

Pulsar 概述

消息队列（Message Queue，简称MQ），是指在消息的传输中保存消息的容器或服务，是一种异步的服务间通信方式，适用于无服务器和微服务架构，是分布式系统实现高性能、高可用、可伸缩等高级特效的重要组件。

常见的主流消息队列有 ActiveMQ、RabbitMQ、ZeroMQ、Kafka、MetaMQ、RocketMQ、Pulsar 等。

Apache Pulsar 是一个多租户、高性能的服务间消息传输解决方案，支持多租户、低延时、读写分离、跨地域复制、快速扩容、灵活容错等特性。Pulsar 最初由 [Yahoo](#) 开发，目前由 [Apache 软件基金会](#) 管理。

Pulsar 的关键特性如下：

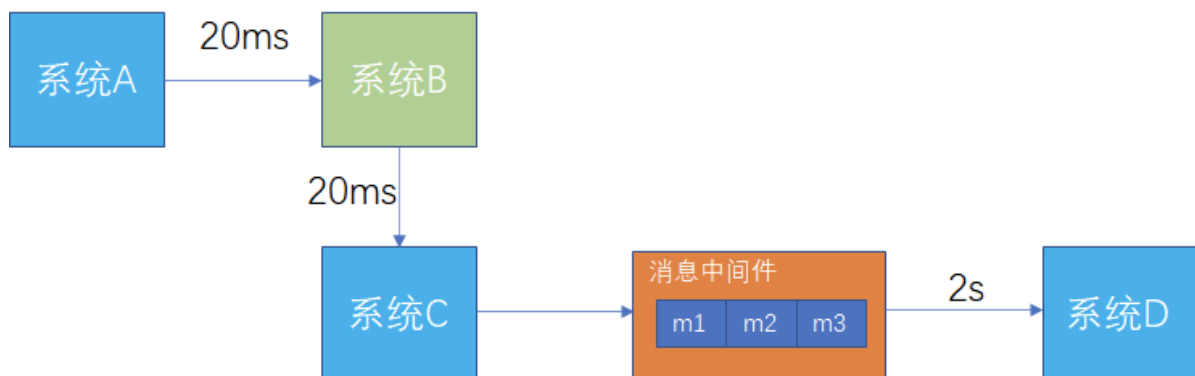
- Pulsar 的单个实例原生支持多个集群，可[跨机房](#)在集群间无缝地完成消息复制。
- 极低的发布延迟和端到端延迟。
- 可无缝扩展到超过一百万个 topic。
- 简单的[客户端 API](#)，支持 [Java](#)、[Go](#)、[Python](#) 和 [C++](#)。
- 支持多种 [topic 订阅模式](#)（[独占订阅](#)、[共享订阅](#)、[故障转移订阅](#)、Key_Shared）。
- 通过 [Apache BookKeeper](#) 提供的[持久化消息存储机制](#)保证消息传递。
 - 由轻量级的 serverless 计算框架 [Pulsar Functions](#) 实现流原生的数据处理。
 - 基于 Pulsar Functions 的 serverless connector 框架 [Pulsar IO](#) 使得数据更易移入、移出 Apache Pulsar。
 - [分层式存储](#)可在数据陈旧时，将数据从热存储卸载到冷/长期存储（如S3、GCS）中。

应用场景

传统消息队列的应用场景

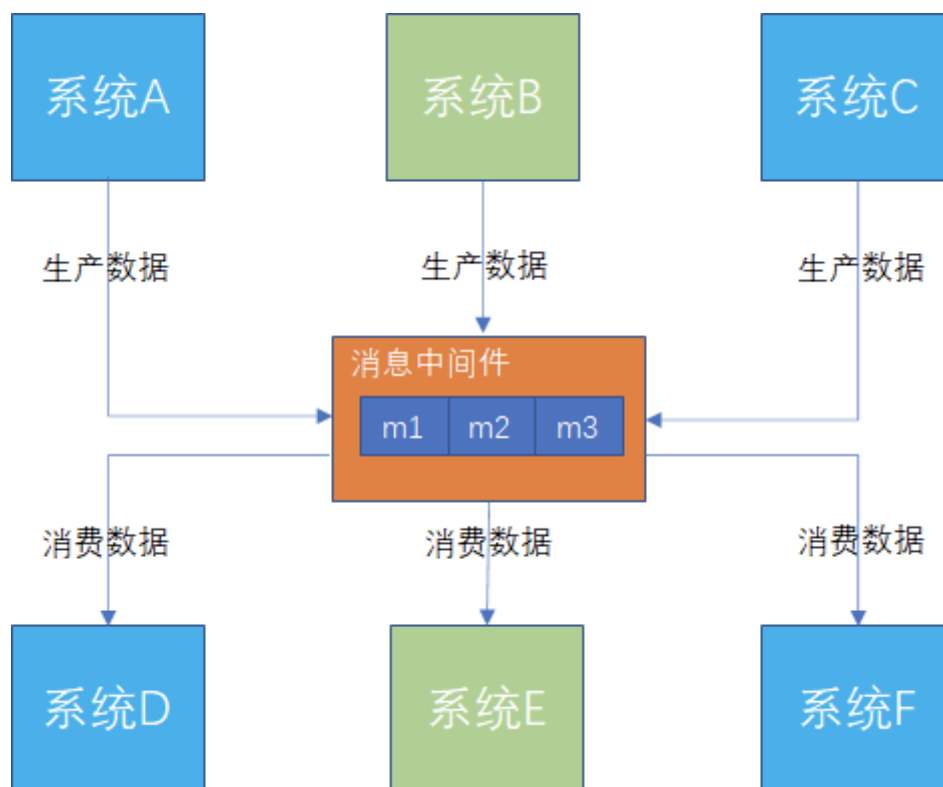
异步调用

假设有一个系统调用链路为 A 调用 B 耗时 20ms，B 调用 C 耗时 20ms，而 C 调用 D 需要 2s，这样下来整个调用需要耗时 2040ms。但实际上 A 调用 B，B 调用 C 只需要 40ms，而 D 系统的引入直接导致系统性能下降约 50 倍。此时我们可以考虑引入消息队列，将 D 系统的调用抽离出来，做一个异步调用：系统 A 到系统 B 再到系统 C 后就直接结束，系统 C 将消息发送到消息队列中，系统 D 从消息队列里取消息进行消费，这样子我们系统的性能就提高了接近 50 倍。



系统解耦

各个业务系统仅需要处理自己的业务逻辑，发送事件消息到消息队列。下游业务系统直接订阅消息队列的队列或主题获取事件。消息队列可用于单体应用被拆解为微服务后不同微服务间的通信。系统解耦的好处是不同系统的迭代不再相互依赖，能有效缩短数据链路长度，提高数据处理效率。



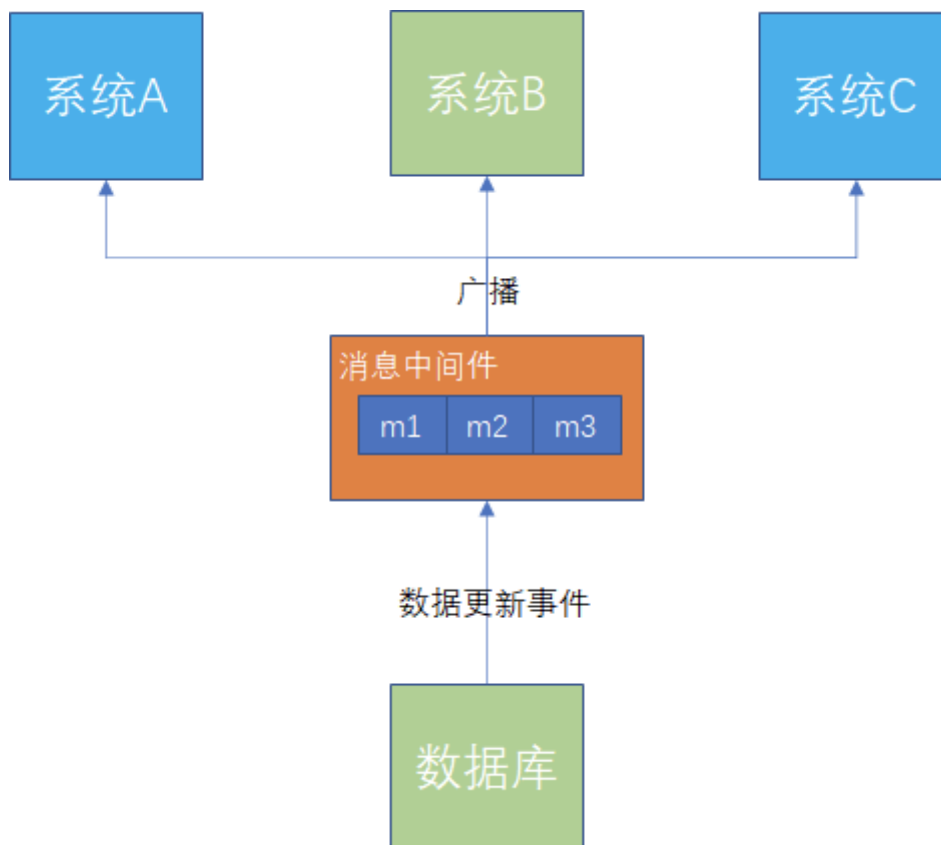
削峰填谷

大型活动带来较高流量时，没有做好相应保护容易导致系统超负荷甚至崩溃，而限制太过则会导致请求大量失败而影响用户体验。消息队列服务拥有高性能的消息处理能力，可以承接流量脉冲而不被击垮，在确保系统可用性的同时，通过快速有效的请求响应技术提升用户体验。其海量消息堆积能力确保下游业务在安全水位内平滑稳定的运行，避免流量高峰的冲击。



广播通知

系统一个状态的改变，需要通知多个相关系统，可通过消息订阅的方式推送给各个订阅者系统。比如数据库值的改变，需要通知所有的缓存系统更新，可以把数据库值改变发送消息给消息队列，然后各缓存订阅相关主题，收到消息后更新自己的缓存。



分布式缓存

在大数据场景中，日志分析往往需要处理大量日志，不可能存储在一台物理机上。消息队列可提供一个集群，用来存储海量消息，将其缓存到消息队列，进一步供实时分析系统分析日志。Kafka 和 TubeMQ 在大数据处理中往往充当分布式缓存的作用。

消息通讯

消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等。



Pulsar的应用场景

Pulsar 作为新一代存储计算分离架构的消息队列服务，**不仅适用于传统消息队列的应用场景，它的一些新特性还为更多应用场景带来可能。**

队列和流的融合-----维护一套 MQ 服务就够了

Apache Pulsar 抽象出了统一的 producer-topic-subscription-consumer 消费模型，既支持队列模型，也支持流模型。在 Pulsar 的消息消费模型中，Topic 是用于发送消息的通道。每一个 Topic 对应着 Apache BookKeeper 中的一个分布式日志。发布者发布的每条消息只在 Topic 中存储一次；存储的过程中，BookKeeper 会将消息复制存储在多个存储节点上；Topic 中的每条消息，可以根据消费者的订阅需求，多次被使用，每个订阅对应一个消费者组。尽管消息仅在主题（Topic）上存储一次，但是用户可以有不同的订阅方式来消费这些消息：

