## KAFKA

**Current offsets vs Committed offsets**

**https://www.logicbig.com/tutorials/misc/kafka.html**

[**https://www.logicbig.com/tutorials/misc/kafka/auto-committing-offsets.html**](https://www.logicbig.com/tutorials/misc/kafka/auto-committing-offsets.html)

* Current Offset (position) is the offset from which next new record will be fetched (when it's available).
* Last Committed Offset is the last committed offset for the given partition. Committing an offset for a partition is the action that the offset has been processed so that Kafka cluster won't send the committed records for the same partition. Committed offset is important in case of a consumer recovery or rebalancing.

**Auto Commit**

For a consumer, we can enable auto commit by setting enable.auto.commit property to true. The default is true. In that case the consumer's offset will be periodically committed in the background. If this property is set to false then no offsets are committed. For a consumer, the property auto.commit.interval.ms, specifies the frequency in milliseconds that the consumer offsets are auto-committed to Kafka if enable.auto.commit is set to true.

**Manually** **Committing** **offsets**

By setting auto.commit.offset=false, offsets can only be committed when the application explicitly chooses to do so. This can be done by using KafkaConsumer#commitSync() and KafkaConsumer#CommitAsync().

**Kafka** - **Auto Committing Offsets**

For a consumer, we can enable/disable auto commit by setting enable.auto.commit = true/false

When set to true, consumer's offset will be periodically committed in the background.If this property is set to false then no offsets are committed

The property auto.commit.interval.ms specifies the frequency in milliseconds that the consumer offsets are auto-committed to Kafka. This only applies if enable.auto.commit is set to true.

Kafka consumer will auto commit the offset of the last message received in response to its poll() call.

KafkaConsumer#position() method

public long position(TopicPartition partition)

Gets the offset of the next record that will be fetched (if a record with that offset exists). This method simply returns the current-offset.

KafkaConsumer#committed() method

public Map<TopicPartition, OffsetAndMetadata> committed(final Set<TopicPartition> partitions)

Gets the last committed offsets for the given partitions.

**kafka** **提交offset**

poll()方法总是返回由生产者写入kafka但还没有被消费者读取过的记录(last committed offset~current offset之间的消息已经被消费者抓取，但是current offset~HW(不包含HW)之间的记录还没poll到，消费者不能poll到HW~LEO间的消息，因为这部分的消息对消费者来说，是不可见的)，因此可以追溯到哪些记录是被群组里的那个消费者读取的。那么消费者是如何提交offset的呢？consumer 往一个叫做\_consumer\_offset的特殊主题发送消息，消息里面包含每个分区的偏移量。如果消费者发生崩溃或者有新的消费者加入群组，就会触发rebanlance(再均衡)，完成在均衡之后，每个消费者可能分配到新的分区，而不是之前处理的那个，为了能够继续之前的工作，消费者需要读取每个分区最后一次提交的偏移量，然后从偏移量指定的地方继续处理。

* 如果提交的偏移量小于客户端处理的最后一个消息的offset，则两者之间的数据就会被重复消费。
* 如果提交的偏移量大于客户端处理的最后一个消息的offset，则两者职期间的数据就会丢失。

所以，偏移量的提交对客户端有很大的影响。

每个consumer消费时会把进度记录在\_\_consumer\_offsets开头的目录中(本质是一个topic)，这样即使consumer宕机了，重启后也不至于从头开始消费。这个消费进度信息在老版本是由zookeeper管理，在新版本存由kafka管理。

**自动提交**

如果enable.auto.commit =true，那么每过5s，consumer会自动把poll()方法接收到的最大offset提交上去。提交时间间隔由auto.commit.interval.ms 控制，默认是 5s。自动提交是在消费者的轮询里进行的。consumer每次在进行查询的时候会检查是否该提交偏移量了，如果是，那么就会提交从上一次轮询返回的偏移量。

**同步提交**

处理完当前批次的消息，在轮询更多的消息之前，调用commitSync方法提交当前批次最新的offset。只要没有发生不可恢复的错误，commitSync()会一直尝试直至提交成功，如果提交失败，我们也只能把异常记录到日志里。

**异步提交**

提交一个offset，然后继续做其他事情，如果提交失败，错误信息和偏移量会被记录下来。commitAsync和commitSync不同在于，它不会一直重试，是因为有可能在它收到服务器响应之前，可能有一个更大的offset已经提交成功。另外commitAsync支持回调。

**Kafka常见的导致重复消费原因和解决方案**

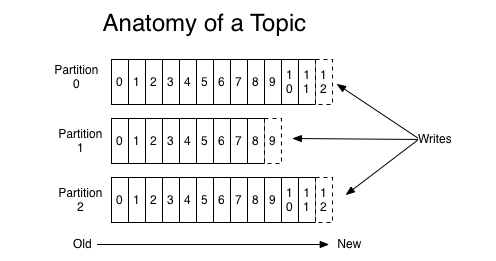
导致kafka的重复消费问题原因在于，已经消费了数据，但是offset没来得及提交（比如Kafka没有或者不知道该数据已经被消费）。总结以下场景导致Kakfa重复消费：

* 原因1：强行kill线程，导致消费后的数据，offset没有提交（消费系统宕机、重启等）。
* 原因2：设置offset为自动提交，关闭kafka时，如果在close之前，调用consumer.unsubscribe()则有可能部分offset没提交，下次重启会重复消费。解决方法是设置offset自动提交为false。一旦设置了enable.auto.commit 为true，Kafka会保证在开始调用poll方法时，提交上次 poll返回的所有消息。从顺序上来说，poll方法的逻辑是先提交上一批消息的位移，再处理下一批消息，因此它能保证不出现消费丢失的情况。
* 原因3:（重复消费最常见的原因）：消费后的数据，当offset还没有提交时，partition就断开连接。比如消费数据处理很耗时，导致超过了Kafka的session timeout时间，那么就会re-blance重平衡，此时有一定几率offset没提交，会导致重平衡后重复消费。
* 原因4：当消费者重新分配partition的时候，可能出现从头开始消费的情况，导致重发问题。
* 原因5：当消费者消费的速度很慢的时候，可能在一个session周期内还未完成，导致心跳机制检测报告出问题。
* 原因6：并发很大，在规定的时间（session.time.out默认30s）内没有消费完，就会可能导致reblance重平衡，导致一部分offset自动提交失败，然后重平衡后重复消费。

**Kafka核心之偏移量机制**

**https://juejin.cn/post/6844903925569568775**

**一、生产者Offset**

首先，Kafka是通过生产者将消息发送给某一个topic，消费者再消费这个topic的消息，当然可能有多个生产者，多个消费者，还可能有消费者组。当生产者将消息发送给某一个topic时，无论写入的是哪一个分区，但只要写入，一定是每一个分区都有一个offset，这个offset就是生产者的offset，同时也是这个分区的最新最大的offset。  


**topic partition offset 这三个唯一确定一条消息**。

**二、消费者Offset**

生产者的offset其实就是最新的offset。消费者的offset是他自己维护的，他可以选择分区最开始，最新，也可以记住他消费到哪了。

**Kafka offset的维护**

**https://blog.csdn.net/WLY19950908/article/details/118308078**

由于Consumer在消费过程中可能会出现断电宕机等故障，Consumer恢复后，需要从故障前的位置继续消费。所以Consumer需要实时记录自己消费到了哪个Offset，以便故障恢复后继续消费。

Kafka 0.9 版本之前，Consumer默认将Offset保存在Zookeeper中，从0.9版本开始，Consumer默认将Offset保存在Kafka一个内置的**\_\_consumer\_offsets** Topic中。

**zookeeper维护**

kafka 0.9以前的版本是将offset存储在zookeeper上的，kafka在传输数据时，数据消费成功就会修改偏移量，这样就可以保证数据不会丢失而导致传输出错；但是存在一个问题：那就是每次消费数据时都要将数据的offset写入一次，效率比较低，而且zookeeper与kafka的offset变化确认也需要走网络IO，这样就会给offset的维护带来不稳定性和低效。

**kafka自己维护offset**

**Kafka offset管理**

**https://www.jianshu.com/p/449074d97daf**

Kafka中的每个partition都由一系列有序的、不可变的消息组成，这些消息被连续的追加到partition中。partition中的每个消息都有一个连续的序号，用于partition唯一标识一条消息。

Offset记录着下一条将要发送给Consumer的消息的序号。Offset从语义上来看拥有两种：Current Offset和Committed Offset。

**Current Offset**

Current Offset保存在Consumer客户端中，它表示Consumer希望收到的下一条消息的序号。它仅仅在poll()方法中使用。

例如，Consumer第一次调用poll()方法后收到了20条消息，那么Current Offset就被设置为20。这样Consumer下一次调用poll()方法时，Kafka就知道应该从序号为21的消息开始读取。这样就能够保证每次Consumer poll消息时，都能够收到不重复的消息。

**Committed Offset**

Committed Offset保存在Broker上，它表示Consumer已经确认消费过的消息的序号。主要通过commitSync和commitAsync API来操作。

举个例子，Consumer通过poll() 方法收到20条消息后，此时Current Offset就是20，经过一系列的逻辑处理后，并没有调用consumer.commitAsync()或consumer.commitSync()来提交Committed Offset，那么此时Committed Offset依旧是0。

Committed Offset主要用于Consumer Rebalance。在Consumer Rebalance的过程中，一个partition被分配给了一个Consumer，那么这个Consumer该从什么位置开始消费消息呢？答案就是Committed Offset。另外，如果一个Consumer消费了5条消息（poll并且成功commitSync）之后宕机了，重新启动之后它仍然能够从第6条消息开始消费，因为Committed Offset已经被Kafka记录为5。

总结一下，

* Current Offset是针对Consumer的poll过程的，它可以保证每次poll都返回不重复的消息；而Committed Offset是用于Consumer Rebalance过程的，它能够保证新的Consumer能够从正确的位置开始消费一个partition，从而避免重复消费。
* Kafka 0.9前，Committed Offset信息保存在zookeeper的[consumers/{group}/offsets/{topic}/{partition}]目录中。而0.9之后，所有的offset信息都保存在了Broker上的一个名为\_\_consumer\_offsets的topic中。
* Kafka集群中offset的管理都是由Group Coordinator中的Offset Manager完成的。

深入解析Kafka的offset管理

<https://segmentfault.com/a/1190000041263079>

https://blog.csdn.net/zjjcchina/article/details/122425951

3、Kafka名词解释以及工作方式

* Broker：一台kafka服务器就是一个broker。一个集群由多个broker组成。一个broker可以容纳多个topic。
* Producer：消息生产者，向kafka broker发送消息的客户端。
* Consumer：消息消费者，向kafka broker取消息的客户端。
* Consumer Group（CG）：若干个Consumer组成的集合。这是kafka用来实现一个topic消息的广播（发给所有的consumer）和单播（发给任意一个consumer）的手段。一个topic可以有多个CG。topic的消息会复制到所有的CG，但每个CG只会把消息发给该CG中的一个consumer。如果需要实现广播，只要每个consumer有一个独立的CG就可以了。要实现单播只要所有的consumer在同一个CG。用CG还可以将consumer进行自由的分组而不需要多次发送消息到不同的topic。
* Topic：Kafka中的消息以Topic为单位进行划分，生产者将消息发送到特定的Topic，而消费者负责订阅Topic的消息并进行消费。Kafka将Topic进行分区，分区可以并发读写。
* Partition：分区，为了实现扩展性，一个topic可以分布在多个broker上，一个topic可以分为多个partition，每个partition都是一个有序的队列。partition中的每条消息都会被分配一个有序的id（offset）。kafka只保证同一个partition中的消息顺序，不保证一个topic的整体（多个partition之间）的顺序。生产者和消费者使用时可以指定topic中的具体partition。
* Replica副本：在kafka中，每个主题可以有多个分区，每个分区又可以有多个副本。这多个副本中，只有一个是leader，而其他的都是follower副本。仅有leader副本可以对外提供服务。多个follower副本通常存放在和leader副本不同的broker中。通过这样的机制实现了高可用，当某台机器挂掉后，其他follower副本也能迅速”转正“，开始对外提供服务。
* offset：消费偏移量，topic中的每个分区都是有序且顺序不可变的记录集，并且不断地追加到结构化的log文件。分区中的每一个记录都会分配一个id号来表示顺序，我们称之为offset，offset用来唯一的标识分区中每一条记录。可以设置为“自动提交”与“手动提交”。

4、AR、ISR、OSR、HW、LEO、LSO等分别代表什么？

* AR: Assigned Replicas 指当前分区中的所有副本。
* ISR: In-Sync Replicas 副本同步队列。ISR中包括Leader和Follower。如果Leader进程挂掉，会在ISR队列中选择一个服务作为新的Leader。replica.lag.max.message(延迟条数)和replica.lag.time.max.ms(延迟时间)两个参数决定一台服务器是否可以加入ISR副本队列，在0.10版本之后移除了replica.lag.max.message(延迟条数)参数，防治服务频繁的进出队列。任意一个维度超过阈值都会把Follower踢出ISR，存入OSR列表，新加入的Follower也会先存放在OSR中。
* OSR：Out-of-Sync Replicas非同步副本队列。与leader副本同步滞后过多的副本组成OSR。如果OSR集合中有follower副本“追上”了leader副本，那么leader副本会把它从OSR集合转移至ISR集合。默认情况下，当leader副本发生故障时，只有ISR集合中的副本才有资格被选举为新的leader，而OSR集合中的副本则没有任何机会， 这个原则可通过修改unclean.leader.election.enable参数配置来改变。
* LogStartOffset，日志文件的起始偏移量，一般情况下，等于第一个日志分段的baseOffset，但不是绝对的，该值可通过DeleteRecordsRequest请求、使用kafka-delete-records.sh 脚本、日志的清理和截断等操作进行修改。
* LW是Low Watermark的缩写，俗称“低水位”，代表AR集合中最小的logStartOffset值。
* LSO(LastStableOffset)，与kafka事务有关。对于未完成的事务而言，LSO的值等于事务中的第一条消息所在的位置（firstUnstableOffset）；对于已经完成的事务而言，它的值等同于HW相同。
* LEO(Log End Offset)，标识当前日志文件中下一条待写入消息的offset。LEO的大小相当于当前日志分区中最后一条消息的offset值加1。ISR 集合中的每个副本都会维护自身的LEO，ISR 集合中最小的LEO即为分区的HW，对消费者而言，只能消费HW之前的消息。
* HW(High Watermark)：replica高水印值，副本中最新一条已提交消息的位移。leader的HW值也就是实际已提交消息的范围，每个replica都有HW值，但仅仅leader中的HW才能作为标示信息，就是说当成功完成消息备份(成功同步给follower replica后)才会更新HW的值，代表消息理论上已经不会丢失，可以认为“已提交”。HW是控制消费者可读取消息范围的重要字段。一个普通消费者只能“看到”Leader 副本上介于LSO和HW(不含)之间的 所有消息。水位以上的消息是对消费者不可见的。

Kafka中的HW、LEO、LSO、LW等分别代表什么？

* LogStartOffset，一般情况下，日志文件的起始偏移量 logStartOffset 等于第一个日志分段的 baseOffset，但这并不是绝对的，logStartOffset 的值可以通过 DeleteRecordsRequest 请求、使用 kafka-delete-records.sh 脚本、日志的清理和截断等操作进行修改。
* LW是Low Watermark的缩写，俗称“低水位”，代表AR集合中最小的logStartOffset值。
* LSO，特指LastStableOffset，与kafka 事务有关。对于未完成的事务，LSO的值等于事务中的第一条消息所在的位置(firstUnstableOffset)；对于已经完成的事务而言，它的值等同于HW相同。

LEO是Log End Offset的缩写，它表示了当前日志文件中下一条待写入消息的offset。分区ISR集合中的每个副本都会维护自身的LEO，而ISR集合中最小的LEO即为分区的HW。

HW是 High Watermark 的缩写，俗称高水位，分区ISR集合中的每个副本都会维护自身的LEO，而ISR集合中最小的LEO即为分区的HW。

所以，HW、LW 是分区层面的概念；而LEO、LogStartOffset 是日志层面的概念；LSO 是事务层面的概念。

<https://blog.csdn.net/u013256816/article/details/88985769>

**LogEndOffset**

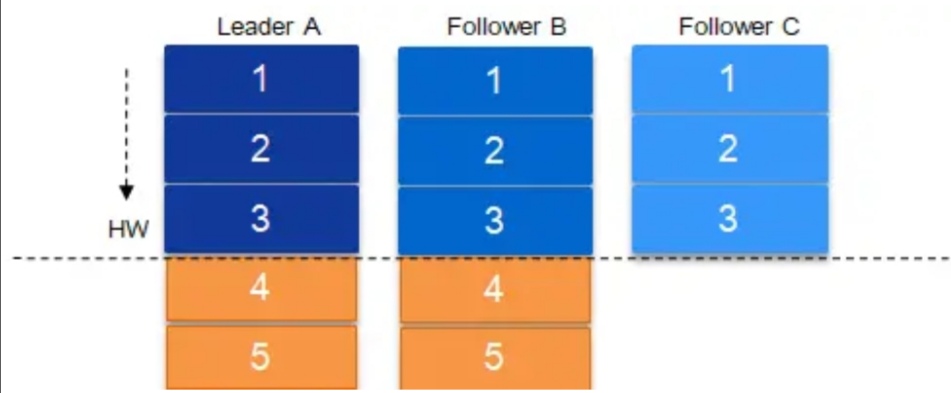
<https://juejin.cn/post/6938234746837139463>



LEO (Log End Offset) 代表Partition的最高日志位移，其值对消费者不可见。LEO是所有的副本都会有的一个offset标记，它指向追加到当前副本的最后一个消息的offset。当生产者向Leader副本追加消息的时候，Leader副本的LEO标记会递增；当FollowerpLeader副本拉取消息并更新到本地的时候，Follower副本的LEO就会增加。注意，数字15所在的方框是虚线，这就说明，这个副本当前只有15条消息，位移值是从0 到14，下一条新消息的位移是15。显然，介于高水位和LEO之间的消息就属于未提交消息。

**主要作用**

Kafka 所有副本都有对应的高水位和 LEO 值，而不仅仅是 Leader 副本。只不过 Leader 副本比较特殊，Kafka使用Leader副本的高水位来定义所在分区的高水位。换句话说，分区的高水位就是其Leader副本的高水位。



比如在ISR（In-Sync-Replicas）副本数等于3的情况下（如上图所示），消息发送到Leader A之后会更新LEO的值，Follower B和Follower C也会实时拉取Leader A中的消息来更新自己。

HW就表示A、B、C三者同时达到的日志位移，也就是A、B、C三者中LEO最小的那个值。由于B、C拉取A消息之间延时问题，所以HW必然不会一直与Leader的LEO相等，即LEO>=HW。

**LastStableOffset**

Last Stable Offset 简称LSO，它与kafka的事物有关。当消费端的参数isolation.level 设置为“read\_committed"的时候，那么消费者就会忽略事务未提交的消息，既只能消费到LSO(LastStableOffset)的位置，默认情况下，”read\_uncommitted",可以消费到HW(High Watermak)的位置。也就是说开启kafka事务的同时，生产者发送了若干消息,(msg1，msg2)到broker中，如果生产者没有提交事务(执行CommitTransaction)那么对于isolation.level=read\_committed的消费者而言是看不多这些消息的，而isolation.level=read\_uncommitted则可以看到。

## PL/SQL

gRPC up and running

grpc与云原生应用开发

Practical gRPC: https://github.com/grpc/grpc/tree/master/examples/

好人：命运的弱者，恋爱的备胎，职场的螺钉

站在领导的立场想问题，站在自己的立场解决问题。

hextagon

Kafka: The Definitive Guide, 2nd Edition

Oracle PLSQL程序设计(第6版)(上下册)

step=3

regex\_count(substr('aa,bb,cc,bb,dd'', 1, instr('aa,bb,cc,bb,dd', 'bb', -1,1)),',') = step

substr('aa,bb,cc,bb,dd', instr('aa,bb,cc,bb,dd', ',', 1, step), length('bb')) = 'bb'

length(str) - length(replace(str, ',', '')): 统计逗号出现的次数

oracle connect by一般用来查找存在父子关系的数据，也就是树形结构的数据；其返还的数据也能够明确的区分出每一层的数据。

select rownum from dual connect by rownum<=10;

select

REGEXP\_SUBSTR('01#02#03#04', '[^#]+', 1, rownum) as newport

from dual

connect by rownum <= REGEXP\_COUNT('01#02#03#04', '[^#]+');

DECLARE

   a number;

   b number;

   c number;

FUNCTION findMax(x IN number, y IN number)  RETURN number IS

    z number;

BEGIN

   IF x > y THEN

      z:= x;

   ELSE

      Z:= y;

   END IF;

   RETURN z;

END;

BEGIN

   a:= 23;

   b:= 45;

   c := findMax(a, b);

   dbms\_output.put\_line(' Maximum of (23,45): ' || c);

END;

DECLARE

   a number;

   b number;

   c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

BEGIN

   IF x < y THEN

      z:= x;

   ELSE

      z:= y;

   END IF;

END;

BEGIN

   a:= 23;

   b:= 45;

   findMin(a, b, c);

   dbms\_output.put\_line(' Minimum of (23, 45) : ' || c);

END;

DECLARE

   a number;

PROCEDURE squareNum(x IN OUT number) IS

BEGIN

  x := x \* x;

END;

BEGIN

   a:= 23;

   squareNum(a);

   dbms\_output.put\_line(' Square of (23): ' || a);

END;

## Cucumber

[Cucumber与Junit4集成示例，以及与Spring Boot+Junit5集成示例](https://juejin.cn/post/7141779735742251039)

Cucumber是一个能够理解用普通语言描述的测试用例的行为驱动开发（BDD）的自动化测试工具。

Cucumber用近似自然的语言去描述Feature和场景，根据Feature驱动开发，即行为驱动开发(BDD)。用作技术人员和非技术人员之间验收测试的桥梁。

Cucumber是一个命令行工具，运行后，会执行features中的内容。feature中的step会调用stepdefinitions。

Cucumber三大组成：Features、Step\_definitions、Cucumber command

对应传统测试工具：

Feature (功能)-------------test suite （测试用例集）

Scenario（情景)-----------test case （测试用例）

Given（给定）------------setup（创建测试所需环境）

When（当）--------------test（触发被测事件）

Then（则）---------------assert(断言，验证结果)

三、Features

语法规则：

1. 基于Gherkin。支持语言：# language: en （zh-CN）等
2. Features文件必须以.features命名。
3. 包含title，多个scenarios，每个scenario包含多个step。
4. Step定义必须以关键字Given，When，Then，And开始。

四、Step\_definitions

根据feature文件中定义的step编写对应的测试代码。

五、Cucumber command

1. 运行：\*.feature文件。cucumber会分析feature文件中定义的step，然后去step - definitions寻找相匹配的step，执行step中的代码。
2. 运行结果以html的形式保存，fail的情况查看对应log日志

六、Cucumber的开发过程

1. 创建feature文件，包括feature，scenarios和step。
2. 创建step\_definitions，即代码实现。如果只有step，没有step definitions，会报undefined错。运行feature，控制台中会输出实现方法，只要实现step\_definitions即可。
3. cucumber --format progress。格式化输出。可修改为html，xml等格式输出。
4. 添加断言：一个scenario失败，Cucumber将它标记失败，转去执行下一个。一个step失败，会跳过这个scenario的其余step，从而终止这个scenario。

[cucumber自动化测试官方教程](https://www.jianshu.com/p/60122d38a08a)

