SOFT130091.01

云原生软件技术

4. 容器编排 Compose和Swarm

复旦大学软件学院

沈立炜

shenliwei@fudan.edu.cn

编排

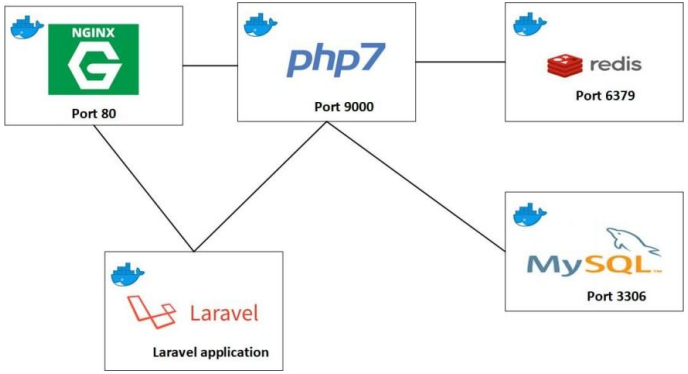
- **编排**
  - ✓ 在计算机领域指对系统的自动配置和管理
- **容器编排**
  - ✓ 容器的配置、部署和管理等任务的自动化
  - ✓ 容器编排可以把开发者从复杂且重复的容器管理工作中解放出来

真实的应用

- 如果只需管理一个容器，不必使用容器编排
- 现实是上线业务时，一般情况下都不会是一个单独的应用容器，通常需要部署多个容器来相互配合
- 一个容器往往无法承载日益复杂的应用，若把应用所有的组件都放到一个容器中运行，就丧失了使用容器的诸多优势
  - ✓ 无法完成快速扩容缩容
  - ✓ 无法对某个组件进行单独部署和升级
  - ✓ 无法灵活调度到其他服务器

真实的应用

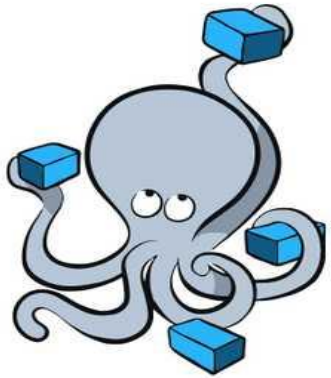
- 一个应用程序由多个容器组成
- 一个容器依赖于另一个容器
- 容器需要按照顺序启动
- 构建容器、部署容器的流程
- 较长的 docker run 命令
- 复杂性与涉及的容器数量成正比



- 多容器的手工管理较为低效，且极易出错，在容器数量增长到一定量级情况下，使用容器编排的必要性就显著增加
- 对于稳定性有极高要求的生产环境，最大可能减少手工操作能够降低出错的风向

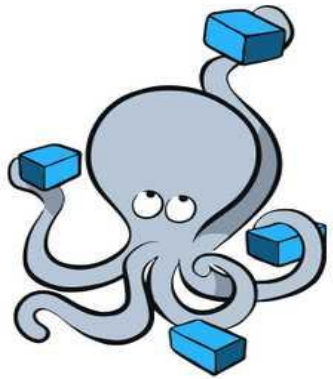
Docker Compose

- Docker提供Compose组件，用于简单的容器编排
- Docker Compose会从文件中读取应用所需的全部容器的定义
  - ✓ Compose文件把项目的运行环境文档化
  - ✓ 默认文件名为docker-compose.yml
- 运行命令启动应用所需的所有容器，准备所需的网络和存储
  - ✓ 一个命令完成烦琐启动多个容器的任务
  - ✓ 命令：docer-compose up
- Docker Compose只能管理单个节点上的容器，更适合在开发和测试环境下使用



Docker Compose

1. 用于定义和运行多容器Docker应用程序的工具
2. 通过Compose，可以使用YAML文件来配置应用程序的服务
3. Compose适用于所有环境：生产(production)、预发布(staging)、开发(development)、测试(testing)以及 CI 工作流程
4. 使用单个命令，可以根据配置创建并启动所有服务



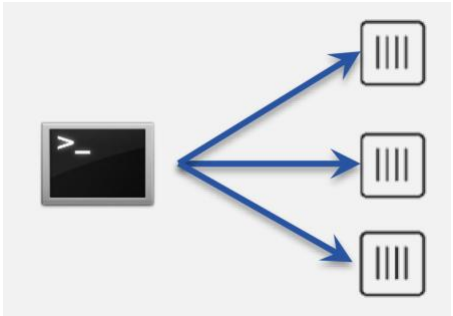
什么是Docker Compose

- Compose中两个重要的概念
  - ✓服务 (service)：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。
  - ✓项目 (project)：由一组关联的应用容器组成的一个完整业务单元，在 docker-compose.yml 文件中定义
- Compose的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷的生命周期管理

