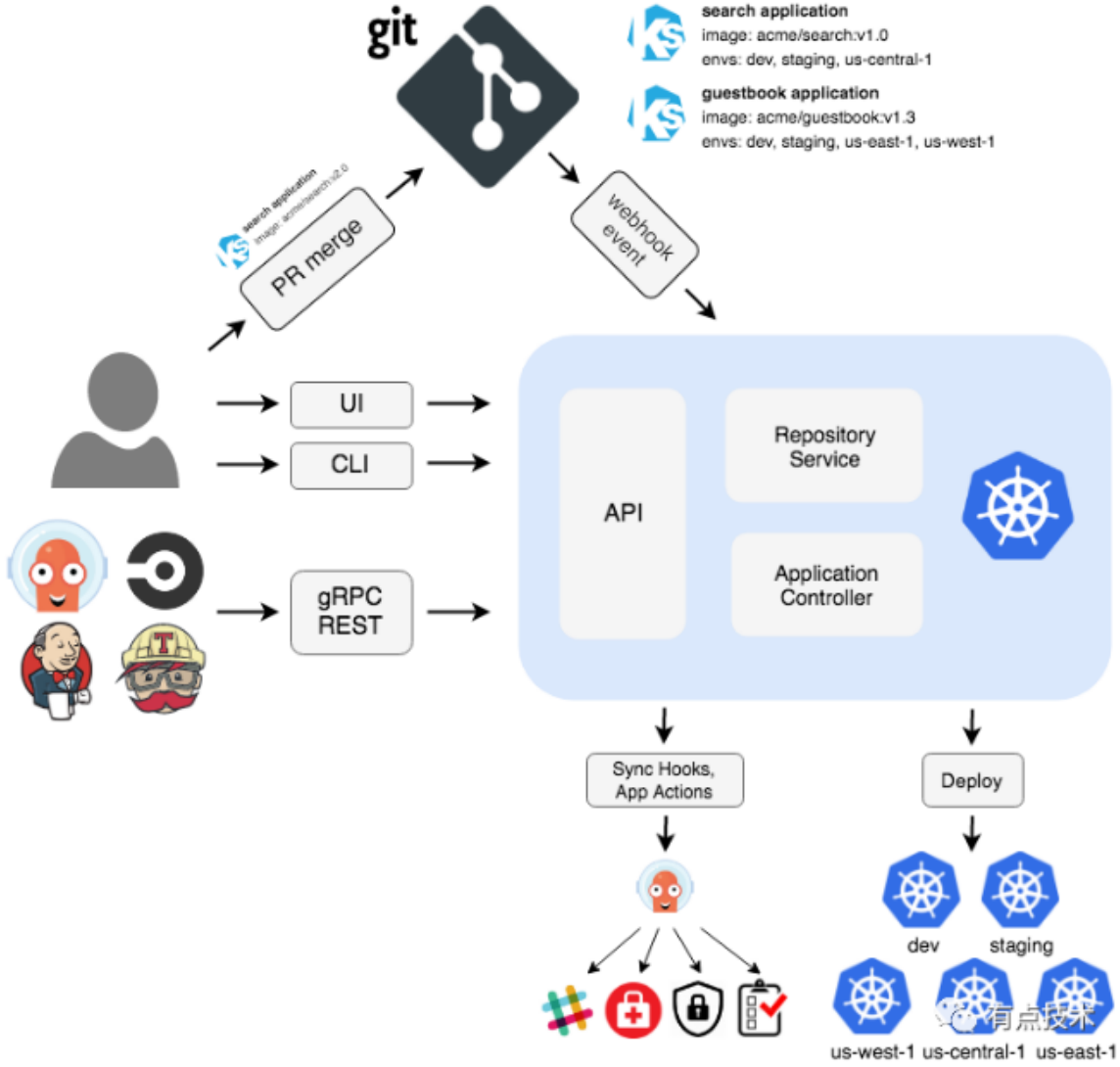
Argocd

Argo CD是用于kubernetes的声明性GitOps连续交付工具，如下都应该是声明性的、并受版本控制

- 应用程序定义
- 配置和环境
- 应用程序部署
- 应用生命周期管理

架构

- Argo CD被实现为K8S的控制器：监控应用的当前状态，并与目标状态（存储在GIT中）比较
- 如果状态有偏离，则被标记为**OutOfSync**
- 报告并可视化差异，同时可以手动或自动将实时状态同步回所需目标状态的功能
- 修改GTI中的目标状态可以被自动应用