[Cucumber 框架Java版从零开始](https://devopstools.cn/2021/11/02/bdd/cucumber-java/)

[JMeter教程](http://www.jmeter.com.cn/2747.html)

## [DevOps 和 SRE](https://zhuanlan.zhihu.com/p/87598465)

DevOps(Development 和 Operations 的组合词)是一种重视“软件开发人员(Dev)”和“IT 运维技术人员(Ops)”之间沟通合作的文化、运动或惯例。

SRE 全称是 Site Reliability Engineering，其职责是确保生产环境服务运行更稳定、健壮、可靠。

DevOps 工程师职能如下：

* 管理应用全生命周期（需求、设计、开发、QA、发布、运行）。
* 关注全流程效率提升，挖掘瓶颈点并将其解决。
* 自动化运维平台设计和研发工作（标准化、自动化、平台化）。
* 支持运维系统，包括 虚拟化技术、资源管理技术、监控技术、网络技术。

SRE 关键词是「高扩展性」「高可用性」。高扩展性是指当服务用户数量暴增时， 应用系统以及支撑其服务（服务器资源、网络系统、数据库资源）可以在不调整系统结构，不强化机器本身性能 ，仅仅增加实例数量方式进行扩容。高可用性是指，应用架构中任何环节出现不可用时，比如应用服务、网关、数据库 等系统挂掉，整个系统可以在可预见时间内恢复并重新提供服务。当然，既然是「高」可用， 那么这个时间一般期望在分钟级别。SRE 职能可以概括为以下：

* 为应用、中间件、基础设施等提供 选型、设计、开发、容量规划、调优、故障处理。
* 为业务系统提供基于可用性、可扩展性考虑决策，参与业务系统设计和实施
* 定位、处理、管理故障，优化导致故障发生相关部件
* 提高各部件资源利用率

## 09/18/2022

### WebSocket

一、什么是WebSocket

WebSocket是一种在单个TCP连接上进行全双工通信的协议。WebSocket 使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。

在WebSocket API中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接， 并进行双向数据传输。

WebSocket是一种计算机网络应用层的协议，用来弥补http协议在持久通信能力上的不足。

WebSocket协议在2008年诞生，2011年成为国际标准。现在最新版本浏览器都已经支持了。

它的最大特点就是，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于服务器推送技术的一种。

WebSocket 的其他特点包括：

1. 建立在TCP协议之上，服务器端的实现比较容易。
2. 与HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用HTTP协议，因此握手时不容易屏蔽，能通过各种HTTP代理服务器。
3. 数据格式比较轻量，性能开销小，通信高效。
4. 可以发送文本，也可以发送二进制数据。
5. 没有同源限制，客户端可以与任意服务器通信。
6. 协议标识符是ws（如果加密，则为wss），服务器网址就是 URL

### [什么是WebSocket，它与HTTP有何不同？](https://segmentfault.com/a/1190000022020185)

**HTTP协议**

HTTP是单向的，客户端发送请求，服务器发送响应。举例来说，当客户端向服务器发送请求时，该请求以HTTP或HTTPS的形式发送，在接收到请求后，服务器会将响应发送给客户端。每个请求都与一个对应的响应相关联，在发送响应后客户端与服务器的连接会被关闭。

每个HTTP或HTTPS请求每次都会新建与服务器的连接，并且在获得响应后，连接将自行终止。

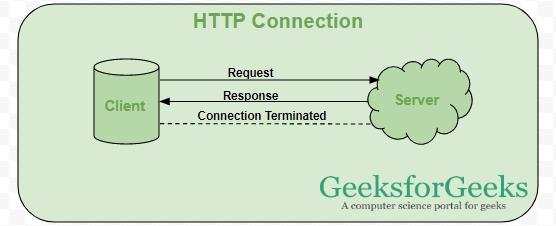
HTTP是在TCP之上运行的无状态协议，TCP是一种面向连接的协议，它使用三向握手方法保证数据包传输的传递并重新传输丢失的数据包。

HTTP可以运行在任何可靠的面向连接的协议（例如TCP，SCTP）的上层。当客户端将HTTP请求发送到服务器时，客户端和服务器之间将打开TCP连接，并且在收到响应后，TCP连接将终止，每个HTTP请求都会建立单独的TCP连接到服务器，例如如果客户端向服务器发送10个请求，则将打开10个单独的HTTP连接。并在获得响应后关闭。

理解上面这段关于 HTTP的描述时我觉得还要了解一下HTTP长连接的概念，以及HTTP与TCP的关系，简单概括一下就是：

* HTTP**协议的长连接和短连接**，实质上是TCP协议的长连接和短连接。
* 每个HTTP连接完成后，其对应的TCP连接并不是每次都会关闭。从 HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头有加入这个头部字段：Connection:keep-alive
* 在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache，Nginx，Nginx中这个默认时间是75s）中设定这个时间。实现长连接要客户端和服务端都支持长连接。
* HTTP属于应用层协议，在传输层使用TCP协议，在网络层使用IP协议。IP协议主要解决网络路由和寻址问题，TCP协议主要解决如何在IP层之上可靠的传递数据包，使在网络上的另一端收到发端发出的所有包，并且顺序与发出顺序一致。TCP有可靠，面向连接的特点。

HTTP消息信息是用ASCII编码的，每个HTTP请求消息均包含HTTP协议版本（HTTP/1.1，HTTP/2），HTTP方法（GET/POST等），HTTP标头（Content-Type，Content-Length），主机信息等。以及包含要传输到服务器的实际消息的正文（请求主体）。HTTP标头的大小从200字节到2KB不等，HTTP标头的常见大小是700-800字节。当Web应用程序在客户端使用更多cookie和其他工具扩展代理的存储功能时，它将减少HTTP标头的荷载。

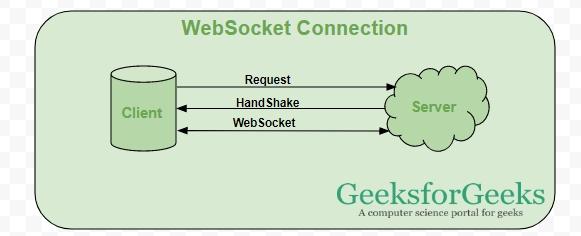


**WebSocket协议**

WebSocket是双向的，在客户端-服务器通信的场景中使用的全双工协议，与HTTP不同，它以ws://或wss://开头。它是一个有状态协议，这意味着客户端和服务器之间的连接将保持活动状态，直到被任何一方（客户端或服务器）终止。在通过客户端和服务器中的任何一方关闭连接之后，连接将从两端终止。

让我们以客户端-服务器通信为例，每当我们启动客户端和服务器之间的连接时，客户端-服务器进行握手随后创建一个新的连接，该连接将保持活动状态，直到被他们中的任何一方终止。建立连接并保持活动状态后，客户端和服务器将使用相同的连接通道进行通信，直到连接终止。

新建的连接被称为WebSocket。一旦通信链接建立和连接打开后，消息交换将以双向模式进行，客户端-服务器之间的连接会持续存在。如果其中任何一方（客户端服务器）宕掉或主动关闭连接，则双方均将关闭连接。套接字的工作方式与HTTP的工作方式略有不同，状态代码101表示WebSocket中的交换协议。



**何时使用WebSocket**

* 即时Web应用程序：即时Web应用程序使用一个Web套接字在客户端显示数据，这些数据由后端服务器连续发送。在WebSocket中，数据被连续推送/传输到已经打开的同一连接中，这就是为什么WebSocket更快并提高了应用程序性能的原因。  
  例如在交易网站或比特币交易中，这是最不稳定的事情，它用于显示价格波动，数据被后端服务器使用Web套接字通道连续推送到客户端。
* 游戏应用程序：在游戏应用程序中，你可能会注意到，服务器会持续接收数据，而不会刷新用户界面。屏幕上的用户界面会自动刷新，而且不需要建立新的连接，因此在WebSocket游戏应用程序中非常有帮助。
* 聊天应用程序：聊天应用程序仅使用WebSocket建立一次连接，便能在订阅户之间交换，发布和广播消息。它重复使用相同的WebSocket连接，用于发送和接收消息以及一对一的消息传输。

**不能使用WebSocket的场景**

如果我们需要通过网络传输的任何实时更新或连续数据流，则可以使用WebSocket。

如果我们要获取旧数据，或者只想获取一次数据供应用程序使用，则应该使用HTTP协议，不需要很频繁或仅获取一次的数据可以通过简单的HTTP请求查询，因此在这种情况下最好不要使用WebSocket。

注意：如果仅加载一次数据，则RESTful Web服务足以从服务器获取数据。

**总结**

| WEBSOCKET | HTTP |
| --- | --- |
| WebSocket是一种双向通信协议，可以通过重用已建立的连接通道将数据从客户端发送到服务器，或者从服务器发送到客户端。连接保持活动状态，直到被客户端或服务器终止。 | HTTP协议是TCP协议之上的单向协议，TCP是面向连接的传输层协议，我们可以在获得响应HTTP连接关闭后再使用HTTP请求方法来创建连接。 |
| 几乎所有的实时应用程序（如（交易，监视，通知）服务）都使用WebSocket在单个通信通道上接收数据。 | 简单的RESTful应用程序使用无状态的HTTP协议。 |
| 所有经常更新的应用程序都应该使用WebSocket，它比HTTP连接更快。 | 当我们不想在特定时间内保留连接或不重复使用单个连接来传输数据时使用HTTP，HTTP连接的速度比WebSocket慢。 |

原文链接：[https://www.geeksforgeeks.org...](https://link.segmentfault.com/?enc=NMdJMwqWFWnKf3TOXQJW%2Bg%3D%3D.BlMRLcOK9PdnWjcVNU4PbfcNEYLZJkTneZgzPfDX%2BtbkxoE%2Bgsb%2F%2B1hqLgKyce3KJLY6z86sbPI4lLw%2BpCBF5XC9Aw9aoFonDHa%2BdkNn%2BY%2FfYBrt64qMYQriactXpO1p)

[HTTP长连接、短连接究竟是什么？](https://www.cnblogs.com/gotodsp/p/6366163.html)

[.](https://www.cnblogs.com/gotodsp/p/6366163.html)

### Web端即时通讯技术

即时通讯技术就是实现这样一种功能：服务器端可以即时地将数据的更新或变化反应到客户端，例如消息即时推送等功能都是通过这种技术实现的。

但是在Web中，由于浏览器的限制，实现即时通讯需要借助一些方法。这种限制出现的主要原因是，一般的Web通信都是浏览器先发送请求到服务器，服务器再进行响应完成数据的现实更新。

实现Web端即时通讯的方法：实现即时通讯主要有四种方式，它们分别是短轮询、长轮询(comet)、长连接(SSE)、WebSocket。它们大体可以分为两类，一种是在HTTP基础上实现的，包括短轮询、comet和SSE；另一种不是在HTTP基础上实现是，即WebSocket。

①短轮询

短轮询的基本思路就是浏览器每隔一段时间向浏览器发送http请求，服务器端在收到请求后，不论是否有数据更新，都直接进行响应。这种方式实现的即时通信，本质上还是浏览器发送请求，服务器接受请求的一个过程，通过让客户端不断的进行请求，使得客户端能够模拟实时地收到服务器端的数据的变化。

这种方式的优点是比较简单，易于理解，实现起来也没有什么技术难点。缺点是显而易见的，这种方式由于需要不断的建立http连接，严重浪费了服务器端和客户端的资源。因此短轮询不适用于那些同时在线用户数量比较大，并且很注重性能的Web应用。

②comet

comet指的是，当服务器收到客户端发来的请求后，不会直接进行响应，而是先将这个请求挂起，然后判断服务器端数据是否有更新。如果有更新，则进行响应，如果一直没有数据，则到达一定的时间限制(服务器端设置)后关闭连接。

长轮询和短轮询比起来，明显减少了很多不必要的http请求次数，相比之下节约了资源。长轮询的缺点在于，连接挂起也会导致资源的浪费。

③SSE

SSE是HTML5新增的功能，全称为Server-Sent Events。它可以允许服务推送数据到客户端。SSE在本质上与之前的长轮询、短轮询不同，虽然都是基于http协议的，但是轮询需要客户端先发送请求。而SSE最大的特点就是不需要客户端发送请求，可以实现只要服务器端数据有更新，就可以马上发送到客户端。

SSE的优势很明显，它不需要建立或保持大量的客户端发往服务器端的请求，节约了很多资源，提升应用性能。

④WebSocket

WWebSocket是HTML5定义的一个新协议，与传统的HTTP协议不同，该协议可以实现服务器与客户端之间全双工通信。简单来说，首先需要在客户端和服务器端建立起一个连接，这部分需要HTTP。连接一旦建立，客户端和服务器端就处于平等的地位，可以相互发送数据，不存在请求和响应的区别。

WebSocket的优点是实现了双向通信，缺点是服务器端的逻辑非常复杂。现在针对不同的后台语言有不同的插件可以使用。

四种Web即时通信技术比较

* 从兼容性角度考虑，短轮询>长轮询>长连接SSE>WebSocket；
* 从性能方面考虑，WebSocket>长连接SSE>长轮询>短轮询。

### Web端即时通讯技术

即时通讯技术就是实现这样一种功能：服务器端可以即时地将数据的更新或变化反应到客户端，例如消息即时推送等功能都是通过这种技术实现的。但是在Web中，由于浏览器的限制，实现即时通讯需要借助一些方法。这种限制出现的主要原因是，一般的Web通信都是浏览器先发送请求到服务器，服务器再进行响应完成数据的现实更新。

实现Web端即时通讯的方法：实现即时通讯主要有四种方式，它们分别是轮询、长轮询(comet)、长连接(SSE)、WebSocket。

* 从兼容性角度考虑，短轮询>长轮询>长连接SSE>WebSocket；
* 从性能方面考虑，WebSocket>长连接SSE>长轮询>短轮询。

它们大体可以分为两类，一种是在HTTP基础上实现的，包括短轮询、comet和SSE；另一种是建立在 TCP 协议基础上实现的，即WebSocket。

下面分别介绍一下这四种轮询方式，以及它们各自的优缺点。

**短轮询**

短轮询的基本思路就是浏览器每隔一段时间向浏览器发送HTTP请求，服务器端在收到请求后，不论是否有数据更新，都直接进行响应。这种方式实现的即时通信，本质上还是浏览器发送请求，服务器接受请求的一个过程，通过让客户端不断的进行请求，使得客户端能够模拟实时地收到服务器端的数据的变化。

这种方式的优点是比较简单，易于理解，实现起来也没有什么技术难点。缺点是显而易见的，这种方式由于需要不断的建立HTTP连接，严重浪费了服务器端和客户端的资源。因此短轮询不适用于那些同时在线用户数量比较大，并且很注重性能的Web应用。

前端用定时器，每间隔一段时间发送请求来获取数据是否更新，这种方式可兼容ie和支持高级浏览器。通常采取setInterval或者setTimeout实现。

**长轮询(comet)**

长轮询通过客户端和服务端的配合，达到主动权在客户端，同时也能保证数据的实时性；长轮询本质上也是轮询，只不过对普通的轮询做了优化处理，服务端在没有数据的时候并不是马上返回数据，会hold住请求，等待服务端有数据，或者一直没有数据超时处理，然后一直循环下去。

长轮询和短轮询比起来，明显减少了很多不必要的http请求次数，相比之下节约了资源。长轮询的缺点在于，连接挂起也会导致资源的浪费。

Comet有两种方式实现，Ajax长轮询和HTTP流。

短轮询与长轮询都是基于HTTP的，两者本身存在着缺陷：

* 短轮询需要更快的处理速度；长轮询则更要求处理并发的能力;
* 两者都是“被动型服务器”的体现:服务器不会主动推送信息，而是在客户端发送请求后进行返回的响应。而理想的模型是"在服务器端数据有了变化后，可以主动推送给客户端",这种"主动型"服务器是解决这类问题的很好的方案。Web Sockets就是这样的方案。

**长连接(SSE)**

严格地说，HTTP协议无法做到服务器主动推送信息。但是，有一种变通方法，就是服务器向客户端声明，接下来要发送的是流信息（streaming）。也就是说，发送的不是一次性的数据包，而是一个数据流，会连续不断地发送过来。这时，客户端不会关闭连接，会一直等着服务器发过来的新的数据流，视频播放就是这样的例子。本质上，这种通信就是以流信息的方式，完成一次用时很长的下载。SSE就是利用这种机制，使用流信息向浏览器推送信息。它基于HTTP协议，目前除了IE/Edge，其他浏览器都支持。

在传统的HTTP请求时（HTTP/1.0版本），客户端发出请求，服务端回复后HTTP连接关闭，接着TCP连接关闭。在下一次HTTP请求时又建立新的TCP连接。这无形中造成了资源的浪费。

为了解决这一问题，HtTTP/1.1想出了持久化连接。只要任意一端没有明确的提出断开连接，则保持TCP连接状态。通过首部字段Connection：Keep-Alive实现。HTTP/1.1默认为长连接。

在客户端，SSE具有三个事件。open、message、error

SS本质是发送的不是一次性的数据包，而是一个数据流。可以使用HTTP 301 和307重定向与正常的HTTP请求一样。服务端连续不断的发送，客户端不会关闭连接，如果连接断开，浏览器会尝试重新连接。如果连接被关闭，客户端可以被告知使用 HTTP 204无内容响应代码停止重新连接。

SSE只适用于高级浏览器，IE不支持。因为IE上的XMLHttpRequest对象不支持获取部分的响应内容，只有在响应完成之后才能获取其内容。

SSE的优势很明显，它不需要建立或保持大量的客户端发往服务器端的请求，节约了很多资源，提升应用性能。并且后面会介绍道，SSE的实现非常简单，并且不需要依赖其他插件。

* SSE使用HTTP协议，现有的服务器软件都支持。WebSocket是一个独立协议。
* SSE属于轻量级，使用简单；WebSocket 协议相对复杂。
* SSE默认支持断线重连，WebSocket需要自己实现。
* SSE一般只用来传送文本，二进制数据需要编码后传送(BASE64)，WebSocket 默认支持传送二进制数据。
* SSE支持自定义发送的消息类型。

**WebSocket**

WebSocket 是一种网络通信协议, 它的最大特点就是，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于服务器推送技术的一种。

SSE与WebSocket作用相似，都是建立浏览器与服务器之间的通信渠道，然后服务器向浏览器推送信息。总体来说，WebSocket更强大和灵活。因为它是全双工通道，可以双向通信；SSE是单向通道，只能服务器向浏览器发送，因为流信息本质上就是下载。如果浏览器向服务器发送信息，就变成了另一次HTTP请求。

WebSocket协议是借用HTTP协议的101 switchprotocol(服务器根据客户端的指定，将协议转换成为 Upgrade首部所列的协议)来达到协议转换的，从HTTP协议切换成WebSocket通信协议。

通过在请求头中增加 upgrade：websocket 及通信密钥（Sec-WebSocket-Key），使双方握手成功，建立全双工通信。

***WebSocket客户端连接报文***

***WebSocket服务端响应报文***

WebSocket是纯事件驱动的，一旦 WebSocket连接建立后，通过监听事件可以处理到来的数据和改变的连接状态。数据都以帧序列的形式传输。服务端发送数据后，消息和事件会异步到达。WebSocket编程遵循一个异步编程模型，只需要对WebSocket对象增加回调函数就可以监听事件。

* 建立在TCP协议之上，服务器端的实现比较容易。
* 与HTTP协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用HTTP协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。
* 数据格式比较轻量，性能开销小，通信高效。
* 可以发送文本，也可以发送二进制数据。
* 没有同源限制，客户端可以与任意服务器通信。
* 协议标识符是ws（如果加密，则为wss），服务器网址就是 URL。

ws://example.com:80/some/path



### [主流服务器推送技术](https://juejin.cn/post/6844903503505293320)

网络营销中的几个常见基本术语：CPA、CPS、CPM、CPT、CPC

他们的英文全称与基本含义分别是：

1. CPA(Cost Per Action) 每行动成本。CPA是一种按广告投放实际效果计价方式的广告，即按回应的有效问卷或注册来计费，而不限广告投放量。电子邮件营销（EDM）现在有很多都是CPA的方式在进行。
2. CPS(Cost Per Sales)：以实际销售产品数量来换算广告刊登金额。CPS是一种以实际销售产品数量来计算广告费用的广告，这种广告更多的适合购物类、导购类、网址导航类的网站，需要精准的流量才能带来转化。
3. CPM(Cost Per Mille) 每千人成本。CPM是一种展示付费广告，只要展示了广告主的广告内容，广告主就为此付费。
4. CPT(Cost Per Time) 每时间段成本。CPT是一种以时间来计费的广告，国内很多的网站都是按照“一个星期多少钱”这种固定收费模式来收费。
5. CPC(Cost Per Click) 每点击成本。CPC是一种点击付费广告，根据广告被点击的次数收费。如关键词广告一般采用这种定价模式，比较典型的有Google广告联盟的AdSense for Content和百度联盟的百度竞价广告。

通过以上信息我们知道：

1. CPT和CPM只在第一步收取广告费用，即媒体只需要将广告对广告受众进行了展示，即可向广告商收取广告费用。
2. CPC只收取第二步费用，消费者看到广告后并进行了点击行为以后，媒体向广告商收取广告费用。
3. CPA和CPS处于第三步，即消费者有看到广告后并点击了广告，进一步了解活动情况后在广告主的网站完成某些特定行为（例如付款消费，填表注册等）。

### gRPC

GRPC 四种通信模式‍

## 09/25/2022

### Devops&Kubernetes

#### [DevOps整合Jenkins+K8S+CICD](https://blog.csdn.net/qq_35583325/article/details/126936804)

##### 一、DevOps介绍

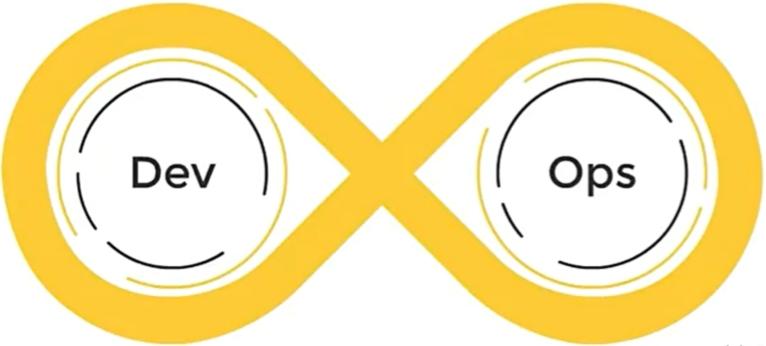
软件开发最开始是由两个团队组成：

* **开发计划**由开发团队从头开始设计和整体系统的构建。需要系统不停的迭代更新。
* **运维团队**将开发团队的Code进行测试后部署上线。希望系统稳定安全运行。

这两个目标不同的团队需要协同完成一个软件的开发。在开发团队指定好计划并完成coding后，需要提供到运维团队。运维团队向开发团队反馈需要修复的BUG以及一些需要返工的任务。因此，开发团队需要经常等待运维团队的反馈。这无疑延长并推迟了整个软件开发的周期。

基于现在的互联网现状，更推**崇敏捷式**开发，这样就导致项目的迭代速度更快。开发团队与运维团队的沟通问题会导致版本上线的高时间成本，同时，违背了敏捷式开发的初衷。