多容器应用的部署

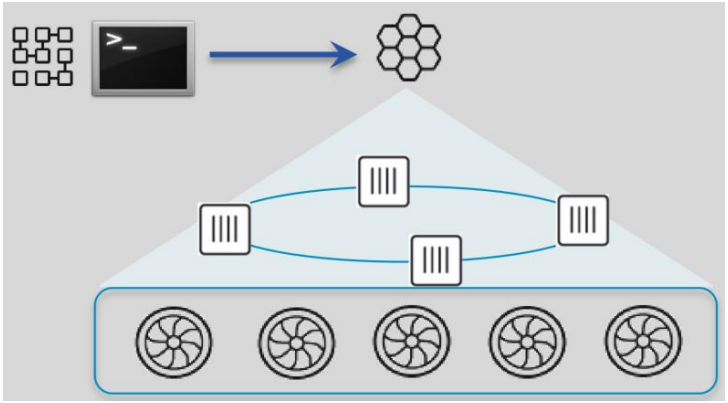
没有Compose

- 一次构建并运行一个容器
- 手动将容器连接在一起
- 必须注意依赖性和启动顺序



有Compose

- 在 compose.yml 文件中定义多容器应用程序
- 单个命令即可部署整个应用程序
- 处理容器依赖性
- 可与 Docker Swarm、网络、卷、通用控制平面配合使用



多容器应用的部署 - WordPress应用示例

没有Compose

```
$docker run -e MYSQL_ROOT_PASSWORD=<pass> -e
MYSQL_DATABASE=wordpress --name wordpressdb -v
"$PWD/database":/var/lib/mysql -d mysql:latest

$docker run -e WORDPRESS_DB_PASSWORD=<pass> --name
wordpress --link wordpressdb:mysql -p 80:80 -v "$PWD/html":/var/www/html
-d wordpress
```

有Compose

```
services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD:
     MYSQL_DATABASE: wordpress
   wordpress:
       depends_on:
           - db
       image: wordpress:latest
    ports:
      - "80:80"
    environment:
      WORDPRESS_DB_PASSWORD=<pass>
```

Docker Compose的命令

docker-compose up

●作用

- ✓ 用于创建并启动 Docker Compose 文件中定义的所有服务
- ✓ 如果服务的容器已经存在，它会尝试重新创建新的容器并启动它们

●特点

- ✓ 创建并启动：第一次运行时，它会根据 docker-compose.yml 文件中的配置创建容器并启动
- ✓ 重新创建容器：如果容器已经存在，docker-compose up 会停止并删除旧的容器，然后创建新的容器
- ✓ 输出日志：默认情况下，docker-compose up 会输出所有服务容器的日志信息，方便用户查看服务的运行状态
- ✓ 交互式运行：通常在前台运行，用户可以通过 Ctrl+C 停止服务

●使用场景

- ✓ 在开发环境中启动服务，因为每次运行都可以确保使用最新的配置和代码
- ✓ 需要查看服务启动过程中的日志信息时

Docker Compose的命令

docker-compose down

- **作用**
  - ✓ 停止并删除由 docker-compose up 启动的容器、网络、卷和图像，具体行为取决于命令的参数
- **特点**
  - ✓ 停止容器：停止正在运行的容器
  - ✓ 删除容器：删除所有服务容器
  - ✓ 删除网络：删除为服务创建的默认网络
  - ✓ 删除镜像：可以指定参数删除为服务创建的镜像
  - ✓ 删除卷：可以指定参数删除与容器关联的卷
- **使用场景**
  - ✓ 开发环境：用于完全关闭并重置运行的容器和服务配置，建议不要删除卷以保留数据
  - ✓ 生产环境：在生产环境中使用时要格外小心，尤其是删除数据卷时，可能会导致数据丢失。确保在生产环境下已经备份了所有重要数据

Docker Compose的命令

docker-compose start

- **作用**
  - ✓ 用于启动已经存在的、停止的 Docker Compose 服务容器
  - ✓ 不会重新创建容器，只是简单地启动已经存在的容器
- **特点**
  - ✓ 启动已停止的容器：只有当容器已经存在并且处于停止状态时，docker-compose start 才能正常工作
  - ✓ 不重新创建容器：不会根据 docker-compose.yml 文件中的配置重新创建容器，只是启动已有的容器
  - ✓ 无日志输出：默认情况下，docker-compose start 不会输出日志信息，它只是启动容器
  - ✓ 后台运行：通常在后台运行，启动后直接返回命令行
- **使用场景**
  - ✓ 当需要快速启动已经存在的服务容器时，比如在服务器重启后恢复服务
  - ✓ 当不需要重新创建容器，只需要启动已停止的容器时

Docker Compose的命令

docker-compose stop

- **作用**
  - ✓停止正在运行的容器，但不删除它们
- **特点**
  - ✓停止容器：停止正在运行的容器
  - ✓保留数据：容器停止后，数据卷和网络配置不会被删除，容器状态保留
- **使用场景**
  - ✓快速停止服务：当需要快速停止服务容器时，如在服务器维护或升级前
  - ✓保留容器状态：如果希望保留容器的状态和数据，以便后续重新启动，使用 docker-compose stop 比较合适