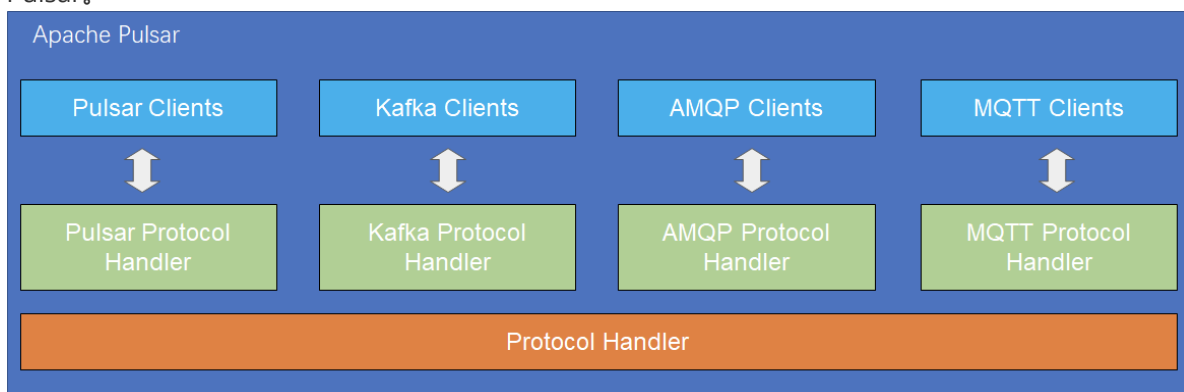
- 消费者被组合在一起以消费消息，每个消费组是一个订阅。
- 每个 Topic 可以有不同的消费组。

- 每组消费者都是对主题的一个订阅。
- 每组消费者可以拥有自己不同的消费方式：独占（Exclusive），故障切换（Failover）、共享（Share）等。

Pulsar 通过这种模型，将队列模型和流模型这两种模型结合在了一起，提供了统一的 API 接口。这种模型，既不会影响消息系统的性能，也不会带来额外的开销，同时还为用户提供了更多灵活性，方便用户程序以最匹配模式来使用消息系统。

多种 MQ 协议兼容-----轻松迁移传统 MQ 服务

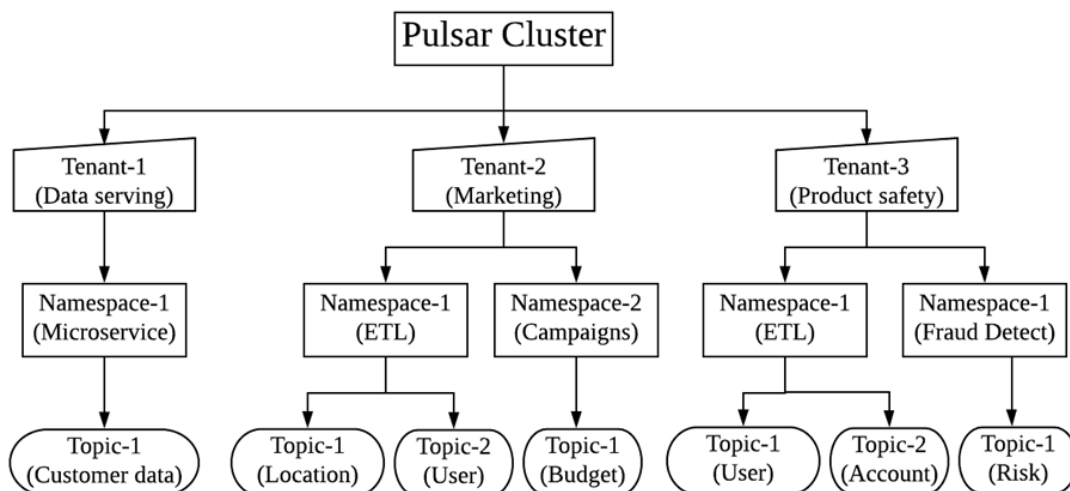
在 Pulsar 架构中，为了处理 Bookie 存储消息和防止消息丢失等，基于 Managed Ledger 实现了一套分布式的流程封装。Pulsar Protocol Handler 处理 Pulsar 中生产者 and 消费者发送出来的 TCP 请求，将其转化为可读取状态的操作。Pulsar 2.5 版本后，将 Protocol Handler 接口单独脱离了出来，利用这个框架就可以单独实现自定义协议的转换，比如 Kafka、AMQP 等，可以帮助存量的 MQ 业务轻松迁移到 Pulsar。



企业级多租户特性-----数据安全有保证

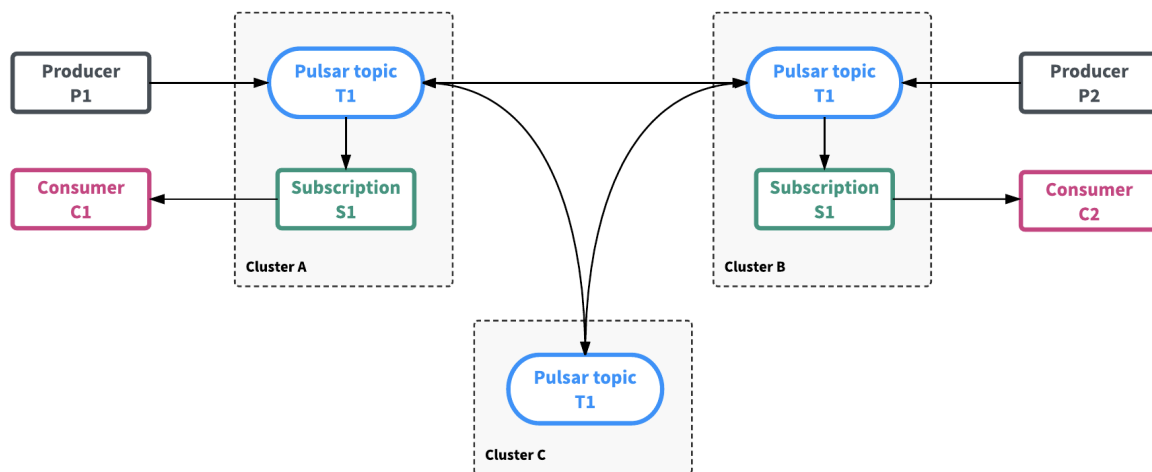
作为企业的消息中枢，Apache Pulsar 自诞生之日起就支持多租户，因为该项目最初就是为了满足 Yahoo 的严格需求，而当时市面上没有任何可用的开源系统能够提供多租户功能。在 Pulsar 的设计中，租户可以跨集群分布，每个租户都可以有单独的认证和授权机制；租户也是存储配额、消息 TTL 和隔离策略的管理单元。Pulsar 通过下列方式满足了多租户场景下的数据安全：

- 通过为每个租户进行身份验证、授权和 ACL（访问控制列表）获得所需安全性。
- 为每个租户强制实施存储配额。
- 以策略的方式定义所有隔离机制，策略可在运行过程中更改，借此降低运维成本并简化管理工作。



跨地域复制-----自带跨机房冗灾能力

在大型的分布式系统中，都会涉及到跨多个数据中心的需求。在对服务质量和灾备要求更高的场景中，会规划将机房部署在地理位置分散的多个数据中心内。在此类多数据中心部署中，通常会使用跨地域复制机制提供额外的冗余，以防某个数据中心故障、自然侵害或其他事件导致服务无法正常运转。Apache Pulsar 在设计之初就加入了对 Yahoo 全球十多个机房的跨地域复制的需求。Apache Pulsar 的跨地域多机房互备特性是 Pulsar 企业级特性的重要组成部分，它在保证数据稳定可靠的同时，为用户提供了便捷的操作和管理。

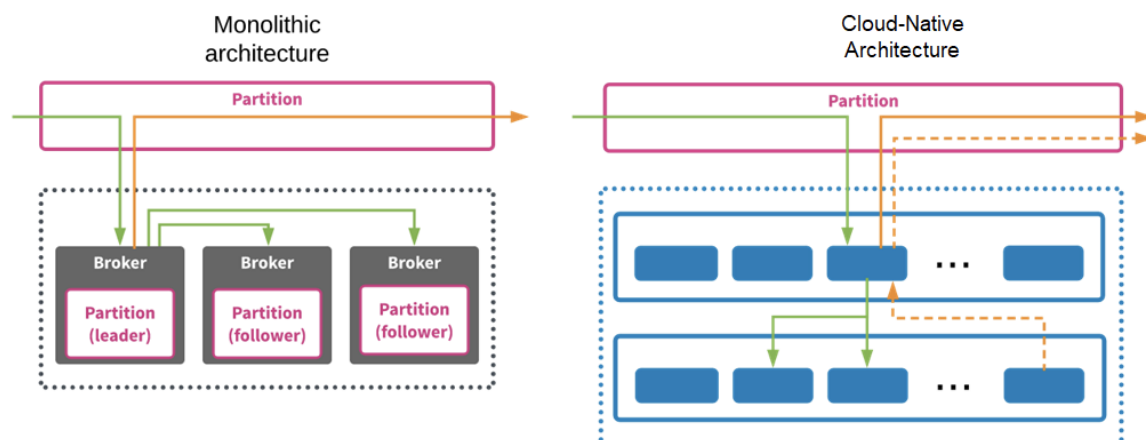


在上图中，无论 Producer P1、P2 和 P3 在什么时候分别将消息发布给 Cluster A、Cluster B 和 Cluster C 中的 Topic T1，这些消息均会立刻复制到整个集群。一旦完成复制，Consumer C1 和 C2 即可从自己所在的集群消费这些消息。

Pulsar 的跨地域复制不仅应用在跨数据中心数据备份的场景，在 PowerFL 联邦学习平台中跨地域复制的能力还被用来做通信服务使用。

云原生支持-----助力服务上云

云原生的原生即软件设计之初就考虑到了将来会被运行在云端的可能，从而在设计层面上就充分利用了云资源的特点，典型的是分布式和弹性伸缩的能力。Pulsar 之所以说是云原生的消息平台，**核心就是它的架构设计能够充分利用分布式的、能够弹性伸缩的云端资源**。以 Pulsar on Kubernetes 为例，Bookie 是有状态的节点，但是节点之间是对等的，可以采用 StatefulSet 来部署；而 Broker 作为无状态的节点，直接使用 ReplicaSet 即可，每个 Pod 支持水平扩展。