开发团队和运维团队整合成一个团队并协同应对被称为DevOps。DevOps，是Development &Operations的缩写，也就是开发&运维。

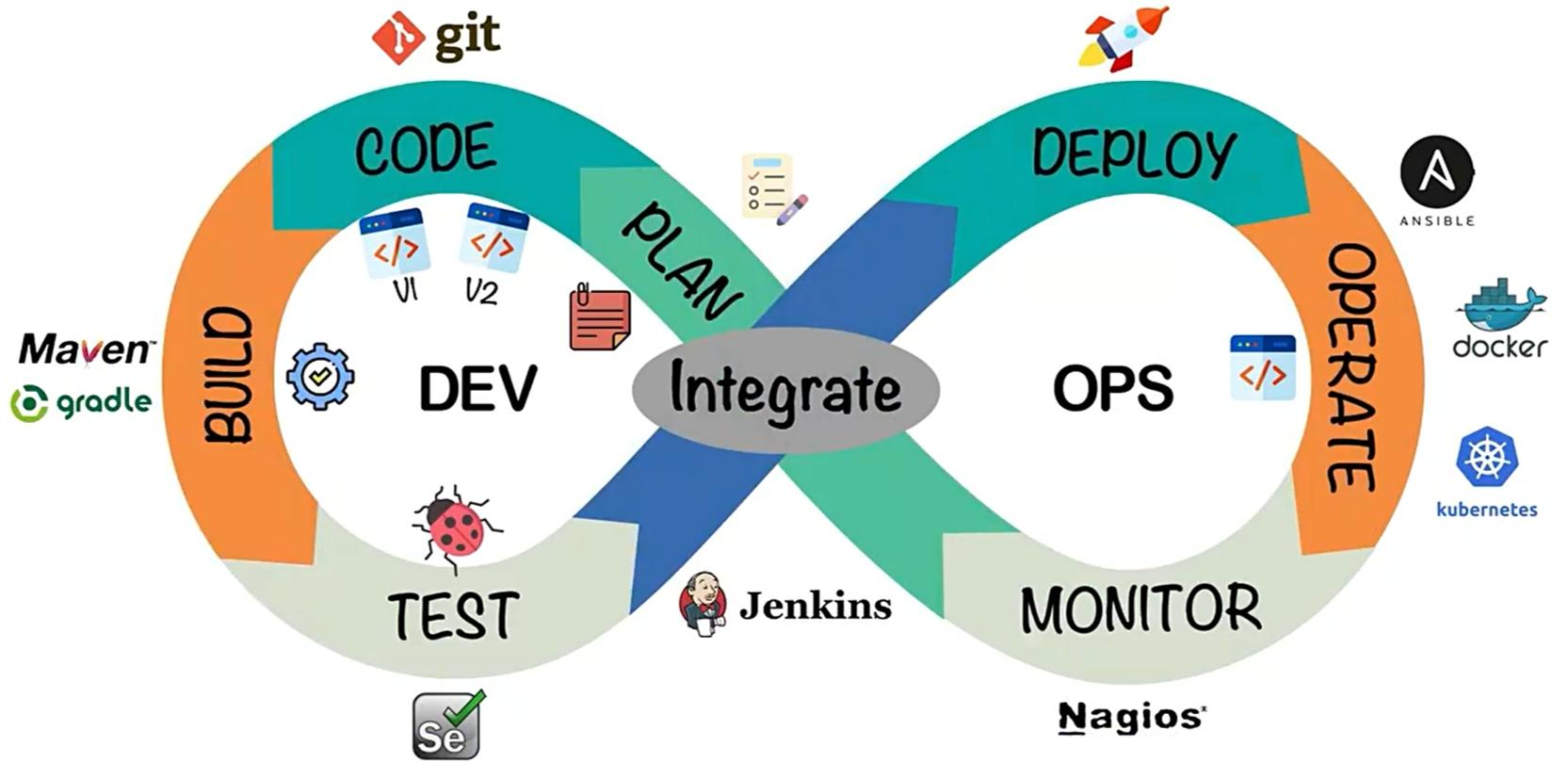


DevOps的符号类似无穷大，表明DevOps是一个不断提高效率并且持续不断工作的过程。DevOps的方式能够更快地应对更新和市场发展变化，开发可以快速交付，部署也更加稳定。核心就在于简化Dev和Ops团队之间的流程，使整体软件开发过程更快速。

整体的软件开发流程包括：

1. PLAN：开发团队根据客户目标制定开发计划。
2. CODE：根据PLAN开始编码，并进行代码版本管理。
3. BUILD：编码完成后，构建代码并且运行。
4. TEST：成功构建项目后，测试代码并修复发现的BUG。
5. DEPLOY：代码测试后，认定代码已经准备好部署并且交给运维团队。
6. OPERATE：运维团队将代码部署到生产环境中。
7. MONITOR：项目部署上线后，需要持续的监控产品。
8. INTEGRATE：将监控阶段收到的反馈发送回PLAN阶段。

整体反复的流程就是DevOps的核心，即持续集成、持续部署。为了保证整体流程可以高效的完成，各个阶段都有比较常见的工具，如下图：

最终可以给DevOps下一个定义：DevOps 强调的是高效组织团队，通过自动化工具协作来完成软件的生命周期管理，从而更快、更频繁地交付更稳定的软件。

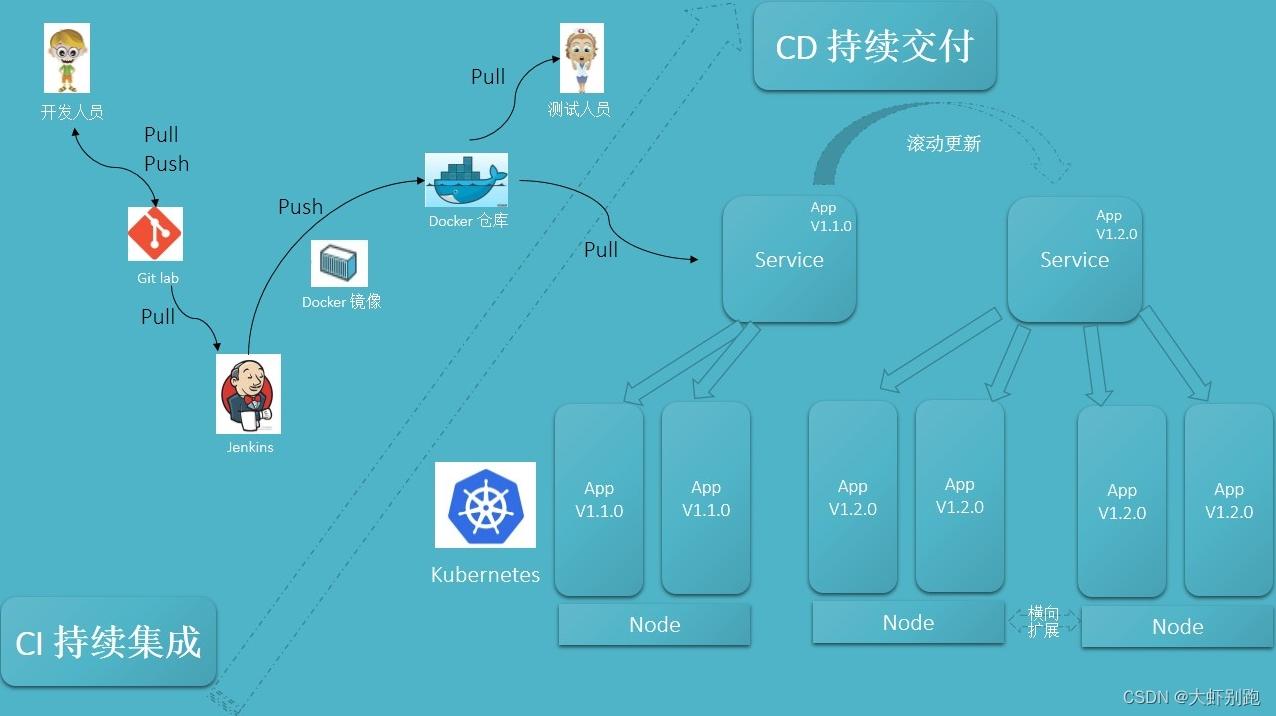
##### 二、安装git工具

##### 三、安装jdk 、maven、Jenkins

Jenkins是基于Java开发的一种持续集成工具。大多都采用Jenkins配合GitLab、Docker、K8S作为实现DevOps的核心工具。Jenkins提供了大量插件库来自动化CI/CD过程中的各种琐碎功能。

CI/CD可以理解为：

* CI(持续集成)过程通过Jenkins拉取代码、构建并制作镜像交给测试人员测试。CI让软件代码可以持续的集成到主干上，并自动构建和测试。
* CD过程通过Jenkins将打好标签的发行版本代码拉取、构建、制作镜像交给运维人员部署。
  + 持续交付：让经过持续集成的代码可以进行手动部署。
  + 持续部署：让可以持续交付的代码随时随地的自动化部。



##### 四、Jenkins实现基础的拉取操作

##### 五、Jenkins实现-Sonarqbue代码质量检测部署

##### 六、Jenkins实现制作自定义镜像并推送harbor部署

1. mv target/\*.jar docker/
2. docker build -t mytest:$tag docker/
3. docker login -u admin -p Harbor12345 192.168.1.6:80
4. docker tag mytest:$tag 192.168.1.6:80/repo/mytest:$tag
5. docker push 192.168.1.6:80/repo/mytest:$tag

[k8s+docker部署jenkins+gitlab实现CICD](https://blog.51cto.com/u_13972012/2473990)

[Jenkins+k8s实现自动化部署](https://juejin.cn/post/6963466680613896206)

[Kubernetes与云原生应用概览](https://blog.csdn.net/weixin_37098404/article/details/102705120)

[Kubernetes与云原生应用概览](http://www.uml.org.cn/yunjisuan/201711101.asp)

https://jimmysong.io/kubernetes-handbook/cloud-native/kubernetes-and-cloud-native-app-overview.html

[Kubernetes Handbook](https://jimmysong.io/kubernetes-handbook/)

[Kubernetes Handbook—Kubernetes中文指南/云原生应用架构实践手册](https://docs.hundan.org/kubernetes-handbook/)

### [DevOps Using Jenkins, Docker, and Kubernetes](https://www.betsol.com/blog/devops-using-jenkins-docker-and-kubernetes/)

DevOps, it’s a hot trend in computing, it’s the new buzz word and everyone’s talking about it.

Continuous integration and delivery or CI/CD is one of the most important parts of DevOps.

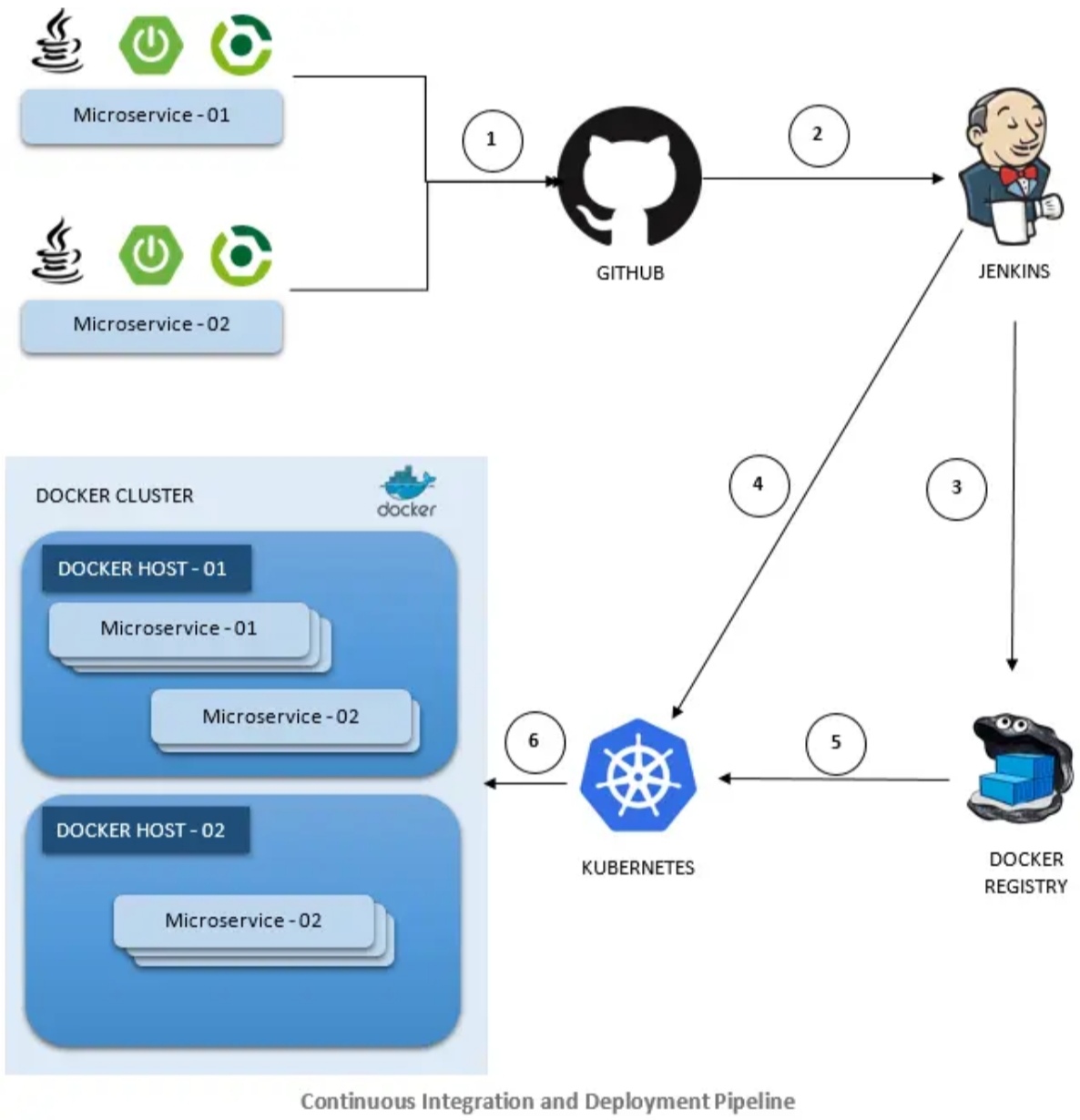
##### Create a CI and CD Pipeline

Our goal would be to automate the below process –

1. Checkout code
2. Compile code
3. Run test cases
4. Build a docker image
5. Push image to **docker registry**
6. Pull new images from the registry
7. Deploy and manage images and containers

The image below shows the CI/CD pipeline and the various tools involved. The various stages in the pipeline are shown in the figure below –

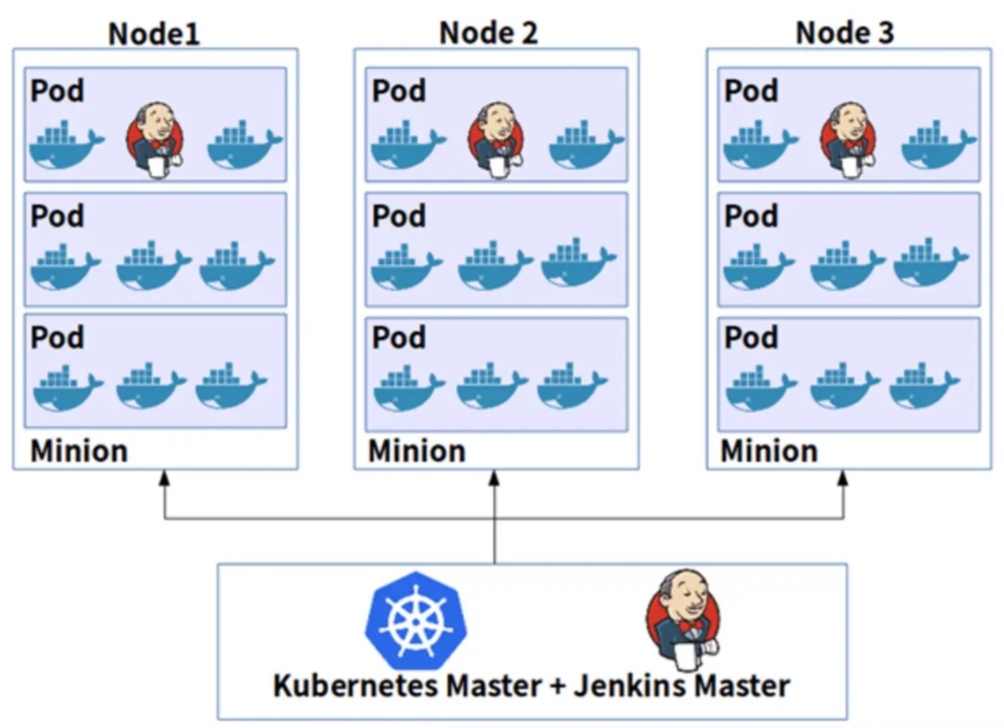
1. Code changes are committed to the version control system – GitHub
2. **Each commit to GitHub automatically triggers Jenkins build**. Jenkins uses Maven to compile the code, run unit test and perform additional checks – code coverage, code quality, etc.
3. Once the code has been successfully compiled and all the tests have been passed. Jenkins **builds a new docker image and pushes it to the Docker registry**.
4. Jenkins notifies Kubernetes of the new image available for deployment.
5. Kubernetes pulls the new docker image from the docker registry.
6. Kubernetes deploys and manages the docker instance/container.



##### Tutorials

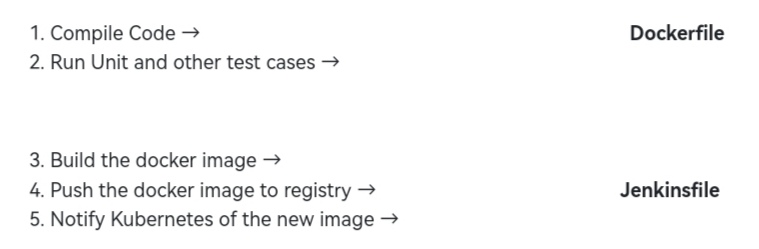
For how to setup the environment on different operating systems and platforms, refer to [this tutorial](https://cloud.google.com/solutions/jenkins-on-kubernetes-engine-tutorial).

By following the above tutorial, your setup consists of a Kubernetes master node with one or many child nodes. You’ll see that **Jenkins is set up to run inside a Kubernetes Engine or Kubernetes cluster**, this reduces the compute resources needed for CI/CD.



Once you have successfully set up the environment, the next step is to configure Jenkins to complete the setup. To do so, please follow the instructions [here](https://cloud.google.com/solutions/configuring-jenkins-kubernetes-engine).

As shown in the architecture diagram above, Jenkins helps in achieving the following steps:



##### Dockerfile

The instructions for Jenkins to execute for steps 1 and 2 are specified in Dockerfile. Dockerfile consists of commands to build and run the microservice. Optionally, we can also include commands to run unit tests and perform additional checks.

Create a new file and name it Dockerfile. Place the file under the project’s root folder. A sample docker file having the instructions for building the microservice that uses maven as the build management tool is provided below.

#Docker base image : Alpine Linux with OpenJDK JRE

FROM openjdk:8-jre-alpine

#Check the java version

RUN ["java", "-version"]

#Install maven

RUN apt-get update

RUN apt-get install -y maven

#Set the working directory for RUN and ADD commands

WORKDIR /code

#Copy the SRC, LIB and pom.xml to WORKDIR

ADD pom.xml /code/pom.xml

ADD lib /code/lib

ADD src /code/src

#Build the code

RUN ["mvn", "clean"]

RUN ["mvn", "install"]

#Optional you can include commands to run test cases.

#Port the container listens on

EXPOSE 8081

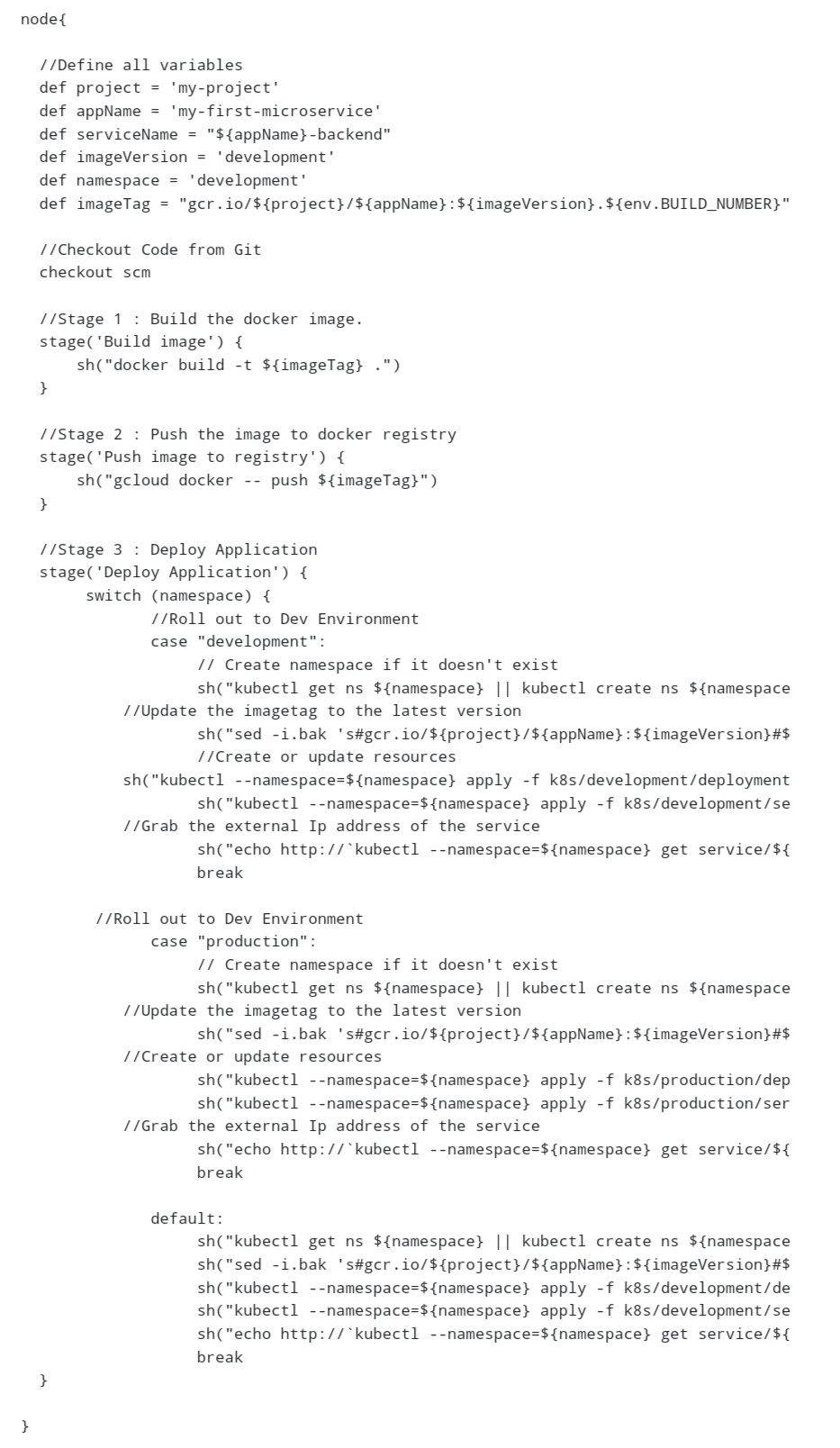
#CMD to be executed when docker is run.

ENTRYPOINT ["java","-jar","target/recruitment-service-0.0.1.jar"]

##### Jenkinsfile

The Instructions for Jenkins to execute for steps 3, 4 and 5 are specified in Jenkinsfile. A Jenkinsfile is a text file that contains the definition of a Jenkins Pipeline and is checked into source control.