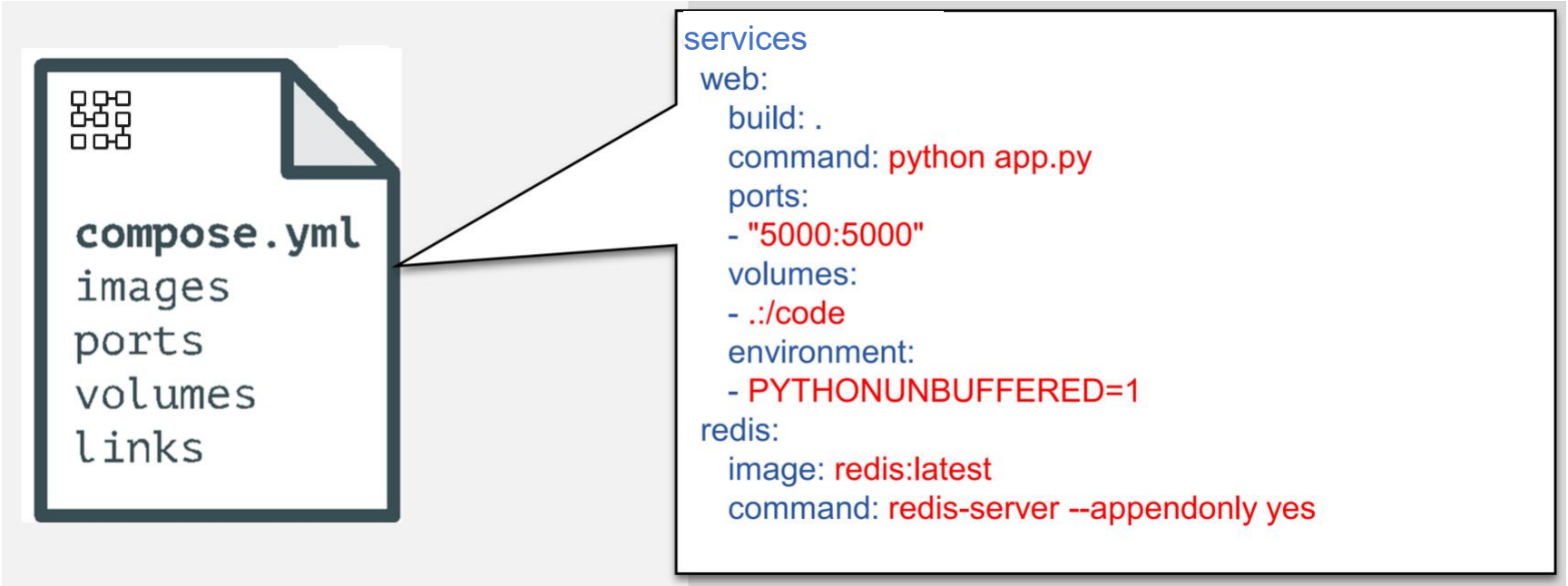
Docker Compose的组成

Services
服务

Volumes
存储卷

Networking
网络连接

Docker Compose的yml文件



Docker Compose的yml文件

```
$ cat docker-compose.yml
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

docker-compose.yml

```
version: "3.9"
```

Compose文件格式版本

```
services:
```

定义应用需要的所有服务

```
web:
  image: bitmyth/accounts-frontend:dev
  depends_on:
    - api
  ports:
    - "8088:80"
```

前端提供的服务

```
api:
  depends_on:
    - mysql
  image: bitmyth/accounts:dev
  ports:
    - "8081:80"
  volumes:
    - ./config:/config
  restart: always
```

后端提供的服务

```
  mysql:
    image: mysql:5.7
    volumes:
      - mysql:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: 123
      MYSQL_DATABASE: accounts
      command: --character-set-server=utf8mb4 --default-time-
zone=+08:00
```

数据库服务

```
  db-migration:
    image: bitmyth/goose:v1.0.0
    depends_on:
      - mysql
    working_dir: /migrations
    volumes:
      - ./src/database/migrations:/migrations
    command: goose up
    environment:
      GOOSE_DRIVER: mysql
      GOOSE_DBSTRING: root:123@tcp(mysql)/accounts
```

数据迁移服务

```
volumes:
  mysql:
```

应用需要用到的数据卷

Docker Compose的环境变量

- 使用 .env 文件

```
$ cat .env
TAG=v1.5

$ cat docker-compose.yml
version: '3'
services:
  web:
    image: "webapp:${TAG}"
```

docker-compose.yml

