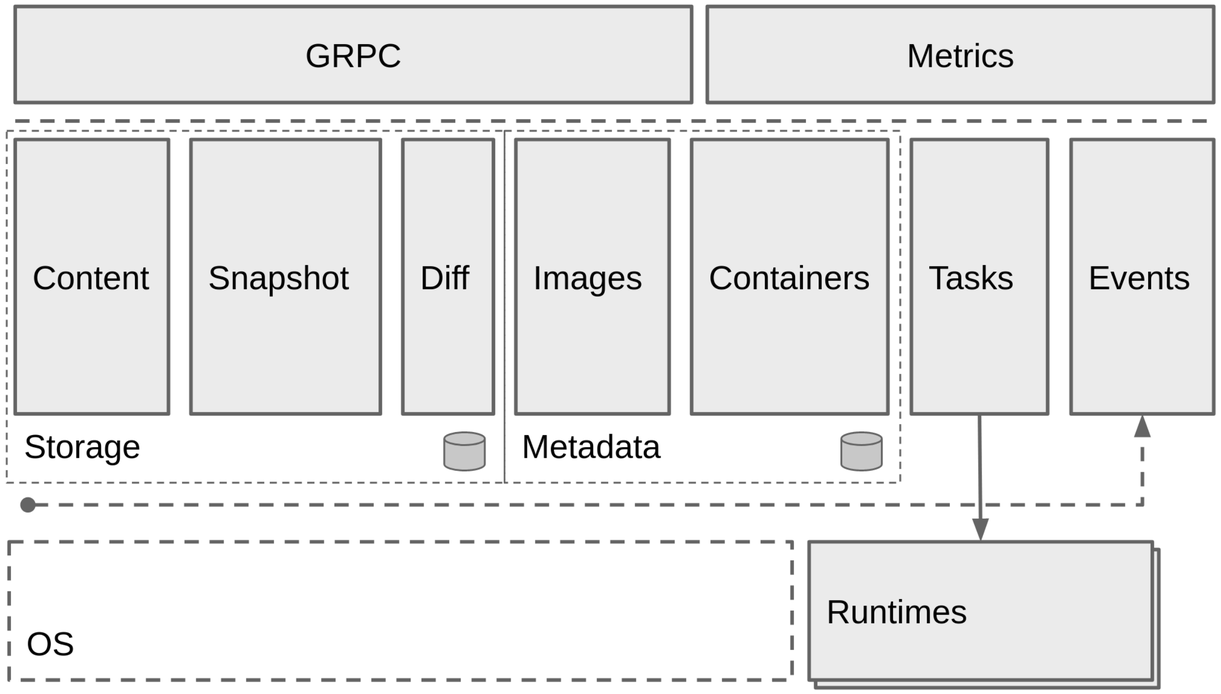
Containerd架构

服务端 -- 组件

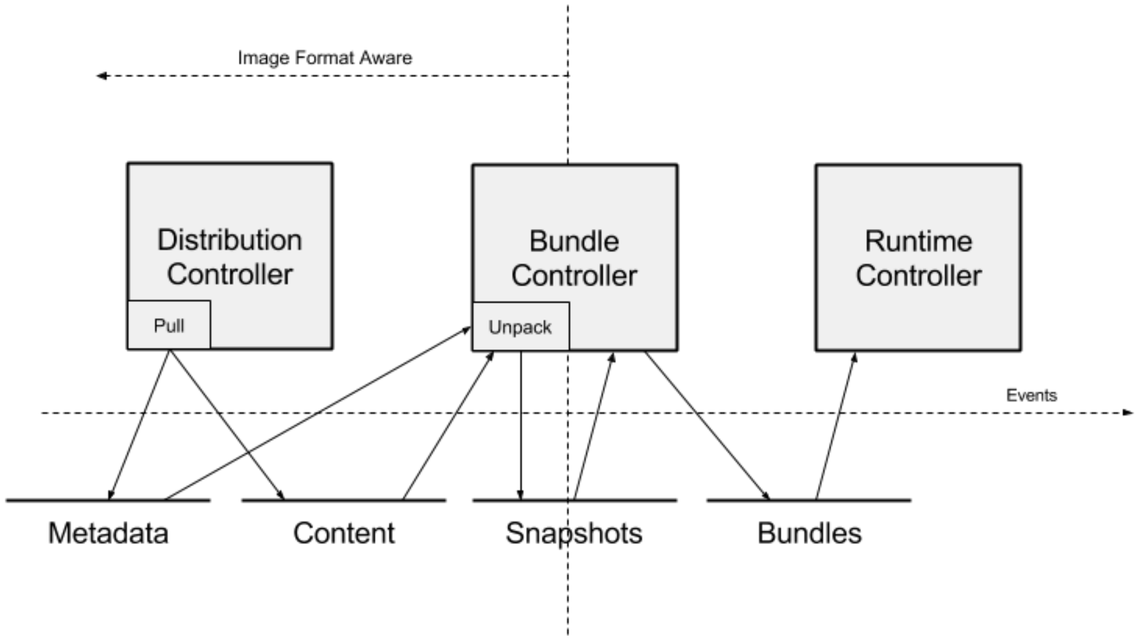
- containerd是典型的**C/S**架构
- 在服务端包含Bundle和Runtime两大Subsystems(子系统)服务，外部用户通过**GRPC** API与服务进行交互。
- 组件
  - These components that may **cross subsystem boundaries**
  - Executor: 实现实际的**容器运行时**
  - Supervisor: 监视并报告**容器状态**
  - Content: 用于保存下载的镜像
    - 一般镜像的内容：index、config、manifest、layers
  - Snapshot: 管理文件系统上**容器镜像的快照**。镜像的层被解压缩到snapshot上，然后snapshotter会通过mount各个层为容器准备rootfs。
  - Metadata: 元数据存储在bolt db，存储对images和bundles的任何持久**引用**
    - bolt db是一个嵌入式的key/value数据库
  - Events: 支持**事件**的收集和使用，提供一致的事件驱动的行为和审计。事件可以replay到各个模块
  - Metrics: 每个组件都将暴露一些**指标**，通过Metrics API进行访问