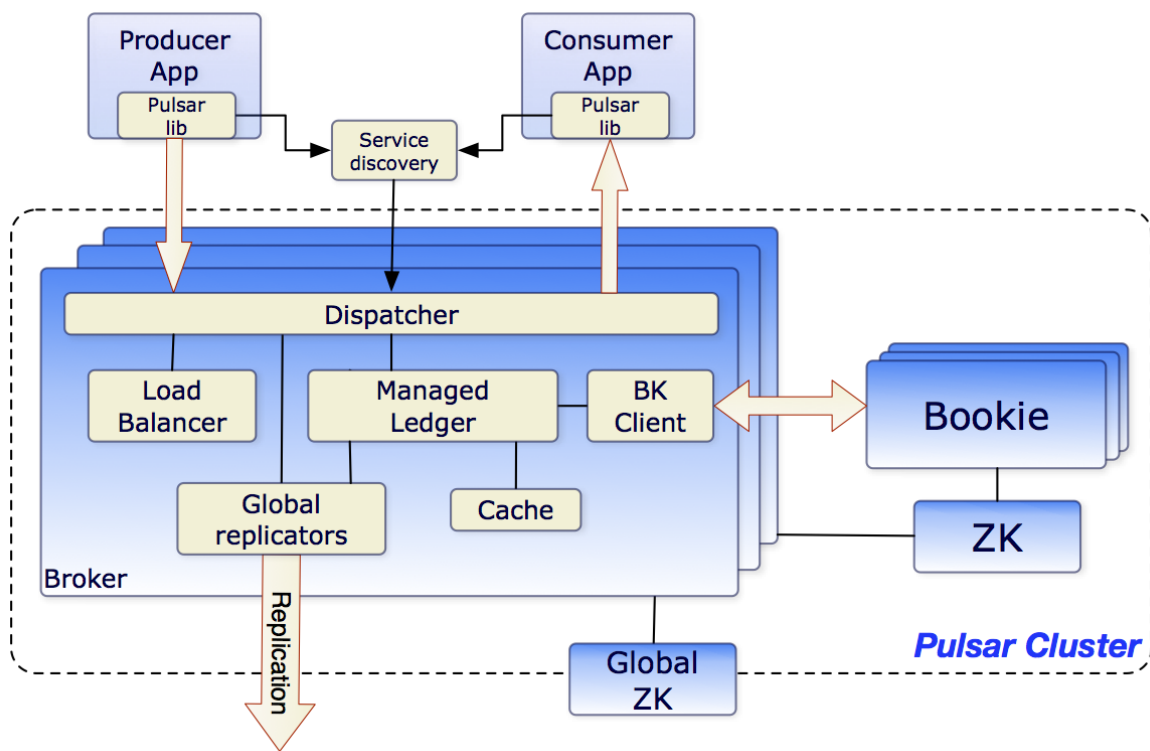
架构概述

在最高级别，单个 Pulsar 实例由一个或多个 Pulsar 集群组成。实例中的集群之间可以[相互复制](#)数据。

单个 Pulsar 集群由以下三部分组成：

- 一个或者多个 broker 负责处理和负载均衡 producer 发出的消息，并将这些消息分派给 consumer；Broker 与 Pulsar 配置存储交互来处理相应的任务，并将消息存储在 BookKeeper 实例中（又称 bookies）；Broker 依赖 ZooKeeper 集群处理特定的任务，等等。
- 包含一个或多个 bookie 的 BookKeeper 集群负责消息的[持久化存储](#)。
- 一个 Zookeeper 集群，用来处理多个 Pulsar 集群之间的协调任务。

下图为一个 Pulsar 集群：



在更细粒度的实例级别，有一个能访问到全部实例的 ZooKeeper 群集处理涉及多个 Pulsar 集群的配置协调任务，例如 [异地复制](#)。

集群

一个 Pulsar 实例包含一个或者多个 Pulsar 集群。集群包括：

- 一个或者多个 Pulsar [brokers](#)
- 一个 ZooKeeper 协调器，用于集群级别的配置和协调
- 一组 BookKeeper 的 Bookies 用于消息的 [持久化存储](#)

集群间可以通过[异地复制](#)进行消息同步

Brokers

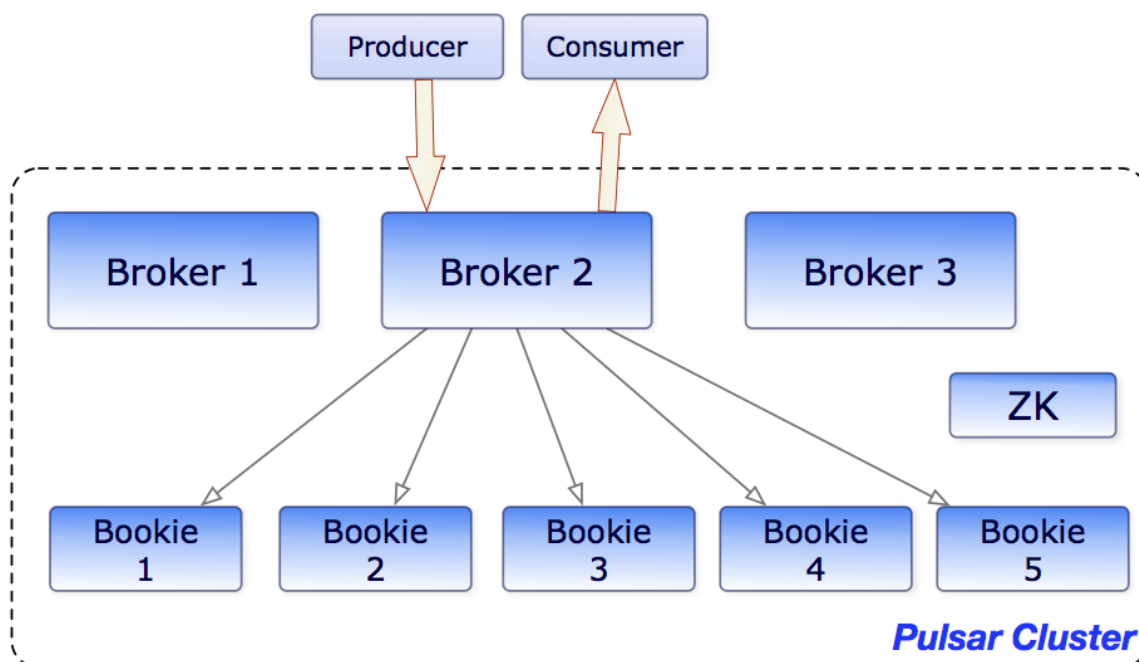
Pulsar 的 broker 是一个无状态组件，负责处理从生产者发出消息、向消费者派发消息、在集群间复制数据等。

- 一个 HTTP 服务器，它为生产者和消费者提供管理任务和主题查找的 [REST API](#)。生产者连接到代理发布消息，消费者连接到代理来消费消息。
- 一个调度分发器，它是异步的 TCP 服务器，通过自定义 [二进制协议](#) 应用于所有相关的数据传输。

BookKeeper

Pulsar 使用 BookKeeper 来存储所有的持久化消息。BookKeeper 是一个分布式的 [预写日志 \(write-ahead log\)](#) 系统。它保证了独立消息日志读取的一致性，称为 ledger。每个 BookKeeper 服务器也被称为 bookie。

下图展示了brokers和bookies是如何交互的



ZooKeeper

ZooKeeper 负责管理与 Pulsar 协调和配置相关的各种基本任务，每个 Pulsar 实例需要两个独立的 ZooKeeper 集群。

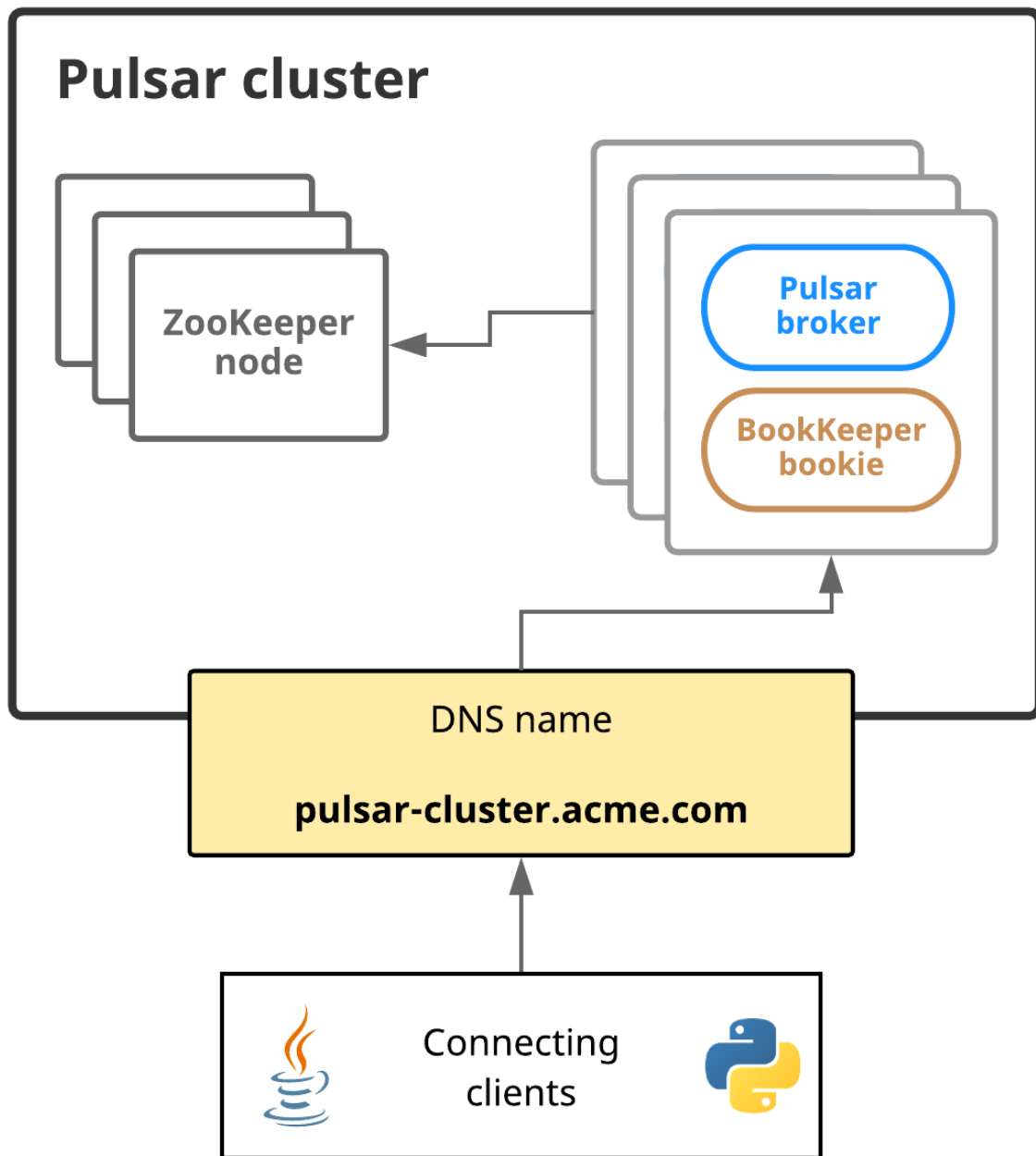
- [本地 ZooKeeper](#) 负责集群级别的操作，提供集群的配置管理和协调。每个 Pulsar 集群都需要一个专用的 ZooKeeper 集群。
- [Configuration Store](#) 在实例层面运行，为整个系统提供配置管理（从而实现跨集群）。配置集群可以部署在独立的机器集群，也可以和本地 Zookeeper 集群部署在同一批机器。