Create a new file and name it Jenkinsfile. Place the file under the project’s root folder. A sample Jenkinsfile which implements three-stage continuous delivery is provided below. For more information click [here](https://jenkins.io/doc/book/pipeline/jenkinsfile/).



##### Kubernetes

Kubernetes relies on a YAML file for information about the containers, replica sets, etc. for deployment. This file is named **deployment.yaml**. The file can be under any path inside the project’s root folder, just remember to update the path for deployment YAML in Jenkisfile.

apiVersion: apps/v1

kind: Deployment

metadata:

name: recruitment-service-deployment

namespace: development

labels:

app: recruitment-service-app

spec:

replicas: 4

template:

metadata:

labels:

apps: recruitment-service

spec:

containers:

- name: recruitment-service

image: gcr.io/bats-qa/recruitment-service:development

ports:

- containerPort: 8081

If the number of replicas is more than 1 then a load balancer is required.

In Kubernetes, we need to define this a **Service**. A sample Service YAML file for creating a load balancer is specified below. Similar to deployment.yaml file, this can be placed anywhere inside the project’s root folder, but just remember to update the path for Service YAML in Jenkisfile.

apiVersion: v1

kind: Service

metadata:

name: recruitment-as-a-service

namespace: development

spec:

ports:

- name: http

port: 8081

type: LoadBalancer

selector:

apps: recruitment-service

[DevOps Tutorial — Docker, Kubernetes, and Azure DevOps](https://dzone.com/articles/devops-tutorial-devops-with-docker-kubernetes-and)

### [DevOps with Kubernetes](https://devopswithkubernetes.com/)

基于Kubernetes的DevOps实践：容器加速软件交付

### Kubernetes开源知识

Kubernetes是一个开源的容器编排部署管理平台，用于管理云平台中多个主机上的容器化应用。Kubernetes的目标是让部署容器化的应用简单高效，Kubernetes提供了应用部署、规划、更新、维护的一种机制。

应用开发者可把Kubernetes看成一个集群操作系统。Kubernetes提供服务发现、伸缩、负载均衡、自愈甚至选举等功能，让开发者从基础设施相关配置等解脱出来。

Docker使用的分层存储以及镜像的技术，使得应用的维护更新更加简单，基于基础镜像进一步扩展镜像也变得非常简单。此外，Docker团队同各个开源项目团队一起维护了大批高质量的官方镜像，既可以直接在生产环境使用，又可以作为基础进一步定制，大大的降低了应用服务的镜像制作成本。

Kubernetes集群包含Master节点(控制节点)和Node节点(计算节点/工作节点），应用部署在node节点上，且可以通过配置选择应用部署在某些特定的节点上。

**Master节点**是集群的控制节点，由以下四个组件构成：

* API Server：各组件互相通讯的中转站，接受外部请求，并将信息写到ETCD中。
* Controller Manager：执行集群级功能，例如复制组件，跟踪Node节点，处理节点故障等等。
* Scheduler：负责应用调度的组件，根据各种条件将容器调度到Node上运行。
* ETCD：一个分布式数据存储组件，负责存储集群的配置信息。

在生产环境中，为了保障集群的高可用，通常会部署多个Master节点。

**Node节点**是集群的计算节点，即运行容器化应用的节点，由以下三个组件构成：

* kubelet：负责同Container Runtime打交道，并与API Server交互，管理节点上的容器。
* kube-proxy：应用组件间的访问代理，解决节点上应用的访问问题。
* Container Runtime：最主要的功能是下载镜像和运行容器。

Kubernetes开放了容器运行时接口(CRI)、容器网络接口(CNI)和容器存储接口(CSI），这些接口让Kubernetes的扩展性变得最大化，而Kubernetes本身则专注于容器调度。

* CRI（Container Runtime Interface）：容器运行时接口，提供计算资源，CRI隔离了各个容器引擎之间的差异，而通过统一的接口与各个容器引擎之间进行互动。
* CNI（Container Network Interface）：容器网络接口，提供网络资源，通过CNI接口，Kubernetes可以支持不同网络环境。
* CSI（Container Storage Interface）：容器存储接口，提供存储资源，通过CSI接口，Kubernetes可以支持各种类型的存储。

环境变量还可以引用ConfigMap和Secret。

Kubernetes提供了Label机制来为资源分类，Kubernetes中几乎所有资源都可以用Label来组织。Pod有Label后，查询Pod时带上--show-labels就可看到Pod的Label 了，使用-L只查询某些固定的Label。对已存在的Pod，可以直接使用kubectl label命令直接添加Label。

Kubernetes提供了Namespace来做资源组织和划分，使用Namespace可以将包含很多组件的系统分成不同的组。

Kubernetes中大部分资源可以用Namespace划分，不过有些资源不行，它们属于全局资源，不属于某一个Namespace。通过kubectl get ns命令可以查询到当前集群下的Namespace。

#### Deployment

Pod是Kubernetes创建或部署的最小单位，但Pod被设计为相对短暂的一次性实体，Pod可以被驱逐(当节点资源不足时)、随着集群的节点崩溃而消失。

Kubernetes提供了Controller(控制器)来管理Pod，Controller可以创建和管理多个Pod，提供副本管理、滚动升级和自愈能力，其中最为常用的就是Deployment。

* Deployment集成了上线部署、滚动升级、创建副本、恢复上线的功能，在某种程度上，Deployment实现无人值守的上线，大大降低了上线过程的复杂性和操作风险。
* Deployment控制ReplicaSet，ReplicaSet控制Pod，也就是说，Deployment 通过ReplicaSet 控制 Pod。

用kubectl describe命令查看Deployment的详情就可看到ReplicaSet，但在实际使用中，不会直接操作ReplicaSet。

在实际应用中，升级是一个常见的场景，Deployment能够很方便的支撑应用升级。Deployment可以设置不同的升级策略，有如下两种：

* RollingUpdate：滚动升级，即逐步创建新Pod再删除旧Pod，为默认策略。
* Recreate：替换升级，即先把当前Pod删掉再重新创建Pod。

Deployment可以通过maxSurge和maxUnavailable两个参数控制升级过程中同时重新创建Pod的比例。

当升级出问题时，Deployment可以非常方便的回滚到老版本，可以执行kubectl rollout undo命令进行回滚。

Deployment之所以能如此容易的做到回滚，是因为Deployment是通过ReplicaSet控制Pod的，升级后之前ReplicaSet都一直存在，Deployment回滚做的就是使用之前的ReplicaSet再次把Pod创建出来。

Deployment中保存ReplicaSet的数量可以使用revisionHistoryLimit参数限制，默认值为10。

#### StatefulSet

Deployment控制器下的每个Pod除了名称和IP地址不同外其余的完全相同。

需要的时候，Deployment可以通过Pod模板创建新的Pod；不需要的时候，Deployment就可以删除任意一个Pod。但在某些场景下，这并不满足需求，比如要求每个Pod都有自己单独的状态，Pod间有依赖等。对应到Kubernetes中就是对Pod有如下要求：

* Pod能够被别的Pod找到，这就要求Pod有固定的标识。
* 每个Pod有单独存储，Pod被删除恢复后，读取的数据必须还是以前那份，否则状态就会不一致。

Kubernetes提供了StatefulSet来解决这个问题，其具体如下：

1. StatefulSet给每个Pod提供固定名称，名称增加从0-N的固定后缀，Pod重新调度后，名称和HostName不变。
2. StatefulSet通过Headless Service给每个Pod提供固定的访问域名。
3. StatefulSet通过创建固定标识的PVC保证Pod重新调度后还是能访问到相同的持久化数据。

创建StatefulSet需要一个Headless Service用于Pod访问，使用如下文件描述Headless Service：

其中：

* spec.clusterIP：必须设置为None，表示Headless Service。
* spec.ports.port：Pod间通信端口号。
* spec.ports.name：Pod间通信端口名称。

StatefulSet创建后，执行kubectl exec nginx-0 -- sh -c 'hostname'命令看到Pod是有固定名称的。

Headless Service使用DNS为Pod提供固定域名，这样Pod间就可以使用域名访问，即便Pod被重新创建而导致Pod的IP地址发生变化，这个域名也不会发生变化。

Headless Service创建后，每个Pod的IP都会有下面格式的域名：

**<pod-name>.<svc-name>.<namespace>.svc.cluster.local**

例如上面的三个Pod的域名就是：

● nginx-0.nginx.default.svc.cluster.local

● nginx-1.nginx.default.svc.cluster.local

● nginx-2.nginx.default.svc.cluster.local

实际访问时可以省略后面的.<namespace>.svc.cluster.local。

StatefulSet可以通过PVC做持久化存储，保证Pod重新调度后还是能访问到相同的持久化数据，在删除Pod时，PVC不会被删除。

#### Job和CronJob

Job和CronJob是负责批量处理短暂的一次性任务（short lived one-off tasks），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。

* Job是用来控制批处理型任务的资源对象。批处理业务与长期伺服业务(Deployment、Statefulset)的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出(Pod自动删除）。
* CronJob是基于时间的Job，类似于Linux的crontab文件中的一行，在指定的时间周期运行指定的Job。

#### DaemonSet

**DaemonSet**是这样一种对象(守护进程），它在集群的每个节点上运行一个Pod，且保证只有一个Pod，这非常适合一些系统层面的应用，例如日志收集、资源监控等，这类应用需要每个节点都运行，且不需要太多实例，一个比较好的例子就是Kubernetes的kube-proxy。

DaemonSet跟节点相关，如果节点异常，也不会在其他节点重新创建。

DaemonSet里没有Deployment或StatefulSet中的replicas参数，因为是每个节点固定一个。

#### ConfigMap和Secret

**ConfigMap**是一种用于存储应用所需配置信息的资源类型，用于保存配置数据的键值对，可以用来保存单个属性，也可以用来保存配置文件。通过ConfigMap可以方便的做到配置解耦，使得不同环境有不同的配置。

**Secret**是一种加密存储的资源对象，可以将认证信息、证书、私钥等保存在Secret中，而不需要把这些敏感数据暴露到镜像或者Pod定义中，从而更加安全和灵活。

ConfigMap和Secret最常见的用法是将环境变量注入到容器中。

Secret与ConfigMap都是key-value键值对形式，使用方式也相同，不同的是Secret会加密存储，所以适用于存储敏感信息。在创建Secret时，Secret的Value必须使用Base64编码。

检测cmb-hz-credentials

#### Network

Kubernetes本身并不负责网络通信，Kubernetes提供了容器网络接口CNI，具体的网络通信交给容器网络接口CNI(Container Network Interface)插件来负责。

Kubernetes不负责网络，但要求集群中的Pod能够互相通信，且Pod必须通过非NAT网络连接，即收到的数据包的源IP就是发送数据包Pod的IP。同时Pod与节点之间的通信也是通过非NAT网络。但是Pod访问集群外部时源IP会被修改成节点的IP。

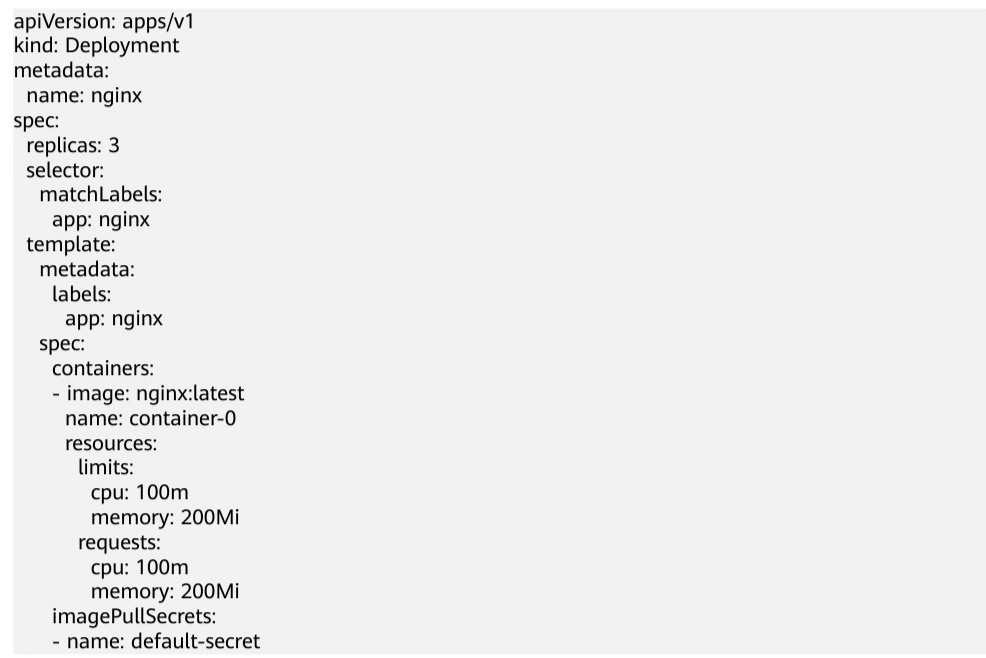
##### Service

Pod创建完成后，如何访问呢？直接访问Pod会有如下几个问题：

* Pod会随时被Deployment这样的控制器删除重建。
* Pod的IP地址是在Pod启动后才被分配。
* 应用往往都是由多个运行相同镜像的一组Pod组成，逐个访问Pod不现实。

Kubernetes Service对象解决了上述Pod的访问问题。Service有一个固定IP地址(在创建集群时会设置一个服务网段，该网段专门用于给Service分配IP地址），Service将访问它的流量转发给Pod，具体转发给哪些Pod通过Label来选择，而且Service可以给这些Pod做负载均衡。

首先创建一个具有3个Pod的Deployment，且Pod上带有标签“app: nginx”，具体如下：



然后创建一个名为“nginx”的Service，通过selector选择到标签“app:nginx”的Pod，目标Pod的端口为80，Service对外暴露的端口为8080。访问服务只需通过“**服务名称:对外暴露的端口**”接口，对应本例即“nginx: 8080”。其他Pod只需通过“nginx:8080”就可以访问到“nginx”关联的Pod。



可以看到Service有个Cluster IP，这个IP是固定不变的，除非Service被删除。所以可以使用ClusterIP在集群内部访问Service。

通过DNS域名解析后，可使用“ServiceName:Port”访问Service，这是Kubernetes中常用的一种使用方式。在创建集群的时候，会默认要求安装CoreDNS插件。在kube-system命名空间下可以查看到CoreDNS的Pod:

**$ kubectl get po --namespace=kube-system**

**NAME READY STATUS RESTARTS AGE**

**coredns-7689f8bdf-295rk 1/1 Running 0 9m11s**

**coredns-7689f8bdf-h7n68 1/1 Running 0 11m**

CoreDNS安装成功后会成为DNS服务器，创建Service后，CoreDNS会将Service的名称与IP记录起来，Pod通过向CoreDNS查询Service的名称就可获得Service的IP地址。

通过nginx.<namespace>.svc.cluster.local访问Service，其中，nginx为Service的名称，<namespace>为名称空间名字，svc.cluster.local为域名后缀。

实际使用中，同一个名称空间下可以省略<namespace>.svc..local，直接使用ServiceName即可。例如上面创建的名为nginx的Service，通过“nginx:8080”就可以访问到Service，进而访问后台Pod。

使用ServiceName的方式有个主要的优点就是可以在开发应用程序时可以将ServiceName写在程序中，这样无需感知具体Service的IP地址。

有了Service后，无论Pod如何变化，Service都能够发现到Pod。用下面命令查看Service的信息时，会看到下如下信息：

**$ kubectl describe svc nginx**

**Name: nginx**

**......**

**Endpoints: 172.16.2.132:80,172.16.3.6:80,172.16.3.7:80**

**......**

可以看到一个Endpoints，Endpoints也是Kubernetes的一种资源对象，可以查询得到：

**$ kubectl get endpoints**

**NAME ENDPOINTS AGE**

**nginx 172.16.2.132:80,172.16.3.6:80,172.16.3.7:80 5h48m**

这里的172.16.2.132:80就是Pod的IP:PORT。通过**kubectl get po -o wide**命令可以查看到Pod的IP，与上面的IP一致。

Kubernetes正是通过Endpoints监控到Pod的IP，从而让Service能够发现Pod。

实际上，Service相关信息都是由节点上的**kube-proxy**来处理的。在创建Service的时候，Kubernetes会分配IP给Service，同时通过API Server通知所有kube-proxy有新的Service创建了，kube-proxy收到通知后，通过iptables记录Service和IP/端口对的关系，从而让Service在节点上可以被查询到。除了记录Service和IP/端口对的关系，kube-proxy还会监控Service和Endpoint的变化，从而保证Pod重建后仍然能通过Service访问到Pod。如果删除一个Pod，Deployment会将Pod重建，新的Pod IP会发生变化。

Service类型除ClusterIP外还有NodePort、LoadBalancer和None，这几种类型的Service有着不同的用途：

* ClusterIP：用于在集群内部互相访问的场景，通过ClusterIP访问Service。
* NodePort：用于从集群外部访问的场景，通过节点上的端口访问Service。
* LoadBalancer：用于从集群外部访问的场景，是NodePort的扩展，通过一个特定的LoadBalancer访问Service，这个LoadBalancer将请求转发到节点的NodePort，而外部只需要访问LoadBalancer。
* None：用于Pod间的互相发现，这种类型的Service又叫Headless Service。

##### Ingress

Service是基于四层TCP和UDP协议转发的，而Ingress可以基于七层的HTTP和HTTPS协议转发，可以通过域名和路径做到更细粒度的划分。

使用Ingress功能前必须在Kubernetes集群上安装Ingress Controller。Ingress Controller有很多种实现，最常见的就是Kubernetes官方维护的NGINX Ingress Controller。

外部请求首先到达Ingress Controller，Ingress Controller根据Ingress的路由规则，查找到对应的Service，进而通过Endpoint查询到Pod的IP地址，然后将请求转发给Pod。

#### ServiceAccount

Kubernetes中的所有访问，无论外部内部，都会通过API Server处理，访问Kubernetes资源前需要经过认证与授权。

ServiceAccount与Namespace绑定，关联一套凭证，存储在Secret中，Pod创建时挂载Secret，从而允许与API Server之间调用。

ServiceAccount也是Kubernetes中的资源，与Pod、ConfigMap类似，且作用于独立的命名空间，也就是ServiceAccount是属于命名空间级别的，创建命名空间时会自动创建一个名为default的ServiceAccount。

在Pod的定义文件中，可以用指定帐户名称的方式将一个ServiceAccount赋值给一个Pod，如果不指定就会使用默认的ServiceAccount。当API Server接收到一个带有认证Token的请求时，API Server会用这个Token来验证发送请求的客户端所关联的ServiceAccount是否允许执行请求的操作。

#### 弹性伸缩

Kubernetes支持Pod和集群节点的自动弹性伸缩，通过设置弹性伸缩规则，当外部条件（如CPU使用率）达到一定条件时，根据规则自动伸缩Pod和集群节点。

使用HPA（Horizontal Pod Autoscaler）配合Metrics Server可以实现基于CPU和内存的自动弹性伸缩，再配合Prometheus还可以实现自定义监控指标的自动弹性伸缩。

HPA（Horizontal Pod Autoscaler）是用来控制Pod水平伸缩的控制器，HPA周期性检查Pod的度量数据，计算满足HPA资源所配置的目标数值所需的副本数量，进而调整目标资源（如Deployment）的replicas字段。

#### Kubernetes Maven Plugin

**Fabric8** is an opinionated and open source integrated developer platform for continuous delivery of microservices using Kubernetes.

fabric8-maven-plugin provides a seamless integration of the Kubernetes lifecycle with Maven.

The plugin serves two purposes:

* Manage Docker images—build images, run containers, and push the images to any Docker registry.
* Manage Kubernetes resources—create Kubernetes resource configuration files, create a Kubernetes development cluster, and apply the resources to a cluster.

###### Stateful Sets

Typically, **pods are stateless**, if a pod is unhealthy or superseded by a newer version, Kubernetes disposes of it. If it is part of a replication controller, then another pod will be started. This notion of treating pods as “cattle” work for stateless applications. Stateful pods have a stronger notion of identity, such pods are called “pets”.

A stateful set ensures that a specified number of “pets” with unique identities are running at any given time. Each pet is a stateful pod. The identity of a pod is composed of:

* A stable hostname, available in DNS
* An ordinal index
* Stable storage, linked to the ordinal and hostname

Stateful applications typically have the following requirements:

* Discovery of peers for quorum
* Stable persistent storage
* Startup/teardown ordering

A stateful set requires 0..N–1 pods, each pod has a deterministic name in the format <statefulset-name>-<ordinal>, and a unique identity. The identity of a pod sticks to it, regardless of which node it is (re)scheduled on.

**Checking the Health of a Pod**

Kubernetes provides diagnostic probes to perform a health check on pods. There are two types of probes: liveness and readiness.

* A liveness probe is identified by livenessProbe and indicate whether the container is live(.i.e., running). If the liveness probe fails, Kubernetes will kill the container and the container will be subjected to its RestartPolicy.
* A readiness probe is identified by readinessProbe and indicates whether the container is ready to service requests. If the readiness probe fails, Kubernetes will remove the pod’s IP address from the endpoints of all services that match the pod.

#### [Docker Desktop](https://hub.docker.com/editions/community/docker-ce-desktop-windows/)

##### [Windows下docker desktop的安装和使用](https://blog.csdn.net/topdeveloperr/article/details/105517877)

Docker Desktop是Microsoft Windows的Docker的社区版本。

Docker Desktop包含一个在Windows主机上运行的独立Kubernetes服务器，因此我们可以测试在Kubernetes上部署Docker work load的方式。

Kubernetes客户端命令kubectl已包括在内，并配置为连接到本地Kubernetes服务器。如果我们自己安装了kubectl并指向其他环境，例如minikube或GKE集群，要确保更改上下文，以使kubectl指向docker-desktop.

**Docker Desktop Dashboard**

Docker Desktop Dashboard提供了一个简单的界面，可以与容器和应用程序进行交互，管理应用程序的生命周期。 Dashboard UI会显示所有正在运行，已停止和已启动的容器及其状态。 它提供了一个直观的界面，可以执行常见的操作来检查，交互和管理Docker对象，包括容器和基于Docker Compose的应用程序。

##### [Docker Desktop 启用 Kubernetes](https://docker-practice.github.io/zh-cn/kubernetes/setup/docker-desktop.html)

##### [Docker Desktop下使用 K8s](https://www.cnblogs.com/ageovb/p/15471084.html)

## 09/30/2022

### Docker\_for\_java\_developers

#### WORA(Write Once Run Anywhere)

allows Java source code to be compiled to byte code and run on any operating system where a Java virtual machine is available.

* Java application typically requires an infrastructure such as a specific version of operating system, an application server, JDK, and a database server.
* Java application may need binding to specific ports and requires a certain amount of memory.
* Java application may need to tune the configuration files and include multiple other dependencies.

The application, dependencies, and infrastructure together may be referred to as the application operating system.

Docker allows you to create an image that contains your application, dependencies and infrastructure together. These images are then used to create Docker containers that run on the container virtualization platform, which is provided by Docker.

Similar to WORA in Java, Docker provides Package Once Deploy Anywhere, or PODA. This allows a Docker image to be created once and deployed on a variety of operating systems where Docker virtualization is available.