Containerd架构

服务端 -- 子系统

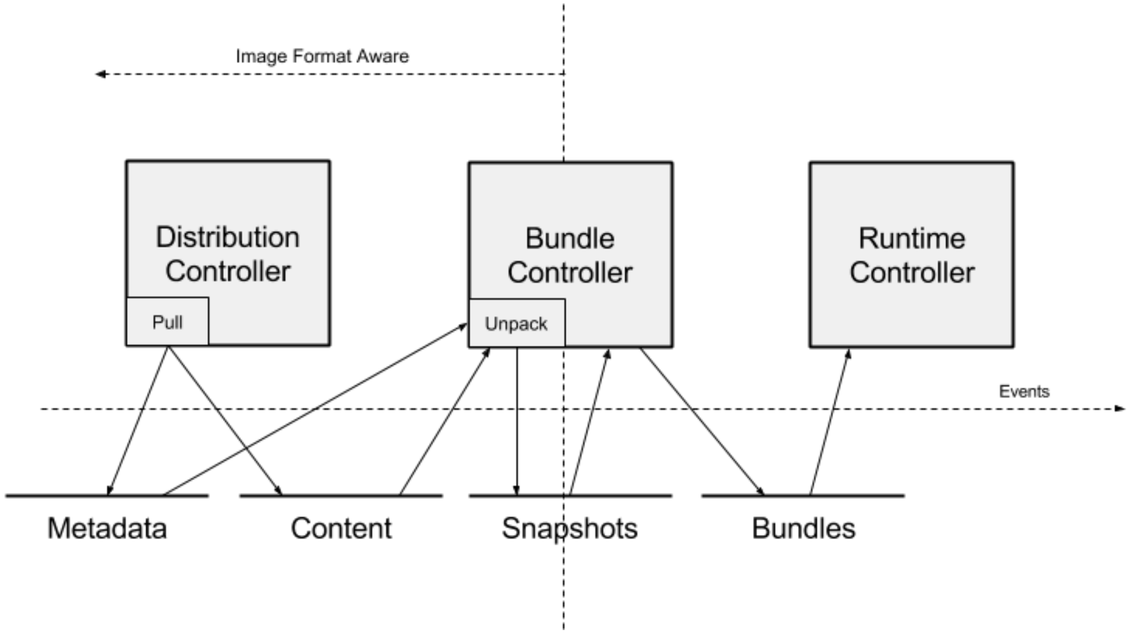
- 在服务端包含Bundle和Runtime两大Subsystems(子系统)服务，外部用户通过 **GRPC** API与服务进行交互。
- 子系统
  - Bundle: 允许用户从磁盘**镜像**中extract和pack bundles.
    - The concept of a bundle is **central** to containerd
    - The definition of a bundle is only concerned with **how a container, and its configuration data, are stored on a local filesystem** so that it can be consumed by a compliant runtime.
    - bundles是指被Runtime使用的config、metadata、rootfs数据
    - **总结**
      - 一个bundle就是一个**运行时的容器**在磁盘上的表现形式（简化为文件系统中的一个目录）
      - 是**镜像到容器的中间形态**，交给runc运行的实际上是bundles
  - Runtime: 支持运行bundles，包括运行时容器的创建
- 每个子系统都有一个或多个相关的**控制器**组件来实现子系统的功能，并以**服务**的形式暴露给外部访问。



Containerd架构

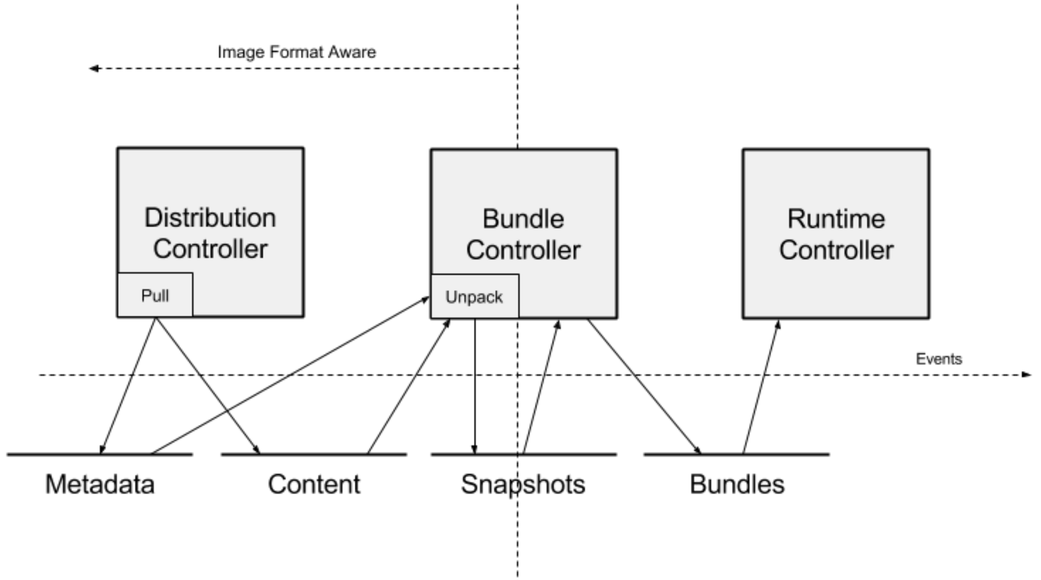
客户端

- 客户端组件(Distribution)
  - 为了灵活性，一些组件是在**客户端**实现的
  - Distribution: 提供**镜像**的拉取和推送上传功能
    - **Fetch** downloads the provided content into containerd's **content store** and returns a non-platform specific **image reference**
    - func (c *Client) Fetch(ctx context.Context, ref string, opts ...RemoteOpt) (**images.Image**, error)