```yaml
version: "3.9"

services:
  web:
    image: ${ACCOUNT_FRONTEND_IMAGE}
    depends_on:
      - api
    ports:
      - "8088:80"

  api:
    depends_on:
      - mysql
    image: ${ACCOUNT_IMAGE}
    ports:
      - "8081:80"
    volumes:
      - ./config:/config
    restart: always
```

```
ACCOUNT_FRONTEND_IMAGE=bitmyth/accounts-frontend:dev
ACCOUNT_IMAGE=bitmyth/accounts:dev
MYSQL_PASSWORD=123
MYSQL_DB=accounts
```

.env

```yaml
  mysql:
    image: mysql:5.7
    volumes:
      - mysql:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_PASSWORD}
      MYSQL_DATABASE: ${MYSQL_DB}
      command: --character-set-server=utf8mb4 --default-time-zone=+08:00

  db-migration:
    image: bitmyth/goose:v1.0.0
    depends_on:
      - mysql
    working_dir: /migrations
    volumes:
      - ./src/database/migrations:/migrations
    command: goose up
    environment:
      GOOSE_DRIVER: mysql
      GOOSE_DBSTRING: root:${MYSQL_PASSWORD}@tcp(mysql)/${MYSQL_DB}
volumes:
  mysql:
```

Docker Compose的网络连接

- 创建一个名为 myapp_default 的网络
- 使用web的配置创建容器；它以 web 名称加入网络 myapp_default
- 使用db的配置创建容器；它以 db 名称加入网络 myapp_default