#### Docker Concepts

Docker simplifies software delivery of distributed applications in three ways:

* **Build**: provides tools to create containerized applications. Developers package the application, its dependencies and infrastructure. These are called the Docker image.
* **Ship**: allows you to share these applications in a secure and collaborative manner. Docker images are stored, shared, and managed in a Docker registry.

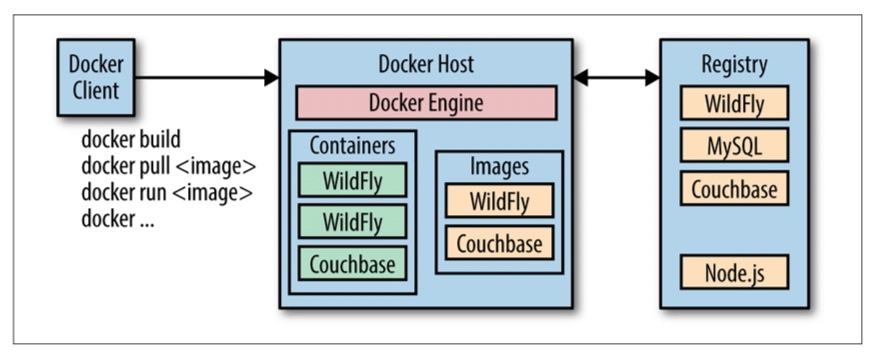
[Docker Hub](https://hub.docker.com/) is a publicly available registry. This is the default registry for all images.

* **Run**: The ability to deploy, manage, and scale these applications.

Docker container is a runtime representation of an image.

Containers can be run, started, scaled, stopped, moved, and deleted.

A typical developer workflow involves running Docker Engine on a host machine as shown

in the following figure

* **Docker host**

A machine, either physical or virtual, is identified to run the Docker Engine.

* **Configure Docker client**

The Docker client binary is downloaded on a machine and configured to talk to this Docker Engine. For development purposes, the client and Docker Engine typically are located on the same machine. The Docker Engine could be on a different host in the network as well.

* **Client downloads or builds an image**

The client can pull a prebuilt image from the preconfigured registry using the **pull** command, create a new image using the **build** command, or run a container using the **run** command.

* **Docker host downloads the image from the registry**

The Docker Engine checks to see if the image already exists on the host. If not, then it downloads the image from the registry.

* **Client runs the container**

The new container can be created using the **run** command, which runs the container using the image definition. Multiple containers, either of the same image or different images, run on the Docker host.

#### Docker Images and Containers

Docker image consists of a series of layers. Docker makes use of a union filesystem to combine these layers into a single image.

Docker images are built on Docker Engine, distributed using the registry, and run as containers.

The multiple versions of an image may be stored in the registry using **image-name:tag** format. **image-name** is the name of the image and **tag** is a version assigned to the image by the user. By default, the tag value is **latest** and typically refers to the latest release of the image.

#### Docker Engine

Docker Engine is a lightweight runtime that builds and runs Docker containers. The runtime consists of a daemon that communicates with the Docker client and execute commands to build, ship, and run containers.

Docker Engine uses Linux kernel features like **cgroups**, kernel namespaces, and a union-capable filesystem. These features allow the containers to share a kernel and run in isolation with their own process ID space, filesystem structure, and network interfaces.

#### Docker Compose

Docker Compose is a tool that allows you to define and run applications with one or more Docker containers.

Typically, an application would consist of multiple containers such as one for the web server, another for the application server, and another one for the database.

With Compose, a multi-container application can be easily defined in a single file. All the containers required for the application can be then started and managed with a single command.

#### Docker Swarm

Running a multiple-container application on a single Docker host makes that host a single point of failure (SPOF).

Docker Swarm allows you to run a multi-container application on multiple hosts. It allows you to create and access a pool of Docker hosts.

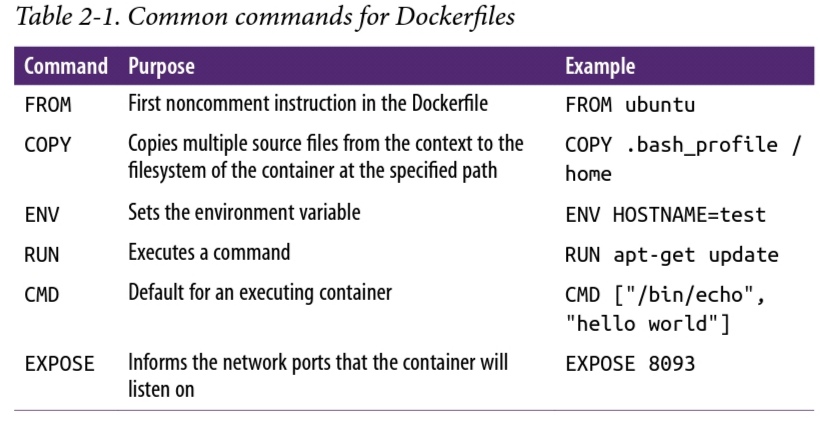
This means a multiple-container application can be seamlessly deployed to multiple hosts.

#### Dockerfile

Docker builds images by reading instructions from a named Dockerfile text document. This file contains all the commands a user can usually call on the command line to assemble an image.

The docker build uses this file and executes all the instructions in this file to create an image. The build command is also passed a context that is used during image creation. This context can be a path on your local filesystem or a URL to a Git repository.

A file named .dockerignore may be included in the root directory of the context. This file has a newline-separated list of patterns for the files and directories to be excluded from the context.

Docker CLI will send the context to Docker Engine to build the image.

#### Your First Docker Image

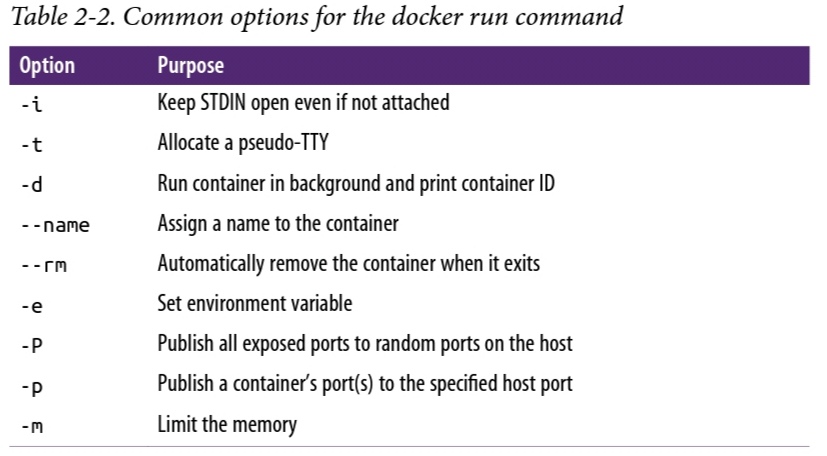
Any valid Dockerfile must have FROM as the first non-comment instruction. The argument to **FROM** defines the base image upon which subsequent instructions in the Dockerfile are executed. There are base images available for different operating systems at the [Docker website](https://hub.docker.com/). Additional packages and software can then be installed on these images.

Each image has a tag associated with it that defines multiple versions of the image. For example, java:8 is the JDK that has OpenJDK 8 included in it.

The Dockerfile can also contain **CMD** instruction. CMD provides defaults for executing the container. If multiple CMD instructions are listed then only the last CMD will **take effect**. This ensures that the Docker container can run one command, and only one.

Each image can optionally be tagged using the **name:tag** format. This allows multiple versions of the image to be created. By default, an image is assigned the latest tag.

You can run a Docker container using the docker run command and specify the image name.

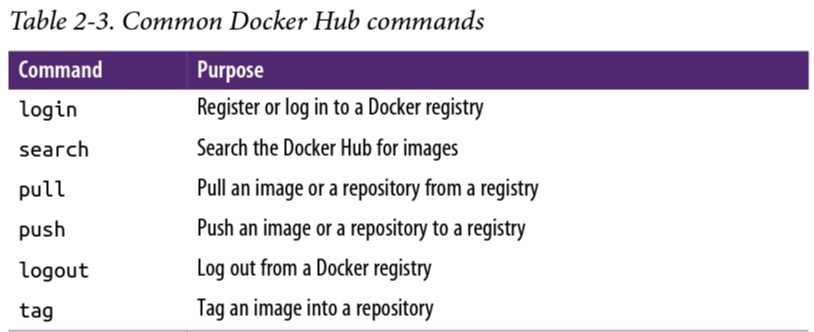


#### Push Image to Docker hub

[Docker Hub](https://hub.docker.com/) is a Software-as-a-Service (**SaaS**) registry service.

You can search, manage, push, and pull images to this registry. The images can be manually pushed to the registry using the docker push command. They can be also built when changes are pushed to a GitHub repository.

User and team collaboration can be facilitated by creating public and private registries.



#### Deploying Multiple-Container Application

This can be achieved by using a framework that allows Docker containers to be scheduled.

the application can be deployed using the following orchestration frameworks:

* Docker Compose and Docker Swarm
* Kubernetes

To deploy an application we need to define the application using a Compose file, set up the Docker Swarm cluster, and then run our application on the Swarm cluster. Let’s look at these steps in more detail.

#### IntelliJ IDEA

Docker support in IntelliJ IDEA is available by installing a plug-in named Docker Integration.

#### Maven

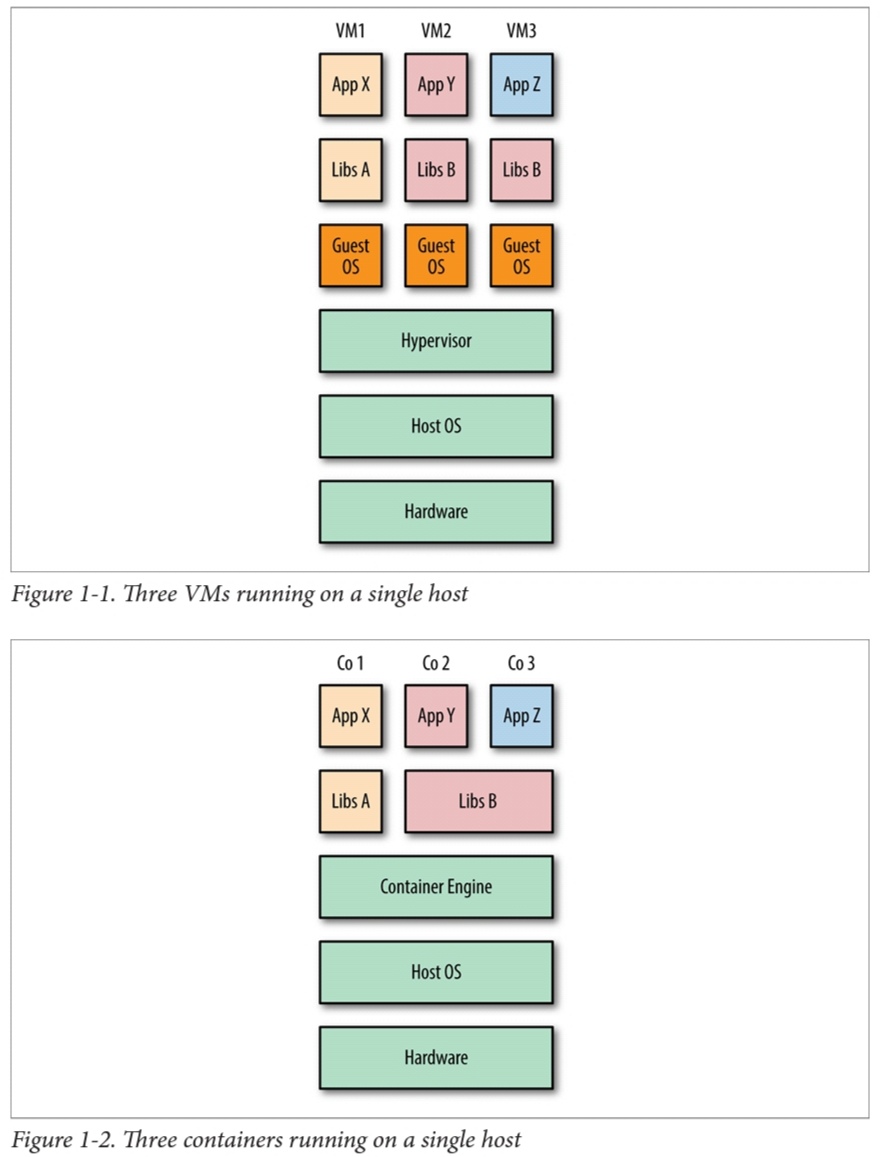
There are a few Maven plug-ins that provide goals to manage Docker images and containers:

* fabric8io/docker-maven-plugin
* spotify/docker-maven-plugin
* wouterd/docker-maven-plugin
* alexec/docker-maven-plugin

### Using Docker

Containers are an encapsulation of an application with its dependencies.

#### Containers Versus VMs



#### Docker and Containers

**Google** started the development of **CGroups** for the Linux kernel and began moving its infrastructure to containers. **Linux Containers (LXC)** project started in 2008 and brought together CGroups, kernel namespaces, and chroot technology to provide a complete containerization solution.

Finally, in 2013, **Docker** brought the final pieces to the containerization puzzle, and the technology began to enter the mainstream.

Docker took the existing Linux container technology, wrapped and extended it in various ways— through **portable images** and a user-friendly interface—to create a complete solution for the creation and distribution of containers.

The Docker platform has two distinct components:

* **Docker Engine**, which is responsible for creating and running containers;
* **Docker Hub**, a cloud service for distributing containers.

**Docker Engine** provides a fast and convenient interface for running containers.

**Docker Hub** provides an enormous number of public container images for download, allowing users to quickly get started and avoid duplicating work already done by others.

Further tooling developed by Docker includes **Swarm**, a clustering manager; **Kitematic**, a GUI for working with containers; and **Machine**, a command-line utility for provisioning Docker hosts.

The portability and isolation guarantees of containers ease collaboration with other developers and operations:

* **Developers** can be sure their code will work across environments.
* **Operations** can focus on hosting and orchestrating containers rather than worrying about the code running inside them.

**LAMP stack** (Linux, Apache, MySQL, PHP) -- common components in a web application runs in a single machine.

The major **turning point** came in March 2013.

Both stability and reliability are declared “production ready”.

Docker Hub, a public repository for containers.

Open Container Initiative(OCI) then called Open Container Project(OCP) to develop a common standard for container formats and runtimes.

#### Microservices and Monoliths

Microservices are a way of developing and composing software systems such that they are built out of small, independent components that interact with one another over the network. This is in contrast to the traditional monolithic way of developing software, where there is a single large program, typically written in C++ or Java.

* Monoliths are designed to scale up(commonly the only choice), where extra demand is handled by using a larger machine with more RAM and CPU power.
* Microservices are designed to **scale out**, where extra demand is handled by provisioning multiple machines the load can be spread over.
* In a microservice architecture, it’s possible to only scale the resources required for a particular service, focusing on the bottlenecks in the system.
* In a monolith architecture, it’s scale everything or nothing, resulting in wasted resources.

In terms of complexity, microservices are a double-edged sword. Individual microservice is easy to understand and modify. But in a system composed of dozens or hundreds of such services, the overall complexity increases due to the interaction between individual components.

The lightweight nature and speed of containers mean they are particularly well suited for running a microservice architecture. Compared to VMs, containers are vastly smaller and quicker to deploy, allowing microservice architectures to use the minimum of resources and react quickly to changes in demand.

#### scale up和scale out

Scale up（scale vertically）纵向扩展，向上扩展

Scale out（scale horizontally）横向扩展，向外扩展

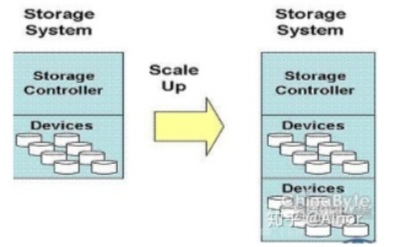
* Scale Out是指Application可以在水平方向上扩展。一般对数据中心的应用而言，当添加更多机器时，应用仍然可以很好的利用这些机器的资源来提升自己的效率从而达到很好的扩展性。
* Scale Up是指Application可以在垂直方向上扩展。一般对单台机器而言，当某个计算节点(机器)添加更多的CPU Cores，存储设备，使用更大的内存时，应用可以很充分的利用这些资源来提升自己的效率从而达到很好的扩展性。

存储系统的升级不只是需要容量，还对其他存储资源有额外需求，即带宽和计算能力:

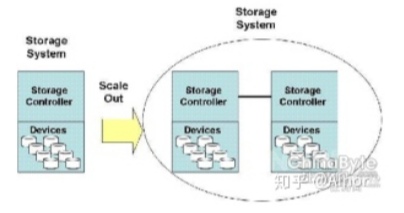
* 如果没有足够的I/O带宽，将出现用户或服务器的访问瓶颈;
* 如果没有足够的计算能力，常用的存储软件如快照、复制和卷管理等服务都将受到限制。

常见的系统扩展方式有scale up和scale out两种。

Scale Up(纵向扩展)主要是利用现有的存储系统，通过不断增加存储容量来满足数据增长的需求。但是这种方式只增加了容量，而带宽和计算能力并没有相应的增加。所以，整个存储系统很快就会达到性能瓶颈，需要继续扩展。这个时候有两种方法，一是采用更强性能的存储引擎，但是这种方式问题在于价格昂贵；另外一种方法是额外购买的独立的存储系统，这样又会增加管理的复杂度。



Scale Out(横向扩展)通常是以节点为单位，每个节点往往将包含容量、处理能力和I / O带宽。一个节点被添加到存储系统，系统中的三种资源将同时升级。



上面的图例可见，容量增长和性能扩展(即增加额外的控制器)是同时进行。

而且，Scale-out架构的存储系统在扩展之后，从用户的视角看起来仍然是一个单一的系统，这一点与我们将多个相互独立的存储系统简单的叠加在一个机柜中是完全不同的。

所以scale out方式使得存储系统升级工作大大简化，用户能够真正实现按需购买，降低TCO。

一个鱼缸的比喻：当你只有六七条鱼的时候， 一个小型鱼缸就够了;可是过一段时间新生了三十多条小鱼，这个小缸显然不够大了。如果用Scale up解决方案，那就需要去买一个大缸，把所有沙、水草、布景、加热棒、温度计都从小缸里拿出来，重新布置到大缸。如果用Scale out方案，就相当于是在这个小缸旁边接了一个同样的小缸，两个缸联通。鱼可以自动分散到两个缸，也就省掉了上面提到的那一系列挪沙、水草、布景等的折腾了。

纵向拓展就好像是普通火车，为了增加容量 只能是加多几节车厢 但是动力源没有增加 就会使速度下降。横向拓展就好像是动车组，每添加一节 都会有动力增加 总而保证性能不变。

* Scale Up：称为单节点系统，指系统中只包括一个有效节点。这种架构的系统只具有垂直扩展能力，当需要扩展系统时，通过在节点上增加更多的CPU、内存和硬盘来扩大系统的能力。
* Scale Out：称为集群系统。指由多个节点组成的系统，这种系统的扩展主要以水平扩展方式（指增加节点的方式）来进行。

谈到系统的可伸缩性，Scale-up(纵向扩展)和Scale-out(横向扩展)是两个常见的术语。

* Scale vertically (scale up)

To scale vertically (or scale up) means to add resources to a single node in a system, typically involving the addition of CPUs or memory to a single computer. Such vertical scaling of existing systems also enables them to leverage Virtualization technology more effectively, as it provides more resources for the hosted set of Operating system and Application modules to share.

* Scale horizontally (scale out)

To scale horizontally (or scale out) means to add more nodes to a system, such as adding a new computer to a distributed software application. An example might be scaling out from one web server system to three.

[高并发(水平扩展，垂直扩展)](https://www.jianshu.com/p/be66a52d2b9b)

[Scale Up vs Scale Out: Data Center Infrastructure](https://www.serverwatch.com/storage/scale-up-vs-scale-out/)

Scaling up (vertical scaling) and scaling out (horizontal scaling) are 2 key methods to add capacity to organizations infrastructure. To an end user, these two concepts may seem to perform the same function. However, they each handle specific needs and solve specific capacity issues for the system’s infrastructure in different ways.

Scaling up is adding further resources, like hard drives and memory, to increase the computing capacity of physical servers.

Scaling out is adding more servers to your architecture to spread the workload across more machines.

Scale Out按字面意思是超过尺寸范围，而Scale Up则是按比例增高。Scale Up靠增加处理器来提升运算能力，Scale Out增加独立服务器来增加运算能力。

对于服务器体系来说必须要考虑的一点就是可扩展性(Scalability)。除非业务永不增长，否则随着使用人数不断增多，服务器就一定会很快达到性能和并发极限。解决这个问题，通常只有两个办法：即代表分布式计算的Scale Out和以主机或机箱式为主的Scale Up。

* Scale Out(向外扩展)：就是指企业可以根据需求增加不同的服务器应用，依靠多部服务器协同运算，借负载平衡及容错等功能来提高运算能力及可靠度。
* Scale Up(向上扩展)：指企业后端大型服务器以增加处理器等运算资源进行升级以获得对应用性能的要求。

更大更强的服务器同时也更昂贵，成本往往会大于部署大量相对便宜的服务器。而且服务器性能所能提高的程度也有一定的上限(分布式的部署相对来说性能提高的上限更高些）。所以应该使用向外扩展(Scale Out)来实现可扩展性，同时可以让使用者得以保留通过增加服务器以提升系统能力的后路。

通常情况下用Scale up扩展单个服务器的性能来满足业务的需求，但一旦遇到服务器性能的上限，那么就需要Scale out来进一步满足要求。

#### Building Images from Dockerfiles

A Dockerfile is simply a text file that contains a set of steps that can be used to create a Docker image. Insert the following contents into Dockerfile:

**FROM debian:wheezy**

**RUN apt-get update && apt-get install -y cowsay fortune**

All Dockerfiles must have a FROM instruction as the first instruction. RUN instructions specify a shell command to execute inside the image.

UFS(Union filesystems) allow multiple file systems to be overlaid, appearing to the user as a single filesytem.

Folders may contain files from multiple filesystems, but if two files have the exact same path, the last mounted file will hide any previous files.

Docker images are made up of multiple layers. Each layer is a read-only fileystem. A layer is created for each instruction in a Dockerfile and sits on top of the previous layers. When an image is turned into a container (from a docker run or docker create command), Docker engine takes the image and adds a read-write filesystem on top (as well as initializing various settings such as the IP address, name, ID, and resource limits).

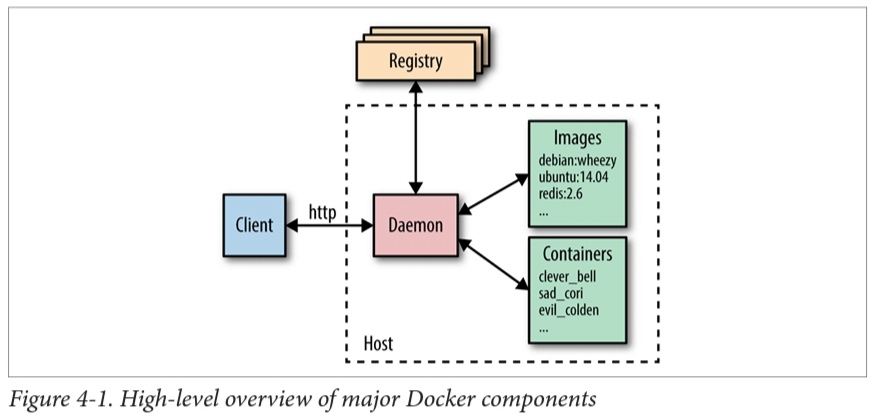
unnecessary layers bloat images (and AUFS filesystem has a hard limit of 127 layers), you will notice that many Dockerfiles try to minimize the number of layers by specifying several UNIX commands in a single RUN instruction.