创建Bundle的流程

- 以redis镜像为例，包含：
  - 1个index, 1个config, 1个manifest和6个layer
- containerd创建bundle的流程
  - 指示Distribution Controller去拉取一个具体的镜像，Distribution将**镜像内容(image content)**存储到内容存储(content store)中，将镜像名和root manifest pointers注册到元数据存储中(metadata store)。
  - 一旦镜像拉取完成，用户可以指示Bundle Controller将镜像unpack到一个bundle中。
    - **镜像中的layers**(OCI标准的tar) 会被解压到**snapshot**组件中，接着会挂载各个layer来准备容器的rootfs；
    - 当rootfs准备好时，Bundle Controller可以使用image manifest和config来准备execution **config** (一部分步骤是将mounts从snapshot module输入到execution config)；
  - 然后将准备好的bundle给Runtime子系统以执行, Runtime子系统将读取bundle配置来创建一个容器。



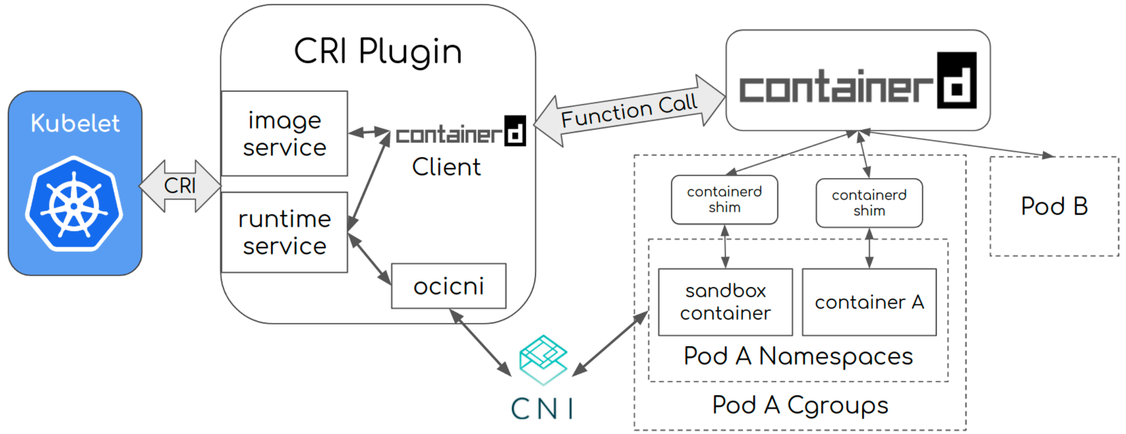
Containerd Plugins System

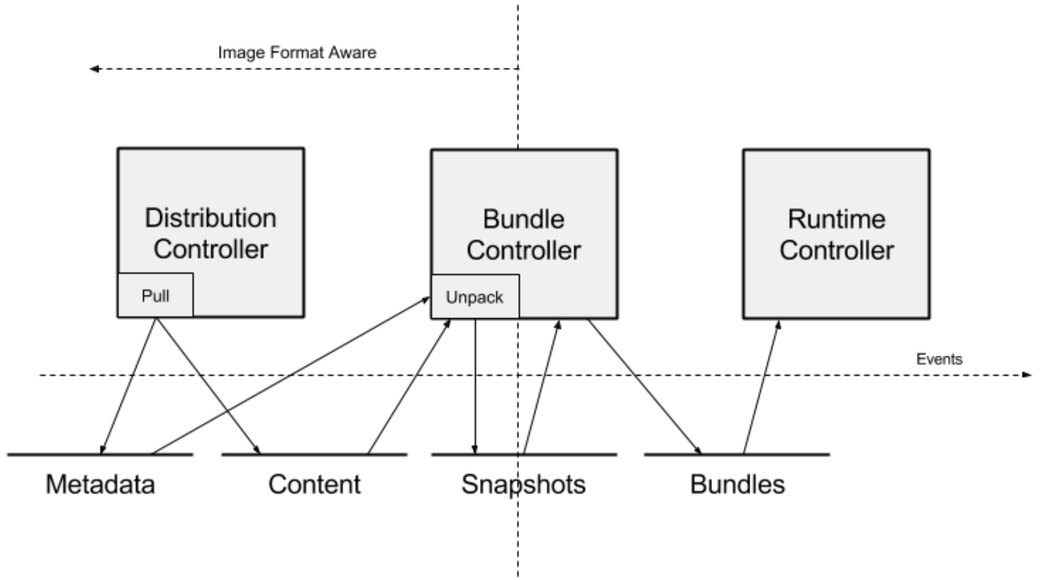
- Containerd 是一个**高度模块化**的高级运行时，所有模块**均可插拔**，模块均以 RPC service 形式注册并调用（gRPC 或者 TTRPC）。
- 不同插件通过声明互相依赖，由 Containerd 核心实现统一加载，使用方可以**自行实现插件以实现定制化**的功能。
- 基于插件我们可以自己定义运行时（runtime）、容器快照模块、存储模块甚至gRPC模块等
  - 外部插件：外部插件可以在不重新编译containerd的情况下对containerd的功能呢进行修正与更改，如cri plugin
  - 内部插件：containerd以**插件**的方式来保证其内部实现的低耦合性，稳定性；虽然名为内部插件，但containerd用**等同**的方式看待内部以及外部插件。
    - 可以通过命令ctr plugins ls查看所有containerd内置插件

```
$ ctr plugins ls
TYPE                                  ID                    PLATFORMS      STATUS
io.containerd.content.v1              content               -              ok
io.containerd.snapshotter.v1          btrfs                 linux/amd64    ok
io.containerd.snapshotter.v1          aufs                  linux/amd64    error
io.containerd.snapshotter.v1          native                linux/amd64    ok
io.containerd.snapshotter.v1          overlayfs             linux/amd64    ok
io.containerd.snapshotter.v1          zfs                   linux/amd64    error
io.containerd.metadata.v1             bolt                  -              ok
io.containerd.differ.v1               walking               linux/amd64    ok
io.containerd.gc.v1                   scheduler             -              ok