```
$docker-compose up

Creating network "acount_default" with the default driver
.....
```

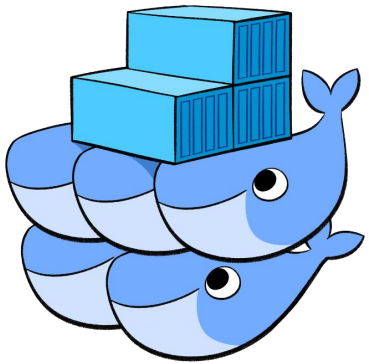
```
version: "3"
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
    ports:
      - "8001:5432"
```

Compose更新应用

- 当更新应用代码后，会构建出新版本的容器镜像
- 不需要手动中止旧容器，然后运行新容器来完成升级
- 只需修改Compose文件中相应的版本镜像，重新运行docker-compose up，会自动中止旧容器，创建新容器

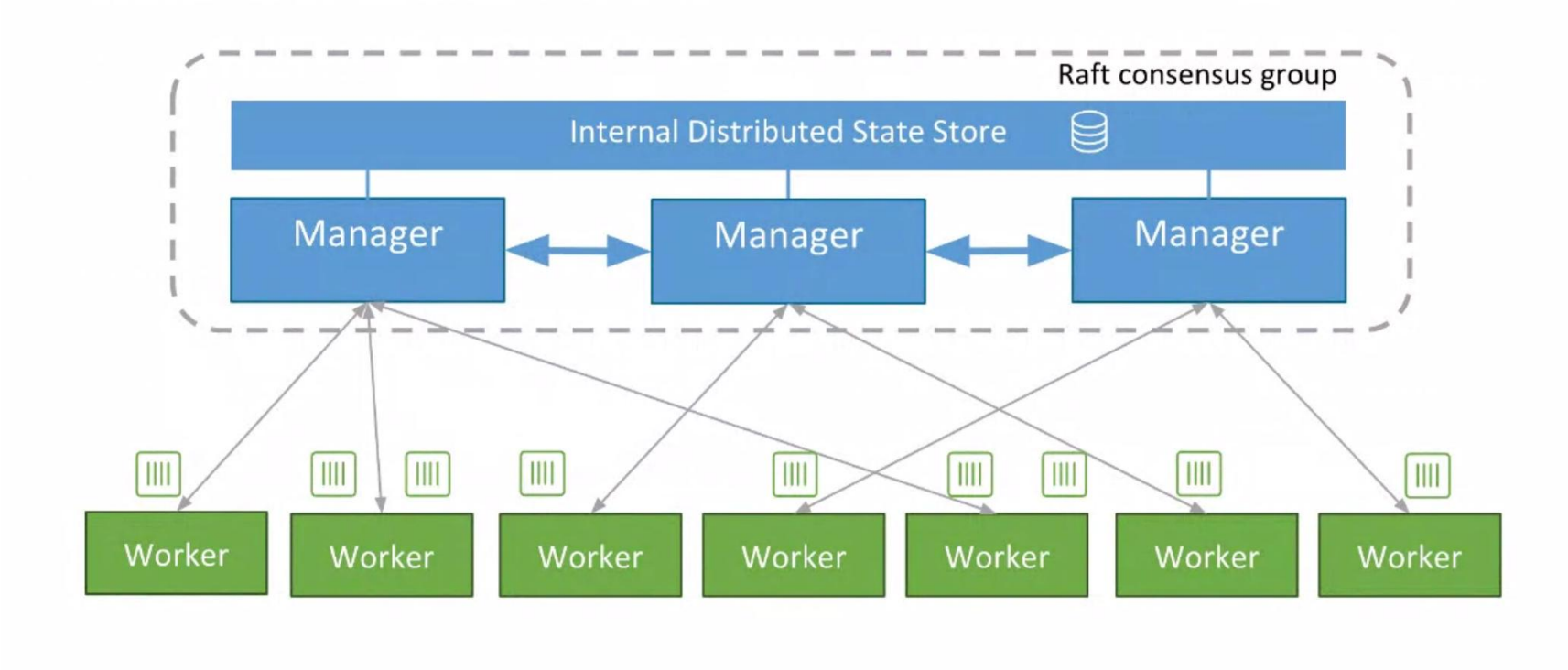
Docker Swarm

- Docker Swarm可以管理多个节点上的容器，具备管理Docker集群的能力，是Docker容器的集群和调度工具
- 借助 Swarm，IT 管理员和开发人员可以将 Docker 节点集群作为单个虚拟系统进行建立和管理



Docker Swarm

Docker Swarm 架构



Docker Swarm 架构

- Swarm集群中的节点分为manager节点和worker节点
  - ✓worker节点只负责运行服务而不负责管理集群
  - ✓manager节点负责管理集群（worker节点），具有leader身份的manager节点负责将任务分配到worker节点
  - ✓manager节点可以执行worker节点的任务，也可设置成只执行manager的任务
  - ✓多个manager节点之间通过Raft算法来达成共识
  - ✓所有manager节点都参与维护Swarm集群内部装填，使其在不同的manager节点保持一致

Raft算法

- Raft 是一种用于分布式系统的共识算法，旨在通过在多个节点之间复制日志来实现高可用性和一致性
- Raft 将系统中的节点分为三种角色：Leader（领导者）、Follower（跟随者）和 Candidate（候选者）。在正常操作下，系统中只有一个 Leader，其他节点都是 Follower。Leader 负责处理所有的客户端请求，并将日志条目复制到所有的 Follower 节点。如果 Leader 失效，系统会通过选举产生一个新的 Leader

Raft算法

- 易于理解：Raft 的设计目标之一是提高可理解性，通过清晰的角色划分和简单的算法流程，使得开发者和运维人员更容易理解和实现
- 高可用性：通过多个节点的协同工作和选举机制，即使部分节点失效，系统仍然能够正常运行，只要大多数节点可用