安装

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

确保argocd-server service类型为NodePort

- access argocd console

  ```
  user：admin

  password：  k get secret -n argocd argocd-initial-admin-secret -oyaml
  ```

- manage repositories->connect repo using https

  ```
  https://github.com/cncamp/test.git
  ```

- **create application**

```
sync policy: manual
path: .
```

- **scale the deploy by cmd**

```
k scale deployment httpserver --replicas=2
```

- **check appstatus and sync again**
- **change the sync policy to auto and see**

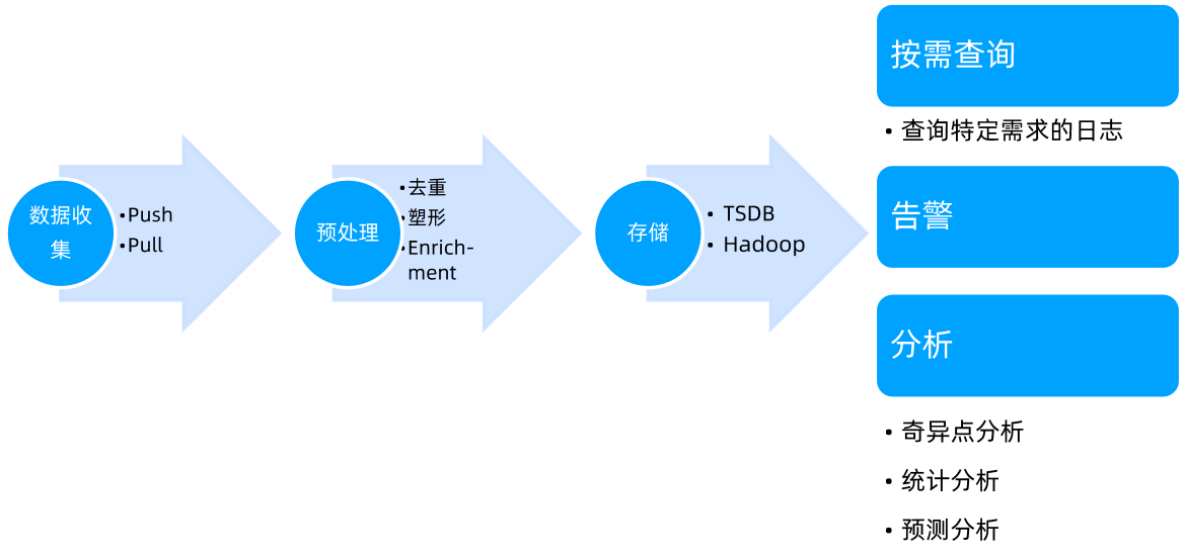
适用场景

1. 低成本的Gitops利器
2. 多集群管理
    1. 不同目的集群：测试，集成，预生产，生产
    2. 多生产集群管理

监控和日志

日志收集和分析

常用数据系统构建模式

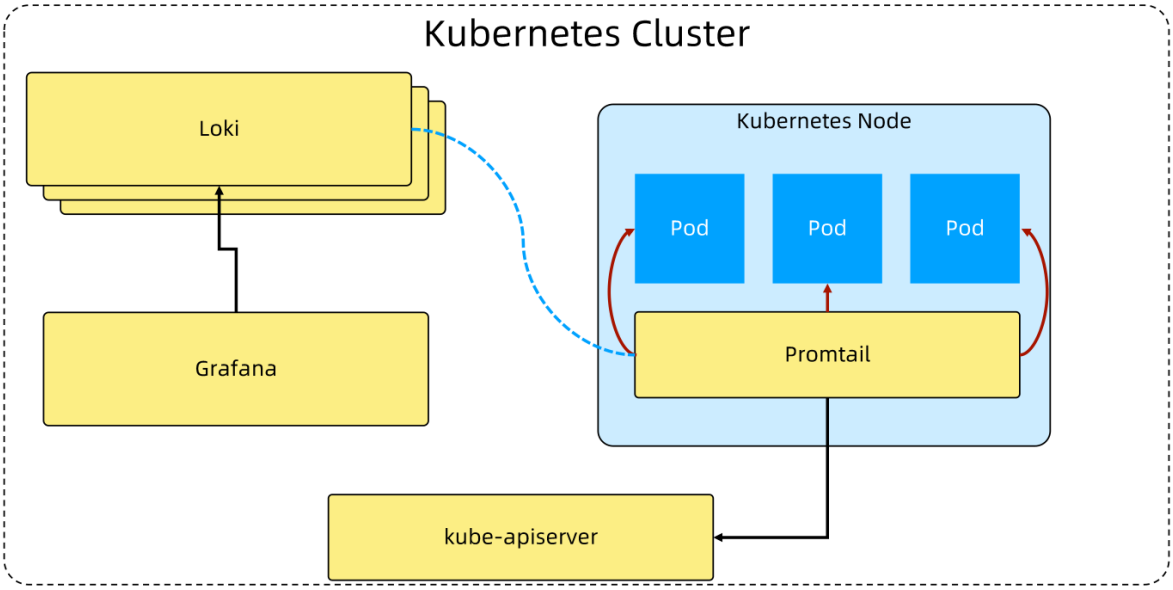


Loki

Grafana Loki是可以组成功能齐全的日志记录栈的一堆组件。Prometheus是K8S监控的事实标准，与Prometheus配套的是Grafana，而Grafana又支持Loki

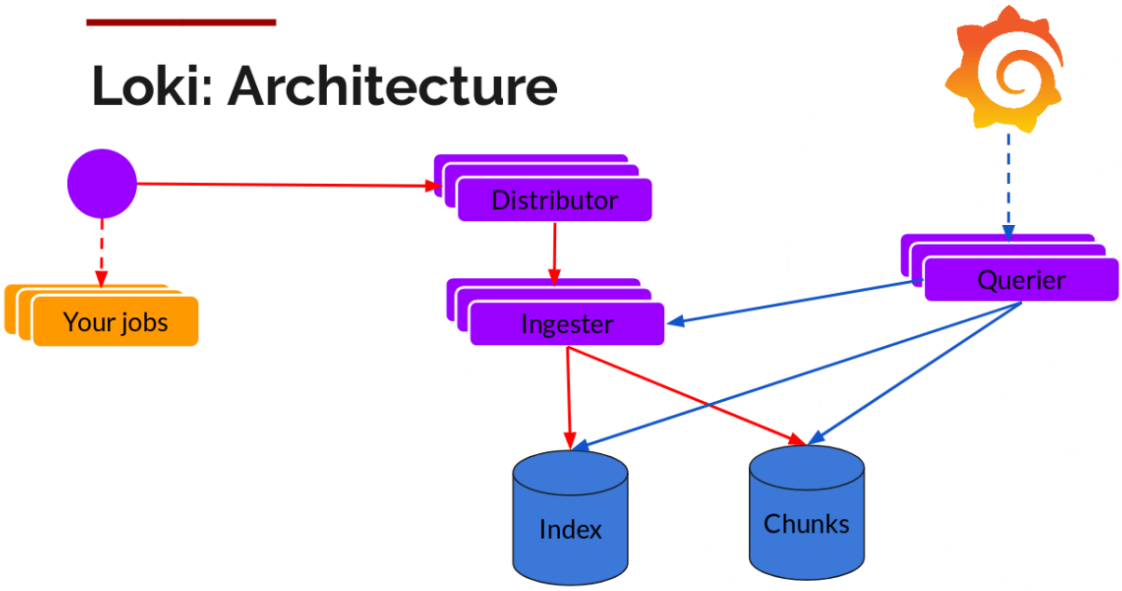
- 与其他日志记录系统不同，Loki是基于仅索引有关日志的元数据的想法而构建的：标签。
- 日志数据本身被压缩并存储在对象存储（例如S3或GCS）中的块中，甚至存储在文件系统本地。
- 小索引和高度压缩的块简化了操作，并大大降低了Loki的成本。