ENTRYPOINT instruction lets us specify an executable that is used to handle any arguments passed to docker run.

The command docker pull amouat/revealjs:latest will download the image tagged latest within amouat/revealjs repository from Docker Hub registry.

#### Docker Fundamentals

##### Docker Architecture



we can see the major components of a Docker installation:

* Docker daemon is responsible for creating, running, and monitoring containers, as well as building and storing images. Docker daemon is launched by running docker daemon, which is normally taken care of by the host OS.
* Docker client is used to talk to Docker daemon via HTTP.
* Docker registries store and distribute images. The default registry is Docker Hub, which hosts thousands of public images as well as “official” images.

Many organizations run their own registries that can be used to store commercial or sensitive images as well as avoiding the overhead of needing to download images from the Internet.

Docker daemon will download images from registries in response to docker pull requests. It will also automatically download images specified in docker run requests and in FROM instruction of Dockerfiles if they are not available locally.

##### Underlying Technologies

Docker daemon uses an “execution driver” to create containers. By default, this is Docker’s own runc driver, Runc is very closely tied to the following kernel features:

* **cgroups**, which are responsible for managing resources used by a container (e.g., CPU and memory usage). They are also responsible for freezing and unfreezing containers, as used in the docker pause functionality.
* **namespaces** are responsible for isolating containers; making sure that a container’s filesystem, hostname, users, networking, and processes are separated from the rest of the system.
* **Union File System (UFS)** is used to store the layers for containers. UFS is provided by one of several storage drivers, either AUFS, devicemapper, BTRFS, or Overlay.

##### How Images Get Built

The docker build command requires a Dockerfile and a build context (which may be empty).

The build context is the set of local files and directories that can be referenced from ADD or COPY instructions in Dockerfile and is normally specified as a path to a directory. All the files and directories under the path form the build context and will be sent to the Docker daemon as part of the build process.

The build context is gathered into a tarball and sent to the Docker daemon.

##### Image Layers

Each instruction in a Dockerfile results in a new image layer, which can also be used to start a container. The new layer is created by starting a container using the image of the previous layer, executing Dockerfile instruction and saving a new image. When a Dockerfile instruction successfully completes, the intermediate container will be deleted.

Since each instruction results in a static image—essentially just a filesystem and some metadata—all running processes in the instruction will be stopped. This means while you can start long-lived processes, such as databases or SSH daemons in a RUN instruction, they will be terminated when the next instruction is processed or a container is started.

If you want a service or process to start with the container, it must be launched from an ENTRYPOINT or CMD instruction.

You can see the full set of layers that make up an image by running the docker history command.

When a build fails, it can be very useful to launch the layer before the failure.

##### Dockerfile Instructions

##### Linking Containers

Docker links are the simplest way to allow containers on the same host to talk to each other.

When using the default Docker networking model, communication between containers will be over an internal Docker network, meaning communications are not exposed to the host network.

##### Managing Data with Volumes and Data Containers

Docker volumes are directories that are not part of the container’s UFS—they are just normal directories on the host that are bind mounted into the container.

There are three different ways to initialize volumes.

* The first way to set up a volume is by declaring a volume at runtime with the -v flag, and find out where the volume lives on the host by running **docker inspect** on the host from a new shell:

$ docker inspect -f {{.Mounts}} container-test

* The second way to set up a volume is by using VOLUME instruction in a Dockerfile.

This has exactly the same effect as specifying -v /data to docker run.

When setting permissions and ownership on a volume or initialize a volume with some default data or configuration files, any instruction after the VOLUME instruction in a Dockerfile will not be able to make changes to that volume.

* The third way to set up volume is to extend the -v argument to docker run with an explicit directory to bind to on the host using the format -v HOST\_DIR:CONTAINER\_DIR. This can’t be done from a Dockerfile (it would be nonportable and a security risk). For example:

$ docker run -v /home/adrian/data:/data debian ls /data

This will mount the directory /home/adrian/data on the host as /data inside the container. Any files already existing in the /home/adrian/data directory will be available inside the container.

##### Sharing Data

-v HOST\_DIR:CONTAINER\_DIR syntax is very useful for sharing files between the host and one or more containers.

We can also share data between containers by using the --volumes-from CONTAINER argument with docker run.

##### Data Containers

A common practice is to create data containers—containers whose sole purpose is to share data between other containers. This approach provides a handy namespace for volumes that can be easily loaded using the –volumes-from command.

### DevOps with Kubernetes

#### RPC design

A monolithic application contains a lot of code that builds a giant exe/war program. The developer starts to divide monolithic applications in to small pieces of application and connect via the network.

Dividing an application in to small pieces and connecting via the network has been attempted back in the 1990s. Sun Microsystems introduced Sun RPC (Remote Procedure Call). It allows you to use the module remotely.

UNIX and C language have the rpcgen tool to support RPC-style functionality. It helps the developer to generate a stub code, which is in charge of network communication code, so the developer can use the C function style and be relieved from difficult network layer programming.

Java Remote Method Invocation (RMI) is similar to Sun RPC, RMI compiler (rmic) generates the stub code that connects remote Java processes to invoke the method and get a result back.

These RPC designs have the benefit to divide an application into multiple programs, individual programs can have separate source code repositories. However, RPC was designed and intended to use the same programming language and also designed for client/server model architecture, instead of a distributed architecture. In addition, there was less security consideration; therefore, it is not recommended to use over a public network.

In the 2000s, there was an initiative web services that:

* used SOAP (HTTP/SSL) as data transport
* used XML as data presentation and service definition Web Services Description Language (WSDL)
* used Universal Description, Discovery, and Integration (UDDI) as the service registry to look up a web services application.

Due to the complexity of web services programming and maintainability, it is not widely accepted by developers.

#### RESTful design

Go to 2010s, the developer starts to utilize powerful machine and network bandwidth to make application code and system structure as easy as possible making the software development cycle quicker.

Based on hardware resources, it is a natural decision to use HTTP/SSL as RPC transport, from experience with web services difficulty, the developer makes it simple as follows:

* use HTTP and SSL/TLS as standard transport.
* use HTTP method for Create/Load/Upload/Delete (CLUD) operation, such as GET/POST/PUT/DELETE.
* use URI as the resource identifier such as: user ID 123 as /user/123/
* use JSON as the standard data presentation.

It is called RESTful design, and has been widely accepted by many developers and become de facto standard of distributed applications. RESTful application allows any programming language as it is HTTP-based, so the RESTful server is Java and client Python is very natural.

It brings freedom and opportunities to the developer that it’s easy to perform code refactoring, upgrade a library and even switch to another programming language. It also encourages the developer to build a distributed modular design by multiple RESTful applications, which is called **microservices**.

If you have multiple RESTful applications, there is a concern on how to manage multiple source code and how to deploy multiple RESTful servers. However, Continuous Integration, and Continuous Delivery automation makes a lower bar to build and deploy a multiple RESTful server application easier. Therefore, microservices design is getting popular for web application developers.

#### Automation and Tools

Automation tools are not an ordinary IT/infrastructure applications. In order to achieve automation, there is an engineer who should have both developer skill set to write a code, especially scripting language, and infrastructure operator skill set such as VM, network, and storage.

DevOps stands for a clipped compound of development and operations that can make automation processes such as Continuous Integration, Infrastructure as code, and Continuous Delivery.

DevOps uses some DevOps tools to make these automation processes.

##### Continuous Delivery tool

Configuration management tools such as Puppet, Chef, and Ansible.

#### DevOps with Container

The key feature of container is isolation. Container isolates an application at OS-layer, while VM-based separation is achieved by the operating system.

Container comprises several building blocks, the two most important being namespaces and cgroups(control groups). Both of them are Linux kernel features.

Namespaces provide logical partitions of certain kinds of system resources, such as mounting point (mnt), process ID (PID), network (net), and so on. With the pid namespace isolation, processes in different namespaces cannot see each other. Nonetheless, if one process eats up a considerable amount of system resources, such as memory, it could cause the system to run out of memory and become unstable. In other words, an isolated process could still disrupt other processes or even crash a whole system if we don't impose resource usage restrictions on it.

cgroups is utilized to limit resource usage, it can set constraint on different kinds of system resources.

When docker run alpine ls is executed, what Docker did behind the scenes is:

1. Find the image alpine locally. If not found, Docker will try to find and pull it from the public Docker registry to the local image storage.
2. Extract the image and create a container accordingly.
3. Execute the entry point defined in the image, which are the arguments after the image name. In this example, it is ls. The entry point by default is /bin/sh -c on the Linux-based Docker.
4. When the entry point process is exited, the container then exits.

An image is an immutable bundle of codes, libraries, configurations, and everything needed to run an application.

A docker container is an instance of a docker image, which would actually be executed during runtime.

We can use docker inspect image and docker inspect container commands to see the difference.

option -i and -t (--interactive and -tty) enable you to enter a container for checking the image or updating something inside.

option -d (--detach) enables you to run a container in detached/background /dameon mode. If you would like to interact with a detached container, exec and attach commands can do us a favor.

An image is immutable, a container is ephemeral, and run an image as a container. An image is a read-only stack that consists of one or more layers, and a layer is a collection of files and directories in the filesystem. You can utilize docker history [image] command to understand how an image is built.

Whenever a container is created, it adds a writable layer on top of the base image. The docker diff [container] command shows the difference between the container and its base image in terms of filesystem states. The data in the writable layer is deleted along with its container. To persist data, you commit the container layer with the docker commit [container] command as a new image, or mount data volumes into a container.

A data volume allows a container's reading and writing to bypass Docker's filesystem, and it can be on a host's directory or other storages. Since the data is persistent outside a container, it can be reused and shared by multiple containers. Mounting a volume is done by specifying the -v(--volume) flag at docker run or docker create.

The following example mounts a volume under /chest in the container, and leaves a file there. Afterwards, we use docker inspect to locate the data volume:

$ docker run --name demo -v /chest alpine touch /chest/coins

$ docker inspect demo

One use case of data volumes is sharing data between containers. To do so, we first create a container and mount volumes on it, and then mount one or more containers and reference the volume with --volumes-from flag.

The following examples create a container with a data volume, /share-vol. Container A can put a file into it, and container B can read it as well:

$ docker create --name box -v /share-vol alpine nop c53e3e498ab05b19a12d554fad4545310e6de6950240cf7a28f42780f382c649 $ docker run --name A --volumes-from box alpine touch /share-vol/wine $ docker run --name B --volumes-from box alpine ls /share-vol/wine

In addition, data volumes can be mounted under a given host path, and the data inside is persistent:

$ docker run --name hi -v $(pwd)/host/dir:/data alpine touch /data/hi $ docker rm hi

$ ls $(pwd)/host/dir

hi

Registry is a service that stores, manages, and distributes images. Public services, such as [Docker Hub](https://hub.docker.com)

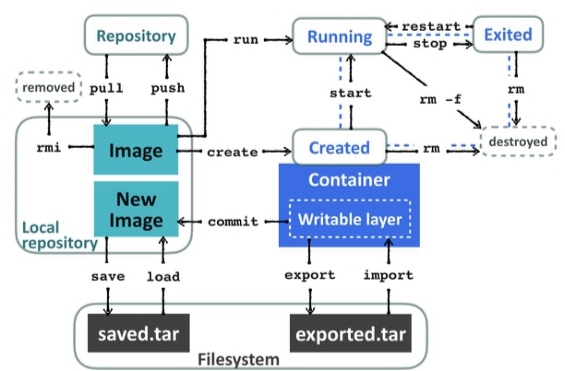
An image name consists of three sections [registry/]name[:tag], and it's resolved with the following rules:

* If the registry field is left out, search for the name on Docker Hub.
* If the registry field is a registry server, search the name for it.
* You can have more than one slash in a name.
* The tag defaults to latest if it's omitted

An image name such as gcr.io/google-containers/guestbook:v3 instructs docker to download v3 of google-containers/guestbook from gcr.io. Likewise, if you want to push an image to a registry, tag your image in the same manner and push it.

To list the images you currently own in the local disk, use docker images, and remove an image with docker rmi [image].

The following diagram depicts the relationship of states between container and images. The captions on the arrows are the corresponding subcommands of Docker:



**Connect containers**

Docker provides three kinds of networks to manage communications within containers and between the hosts, namely **bridge**, **host**, and **none**.

$ docker network ls

NETWORK ID NAME DRIVER SCOPE

1224183f2080 bridge bridge local

801dec6d5e30 host host local

f938cd2d644d none null local

By default, every container is connected to the **bridge** **network** upon creation. In this mode, every container is allocated a virtual interface as well as a private IP address, and the traffic going through the interface is bridged to the host's docker0 interface. Also, other containers within the same bridge network can connect to each other via their IP address.

##### Kubernetes

[minikube](https://github.com/kubernetes/minikube) is a tool to run Kubernetes on a single node locally. It supports to run on Windows, Linux, and macOS.

kubeconfig is a configuration file to define the context and authentication settings of the cluster, with it, we are able to switch to different clusters via the kubectl command, use kubectl config view command to see current settings in kubeconfig

**Namespace**

Kubernetes namespace is considered to be an isolation as multiple virtual clusters, and objects in different namespaces are invisible to each other.

This is useful when different teams or projects are sharing the same cluster. Most of the resources are under a namespace (a.k.a. namespaced resources); however, some generic resources, such as nodes or namespace itself, don't belong to any namespace.

Kubernetes has three namespaces by default:

* default
* kube-system
* kube-public

If we never add a new namespace, a **default** namespace will be used.

**Annotation**

Annotation is a set of user-specified key/value pairs, used for specifying non-identifying metadata. With annotation, for example, a user could add timestamp, commit hash, or build number to annotation.

Some of the kubectl commands support the --record option to record the commands that make the changes to the objects to the annotation.

Another use case of annotation is storing the configuration, such as Kubernetes Deployments

##### Pods

Pod is the smallest deployable unit in Kubernetes. It can contain one or more containers.

Most of the time, we just need one container per pod, but in some special cases, more than one container is included in the same pod, such as sidecar containers.

The containers in the same pod run in a shared context, on the same node, sharing the network namespace and shared volumes.

use kubectl explain <resource> to get the detailed description for the resource.

**ReplicaSet (RS) and ReplicationController (RC)**

A pod is not self-healing, when a pod encounters failure, it won't recover on its own. RS and RC therefore come into play.

Both RS and RC ensure a specified number of replica pods are always up and running in the cluster. If a pod crashes for any reason, RS and RC will request to spin up a new Pod.

After the latest Kubernetes:

* RC is replaced by RS gradually.
* RC and RS share the same concept, just using different requirements for the pod selector.
* RC uses equality-based pod selector while RS uses set-based selector requirements.
* RS is not created by users but by Kubernetes Deployments objects, while RC is created by users ourselves.

RC manages all the pods matching the selector, we shouldn't create RS on our own, RS should be always managed by Kubernetes Deployment object.

##### Deployments

Deployment is the best primitive to manage and deploy our software in Kubernetes after version 1.2.

Deployment supports gracefully deploying, rolling updating, and rolling back pods and ReplicaSets.

We define the desired update of the software by deployment declaratively, and then deployment will do it for us progressively.

First, we could use kubectl run command to create a deployment for us:

// using kubectl run to launch the Pods

# kubectl run nginx --image=nginx:1.12.0 --replicas=2 --port=80 deployment "nginx" created

// check the deployment status

# kubectl get deployments

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx 2 2 2 2 4h

There are two pods that are deployed by deployment:

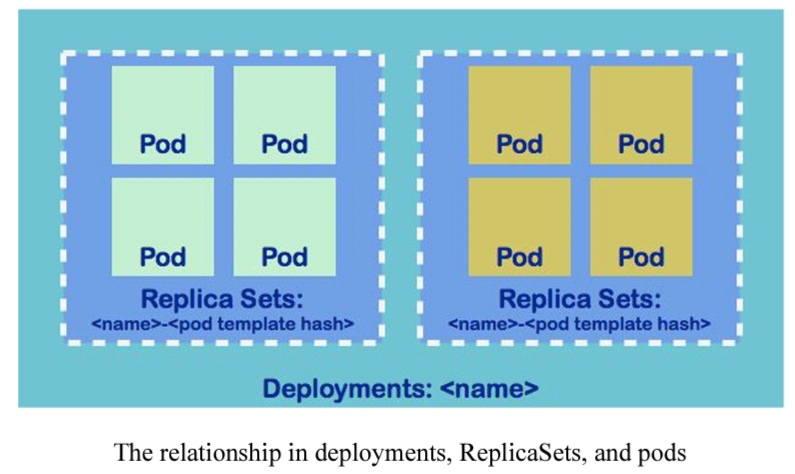
// check if pods match our desired count

# kubectl get pods

NAME READY STATUS RESTARTS AGE

nginx-2371676037-2brn5 1/1 Running 0 4h

nginx-2371676037-gjfhp 1/1 Running 0 4h



If one of the pods is deleted, a new pod will be scheduled and immediately launched due to the Department ReplicaSet behind the scenes, which will ensure the number of replicas is matched with our desired count.

In general, Deployment manage ReplicaSets, and ReplicaSets manage pods. We shouldn't manually manipulate ReplicaSets, just like there is no sense to change pods directly because they're managed by ReplicaSets.

##### Service

Service in Kubernetes is an abstraction layer for routing traffic to a logical set of pods.

With service:

* No need to trace IP address of each pod.
* Service usually uses label selector to select the pods that it needs to route to.
* Service doesn't care how the pod is created, it only cares that the pods match its label selectors.
* Service supports TCP and UDP.

There are four types of services: ClusterIP, NodePort, LoadBalancer, and ExternalName.

ClusterIP is the default service type. It exposes the service on a cluster-internal IP. Pods in the cluster could reach the service via the IP address, environment variables, or DNS.

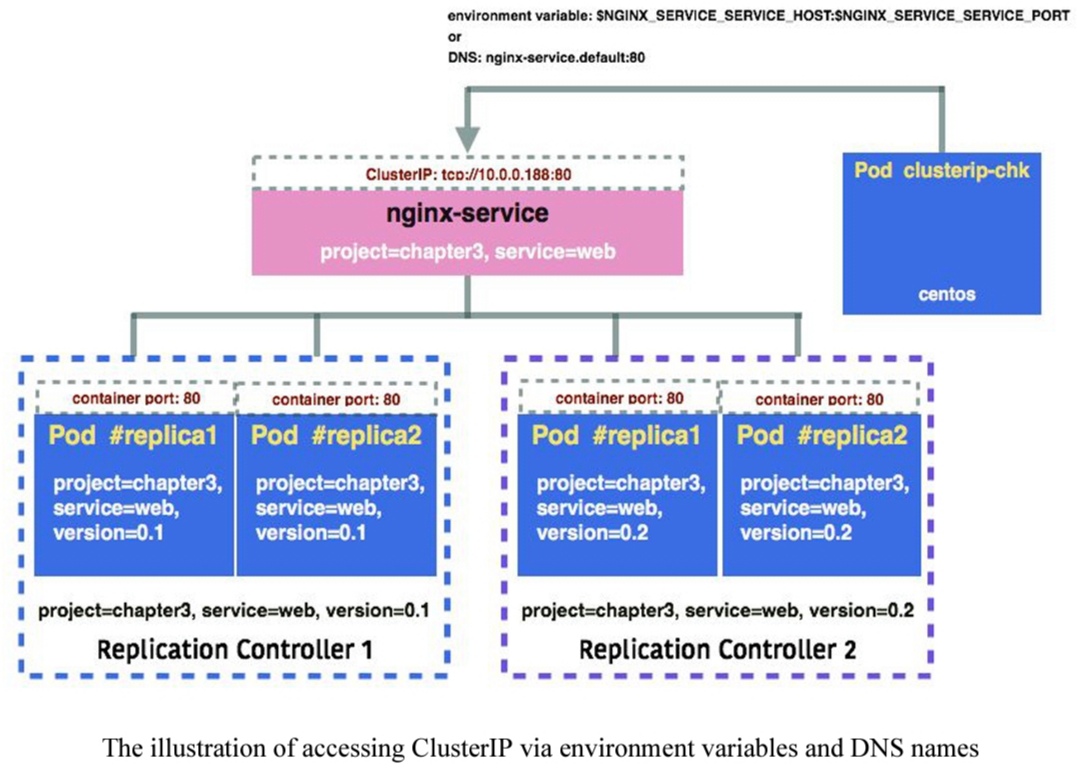
An endpoints object are created along with a Service object for routing the traffic to matching pods.

When a service is created with selectors, Kubernetes create corresponding endpoints entries and keep updating, which tells the destination that service routes into.

By default, Kubernetes will expose seven environment variables for each service. In most cases, the first two will be used for using kube-dns addon to do service discovery for us:

* ${SVCNAME}\_SERVICE\_HOST
* ${SVCNAME}\_SERVICE\_PORT
* ${SVCNAME}\_PORT${SVCNAME}\_PORT\_${PORT}\_${PROTOCAL}
* ${SVCNAME}\_PORT\_${PORT}\_${PROTOCAL}\_PROTO
* ${SVCNAME}\_PORT\_${PORT}\_${PROTOCAL}\_PORT
* ${SVCNAME}\_PORT\_${PORT}\_${PROTOCAL}\_ADDR

In the following example, use ${SVCNAME}\_SERVICE\_HOST in another pod to check if we could access our nginx pods:



With kube-dns addon enabled, the pods in the same cluster and same namespace with services could access services via services DNS records.

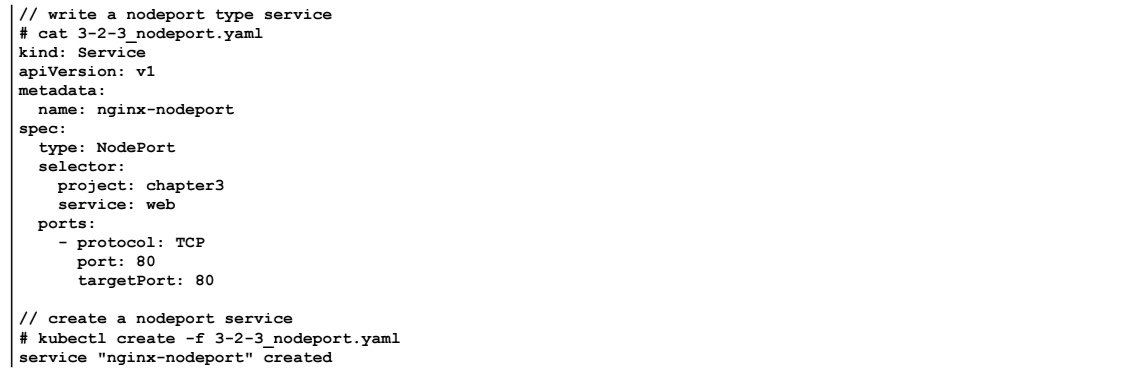
Kube-dns creates DNS records for newly created services by watching the Kubernetes API, and the DNS format

* the cluster IP is $servicename.$namespace
* the port is \_$portname\_$protocal.$servicename.$namespace.

**NodePort**

If the service is set as NodePort, Kubernetes will allocate a port within a certain range on each node. Any traffic going to nodes on that port will be routed to the service port. Port number could be user-specified. If not specified, Kubernetes will randomly choose a port from range 30000 to 32767 without collision.If specified, the user should be responsible to manage the collision by themselves.

NodePort includes the feature of ClusterIP. Kubernetes assigns an internal IP to the service.