- 强一致性：Raft 确保所有节点上的日志最终是一致的，从而保证了分布式系统中数据的一致性
- 灵活性：Raft 可以应用于多种分布式系统场景，如分布式存储、配置管理、服务发现等

Raft算法

- 选举过程
  - ✓启动选举：当一个 Follower 节点在超时时间内没有收到 Leader 的心跳时，它会将自己转换为 Candidate 角色，并发起选举
  - ✓投票过程：Candidate 会向其他节点发送投票请求，请求它们投票给自己。其他节点如果在任期内没有投票给其他节点，会投票给第一个请求投票的 Candidate
  - ✓选举成功或失败：如果一个 Candidate 收到了超过半数的投票，它将成为新的 Leader。如果没有任何一个 Candidate 获得足够的投票，系统会进入一个新的选举任期，直到选出新的 Leader

Raft算法

- 日志复制过程
  - ✓处理客户端请求：Leader 收到客户端的请求后，会将请求作为新的日志条目添加到自己的日志中
  - ✓复制日志：Leader 会将新的日志条目发送给所有的 Follower 节点，Follower 收到后会将日志条目添加到自己的日志中，并返回确认信息给 Leader
  - ✓提交日志：当 Leader 收到超过半数的 Follower 的确认后，它会将日志条目标记为已提交，并通知 Follower 节点提交日志条目。提交后的日志条目会被应用到状态机中，产生实际的输出结果

Docker Swarm 的容错能力

- Swarm集群中的manager节点数量决定了集群的容错能力
  - ✓如果集群节点数是n，则集群最多可容忍（n-1/2）个节点失败
  - ✓例如，3个节点的集群，最多容忍1个节点失败；4个节点的集群，也最多容忍1个节点失败
  - ✓建议使用奇数个manager节点

Docker Swarm 特点

- 与 Docker Engine 集成的集群管理
  - ✓节点即Docker主机，运行Docker Engine，可部署应用程序服务
- 声明式服务模型
  - ✓使用声明式方法定义应用程序堆栈中各种服务的所期望状态
- 缩放
  - ✓当扩大或缩小规模时，Swarm的manger节点会通过添加或删除任务来自动适应以维持期望的状态

Docker Swarm 特点

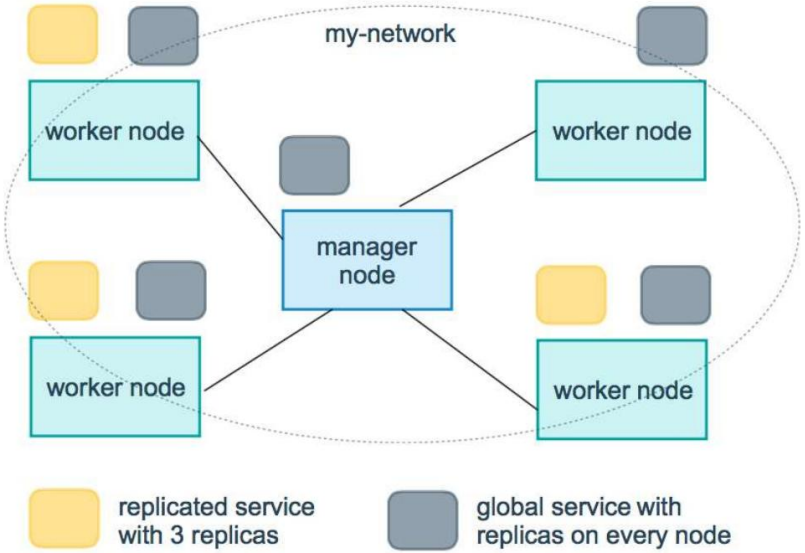
- 期望的状态协调
  - ✓Swarm的manager节点持续监控集群状态并协调实际状态与表达的期望状态之间的任何差异
- 服务发现
  - ✓Swarm 管理器节点为 Swarm 中的每个服务分配唯一的 DNS 名称，并对运行的容器进行负载平衡