io.containerd.service.v1              containers-service    -              ok
io.containerd.service.v1              content-service       -              ok
io.containerd.service.v1              diff-service          -              ok
io.containerd.service.v1              images-service        -              ok
io.containerd.service.v1              leases-service        -              ok
io.containerd.service.v1              namespaces-service    -              ok
io.containerd.service.v1              snapshots-service     -              ok
io.containerd.runtime.v1              linux                 linux/amd64    ok
io.containerd.runtime.v2              task                  linux/amd64    ok
io.containerd.monitor.v1              cgroups               linux/amd64    ok
io.containerd.service.v1              tasks-service         -              ok
io.containerd.internal.v1             restart               -              ok
io.containerd.grpc.v1                 containers            -              ok
io.containerd.grpc.v1                 content               -              ok
io.containerd.grpc.v1                 diff                  -              ok
io.containerd.grpc.v1                 events                -              ok
io.containerd.grpc.v1                 healthcheck           -              ok
io.containerd.grpc.v1                 images                -              ok
io.containerd.grpc.v1                 leases                -              ok
io.containerd.grpc.v1                 namespaces            -              ok
io.containerd.grpc.v1                 snapshots             -              ok
io.containerd.grpc.v1                 tasks                 -              ok
io.containerd.grpc.v1                 version               -              ok
io.containerd.grpc.v1                 cri                   linux/amd64    ok
```