Then you should be able to access the service via http://${NODE\_IP}:80. Node could be any node. kube-proxy watches any update of service and endpoints, and updates iptables rules accordingly (if using default iptables proxy-mode).

##### Secrets

Secret object stores the secrets in key-value format for providing sensitive information to pods, which could be a password, access key, or token.

Secret is not landed to the disk but stored in a per-node **tmpfs** filesystem. Kubelet on the nod e will create a tmpfs filesystem to store secret.

Secret is not designed to store large amounts of data, the current size limit of one secret is 1MB.

We can create a secret based on a file, directory, or specified literal value by launching kubectl to create a secret command or by spec. There are three types of secret format:

* generic (or opaque, if encoded) is the text that will be used in our application.
* docker registry secret is used to store the credential of a private docker registry.
* TLS secret is used to store the CA certificate bundle for cluster administration.

The docker-registry type of secret is also called imagePullSecrets, which is used to pass the

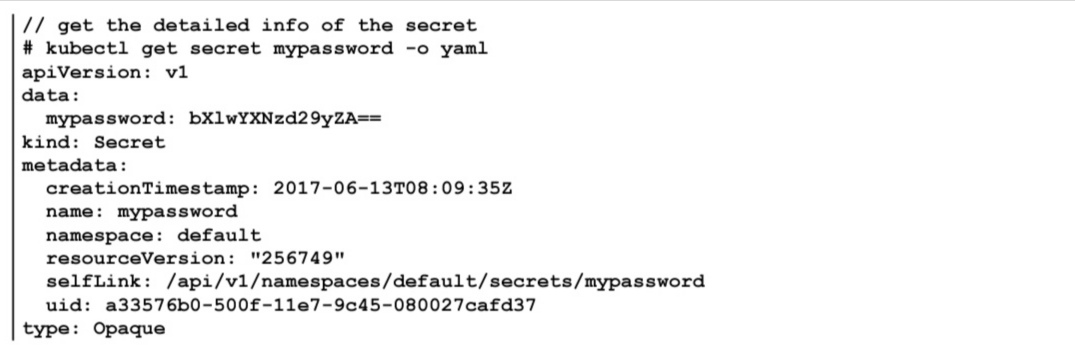
password of a private docker registry via kubelet when pulling the image.

// create a secret by command line

# kubectl create secret generic mypassword --from-file=./mypassword.txt

secret "mypassword" created

We can use the kubectl get secret <secret\_name> -o yaml command could check out the detailed information of the secret:



The type of the secret becomes Opaque since the text has been encrypted by kubectl. It's base64 encoded. We could use a simple bash command to decode it:

# echo "bXlwYXNzd29yZA==" | base64 –decode

mypassword

There are two ways for a pod to retrieve the secret. The first one is by file, and the second one is by environment variable. The first method is implemented by volume.

We just learn how to create a secret by command line. Next we'll briefly introduce its spec



the spec is plain text, so we need to encode the secret by our own by using the command:

echo -n <password> | base64.

also note that the type here becomes **Opaque**.

##### ConfigMap

ConfigMap is able to leave your configuration outside of a Docker image. It injects the configuration data as key-values pairs into pods. Its properties are similar to secret, more specifically, secret is used to store sensitive data, such as password, and ConfigMap is used to store insensitive configuration data. Same as secret, ConfigMap could be based on a file, directory, or specified literal value. ConfigMap uses kubectl create configmap command.

There are two ways to use ConfigMap inside a pod too: by volume or environment variables.

##### Resource Quality of Service

Kubernetes has the concept of Resource QoS (Quality of Service), which helps an administrator to assign and manage pods by different priorities. Based on the pod's setting, Kubernetes classifies each pod as:

* Guaranteed pod
* Burstable pod
* BestEffort pod

The priority would be Guaranteed > Burstable > BestEffort, which means if BestEffort pod and Guaranteed pod exist in the same node, when one of the pods consumes memory and causes a node resource shortage, one of the BestEffort pods will be terminated to save the Guaranteed pod.

In order to configure Resource QoS, you have to set the resource request and/or resource limit in the pod definition.

For CPU resource, acceptable value expressions for either cores (0.1, 0.2 ... 1.0, 2.0) or millicpu (100m, 200m ... 1000m, 2000m). 1000 m is equivalent to 1 core. For example, if Kubernetes node has 2 cores CPU, there are total of 2.0 cores or 2000 millicpu.

When you start to configure to set a resource request and/or limit, your pod may not be scheduled to deploy by Kubernetes scheduler due to insufficient resources. In order to understand allocatable resources and available resources, use the kubectl describe nodes command to see the status.

The error event can be captured by the kubectl describe pod command.

##### Network and Security

Kubernetes itself doesn't care how you implement it, but you must meet its three fundamental requirements:

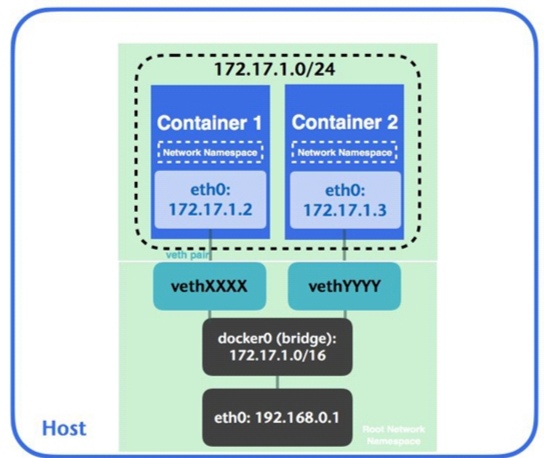
1. All containers should be accessible to each other without NAT, regardless of which nodes they are on
2. All nodes should communicate with all containers
3. The IP container should see itself the same way as the others see it

**Docker networking**

Bridge is the default networking model. Docker creates and attaches virtual Ethernet device (also known as veth) and assigns network namespace to each container.

The network namespace is a feature in Linux, which is logically another copy of a network stack. It has its own routing tables, arp tables, and network devices. It's a fundamental concept of container networking.

Veth always comes in a pair, one is in network namespace and the other is in the bridge. When the traffic comes into the host network, it will be routed into the bridge. The packet will be dispatched to its veth, and will go into the namespace inside the container, as shown in the following figure:



**Kubernetes Networking**

**Container-to-container communications**

Pods in Kubernetes have their own real IP addresses. Containers within a pod share network namespace, so they see each other as localhost. This is implemented by the network container by default, which acts as a bridge to dispatch the traffic for every container in a pod.

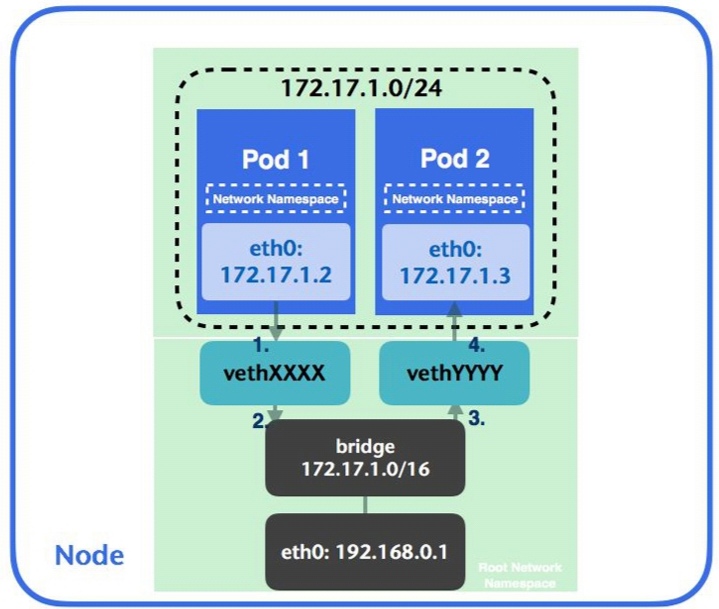
Containers in the same network namespace share the same IP address and same network configuration. This is the default implementation in Kubernetes to achieve container-to-container communications.

**Pod-to-pod communications**

Pod IP addresses are accessible from other pods no matter which nodes they're on.

Pod-to-pod communication within the same node goes through the **bridge** by default.

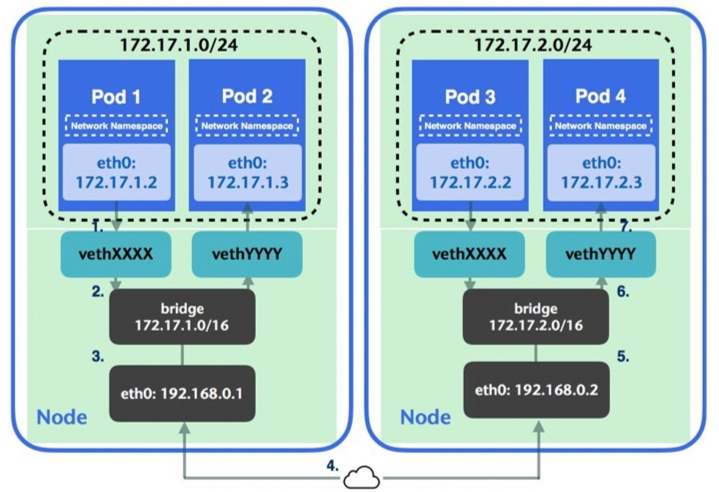
Let's say we have two pods, which have their own network namespaces. When pod1 wants to talk to pod2, the packet passes through pod1's namespace to the corresponding veth pair vethXXXX and eventually goes to the bridge. The bridge then broadcasts the destination IP to help the packet find its way, vethYYYY responses. The packet then arrives at pod2:



**Pod communication across nodes**

Kubernetes is all about clusters, so how does traffic get routed when the pods are in different nodes? According to the second requirement, all nodes must communicate with all containers. Kubernetes delegates the implementation to container network interface (CNI). Users could choose different implementations, by L2, L3, or Overlay. Overlay networking is one of the common solutions, known as **packet encapsulation**. It wraps a message before leaving the source, gets delivered, and unwraps the message at the destination. This leads to a situation where overlay increases the network latency and complexity.

As long as all the containers can access each other across nodes, you're free to use any technology, such as L2 adjacency or L3 gateway.



Let's say we have a packet from pod1 to pod4. The packet leaves from container interface and reaches to the veth pair, then passes through the bridge and node's network interface. Network implementation comes into play in step 4.

**Pod-to-service communications**

Kubernetes is dynamic. Pods are created and deleted all the time.

Kubernetes service is an abstraction to define a set of pods by label selectors. We normally use the service to access pods instead of specifying a pod explicitly.

When we create a service, an **endpoint** object will be created, which describes a set of pod IPs that the label selector in that service has selected.

In some cases, endpoint object will not be created with service creation. For example, services without selectors will not create a corresponding endpoint object.

By default, Kubernetes uses **iptables** to perform the magic by **kube-proxy**.

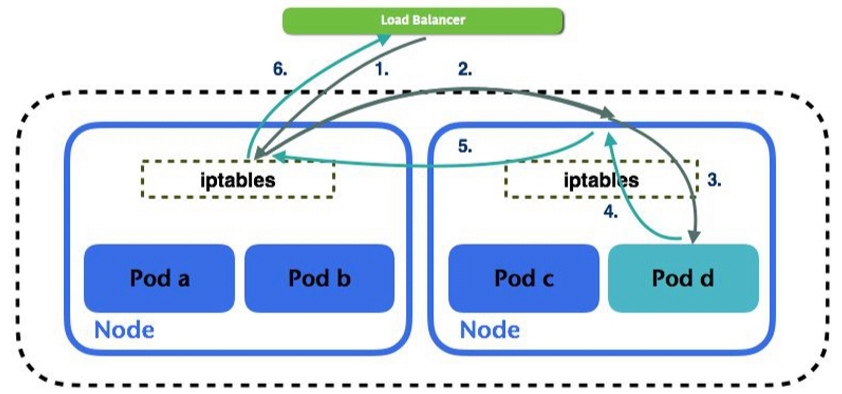
**External-to-service communications**

The ability to serve external traffic to Kubernetes is critical. Kubernetes provides two API objects to achieve this:

* Service: external network LoadBalancer or NodePort (L4)
* Ingress: HTTP(S) LoadBalancer (L7)

Based on what we've learned about pod-to-pod communication across nodes, and how the packet goes in and out between service and pod.

The following figure shows how it works. Let's say we have two services, service A has three pods (pod a, pod b, and pod c) and service B gets only one pod (pod d).



When the traffic comes in from LoadBalancer, the packet will be dispatched to one of the nodes. Most of the cloud LoadBalancer itself is not aware of pods or containers. It only knows about the node. If the node passes the health check, then it will be the candidate for the destination.

Assume that we want to access service B, it currently only has one pod running on one node. However, LoadBalancer sends the packet to another node that doesn't have any of our desired pods running.

The packet routing journey will be:

1. LoadBalancer will choose one of the nodes to forward the packet. In GCE, it selects the instance based on a hash of the source IP and port, destination IP and port, and protocol. In AWS, it's based on a round-robin algorithm.
2. Here, the routing destination will be changed to pod d (DNAT) and forward it to the other node similar to pod-to-pod communication across nodes.
3. Then, comes service-to-pod communication. The packet arrives at pod d with the response accordingly.
4. Pod-to-service communication is manipulated by iptables as well.
5. The packet will be forwarded to the original node.
6. The source and destination will be un-DNAT to LoadBalancer and client, and sent all the way back.

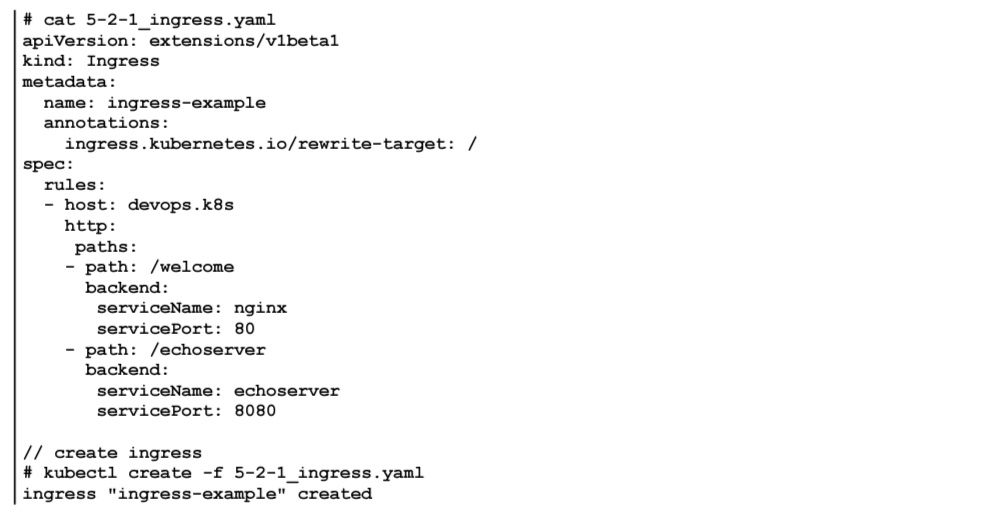
**Ingress**

Pods and services in Kubernetes have their own IP; but it is not the interface you'd provide to the external internet. Though there is service with node IP configured, the port in the node IP can't be duplicated among the services. It is cumbersome to decide which port to manage with which service. Furthermore, the node comes and goes, it wouldn't be clever to provide a static node IP to external service.

**Ingress** defines a set of rules that allows the inbound connection to access Kubernetes cluster services. It brings the traffic into the cluster at L7, allocates and forwards a port on each VM to the service port.

We define a set of rules and post them as source type ingress to the API server. When the traffic comes in, the ingress controller will route the ingress by the ingress rules.

Ingress is used to route external traffic to the kubernetes endpoints by different URLs:



There is an annotation named ingress.kubernetes.io/rewrite-target. This is required if the service requests are coming from the root URL. Without a rewrite annotation, we'll get 404 as response.

Since our host is set to devops.k8s, it will only return if we access it from that hostname. You could either configure the DNS record in the DNS server, or modify the hosts file in local. For simplicity, we'll just add a line with the ip hostname format in the host file. Then we should be able to access our service by the URL directly:

# curl <http://devops.k8s/welcome>

The pod ingress controller dispatches the traffic based on the URL path. The routing path is similar to external-to-service communication. The packet hops between nodes and pods.

**Network policy**

[JSON转YAML,YAML转JSON - 在线工具](https://oktools.net/json2yaml)

**Helm** is the package manager for Kubernetes.

## Quarkus

https://cn.quarkus.io/guides/

https://cn.quarkus.io/guides/kafka

https://quarkus.io/guides/kafka

https://quarkus.io/blog/getting-started-kafka/

https://quarkus.io/guides/kafka-reactive-getting-started

https://quarkus.pro/guides/kafka-streams.html

**https://quarkus.io/guides/lifecycle**

**https://quarkus.io/guides/command-mode-reference**

https://smallrye.io/smallrye-reactive-messaging/3.21.0/getting-started/

https://www.infoq.cn/article/K14m6yPhZ63iurHJhBWA

https://www.bookstack.cn/read/quarkus-1.7-en/48a136c995fdfe54.md

http://www.mastertheboss.com/soa-cloud/quarkus/quarkus-reactive-messaging-with-kafka/?amp=1

<https://github.com/rootsongjc/kubernetes-handbook>

### Troubleshooting Kafka Connect

[Troubleshooting Kafka Connect Common Issues and How to Debug Them](https://developer.confluent.io/learn-kafka/kafka-connect/troubleshooting-kafka-connect/)

Kafka Connect is a data integration framework. Connectors are comprised of tasks, one of the first things you should consider is that one or more of its tasks has failed, independently of the connector. To verify this, you’ll need to gather more information from the Connect API.

**Getting Task Status**

You can use **curl** to get task status and pipe the results through jq (a remarkably capable JSON formatter). The command as written requests the stack trace for the first element in the tasks array.

**curl -s "http://localhost:8083/connectors/source-debezium-orders-00/status" | jq '.tasks[0].trace**

**Dynamic Log Configuration**

Dynamic log configuration arrived in Apache Kafka 2.4. It means you can change the level of logging detail without having to restart the worker.

# List current logger configuration

curl -s http://localhost:8083/admin/loggers/ | jq

{

"org.apache.kafka.connect.runtime.rest": {

"level": "WARN"

},

"io debezium": {

"level": "WARN"

},

"root": {

"level": "INFO"

}

}

For example, there is a particular connector such as **io.debezium** that you’d like to log at **TRACE** level to try and troubleshoot. If you set everything to TRACE, it would be overwhelming. Using dynamic log configuration, you can conveniently do so at runtime via REST without restarting Connect, and targeting the specific logger of interest.

curl -s -X PUT -H "Content-Type:application/json" \

http://localhost:8083/admin/loggers/io.debezium \

-d '{"level": "TRACE"}'

<https://developer.confluent.io/learn-kafka/kafka-connect/intro/>

[confluent kafka connect remote debugging](https://www.cnblogs.com/lenmom/p/10943661.html)

#### [debezium系列之：Kafka Connect](https://zhuanlan.zhihu.com/p/484760387?utm_id=0)

Kafka Connect是一个工具，为Kafka和外部数据系统之间移动数据提供了一种可靠且可伸缩的实现方式。

Kafka Connect可以将数据从Kafka中导入或导出，数据范围涵盖关系型数据库、日志和度量数据、Hadoop和数据仓库、NoSQL数据存储、搜索索引等。

1. **Source和Sink**

Kafka有两个核心概念：Source和Sink，Source和Sink都被称为Connector连接器。

* Source负责导入数据到Kafka
* Sink负责从Kafka导出数据

1. **Task和Worker**

在Kafka Connect中还有两个重要的概念：Task和Worker。

* Task是Kafka Connect的主角，每一个Connector都会协调一系列的Task去执行任务，Connector把一项工作分割成许多Task，然后把Task分发到各个Worker进程中去执行（分布式模式下），Task不保存自己的状态信息，而是交给特定的Kafka主题去保存。
* Connector和Task都是逻辑工作单位，必须安排在进程中执行，而在Kafka Connector中，这些进程就是Worker。

**三、Kafka Connect特性**

* 通用性：规范化其他数据系统与Kafka的集成，简化了连接器的开发、部署和管理
* 支持独立模式(standalone)和分布式模式(distributed)
* REST接口：使用REST API提交和管理Connector
* 自动Offset管理：自动管理offset提交，不需要开发人员干预，降低了开发成本
* 分布式和可扩展性：Kafka Connect基于现有的组管理协议来实现扩展Kafka Connect集群
* 流式计算/批处理的集成。

1. **独立模式**

connect-standalone.sh脚本用来实现以独立模式运行Kafka Connector，在独立模式下所有的操作都是在一个进程中完成的。

独立模式适合测试或功能验证的场景，由于是单进程，所以独立模式无法充分利用Kafka自身所提供的负载均衡和高容错等特性。

在执行这个脚本时需要指定两个配置文件，一个是用于**Worker**进程运行的相关配置文件。另一个是指定**Source**连接器或**Sink**连接器的配置文件，可以同时指定多个连接器配置，每个连接器配置文件对应一个连接器，因此，要保证连接器名称全局唯一，连接器名称通过name参数指定。

1. **REST API**

可以通过Kafka Connect提供的基于REST风格的API接口来管理连接器，默认端口号8083，可以通过Worker进程的配置文件中的**rest.port**参数来修改端口号。

Kafka Connect REST API接口如下所示：

REST API 释义

GET / 查看kafka集群版本信息

GET /connectors 查看当前活跃的连接器列表，显示连接器的名字

POST /connectors 根据指定配置，创建一个新的连接器

GET /connectors/{name} ，查看连接器的信息

GET /connectors/{name}/config 查看连接器的配置信息

GET /connectors/{name}/status 查看连接器的状态

POST /connectors/{name}/restart 重启指定的连接器

PUT /connectors/{name}/pause 暂停指定的连接器

GET /connectors/{name}/tasks 查看指定连接器正在运行的Task

POST /connectors/{name}/tasks 修改Task的配置

GET /connector/{name}/tasks/{taskId}/status 查看指定连接器中指定Task的状态

重启指定连接器中指定的Task

Delete /connectors/{name} 删除指定的连接器

1. **分布式模式**

与独立模式不同，分布式模式结合了Kafka提供的负载均衡和故障转移功能，能够自动在多个节点机器上平衡负载。**分布式模式只能通过访问REST API来创建连接器。**

在运行分布式模式的连接器前，同样要修改**Worker**进程的相关配置文件($KAFKA\_HOME/config/connect-distributed.properties)，内容参考如下：

bootstrap.servers=localhost1:9092,localhost2:9092,localhost3:9092

group.id=connect-cluster

key.converter=org.apache.kafka.connect.json.JsonConverter

value.converter=org.apache.kafka.connect.json.JsonConverter

启动分布式模式，运行脚本变成了对应的connect-distributed.sh，示例如下所示：

bin/connect-distributed.sh config/connect-distributed.properties

#### [Kafka核心API——Connect API](https://cloud.tencent.com/developer/article/1702387)

[Your Complete Guide to Apache Kafka Streams](https://www.instaclustr.com/blog/kafka-streams-guide/).