- 滚动更新
  - ✓在升级时，可以增量地将服务更新应用到节点； 如果出现任何问题，可以将任务回滚到服务的先前版本

服务（Service）

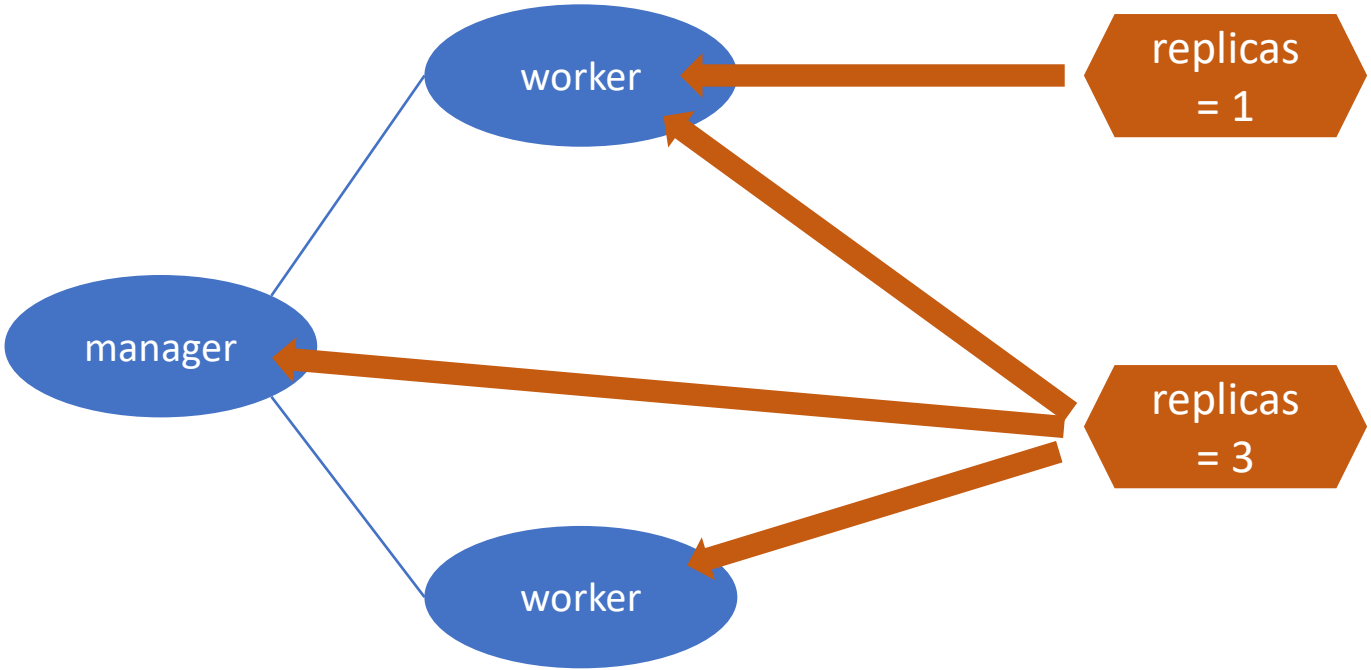
- 服务实际上是“生产中的容器”
- 一项服务仅运行一个镜像，但它对镜像的运行方式进行了编码：应使用哪些端口、应运行多少个容器副本以便服务具有所需的容量等
- 扩展（Scaling）服务会改变运行该软件的容器实例的数量

服务（Service）

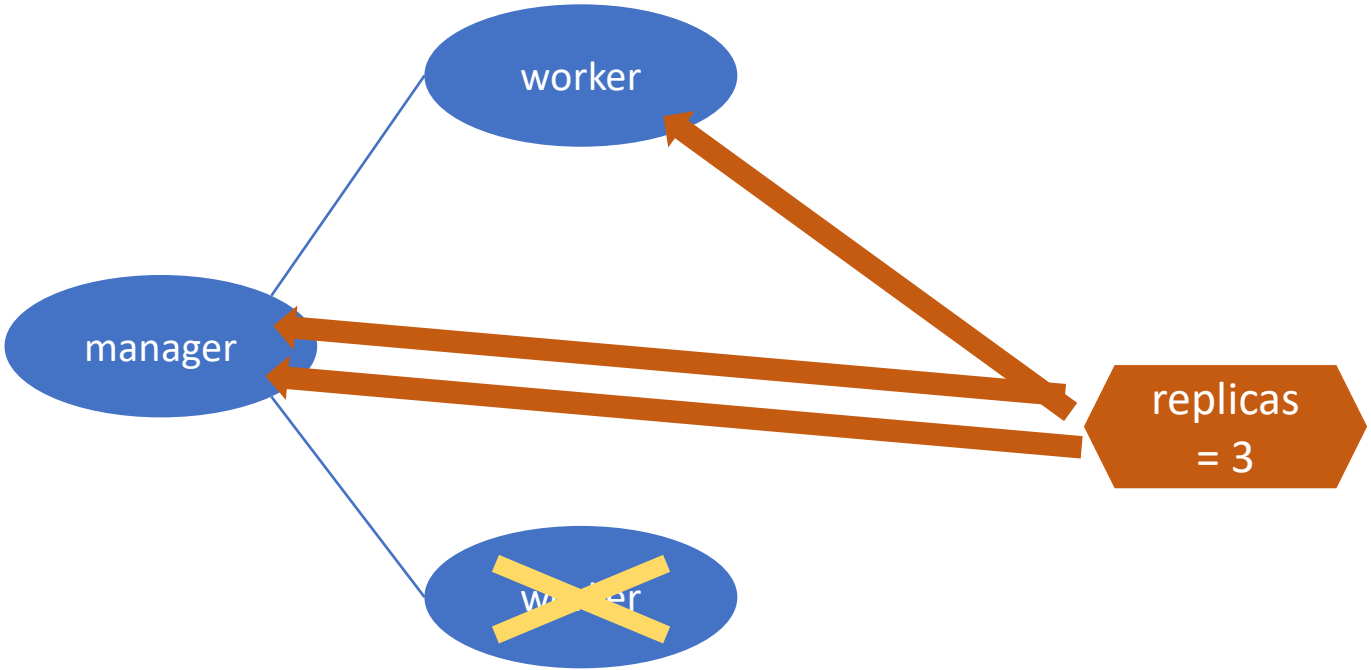
- 两种类型的服务
- 复制（Replicated）
  - ✓指定想要的执行相同任务的副本数量
- 全局（Global）
  - ✓在每个节点上运行一项任务的服务



服务部署、伸缩与调度



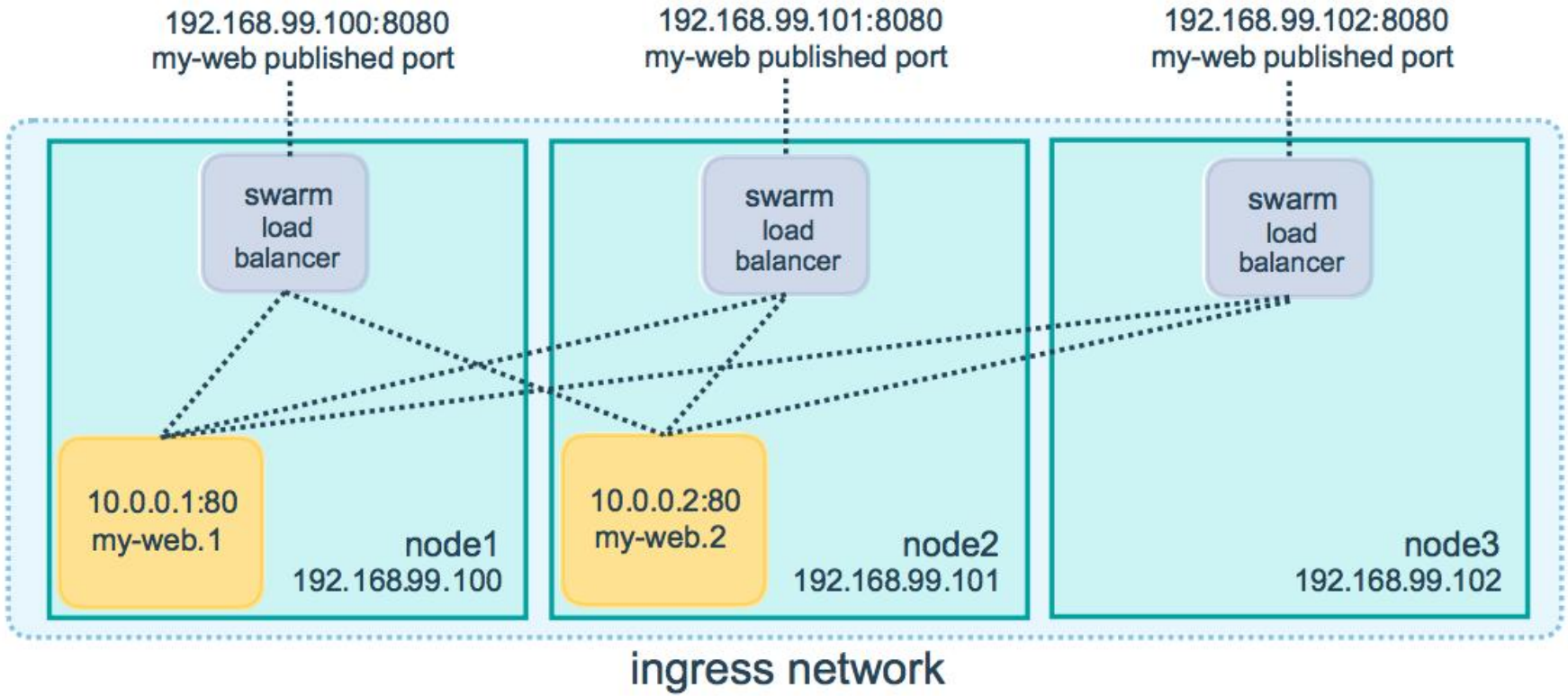
服务部署、伸缩与调度



路由网格

- Swarm集群中的服务一般会部署到多个节点上运行，当Swarm中的服务通过某个网络端口对外提供服务时，Swarm的路由网络功能使集群中任何一个节点都能在服务暴露的端口上接受请求，包括没有运行服务实例的节点，也可以在服务暴露的端口接受请求
- Swarm路由网格会把节点收到的请求自动转发到运行服务的容器实例中
- 有利于提高服务的可用性，例如当某个节点上运行服务的容器实例意外中止后，这个节点上收到的请求会被转发到正常的节点上进行处理

路由网格


192.168.99.100:8080
my-web published port
192.168.99.101:8080
my-web published port
192.168.99.102:8080
my-web published port
swarm load balancer
swarm load balancer
swarm load balancer
10.0.0.1:80
my-web.1
node1
192.168.99.100
10.0.0.2:80
my-web.2
node2
192.168.99.101
node3
192.168.99.102
ingress network

Swarm部署

- 与Docker Compose类似，Swarm也支持在描述应用的清单文件中读取容器的所有配置，包括镜像、端口、存储和网络等，并且文件格式和Compose是兼容的，称为stack文件
- 部署应用accounts
  - ✓docker stack deploy --compose-file stack.yaml accounts
- 关闭应用
  - ✓docker stack rm accounts