Loki-stack的组成



- Promtail
  - 将容器日志发送到 Loki 或者 Grafana 服务上的日志收集工具
  - 发现采集目标以及给日志流添加上 Label，然后发送给 Loki
  - Promtail 的服务发现是基于 Prometheus 的服务发现机制实现的，可以查看configmap loki-promtail了解细节
- Loki
  - Loki 是可以水平扩展、高可用以及支持多租户的日志聚合系统
  - 使用和 Prometheus 相同的服务发现机制，将标签添加到日志流中而不是构建全文索引
  - Promtail 接收到的日志和应用的 metrics 指标就具有相同的标签集
- Grafana
  - Grafana 是一个用于监控和可视化观测的开源平台，支持非常丰富的数据源
  - 在 Loki 技术栈中它专门用来展示来自 Prometheus 和 Loki 等数据源的时间序列数据
  - 允许进行查询、可视化、报警等操作，可以用于创建、探索和共享数据 Dashboard

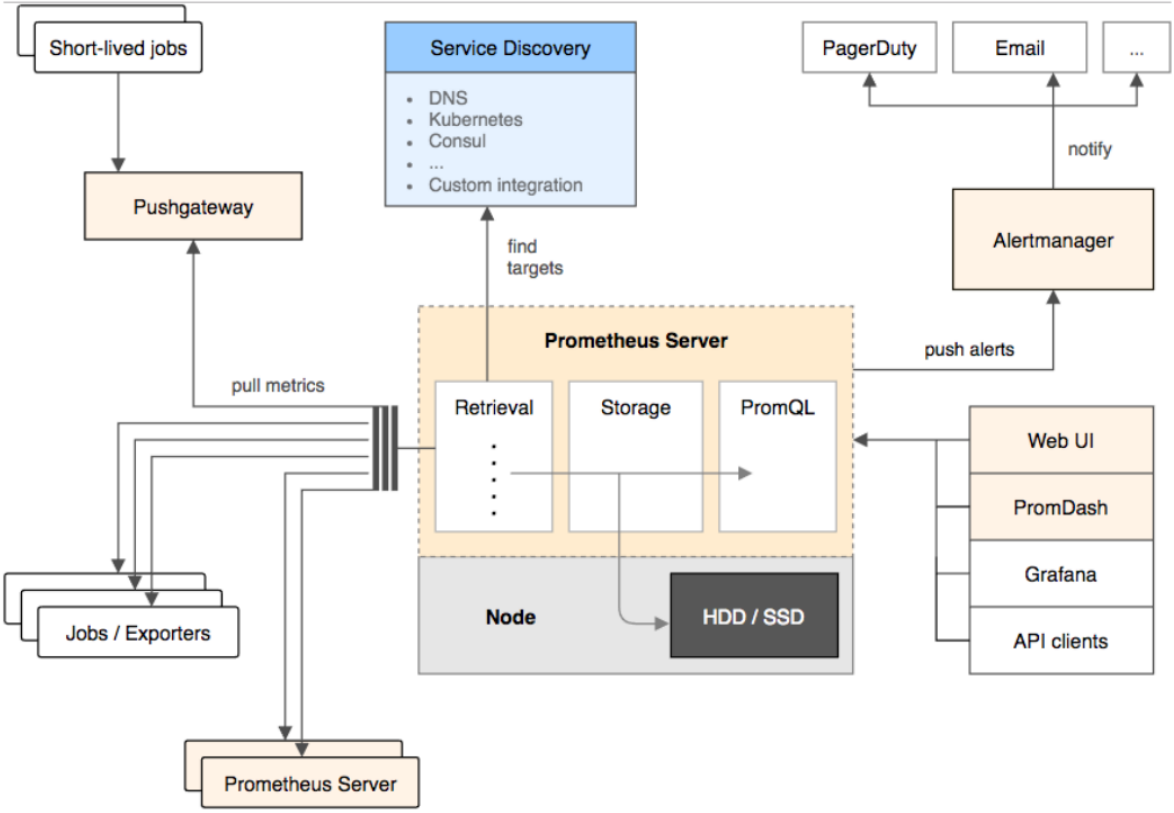
Loki架构



- **Distributor（分配器）**
    - 分配器服务负责处理客户端写入的日志。
    - 一旦分配器接收到日志数据，它就会把它们分成若干批次，并将它们并行地发送到多个采集器去。
    - 分配器通过 gRPC 和采集器进行通信。
    - 它们是无状态的， 基于一致性哈希，我们可以根据实际需要对他们进行扩缩容。
- **Ingester（采集器）**
    - 采集器服务负责将日志数据写入长期存储的后端（DynamoDB、S3、Cassandra 等等）。
    - 采集器会校验采集的日志是否乱序。
    - 采集器验证接收到的日志行是按照时间戳递增的顺序接收的，否则日志行将被拒绝并返回错误。
- **Querier（查询器）**
    - 查询器服务负责处理 LogQL 查询语句来评估存储在长期存储中的日志数据。

监控系统

这里主要是指Prometheus



在K8S中汇报指标

- 应用POD需要声明上报指标端口和地址

```
apiVersion: v1
kind: Pod
metadata:
 annotations:
  prometheus.io/port: http-metrics
  prometheus.io/scrape: "true"
name: loki-0
 namespace: default
spec:
 ports:
  - containerPort: 3100
   name: http-metrics
   protocol: TCP
```

- 声明了指标就一定要汇报，否则Prometheus就会认为指标DOWN了从而引发告警
  - 应用启动时，需要注册metrics

```
http.Handle("/metrics", promhttp.Handler())
http.ListenAndServe(sever.MetricsBindAddress, nil)
```

  - 注册指标

```
func RegisterMetrics() {
  registerMetricOnce.Do(func() {
    prometheus.MustRegister(APIServerRequests)