Kafka Streams real-time Stream Processing

### Oracle执行计划

[Oracle调优之看懂SQL执行计划explain](https://cloud.tencent.com/developer/article/1648496)

执行计划是一条查询语句在Oracle中的执行过程或访问路径的描述。执行计划描述了SQL引擎为执行SQL语句进行的操作；分析SQL语句相关的性能问题或仅仅质疑查询优化器的决定时，必须知道执行计划；所以执行计划常用于SQL调优。

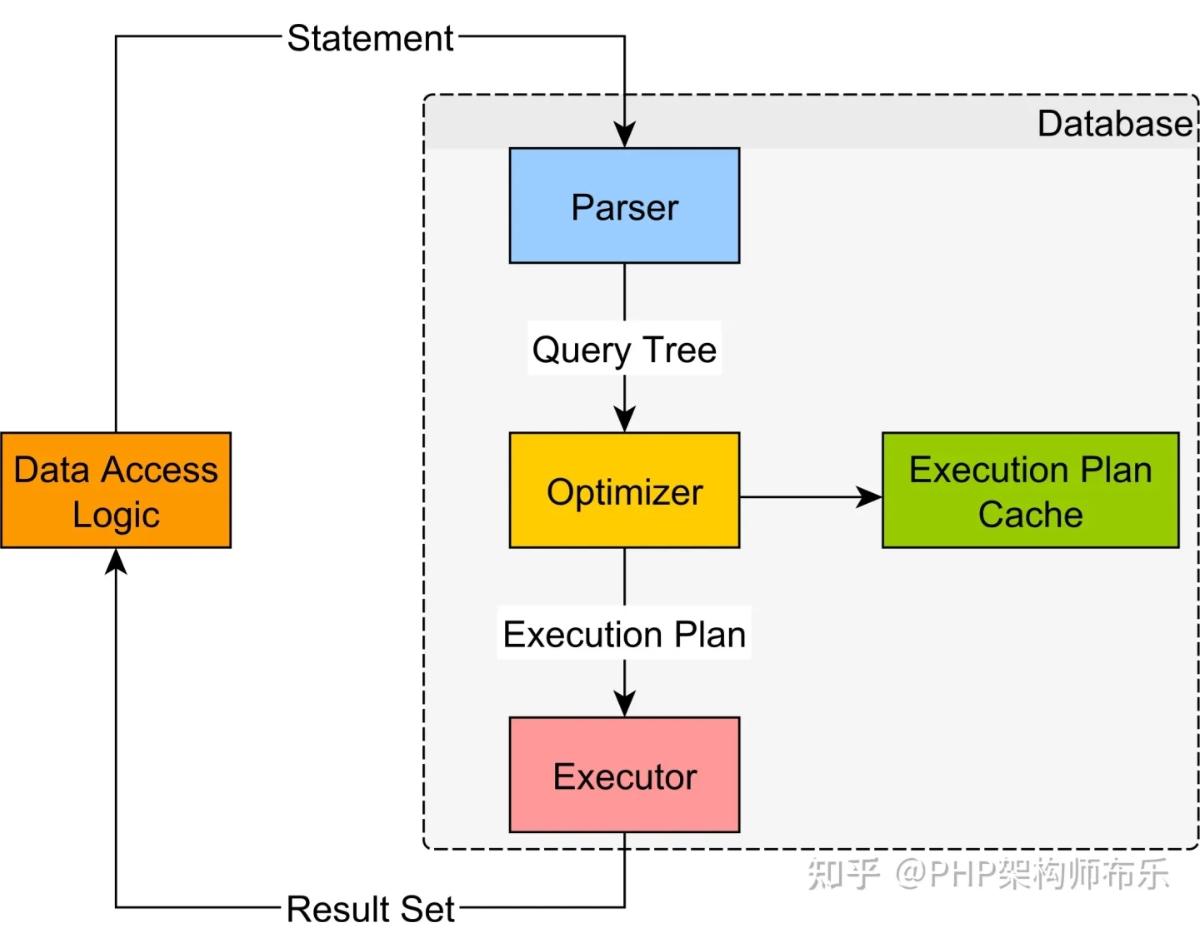
收获，不止sql调优

[Oracle执行计划详解](https://cloud.tencent.com/developer/article/1033609)

[生成、查看、理解Oracle的执行计划](https://zhuanlan.zhihu.com/p/353506250?utm_id=0)

SQL语句只是定义了要查询什么样的数据，但是没有定义数据是怎样被查询出来。

1. 当DB接收到一条SQL查询语句(INSERT, UPDATE, SELECT, DELETE)时，DB语法解析器(Parser)先将其解析成一棵抽象语法树(Abstract Syntax Tree, AST)，解析器在不影响执行结果的前提下可能会重写这个SQL来优化执行。
2. 这棵生成的语法树会被送到DB基于开销的优化器(Cost-Based Optimizer, CBO)，优化器会创建执行计划(Execution Plan)，并通过选择join类型(Nested Loops, Merge Join, Hash Join等)、join顺序、是否使用索引、使用什么索引等方式找到使用开销(CPU, IO, 内存)最小的计划作为最佳的执行计划。
3. 然后这个最佳的执行计划被送到DB的执行器(Executor)，执行器执行这个计划，并将取到的数据返回给DB调用方。



执行计划是DB优化器生成的，为了完成一个SQL查询的一系列必要且有序的数据库操作(Operation)的集合。

##### 生成执行计划

**预估执行计划**

每执行一次EXPLAIN PLAN，DB优化器(Optimizer)就会生成执行计划而不执行SQL，并写入PLAN\_TABLE表中(Oracle默认使用PLAN\_TABLE临时表来保存EXPLAIN PLAN的结果)

EXPLAIN PLAN

--- 指定该条语句的statement\_id，后续可用于查询PLAN\_TABLE表中的执行计划

SET STATEMENT\_ID = 'statement1'

--- 可以不使用默认的PLAN\_TABLE而使用自定义表

INTO my\_plan\_table

FOR

--要分析的SQL语句。INSERT, UPDATE, SELECT和DELETE语句。

SELECT last\_name FROM employees;

**实际执行计划**

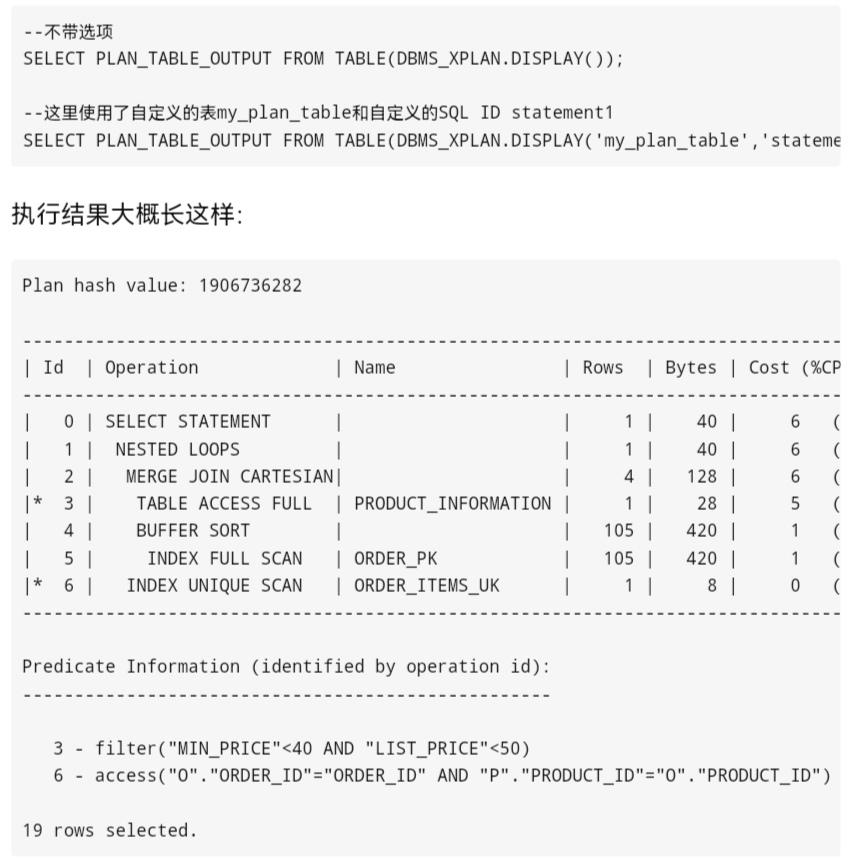
在SQL执行的时候由DB优化器(Optimizer)创建的执行计划。

EXPLAIN PLAN的执行结果代表在EXPLAIN这条SQL的时候，DB应该如何去执行这条SQL。而实际在执行的时候可能由于数据数量，结构(例如增加了索引等)等的变化，使得实际的执行计划与预估执行计划略有不同。

##### 查询执行计划

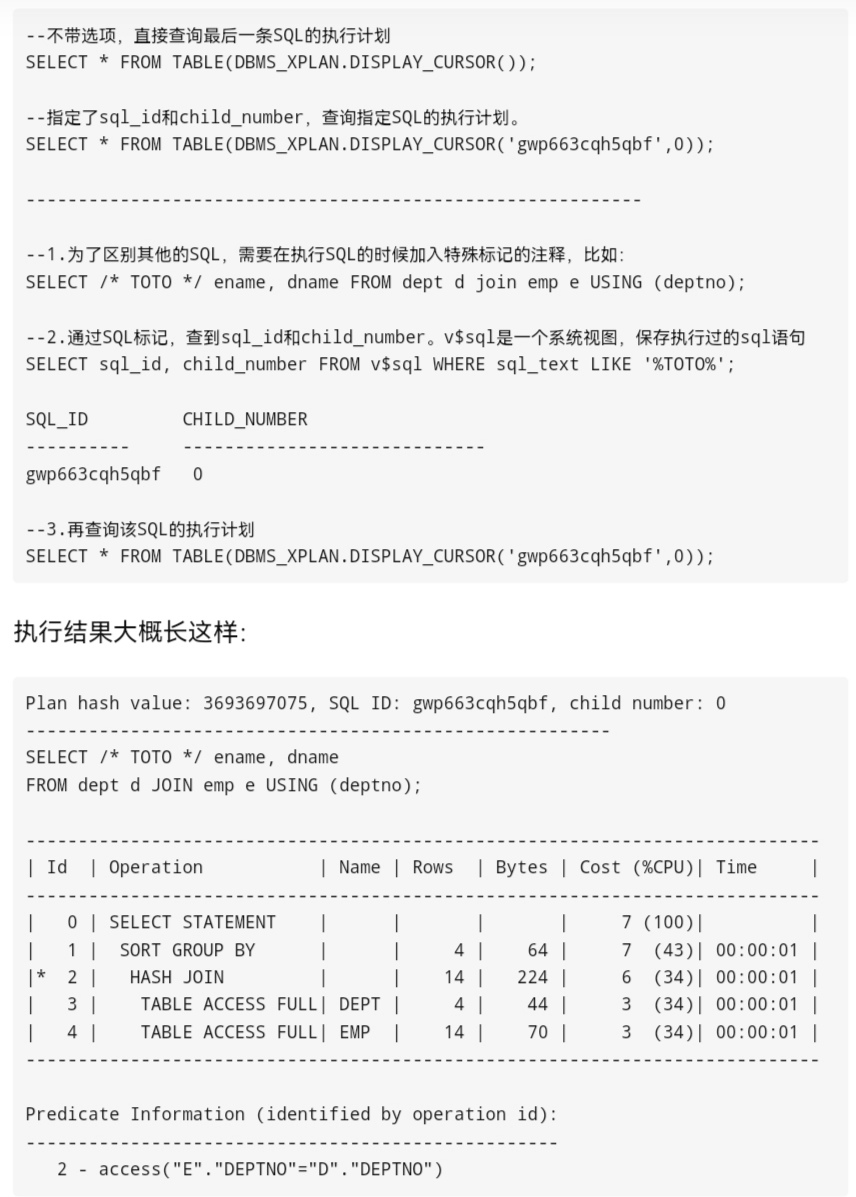
**预估执行计划**

DBMS\_XPLAN.DISPLAY是一个Oracle自带的函数，它可以接受一些选项用于展示执行计划。



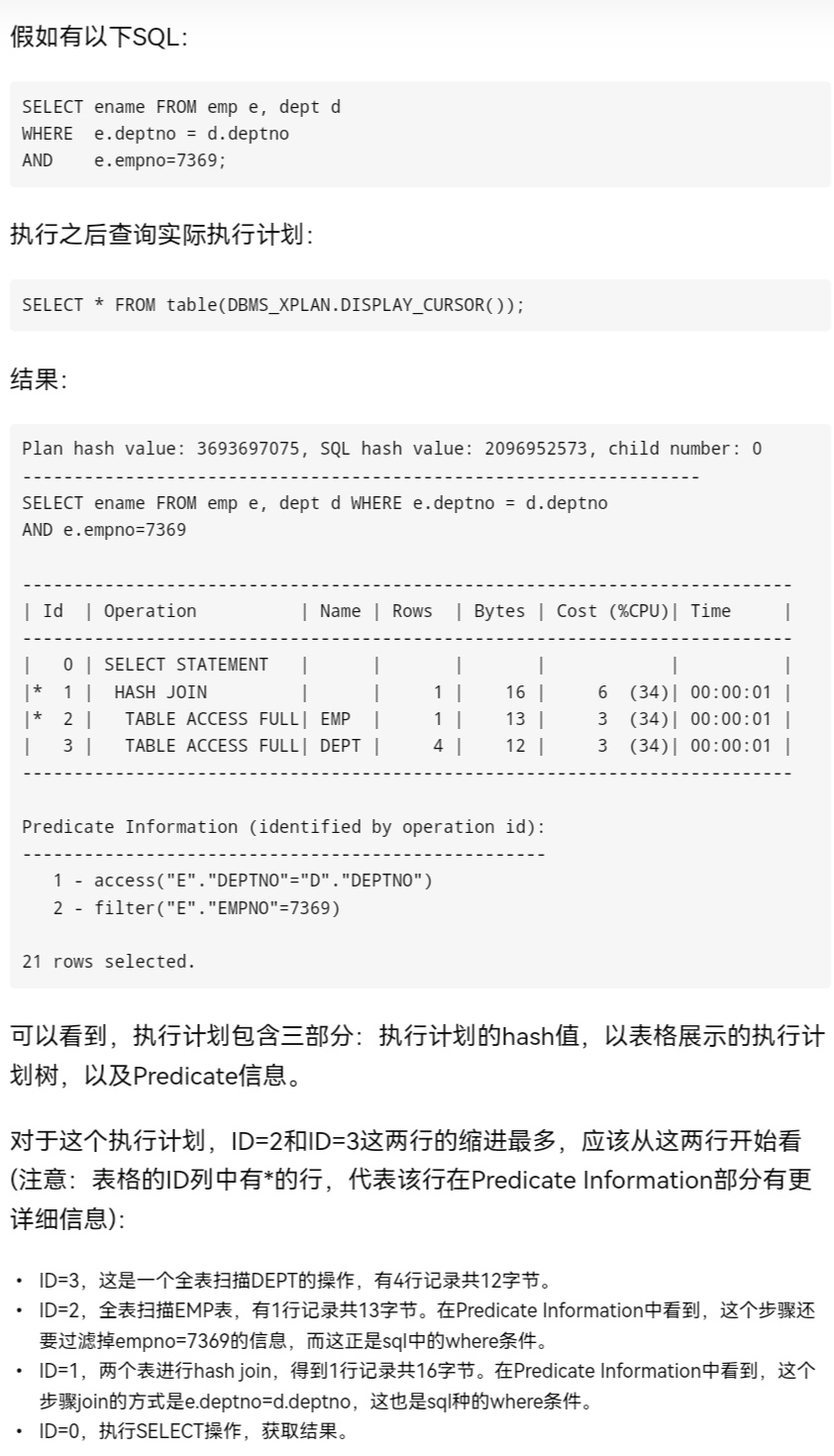
**实际执行计划**

实际执行计划要在执行SQL时才会生成，查看执行计划前，要先执行SQL。DBMS\_XPLAN.DISPLAY\_CURSOR也是一个Oracle自带的函数，它可以展示已经加载到cache中的执行计划：



##### 理解执行计划

Oracle的执行计划是以tab符和行的方式展示的，它是一棵执行树，在阅读执行计划时，要从树的叶子节点看起，即从执行计划表里Operation列缩进最多的一行看起。



[Oracle物化视图的用法与总结](https://www.jianshu.com/p/a06b3436f5ed)

物化视图是一个包括一个查询结果的数据库对象，它是远程数据的的本地副本，或者用来生成基于数据表求和的汇总表。物化视图存储基于远程表的数据，也可以称为快照。物化视图允许你在本地维护远程数据的副本，这些副本是只读的。

物化视图也是物理表，只不过这张表通过oracle的内部机制可以定期更新，将一些大的耗时的表连接用物化视图实现，会提高查询的效率。当然要打开查询重写选项。

物化视图一些重要而想详细的功能：

* 能够提高查询速度，因为物化视图存储了实际的数据，其次具有查询重写功能。最后，物化视图具有实体表，你也可以在上面建立索引。
* 简化了开发任务,开发的人员有的时候，无需直接关注部分sql的性能，而通过dba的努力，使用查询重写来完成性能的提升。
* 减少了工作量，物化视图可以定义两种刷新方式：立即刷新，按需刷新。所谓按需刷新就是你自己手动刷新，或者是定时刷新；所谓立即刷新，即视图主表发生变化的时候，视图立即刷新内容。
* 刷新量的灵活限制，可以快速是刷新（只刷新变化的），也可以全刷新。

##### [Oracle SQL 性能优化40条](https://zhuanlan.zhihu.com/p/143995493?utm_id=0)

1. ORACLE解析器按照从右到左的顺序处理FROM子句中的表名，因此FROM子句中写在最后的表(driving table)将被最先处理。
2. ORACLE解析器按照自下而上或自右向左的顺序解析WHERE子句。根据这个原理，表之间的连接必须写在其他WHERE条件之前，那些可以过滤掉最大数量记录的条件必须写在WHERE子句的末尾。
3. 避免在索引列上使用计算WHERE子句中，如果索引列是函数的一部分。优化器将不使用索引而使用全表扫描。

[Oracle SQL性能优化40条，值得收藏](https://cloud.tencent.com/developer/article/1424554)

## Apache Project

[Apache Mina](https://mina.apache.org/mina-project/)

Apache MINA is a network application framework which helps users develop high performance and high scalability network applications. It provides an abstract event-driven asynchronous API over various transports such as TCP/IP and UDP/IP via Java NIO.

[Apache SSHD](https://mina.apache.org/sshd-project/)

Apache MINA SSHD is a 100% pure java library to support the SSH protocols on both the client and server side. It does not aim at being a replacement for the SSH client or SSH server from Unix operating systems, but rather provides support for Java based applications requiring SSH support

## Cloud Native

云原生应用程序的各种模式以下列举的模式大致按照它们在本书中出现的顺序排列：

■ 请求/响应模式—这是一种服务间的通信协议，一个客户端会向某个远程服务发起一个请求，并且在大多数情况下希望接收到一个响应。这种通信可以是同步的，也可以是异步的，通常基于HTTP完成。

■ 事件驱动模式—这是一个分布式系统中各个实体之间进行事件通信的协议，在一个云原生的应用程序中，各个服务之间通过这些事件来保持最新的状态。

■ CQRS（Command Query Responsibility Segregation，命令查询职责分离）模式—在这种模式中，对某个领域实体的查询处理（读）与命令处理（写）是分开的。

■ 多服务实例模式—通过部署应用程序/服务的多个实例来保证弹性、可伸缩性及云原生的运维实践。

■ 水平伸缩模式—通过创建额外的应用程序实例来增加某个服务的容量。

■ 无状态服务模式—应用程序/服务不会在内存或者本地磁盘中不会存储任何状态。

■ 有状态服务模式—指用来持久化状态的服务，例如，数据库和消息队列。这些服务用来为无状态服务提供持久化数据。

■ **通过环境变量配置应用程序的模式**—在应用程序启动时，通过环境变量将参数值注入应用程序。

■ 配置服务模式—这是一个（有状态的）服务，用来将配置参数传递给多个应用程序的实例，以确保各个实例的行为是一致的。

■ 配置即代码模式—通过对配置文件进行版本控制来管理配置。

■ 零停机时间升级模式—一种让应用程序/服务在升级所有实例时仍然可以提供完整功能的方法。

■ 滚动升级模式—通过分批次、增量式升级部分服务实例来实现应用程序整体零停机时间升级的技术。

■ 蓝/绿升级—一种通过部署一组新的应用程序实例，然后全部切换到这些实例来升级应用程序的技术。

■ 应用程序健康检查—实现一个可以被调用的端点，用来访问某个应用程序的健康状态。

■ 活性探测—定期调用应用程序的健康端点，如果健康检查失败，则重新创建一个应用程序实例。

■ 服务端负载均衡—一种将请求在多个应用程序实例之间路由的方法，使得客户端只需向一个单独实体（负载均衡器）发送请求。

■ 客户端负载均衡—一种将请求在多个应用程序实例之间路由的方法，客户端知道并且可以控制路由到某个服务的多个实例。

■ 服务发现—客户端可以发现它所要调用服务的一个或多个地址。

■ 重试—当客户端无法接收到服务的响应时，可以重复发起一次请求。

■ 安全服务—一个服务无论被调用零次或者多次，返回的结果都是一样的。■ 幂等服务—一个服务无论被调用一次或者多次，返回的结果都是一样的。■ 回调—当发往下游服务的请求无法生成一个结果时，执行的应用程序逻辑。■ 断路器—一种用来阻止不断向某个故障服务实例发送请求，并且当服务恢复正常后允许继续发送请求的技术。

■ API网关—一个具有多种用途的服务代理，包括访问控制、审计、路由等。■ 挎斗—一种服务代理方法，代理服务本身与服务在一起。

■ 服务网格—多个挎斗的网络和控制台。

## Rate Limit

[5种限流算法，7种限流方式，挡住突发流量](https://www.cnblogs.com/niumoo/p/16007224.html)

[面试必备：4种经典限流算法讲解](https://zhuanlan.zhihu.com/p/376564740?utm_id=0)

[5种限流算法，7种限流方式，挡住突发流量](https://segmentfault.com/a/1190000041548601)

[Quick Guide to the Guava RateLimiter](https://www.baeldung.com/guava-rate-limiter#:~:text=In%20this%20article%2C%20we%27ll%20be%20looking%20at%20the,can%20issue%20at%20most%20N%20permits%20per%20second.)

[基于RateLimiter的分布式限流实现](https://zhuanlan.zhihu.com/p/165503126?utm_id=0)

https://www.baeldung.com/resilience4j

https://builtin.com/software-engineering-perspectives/rate-limiter

<https://zhuanlan.zhihu.com/p/141823121?utm_id=0>

message validation，在YAML里增加

ATOM\_PHACH\_ENTITY: F01HSBCBHPM

shell script: file level validation

message-validation: field validation

qurkus\_sftp\_connector: 可配置的行级校验(行长，不支持的字符), 文件完整性检查，可配置的限流

pol-lib-ach-sftp-kafka-connector:根据文件名后缀决定parser

## Books Repository

1. [搬书匠](http://www.banshujiang.cn/)
2. [资源收集站](https://www.9080hou.com/)
3. [程序员书库](https://www.werebook.com/book-category/software)
4. [PDF书籍之家](https://homeofbook.com/index.html)
5. [ComputerScienceFreeDownload](https://www.dbooks.org/subject/computer-science/)
6. [DockerKubernetesIstioELKBooks](https://github.com/ramosITBooks/DockerKubernetesIstioELKBooks)

MyResources(hakuxpaul/Cock@\*\*\*\*)

1. <https://hub.docker.com/>
2. https://github.com/hakuxpaul/dwolf