服务发现

[客户端](#) 需要能够使用单个 URL 与整个 Pulsar 实例进行通信。Pulsar 内部提供了服务发现的机制，你可以通过 [配置 Pulsar 实例指南](#) 设置。

你也可以用你自己的服务发现系统。如果你用你自己的系统，只需满足一个需求：当客户端发送一个 HTTP 请求，例如发到 `http://pulsar.us-west.example.com:8080`，客户端需要被重定向到某些所需的集群中活跃的 broker，或者通过 DNS，或者通过 HTTP 和 IP 重定向，或者其他机制。

下面这张图展示了 Pulsar 服务发现机制：



多租户

成为一个多租户系统是 Pulsar 最初设计理念的一部分。为了支持多租户，Pulsar 有租户的概念。租户可以跨集群分布，每个租户都可以有单独的[认证和授权](#)机制。租户也是存储配额、[消息 TTL](#) 和隔离策略的管理单元。Pulsar 的多租户性质主要体现在 Topic 的 URL 中，结构如下：

`persistent://tenant/namespace/topic` 可以看到，租户是 topic 的最基本单位（比命名空间和 topic 名称更为基本）。

消息

Pulsar 采用 [发布-订阅](#) 的设计模式(简称 pub-sub)，在这种模式中，[producers](#) 向 [topics](#) 发送消息；[consumers](#) 订阅 topics，处理传入的消息，并且处理完成后向 broker 发送 ACK 确认。

一旦创建订阅，即使 consumer 断开连接，Pulsar 仍然可以[保存](#)所有消息。只有当消费者确认所有这些消息都已成功处理时，才会丢弃保留的消息。

消息是 Pulsar 的基本“单元”。下表列出了消息的组成部分。

组件	说明
Value / data payload	消息所承载的数据。 尽管消息数据也可以符合数据 schemas ，但所有 Pulsar 消息都包含原始字节。
Key	消息可以选择用键进行标记，这在 topic 压缩 等操作很有用。
Properties	用户自定义属性的键值对（可选）。
Producer 名称	生成消息的 producer 的名称。 如果不指定，则使用默认名称
序列 ID	每个 Pulsar 消息都存储在其主题上的有序序列中。消息的序列 ID 是其在序列中的顺序。
发布时间	消息发布的时间戳，由 producer 自动添加。
事件时间	应用程序可以附加到消息的时间戳（可选）， 例如处理消息的时间。如果没有明确设置，则消息的事件时间为 0。
TypedMessageBuilder	用于构造消息。 您可以使用 <code>TypedMessageBuilder</code> 设置消息的键值对属性。 在设置 <code>TypedMessageBuilder</code> 时，最佳的选择是将 key 设置为字符串。 如果将 key 设置为其他类型（例如，AVRO 对象），则 key 会以字节形式发送，这时 consumer 就很难使用了。

Topic

与其他 pub-sub 系统一样，Pulsar 中的 Topic 是用于将消息从生产者传输到消费者的命名通道。 主题名称是具有明确定义结构的 URL：

```
{persistent|non-persistent}://tenant/namespace/topic
```

Topic 名称 组成	说明
持久化 / 非持久化	用来标识 topic 的类型。 Pulsar 支持两种主题类型： 持久化 和 非持久化 。 主题默认是持久化类型，如果不特殊指定主题类型，那主题就是持久化的。 对于持久化的主题，所有的消息都会被持久化的保存到磁盘当中(如果 broker 不是单机模式，消息会被持久化到多块磁盘)，而非持久化的主题的数据不会被保存到磁盘里面。
租户	实例中的主题租户。 租户是 topic 的最基本单位（比命名空间和 topic 名称更为基本），租户是 Pulsar 多租户的基本要素，租户可以跨集群分布，每个租户都可以有单独的 认证和授权 机制。
命名空间	将相关联的 topic 作为一个组来管理，是管理 Topic 的基本单元。 大多数对 topic 的管理都是对 命名空间 的一项配置。 每个租户里面可以有一个或者多个命名空间。
topic	主题名称是主题命名结构的最后一个部分，主题名字在 Pulsar 实例当中没有特殊意义。

角色标识

在 Pulsar 中，*role* 是一个字符串，比如 `admin` 或 `app1`，它可以代表单个或者多个客户端。 您可以使用这些角色来控制客户端生产或消费某些特定 topic 的权限，管理租户的配置等。

Apache Pulsar 使用[Authentication Provider](#)在建立连接的时候来验证客户端的身份，然后将角色令牌分配给这个客户端。这个角色标识用于 [授权和 ACL](#)，确定了客户端被授权做什么。

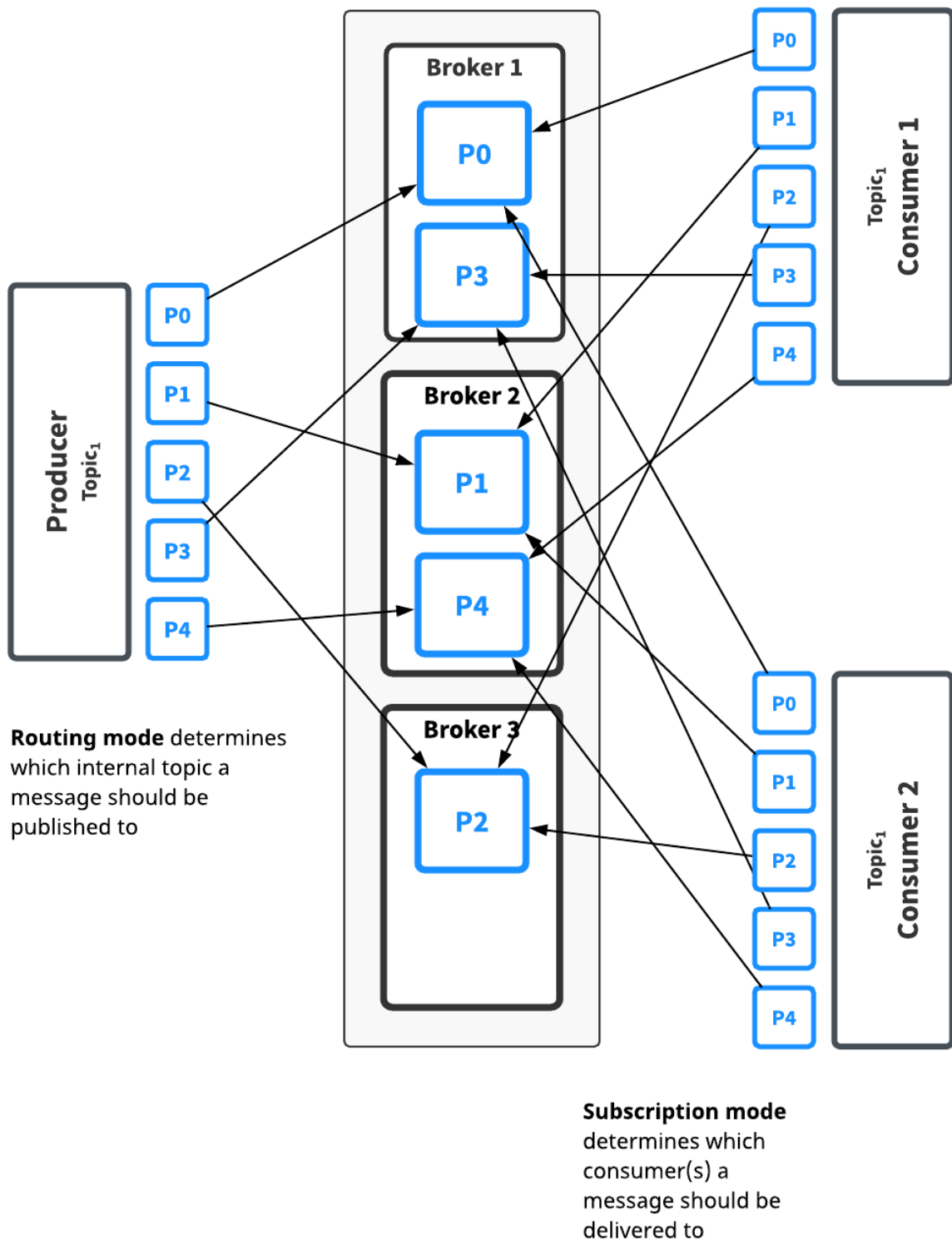
分区 topic

普通的主题仅仅被保存在单个 broker 中，这限制了主题的最大吞吐量。分区主题 是一种特殊类型的主题，由多个代理处理，从而获得更高的吞吐量。

分区主题实际是通过在底层拥有 N 个内部主题来实现的，这个 N 的数量就是等于分区的数量。当向分区的topic发送消息，每条消息被路由到其中一个broker。Pulsar自动处理跨broker的分区分布。

下图对此做了阐明：

Pulsar cluster



主题**Topic1** 有5个分区（**P0 ~P4**），3个brokers。因为分区多于broker数量，其中有两个broker要处理两个分区。第三个broker则只处理一个。（再次强调，分区的分布是Pulsar自动处理的）。