    prometheus.MustRegister(WorkQueueSize)
  })
}
```

  - 代码中输出指标

```
metrics.AddAPIServerRequest(controllerName,
constants.CoreAPIGroup, constants.SecretResource,
constants.Get, cn.Namespace)
```

> K8S中控制平面包括各种控制器都原生暴露Promethues格式的Metrics。

Promethues中的指标类型

- Counter （计数器器）
    - Counter 类型代表一种样本数据单调递增的指标，即只增不减，除非监控系统发生了重置。
- Gauge（仪表盘）
    - Guage 类型代表一种样本数据可以任意变化的指标，即可增可减。
- Histogram（直方图）
    - Histogram 在一段时间范围内对数据进行采样（通常是请求持续时间或响应大小等），并将其计入可配置的存储桶（bucket）中，后续可通过指定区间筛选样本，也可以统计样本总数，最后一般将数据展示为直方图
    - 样本的值分布在 bucket 中的数量，命名为 <basename>_bucket{le="<上边界>"}
    - 所有样本值的大小总和，命名为 <basename>_sum
    - 样本总数，命名为 <basename>_count。值和 <basename>_bucket{le="+Inf"} 相同
- Summary（摘要）
    - 与 Histogram 类型类似，用于表示一段时间内的数据采样结果（通常是请求持续时间或响应大小等），但它直接存储了分位数（通过客户端计算，然后展示出来），而不是通过区间来计算
    - 它们都包含了 <basename>_sum 和 <basename>_count 指标
    - Histogram 需要通过 <basename>_bucket 来计算分位数，而 Summary 则直接存储了分位数的值。

开启告警

修改Promitheus的配置文件promitheus.yml添加配置即可（如果是在K8S中安装，修改一个configmap即可）

```
rule_files:
  - /etc/prometheus/rules/*.rules
```

在目录/etc/prometheus/rules/下创建告警文件hoststats-alert.rules内容如下：

```
groups:
- name: hostStatsAlert
  rules:
  - alert: hostCpuUsageAlert
    expr: sum(avg without (cpu)(irate(node_cpu{mode!='idle'}[5m]))) by (instance) > 0.85
    for: 1m
    labels:
      severity: High
    annotations:
      summary: "Instance {{ $labels.instance }} CPU usgae high"
      description: "{{ $labels.instance }} CPU usage above 85% (current value: {{ $value }})"
```