扩展containerd的思路

- CRI Plugin实现了CRI api
- 能否**新增一个CRI api的实现**，让其调用**新实现的插件**
  - 新的插件们实现拉取、存储、运行的功能
    - 省去准备bundle的过程，拉取后直接交给runtime运行；

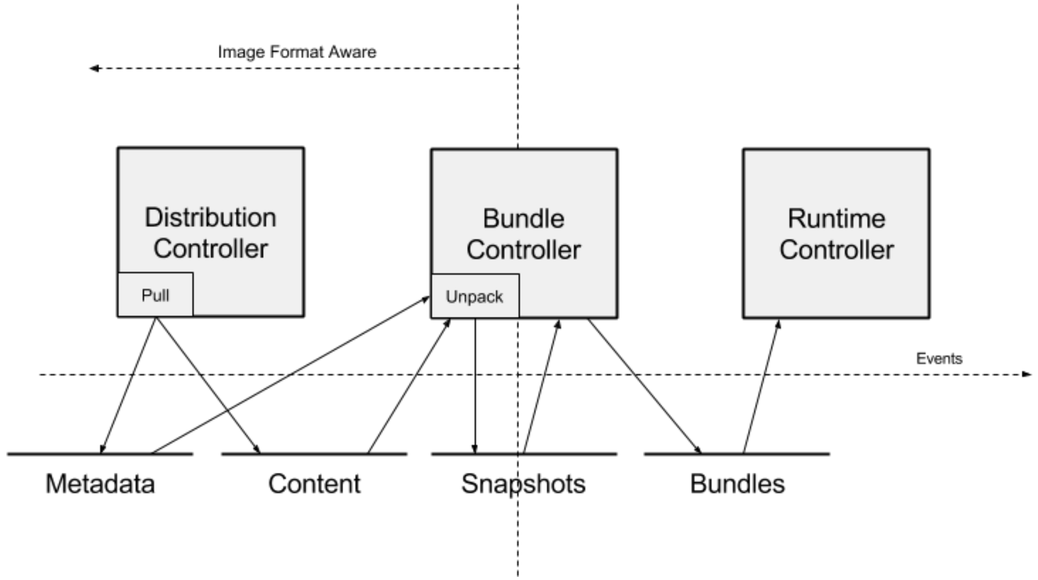




fission从k8s crd拉取package的原理

```
crdsExpected := []string{
    "canaryconfigs.fission.io",
    "environments.fission.io",
    "functions.fission.io",
    "httptriggers.fission.io",
    "kuberneteswatchtriggers.fission.io",
    "messagequeuetriggers.fission.io",
    "packages.fission.io",
    "timetriggers.fission.io",
}
```

- crd是否能够作为wasm registry
  - 可能。通过类似的方式，将其存储在crd，然后通过http方式将其拉取到本地




```
func (c *Package) Get(m *metav1.ObjectMeta) (*fv1.Package, error) {
    relativeUrl := fmt.Sprintf( format: "packages/%v", m.Name)
    relativeUrl += fmt.Sprintf( format: "?namespace=%v", m.Namespace)

    resp, err := c.client.Get(relativeUrl)
    if err != nil : nil, err
    defer resp.Body.Close()

    body, err := handleResponse(resp)
    if err != nil : nil, err

    var f fv1.Package
    err = json.Unmarshal(body, &f)
    if err != nil {
        return nil, err
    }

    return &f, nil
}
```

```
func (c *RESTClient) Get(relativeUrl string) (*http.Response, error) {
    return c.sendRequest(http.MethodGet, c.v2CrdUrl(relativeUrl), headers: nil, reader: nil)
}
```

参考

- containerd/docs
- 重学容器08: 简单理解Containerd的架构
- 重学容器09: Containerd是如何存储容器镜像和数据的
- https://github.com/containerd/containerd/blob/main/docs/cri/architecture.md
- runc和bundle