这个topic的消息被广播给两个consumer。路由模式确定每条消息该发往哪个分区，而订阅模式确定消息传递给哪个消费者。

大多数境况下，路由和订阅模式可以分开制定。通常来讲，吞吐能力的要求，决定了分区/路由的方式。订阅模式则应该由应用的语义来做决定。

分区topic和普通topic，对于订阅模式如何工作，没有任何不同。分区只是决定了从生产者生产消息到消费者处理及确认消息过程中发生的事情。

分区topic需要通过[admin API](#)指定创建。创建的时候可以指明分区的数量。

路由模式

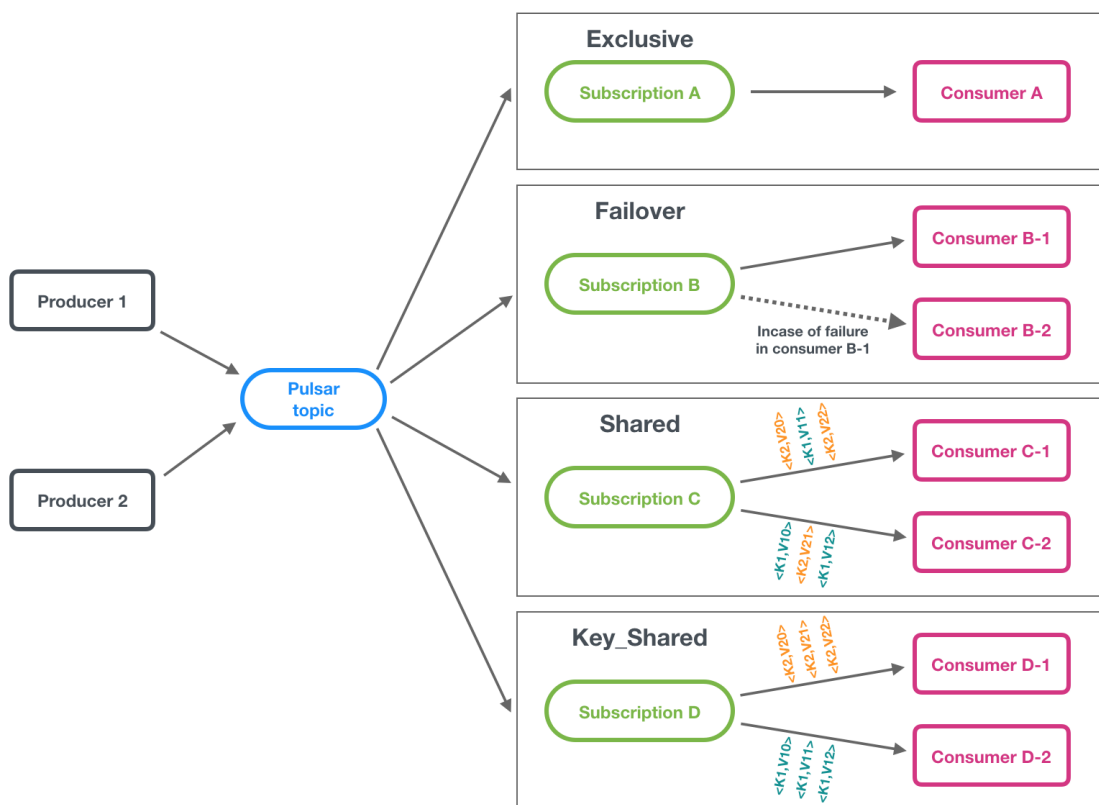
当发布消息到分区topic，你必须要指定路由模式。路由模式决定了每条消息被发布到的分区---其实是内部主题。

有三种 [MessageRoutingMode](#) 可用:

发送模式	说明
<code>RoundRobinPartition</code>	如果消息没有指定 key，为了达到最大吞吐量，生产者会以 round-robin 方式将消息发布到所有分区。请注意round-robin并不是作用于每条单独的消息，而是作用于延迟处理的批次边界，以确保批处理有效。如果消息指定了key，分区生产者会根据key的hash值将该消息分配到对应的分区。这是默认的模式。
<code>SinglePartition</code>	如果消息没有指定 key，生产者将会随机选择一个分区，并发布所有消息到这个分区。如果消息指定了key，分区生产者会根据key的hash值将该消息分配到对应的分区。
<code>CustomPartition</code>	使用自定义消息路由器实现来决定特定消息的分区。用户可以创建自定义路由模式：使用 Java client 并实现 MessageRouter 接口。

订阅模式

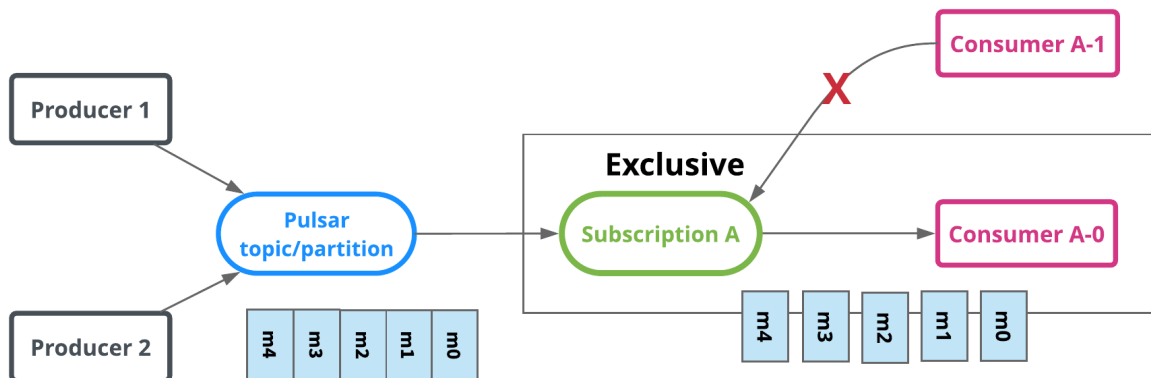
订阅是命名好的配置规则，指导消息如何投递给消费者。Pulsar 中有四种订阅模式: [独占](#)，[共享](#)，[灾备](#) 和[key共享](#) 下图展示了这四种模式：



独占模式

在独占模式下，只允许一个消费者附加到订阅。如果多个消费者使用同一个订阅订阅一个主题，则会发生错误。

在下图中，仅允许消费者 A-0 消费消息。



- 不设置订阅模式时，默认为独占模式。

多个 consumer，如果使用相同的 subscriptionName 订阅同一个主题 topic，只有第一个 consumer 能够连接成功

第二个 consumer 连接 pulsar 时:

```
./recvccc test-topic aaa111
2021-08-30 17:35:58.983 INFO [140178222058176] Client:88 | Subscribing on Topic :test-topic
2021-08-30 17:35:58.983 INFO [140178222058176] ClientConnection:190 | [<none> -> pulsar://localhost:6650] Create ClientConnection, timeout=10000
2021-08-30 17:35:58.983 INFO [140178222058176] ConnectionPool:84 | Created connection for pulsar://localhost:6650
2021-08-30 17:35:58.985 WARN [140178127075072] ClientConnection:437 | [<none> -> pulsar://localhost:6650] Failed to establish connection: Connection refused
2021-08-30 17:35:58.985 INFO [140178127075072] ClientConnection:375 | [127.0.0.1:49224 -> 127.0.0.1:6650] Connected to broker
2021-08-30 17:35:58.987 INFO [140178127075072] HandlerBase:64 | [persistent://public/default/test-topic-partition-0, aaa111, 0] Getting connection from pool
2021-08-30 17:35:58.987 INFO [140178127075072] HandlerBase:64 | [persistent://public/default/test-topic-partition-1, aaa111, 1] Getting connection from pool
2021-08-30 17:35:58.987 INFO [140178127075072] HandlerBase:64 | [persistent://public/default/test-topic-partition-2, aaa111, 2] Getting connection from pool
2021-08-30 17:35:58.987 INFO [140178127075072] HandlerBase:64 | [persistent://public/default/test-topic-partition-3, aaa111, 3] Getting connection from pool
2021-08-30 17:35:58.996 WARN [140178127075072] ClientConnection:1030 | [127.0.0.1:49224 -> 127.0.0.1:6650] Received error response from server: ConsumerBusy (Exclusive consumer is already connected) -- req_id: 0
2021-08-30 17:35:58.996 ERROR [140178127075072] ConsumerImpl:263 | [persistent://public/default/test-topic-partition-0, aaa111, 0] Failed to create consumer: ConsumerBusy
2021-08-30 17:35:58.996 ERROR [140178127075072] PartitionedConsumerImpl:326 | Closing the consumer failed for partition - 0
```

```

2021-08-30 17:35:58.996 ERROR [140178127075072] PartitionedConsumerImpl:297 |
Unable to create Consumer for partition - 0 Error - ConsumerBusy
2021-08-30 17:35:58.996 WARN [140178127075072] ClientConnection:1030 |
[127.0.0.1:49224 -> 127.0.0.1:6650] Received error response from server:
ConsumerBusy (Exclusive consumer is already connected) -- req_id: 1
2021-08-30 17:35:58.996 ERROR [140178127075072] ConsumerImpl:263 |
[persistent://public/default/test-topic-partition-1, aaa111, 1] Failed to create
consumer: ConsumerBusy
2021-08-30 17:35:58.996 WARN [140178127075072] ClientConnection:1030 |
[127.0.0.1:49224 -> 127.0.0.1:6650] Received error response from server:
ConsumerBusy (Exclusive consumer is already connected) -- req_id: 2
2021-08-30 17:35:58.996 ERROR [140178127075072] ConsumerImpl:263 |
[persistent://public/default/test-topic-partition-2, aaa111, 2] Failed to create
consumer: ConsumerBusy
2021-08-30 17:35:58.996 WARN [140178127075072] ClientConnection:1030 |
[127.0.0.1:49224 -> 127.0.0.1:6650] Received error response from server:
ConsumerBusy (Exclusive consumer is already connected) -- req_id: 3
2021-08-30 17:35:58.996 ERROR [140178127075072] ConsumerImpl:263 |
[persistent://public/default/test-topic-partition-3, aaa111, 3] Failed to create
consumer: ConsumerBusy
Failed to create subscribe to topic: ConsumerBusy

```

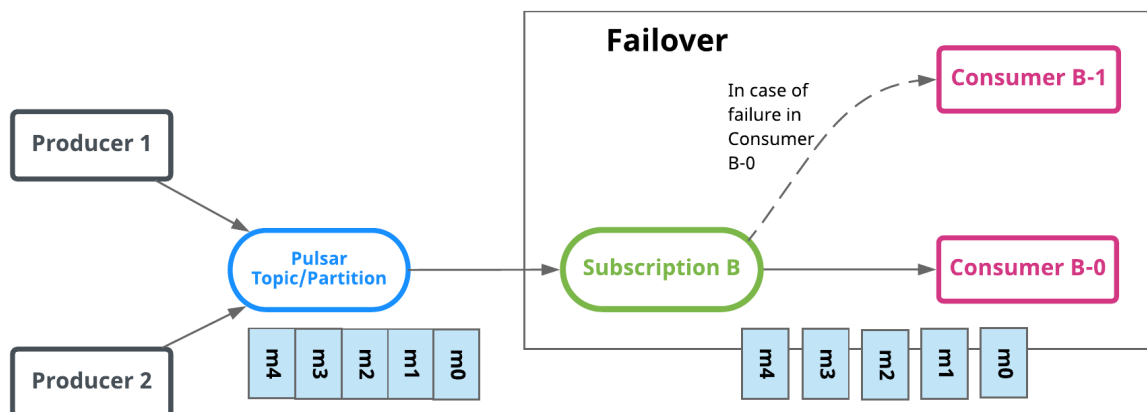
灾备模式

*Failover*模式中，多个consumer可以绑定到同一个subscription。主消费者会消费非分区主题或者分区主题中的每个分区的信息。当主消费者断开连接时，所有（未确认的和后续的）消息都会传递给下一个消费者。