```yaml
version: "3.9"
services:
  web:
    image: ${ACCOUNT_FRONTEND_IMAGE}
    depends_on:
      - api
    ports:
      - "8080:80"
  api:
    image: ${ACCOUNT_IMAGE}
    ports:
      - "8081:80"
    configs:
      - source: plain_config
        target: /config/plain.yaml
        mode: 0440
    secrets:
      - source: secret_config
        target: /config/secret.yaml
        mode: 0440
    deploy:
      replicas: 1
  mysql:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_PASSWORD}
      MYSQL_DATABASE: ${MYSQL_DB}
    # 部署(调度)策略，指定部署到 manager 节点
    # deploy:
    #   placement:
    #     constraints:
    #       - node.role == manager
volumes:
  mysql:
configs:
  plain_config:
    file: ./config/plain.yaml
secrets:
  secret_config:
    file: ./config/secret.yaml
```

```
root@docker-swarm-manager:/home/osboxes/docker-swarm/05-stack# docker stack deploy --compose-file stack.yaml accounts
Since --detach=false was not specified, tasks will be created in the background.
In a future release, --detach=false will become the default.
Creating network accounts_default
Creating secret accounts_secret_config
Creating config accounts_plain_config
Creating service accounts_mysql
Creating service accounts_web
Creating service accounts_api
```

```
root@docker-swarm-manager:/home/osboxes/docker-swarm/05-stack# docker stack remove accounts
Removing service accounts_api
Removing service accounts_mysql
Removing service accounts_web
Removing secret accounts_secret_config
Removing config accounts_plain_config
Removing network accounts_default
```

SOFT130091.01
云原生软件技术

End

4. 容器编排 Compose和Swarm