对于分区主题来说，Broker 将按照消费者的优先级和消费者名称的词汇表顺序对消费者进行排序。然后试图将主题均匀的分配给优先级最高的消费者。

对于非分区主题来说，broker 会根据消费者订阅非分区主题的顺序选择消费者。

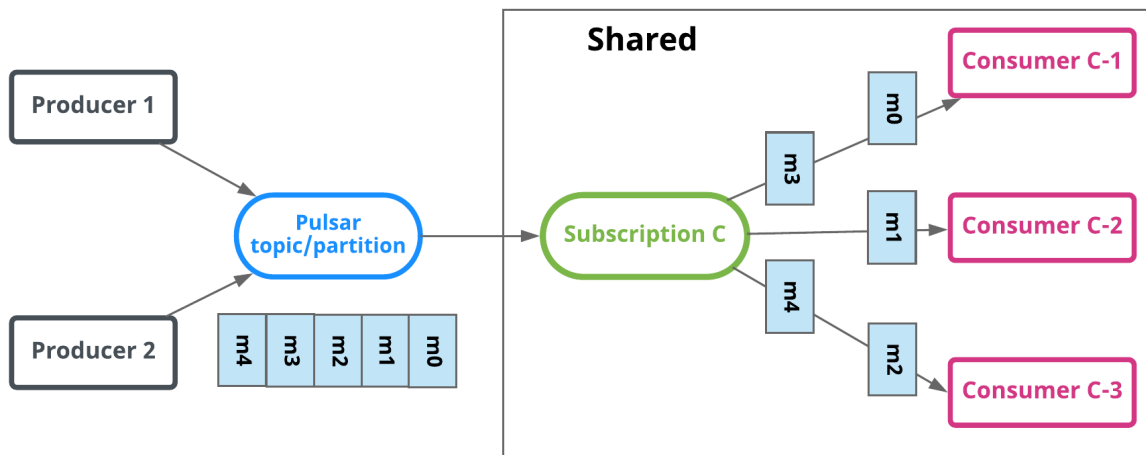
在下图中，Consumer-B-0 是主消费者，而如果 Consumer-B-0 断开连接，Consumer-B-1 将是下一个接收消息的消费者。



共享模式

*shared*或者*round robin*模式中，多个消费者可以绑定到同一个订阅上。消息通过*round robin*轮询机制分发给不同的消费者，并且每个消息仅会被分发给一个消费者。当消费者断开连接，所有被发送给他，但没有被确认的消息将被重新安排，分发给其它存活的消费者。

在下图中，Consumer-C-1 和 Consumer-C-2 能够订阅该主题，但 Consumer-C-3 和其他人也可以。



多个 consumer，使用相同的订阅名称 subscriptionName 订阅同一个主题 topic，可以将该 topic 下的消息分发至各个 consumer

例如Producer发送10条消息，两个 consumer 使用共享模式订阅

```

./recvccc test-topic aaa111
./recvccc test-topic aaa111

./sendccc test-topic 123
  Sent message 0
  Sent message 1
  Sent message 2
  Sent message 3
  Sent message 4
  Sent message 5
  Sent message 6
  Sent message 7
  Sent message 8
  Sent message 9

```

两个 consumer 收到的消息分别为

```

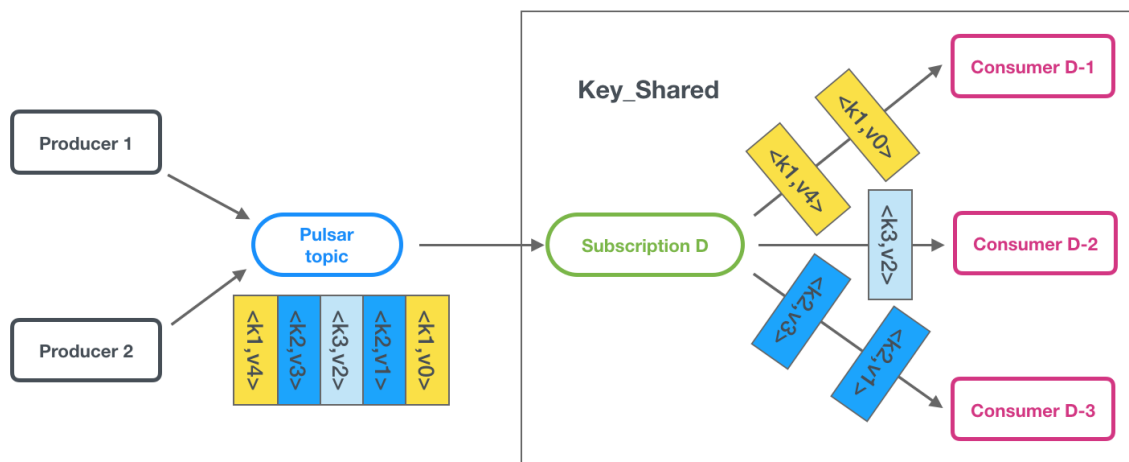
Received message with payload: '1 : 123'
Received message with payload: '3 : 123'
Received message with payload: '4 : 123'
Received message with payload: '5 : 123'
Received message with payload: '8 : 123'
Received message with payload: '9 : 123'

Received message with payload: '0 : 123'
Received message with payload: '2 : 123'
Received message with payload: '6 : 123'
Received message with payload: '7 : 123'

```


Key_Shared

In *Key_Shared* mode, 在 *Key_Shared* 模式下, 多个消费者可以附加到同一个订阅。消息在跨消费者的分布中传递, 具有相同键或相同排序键的消息仅传递给一个消费者。无论消息被重新传递多少次, 它都会传递给同一个消费者。当消费者连接或断开连接时, 将导致服务的消费者更改某些消息的关键字。



消息队列对比

Key_Shared

目前在市面上比较主流的消息队列中间件主要有, **Kafka**、**ActiveMQ**、**RabbitMQ**、**RocketMQ** 等这几种。

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比 RocketMQ、Kafka 低一个数量级	同 ActiveMQ	10 万级，支撑高吞吐	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
时效性	ms 级	微秒级，这是 RabbitMQ 的一大特点，延迟最低	ms 级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用	同 ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据	基本不丢	经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于 erlang 开发，并发能力很强，性能极好，延时很低	MQ 功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用
社区活跃度	低	中	高	高

知乎 @敖丙

Pulsar与Kafka的对比

对比	Pulsar	Kafka
主题可扩展性	轻松处理百万个主题，随意添加更多Bookies,无需rebalance	不擅长处理大量主题，Broker是绑定存储状态的，集群扩展时必须做rebalance
性能	分层架构，IO隔离读写分离，支持 non-persistent,更高的吞吐量	服务和数据绑定，持久主题
消费模式	流（Stream）模式：独占和故障切换订阅方式 队列（Queue）模式：共享订阅的方式	主要集中在流（Stream）模式，对单个partition是独占消费。无队列模式
操作更简单	使用专门的Cursor管理，并存在Ledger中	使用偏移量Offset,存在zk中或者_consumer_topic中
消费重复消息删除	可设置启动消息重复删除	无，需要自己逻辑实现
无限的数据保留期	消息只有被所有订阅消费后才会删除，不会丢失数据，但允许设置保留期，保留被消费的数据。支持TTL(Time To Live)	根据设置的保留期来删除消息，有可能消息未被消费，过期后就删除了。不支持TTL。

为什么选择Pulsar而不是Kafka？

对比	Pulsar	Kafka
分区存储优势	<p>在 Apache Pulsar 中，同样是以分区为逻辑单元，但是以 Segment 为物理存储单元。分区随着时间的推移会进行分段，并在整个集群中均衡分布，旨在有效地迅速地扩展。</p> <p>Pulsar 是以 Segment 为中心的，因此在扩展容量时不需要数据重新平衡和拷贝，旧数据不会被重新复制，这要归功于在 Apache BookKeeper 中使用可扩展的以 Segment 为中心的分布式日志存储系统。</p>	<p>在 Apache Kafka 中，分区只能存储在单个节点上并复制到其他节点，其容量受最小节点容量的限制。这意味着容量扩展需要对分区重新平衡，这反过来又需要重新复制整个分区，以平衡新添加的代理的数据和流量。重新传输数据非常昂贵且容易出错，并且会消耗网络带宽和 I/O。维护人员在执行此操作时必须非常小心，以避免破坏生产系统。</p>
支持分区，但不是必需的	<p>Pulsar 就可以做到。如果你只需要一个主题，而不需要分区，那使用一个主题就好了。如果你需要使用多个消费者实例来提升处理速率，其实也不需要分区，因为 Pulsar 的共享订阅可以达到你的目的。如果你确实需要分区来进一步提升性能，你也可以使用分区。</p>	<p>Kafka 中的所有主题都是分区的，这样可以增加吞吐量。通过分区，单个主题的处理速率可以得到大幅提升。但如果某些主题不需要太高的处理速率，那该怎么办？对于这类情况，就不需要考虑分区了，以避免复杂的 API 和管理方面的工作，这样不是更好吗？</p>
日志固然不错，但 ledger 更胜一筹	<p>Pulsar 对日志进行分段，从而避免了拷贝大块的日志。它通过 BookKeeper 将日志分段分散到多台不同的服务器上。也就是说，日志并不是保存在单台服务器上，所以任何一台服务器都不会成为整个系统的瓶颈。这样就可以更容易地处理故障，要进行伸缩也很容易，只需要加入新的服务器，不需要进行再均衡。</p>	<p>Kafka 开发团队预见了日志对于一个实时数据交换系统的重要性。因为日志是通过追加的方式写入系统的，所以数据写入速度很快。又因为日志中的数据是串行的，所以可以按照写入的顺序快速读取数据。相比随机读取和写入，串行读取和写入速度更快。对于任何一个提供数据保证的系统来说，持久化存储方面的交互都是一个瓶颈，而日志抽象最大限度地提升了这方面的效率。日志固然是好，但当它们的量增长到很大的时候，也会给我们带来一些麻烦。在单台服务器上保存所有日志已经成为一个挑战。在服务器存储被日志填满之后该怎么办？如何进行伸缩？或者保存日志的服务器宕机，需要重新从副本创建新的服务器，该怎么办？将日志从一台服务器拷贝到另一台服务器是很耗费时间的，特别是如果你想要在保持系统实时数据的情况下完成这个操作就更难了。</p>

对比	Pulsar	Kafka
无状态	<p>在 Pulsar 架构中, broker 是无状态的。但是完全无状态的系统是无法用来持久化消息的, 所以 Pulsar 其实是有维护在状态的, 只是不是在 broker 上。</p> <p>在 Pulsar 架构中, 数据的分发和保存是相互独立的。broker 从生产者接收数据, 然后将数据发送给消费者, 但数据是保存在 BookKeeper 中的。</p> <p>因为 Pulsar 的 broker 是无状态的, 所以如果工作负载很高, 就可以直接添加新的 broker。</p>	<p>Kafka 不是无状态的, 因为每个 broker 都包含了分区的所有日志, 如果一个 broker 宕机, 并非任意一 broker 都可以接替它的工作。如果工作负载太高, 也不能随意添加新的 broker 来分担。broker 之间必须进行状态同步。</p>

Pulsar使用

Pulsar安装&&启动（服务端）

参考链接: <https://pulsar.apache.org/docs/zh-CN/standalone/>

下载[Pulsar 2.8.0 binary release](#) (速度慢的话, 可以去上边的网页换个下载链接试试)

下载好压缩文件后, 解压缩并使用 `cd` 命令进入文件所在位置:

```
$ tar xvfz apache-pulsar-2.8.0-bin.tar.gz
$ cd apache-pulsar-2.8.0
```

使用 `pulsar` 命令启动本地集群 (该命令存储在 `bin` 目录中), 并且可以指定为以单机模式启动 Pulsar。

```
$ bin/pulsar standalone
```

如果启动失败, 可以尝试

```
$ bin/pulsar standalone -nss
```

成功启动 Pulsar 后, 可以看到类似如下所示的 `INFO` 级日志消息:

```
09:08:26.454 [main-EventThread] INFO
org.apache.bookkeeper.zookeeper.ZooKeeperWatcherBase - Zookeeper client is
connected now.
09:08:26.455 [main] INFO
org.apache.pulsar.broker.authentication.AuthenticationService - Authentication
is disabled
09:08:26.455 [main] INFO org.apache.pulsar.websocket.WebSocketService - Pulsar
WebSocket Service started
```

使用 `Ctrl+C` 终止单机模式 Pulsar 的运行。

Pulsar C++ 客户端安装

参考链接: <https://pulsar.apache.org/docs/zh-CN/client-libraries-cpp/>

系统要求

需要先安装下列组件才能使用 C++ 客户端:

- [CMake](#)
- [Boost](#)
- [Protocol Buffers](#) 2.6
- [libcurl](#)
- [Google Test](#)

编译

1. 克隆Pulsar项目仓库

```
$ git clone https://github.com/apache/pulsar
```

2. 安装所有必要的依赖项。

```
$ apt-get install cmake libssl-dev libcurl4-openssl-dev liblog4cxx-dev \
libprotobuf-dev protobuf-compiler libboost-all-dev google-mock libgtest-dev
libjsoncpp-dev
```

3. 编译并安装 [Google 测试](#)。

```
# libgtest-dev version is 1.18.0 or above
$ cd /usr/src/googletest
$ sudo cmake .
$ sudo make
$ sudo cp ./googletest/libgtest.a ./googletest/gtest/libgtest.a /usr/lib/

# less than 1.18.0
$ cd /usr/src/gtest
$ sudo cmake .
$ sudo make
$ sudo cp libgtest.a /usr/lib

$ cd /usr/src/gmock
$ sudo cmake .
```

```
$ sudo make
$ sudo cp libgmock.a /usr/lib
```

4. 在 Pulsar 仓库中编译 Pulsar C++ 客户端库。

```
$ cd pulsar-client-cpp
$ cmake .
$ make
```

安装组件成功后，文件 `libpulsar.so` 和 `libpulsar.a` 位于资源库的 `lib` 文件夹中。工具 `famerProducer` and `familConsumer` 在 `perf` 目录中。

客户端代码

Pulsar C++ 客户端安装完成后，可以编译客户端代码。

编译命令

```
$ g++ --std=c++11 /usr/lib/libpulsar.so -I /home/tianyu8867/pulsar-master/pulsar-client-cpp/ -g SampleProducerCapi.c -o sendccc
```

Producer示例（C语言）

```
#include <pulsar/c/client.h>
#include <pulsar/c/authentication.h>

#include <stdio.h>
#include <string.h>

//返回值为指定生产者发送到某个分区
int router_callback(pulsar_message_t *msg,
                   pulsar_topic_metadata_t *topicMetadata,
                   void *ctx)
{
    return 2;
}

int main(int argc, char **argv){
    if(argc != 3) {
        printf("input error!!!!!!!!!!!!!!!!!!!!\n");
        return -1;
    }
    //设置client conf, 内存限制
    pulsar_client_configuration_t *conf = pulsar_client_configuration_create();
    pulsar_client_configuration_set_memory_limit(conf, 64 * 1024 * 1024);

    //设置auth conf, token
    pulsar_authentication_t* auth_conf =
    pulsar_authentication_token_create("eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0LXVzZXIifQ.xZjctF7p5qGxLZ0nDVTkDzwEsi5L0iMkDeaziuGP8I");
    pulsar_client_configuration_set_auth(conf, auth_conf);

    pulsar_producer_configuration_t* producer_conf =
    pulsar_producer_configuration_create();
    void *ctx = NULL;
```



```

//设置批量，路由模式，路由分发函数
pulsar_producer_configuration_set_batching_enabled(producer_conf, 1);
pulsar_producer_configuration_set_partitions_routing_mode(producer_conf,
pulsar_CustomPartition);
pulsar_producer_configuration_set_message_router(producer_conf,
router_callback, ctx);

//创建连接client
pulsar_client_t *client = pulsar_client_create("pulsar://localhost:6650",
conf);

pulsar_producer_t *producer;

//创建producer，设置potic
pulsar_result err = pulsar_client_create_producer(client, argv[1],
producer_conf, &producer);
if (err != pulsar_result_ok) {
    printf("Failed to create producer: %s\n", pulsar_result_str(err));
    return 1;
}
char data[32];

for (int i = 0; i < 10; i++) {
    int size_t = sprintf(data, "%d : %s", i, argv[2]);
    data[size_t] = '\0';

    pulsar_message_t* message = pulsar_message_create();
    pulsar_message_set_content(message, data, strlen(data));

    //send
    err = pulsar_producer_send(producer, message);
    if (err == pulsar_result_ok) {
        printf("Sent message %d\n", i);
    } else {
        printf("Failed to publish message: %s\n", pulsar_result_str(err));
        return 1;
    }

    pulsar_message_free(message);
}
scanf("%s", data);
// cleanup
pulsar_producer_close(producer);
pulsar_producer_free(producer);
pulsar_producer_configuration_free(producer_conf);

pulsar_client_close(client);
pulsar_client_free(client);
pulsar_client_configuration_free(conf);
}

```

1. 生产者生产消息，test-topic为指定的topic，123为生产的消息

```
$ ./sendccc test-topic 123
```

Consumer示例 (C语言)

```
#include <stdio.h>
#include <pulsar/c/client.h>
#include <pulsar/c/authentication.h>

int main(int argc, char **argv){
    if(argc != 3) {
        printf("input error!!!!!!!!!!!!!!!!!!!!\n");
        return -1;
    }
    //argv[1]是topic, argv[2]是订阅主题

    //设置auth的token
    pulsar_client_configuration_t *conf = pulsar_client_configuration_create();
    pulsar_authentication_t* auth_conf =
    pulsar_authentication_token_create("eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0LXVzZXIifQ.xZjCtF7p5qGxLZonDvTRkDzwEsi5L0iMkDeaziuGP8I");
    pulsar_client_configuration_set_auth(conf, auth_conf);

    //生成client
    pulsar_client_t *client = pulsar_client_create("pulsar://localhost:6650",
    conf);

    //设置consumer订阅模式
    pulsar_consumer_configuration_t *consumer_conf =
    pulsar_consumer_configuration_create();
    pulsar_consumer_configuration_set_consumer_type(consumer_conf,
    pulsar_ConsumersShared);

    //设置consumer的topic和订阅主题
    pulsar_consumer_t *consumer;
    pulsar_result res = pulsar_client_subscribe(client, argv[1], argv[2],
    consumer_conf, &consumer);
    if (res != pulsar_result_ok) {
        printf("Failed to create subscribe to topic: %s\n",
    pulsar_result_str(res));
        return 1;
    }

    for (;;) {
        pulsar_message_t *message;
        //recv
        res = pulsar_consumer_receive(consumer, &message);
        if (res != pulsar_result_ok) {
            printf("Failed to receive message: %s\n", pulsar_result_str(res));
            return 1;
        }

        printf("Received message with payload: '%.*s'\n",
        pulsar_message_get_length(message),
        pulsar_message_get_data(message));

        pulsar_consumer_acknowledge(consumer, message);
        pulsar_message_free(message);
    }
}
```

```
// cleanup
pulsar_consumer_close(consumer);
pulsar_consumer_free(consumer);
pulsar_consumer_configuration_free(consumer_conf);

pulsar_client_close(client);
pulsar_client_free(client);
pulsar_client_configuration_free(conf);
}
```

1. 消费者消费消息，test-topic为指定的topic，my-subscription为订阅的主题

```
$ ./recvccc test-topic my-subscription
```

代码API

Producer和Consumer使用C语言连接Pulsar的方法，可以参考上面的示例。

1. client config的结构体，用于配置client

```
pulsar_client_configuration_t *pulsar_client_configuration_create();
void pulsar_client_configuration_free(pulsar_client_configuration_t *conf);
```

2. 设置client内存限制

```
/**
 * Configure a limit on the amount of memory that will be allocated by this
 * client instance.
 * Setting this to 0 will disable the limit. By default this is disabled.
 *
 * @param memoryLimitBytes the memory limit
 */
void pulsar_client_configuration_set_memory_limit(pulsar_client_configuration_t
*conf,
                                                    unsigned long long
memoryLimitBytes);
```

3. 设置client的认证模式，auth创建

```
pulsar_authentication_t *pulsar_authentication_token_create(const char *token);

/**
 * Set the authentication method to be used with the broker
 *
 * @param authentication the authentication data to use
 */
PULSAR_PUBLIC void
pulsar_client_configuration_set_auth(pulsar_client_configuration_t *conf,
                                      pulsar_authentication_t
*authentication);
```

4. producer config结构体，用于配置producer

```
PULSAR_PUBLIC pulsar_producer_configuration_t
*pulsar_producer_configuration_create();
PULSAR_PUBLIC void
pulsar_producer_configuration_free(pulsar_producer_configuration_t *conf);
```

4. 设置producer 批量模式, 路由模式

```
// Zero queue size feature will not be supported on consumer end if batching is
enabled
PULSAR_PUBLIC void
pulsar_producer_configuration_set_batching_enabled(pulsar_producer_configuration
_t *conf,
                                                    int
batchingEnabled);

PULSAR_PUBLIC void pulsar_producer_configuration_set_partitions_routing_mode(
    pulsar_producer_configuration_t *conf, pulsar_partitions_routing_mode mode);

PULSAR_PUBLIC void
pulsar_producer_configuration_set_message_router(pulsar_producer_configuration_t
*conf,

    pulsar_message_router router, void *ctx);
// pulsar_producer_configuration_set_message_router中的回调函数, 返回值为发送的分区ID
typedef int (*pulsar_message_router)(pulsar_message_t *msg,
    pulsar_topic_metadata_t *topicMetadata,
    void *ctx);
```

5. 使用client config创建client

```
/**
 * Create a Pulsar client object connecting to the specified cluster address and
 * using the specified
 * configuration.
 *
 * @param serviceUrl the Pulsar endpoint to use (eg: pulsar://broker-
 * example.com:6650)
 * @param clientConfiguration the client configuration to use
 */
PULSAR_PUBLIC pulsar_client_t *pulsar_client_create(const char *serviceUrl,
                                                    const
pulsar_client_configuration_t *clientConfiguration);
```

6. 使用client, producer config, topic, 创建producer

```
/**
 * Create a producer with default configuration
 *
 * @see createProducer(const std::string&, const ProducerConfiguration&,
 * Producer&)
 *
 * @param topic the topic where the new producer will publish
 * @param producer a non-const reference where the new producer will be copied
```

```

    * @return ResultOk if the producer has been successfully created
    * @return ResultError if there was an error
    */
PULSAR_PUBLIC pulsar_result pulsar_client_create_producer(pulsar_client_t
*client, const char *topic,
                                                    const
pulsar_producer_configuration_t *conf,
                                                    pulsar_producer_t
**producer);

```

7. 设置consumer订阅模式

```

/**
 * Specify the consumer type. The consumer type enables
 * specifying the type of subscription. In Exclusive subscription,
 * only a single consumer is allowed to attach to the subscription. Other
consumers
 * will get an error message. In Shared subscription, multiple consumers will be
 * able to use the same subscription name and the messages will be dispatched in
a
 * round robin fashion. In Failover subscription, a primary-failover
subscription model
 * allows for multiple consumers to attach to a single subscription, though only
one
 * of them will be “master” at a given time. Only the primary consumer will
receive
 * messages. When the primary consumer gets disconnected, one among the failover
 * consumers will be promoted to primary and will start getting messages.
 */
PULSAR_PUBLIC void pulsar_consumer_configuration_set_consumer_type(
    pulsar_consumer_configuration_t *consumer_configuration,
    pulsar_consumer_type consumerType);

```

8. 使用client, consumerconfig, topic, , 订阅主题, 创建consumer

```

PULSAR_PUBLIC pulsar_result pulsar_client_subscribe(pulsar_client_t *client,
const char *topic,
                                                    const char
*subscriptionName,
                                                    const
pulsar_consumer_configuration_t *conf,
                                                    pulsar_consumer_t
**consumer);

```

命令行API

Pulsar 提供了一些命令行工具，可以用于管理 Pulsar 的安装，测试性能，使用命令行生产者与消费者，等等。

所有的 Pulsar 命令行工具都可以从 [已安装的 Pulsar 包](#) 的 `bin` 目录里运行，下面是当前已有文档支持的工具：

- `pulsar`

- `pulsar-client`
- `pulsar-daemon`
- `pulsar-perf`
- `bookkeeper`
- `broker-tool`

Token 认证概述

Pulsar 支持通过加密 Token 的方式认证客户端，它是基于[JSON Web Tokens \(RFC-7519\)](#) 来实现的。

你可以根据 Token 来识别 Pulsar 客户端。通过给 Token 关联一些 "principal" 或 "role"，可以用来授权客户端允许执行某些操作 (例如：发布消息主题或从主题消费消息)。

用户通常需要从管理员 (或某些自动化服务) 获得 Token 字符串。

如下是一个简短的 JWT 签名字符串：

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJKb2UifQ.ipevRNuRP6Hf1G8cFKnmUPtypruRC4fb1DWtoLL62SY
```

应用程序需要在创建客户端实例时，指定所使用的 Token。一个替代方法是通过“令牌供应商”(即通过一个函数返回所需要的Token)。

输入下面的命令可以生成 base64 编码的密钥。

```
$ bin/pulsar tokens create-secret-key --output /opt/my-secret.key --base64
```

创建密钥对

若需要使用公钥和私钥方式，需要先创建一对密钥。Pulsar 支持 Java JWT 库 ([这里](#)) 支持的所有算法。

默认在 Pulsar 安装目录的根目录生成输出文件。您也可以使用下面的命令为输出文件指定绝对路径。

```
$ bin/pulsar tokens create-key-pair --output-private-key my-private.key --output-public-key my-public.key
```

- 保存 `my-private.key` 到一个安全的位置，只有管理员能使用 `my-private.key` 来生成新的 tokens。
- `my-public.key` 需要分发给所有 Pulsar brokers。您可以公开共享此文件，不用担心安全问题。

生成 Token

A token is the credential associated with a user. 连接是通过 "principal" 或 "role" 完成的。对于 JWT tokens，这个字段通常被称为 **subject**，尽管它们是完全相同的概念。

接下来，您需要使用这个命令来要求生成的具有一个 **subject** 字段的 token。

```
$ bin/pulsar tokens create --secret-key file:///path/to/my-secret.key \
--subject test-user
```

命令将在 stdout 上打印出 token 字符串。

类似地，您可以通过下面的命令使用 "private" key 来创建一个 token。

```
$ bin/pulsar tokens create --private-key file:///path/to/my-private.key \
    --subject test-user
```

最后，你可以通过以下命令来创建一个带有过期时间的 Token。Token 到期后会自动失效。

```
$ bin/pulsar tokens create --secret-key file:///path/to/my-secret.key \
    --subject test-user \
    --expiry-time 1y
```

授权

Token 本身没有关联任何权限。授权引擎确定 token 是否有权限。一旦创建 token，就可以授予该 Token 执行某些操作的权限。The following is an example.

```
$ bin/pulsar-admin namespaces grant-permission my-tenant/my-namespace \
    --role test-user \
    --actions produce,consume
```

启用 token 认证

要让 brokers 能对客户端进行身份认证，请在 `conf/broker.conf` 中添加以下参数：

PS：如果是单机模式启动，需要改的是 `standalone.conf`

```
authenticateOriginalAuthData=true
authenticationEnabled=true
authorizationEnabled=true
authenticationProviders=org.apache.pulsar.broker.authentication.AuthenticationPr
oviderToken
superUserRoles=test-user
brokerClientAuthenticationPlugin=org.apache.pulsar.client.impl.auth.Authenticati
onToken
brokerClientAuthenticationParameters=token:eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0
LXVzZXIifQ.xZjCtF7p5qGxLZOnDvTRkDzwEsi5L0iMkDeaziuGP8I
tokenSecretKey=file:///mnt/hgfs/share/apache-pulsar-2.8.0/my-secret.key
```

分区主题

1. 在 `toolset` 容器里，创建一个名为 `apache` 的租户。

```
bin/pulsar-admin tenants create apache
```

2. 然后你可以列出所有租户，来查看租户是否创建成功。

```
bin/pulsar-admin tenants list
```

你应该能看到类似如下的输出：租户 `apache` 已经被成功地创建了。

```
"apache"
"public"
"pulsar"
```


3. 在 `toolset` 容器里，在租户 `apache` 下创建一个名为 `pulsar` 的命名空间。

```
bin/pulsar-admin namespaces create apache/pulsar
```

然后你可以列出租户 `apache` 的所有命名空间，来查看命名空间是否创建成功。

```
bin/pulsar-admin namespaces list apache
```

你应该能看到类似如下的输出：命名空间 `apache/pulsar` 已经被成功地创建了。

```
"apache/pulsar"
```

4. 在命名空间 `public/default` 下创建一个包含 4 个分区的话题 `test-topic`

```
bin/pulsar-admin topics create-partitioned-topic public/default/test-topic -  
p 4
```

5. 在 `toolset` 容器里，列出命名空间 `public/default` 中的所有分区主题

```
bin/pulsar-admin topics list-partitioned-topics public/default
```

然后你可以看见命名空间 `public/default` 下的所有分区主题

```
"persistent://public/default/test-topic"
```

查看分区信息

1. 列出topic

```
bin/pulsar-admin topics list public/default
```

2. 查看topic状态

```
bin/pulsar-admin topics stats persistent://public/default/test-topic
```

Pulsar stats字段信息

分区 topic

统计信息	说明
msgRateIn	所有本地 producer 和复制 producer 的发布频率总和，以消息数/秒为单位。
msgThroughputIn	与 msgRateIn 相同，但以字节/秒作为单位。
msgRateOut	所有本地 consumer 和复制 consumer 的调度频率总和，以消息数/秒为单位。
msgThroughputOut	与 msgRateOut 相同，但以字节/秒为单位。
averageMsgSize	Producer 在最后一段时间间隔内发布消息的平均大小，以字节为单位。
storageSize	Topic 中所有 ledger 存储大小的总和。
publishers	Topic 中所有本地 producer 的列表，数量从零到几千不等。
producerId	Topic 上 producer 的内部标识符。
producerName	由客户端库生成的 producer 的内部标识符，。
address	Producer 连接的 IP 地址和 source 端口。
connectedSince	Producer 创建或最后一次重新连接的时间戳。
subscriptions	Topic 所有本地订阅的列表。
my-subscription	订阅名称（由客户端定义）。
msgBacklog	订阅中积压消息的数量。
type	订阅的类型。
msgRateExpired	由于 TTL，订阅中丢弃消息，而非调度消息的频率。
consumers	订阅已连接 consumer 的列表。
consumerName	由客户端库生成的 consumer 的内部标识符。
availablePermits	Consumer 在客户端监听队列中所能容纳的最大消息数。值为 0 意味着客户端库队列已满，receive() 不再被调用。非零值意味着 consumer 可以接收消息。
replication	Topic 上 cross-colo 复制的统计信息。
replicationBacklog	待复制到远程集群的消息数。
connected	出站复制器是否已连接。
replicationDelayInSeconds	已连接状态下，若通过此连接发送最旧的消息，则此消息已经等待的时间。
inboundConnection	远程集群到此集群 producer 连接中，broker 的 IP 和端口。
inboundConnectedSince	正在向远程集群发布消息的 TCP 连接。如果本地 producer 没有连接，则此连接在一分钟后自动关闭。

Topic

统计信息	说明
entriesAddedCounter	此 broker 加载此 topic 后发布的消息。
numberOfEntries	正在追踪的消息总数。
totalSize	所有消息存储大小的总和（以字节为单位）。
currentLedgerEntries	当前开放写入的 ledger 中，写入消息的总数。
currentLedgerSize	当前开放写入的 ledger 中，写入消息的大小（以字节为单位）。
lastLedgerCreatedTimestamp	上一个 ledger 的创建时间。
lastLedgerCreationFailureTimestamp	上一次 ledger 创建失败的时间。
waitingCursorsCount	跟上消息发布的进度，正在等待新消息发布的游标数量。
pendingAddEntriesCount	正在等待完成（异步）写入请求的消息数。
lastConfirmedEntry	Ledgerid：最后一条成功写入的消息的 entryid。如果 entryid 值为 -1，则此 ledger 已开启或正在开启，但 ledger 中没有已写入的 entry。
state	游标 ledger 的状态。Open 表示有一个游标 ledger，可用于存储 markDeletePosition 的更新值。
ledgers	Topic 上用于保存消息的所有 ledger 的有序列表。
cursors	Topic 上所有游标的列表。Topic 统计信息中的每个订阅都对应一个此列表。
markDeletePosition	The ack position：订阅者确认的最后一条消息。
readPosition	订阅者最后读取消息的位置。
waitingReadOp	若订阅已读取发布到 topic 的最新消息，并且正在等待新消息的发布，则值为 true。
pendingReadOps	BookKeeper 中正在处理的读取请求数量计数。
messagesConsumedCounter	Broker 加载 topic 后，此游标已确认的消息总数。
cursorLedger	用于持久存储当前 markDeletePosition 的 ledger。
cursorLedgerLastEntry	最后一个用于持久存储当前 markDeletePosition 的 entryid。
individuallyDeletedMessages	如果消息被乱序确认，则显示在 markDeletePosition 和 read-position 之间已确认消息的范围。
lastLedgerSwitchTimestamp	游标 ledger 上一次轮询的时间。

