1. **绪 论**
   1. **研究背景与意义**

随着信息化的高速发展，出现了很多互联网公司。互联网公司的主要用户群体是终端客户，因此“以客户为中心”的理念也随之得到的越来越广泛的认同。很多互联网公司都关注客户在企业经营活动中的体验和感受。因此，面向客户的系统或者说C端（Customer）系统（如，网上商城系统，支付系统，订单系统，物理系统等）也越来越受到互联网公司的关注和重视。

然而，在这些面向客户的系统建设和运营过程中，有大量信息（如，购物信息，订单信息，支付信息，物流信息，评价信息）产生，这些信息一般被作为消息在各个系统和客户中进行交互和流转。如果各个系统单独开发消息通道（如，短信通道，推动通道）进行对接，需要花费很大的开发成本，并且系统间在推送消息的功能和性能方便都需要经过严格测试，使得系统的建设周期，研发成本和效率不能满足既定的要求。并且，系统直接直接对接，造成了系统间的紧耦合，造成系统的可扩展性，可用性和容错性降低。所以，如果建设一个统一的，通用的，安全且扩展性高的消息推送系统是值得研究的，这也对企业信息化建设，特别是互联网公司的面向客户的业务系统建设具有重要的意义。

* 1. **国内外研究现状**

目前，国内外消息系统的研究和实践，可以从理论和实践两个方面来进行深入分析。

**1.2.1国内外理论研究现状**

从理论上还说，消息系统属于MOM（Message-oriented middleware）的范畴。目前，主要的业界标准有：

1. JMS（Java Message Service）

JMS用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。JMS务是一个与具体平台无关的API，绝大多数MOM提供商都对JMS提供支持。JMS是一个在 Java标准化组织（JCP）内开发的标准（代号JSR 914）。2001年6月25日，Java消息服务发布JMS 1.0.2b，2002年3月18日Java消息服务发布 1.1，统一了消息域。目前JMS最新规范为2.0版本。

1. AMQP（Advanced Message Queueing Protocol）

AMQP是一个提供统一消息服务的应用层标准高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同的开发语言等条件的限制。

1. DDS（Data Distribution Service）

DDS是对象管理组织(OMG)在HLA及CORBA等标准的基础上制定的新一代分布式实时通信中间件技术规范，DDS采用发布/订阅体系架构，强调以数据为中心，提供丰富的QoS服务质量策略，能保障数据进行实时、高效、灵活地分发，可满足各种分布式实时通信应用需求。DDS信息分发中间件是一种轻便的、能够提供实时信息传送的中间件技术。

1. MQTT（Message Queuing Telemetry Transport）

MQTT是一种基于发布/订阅（publish/subscribe）模式的“轻量级”通讯协议，该协议构建于TCP/IP协议上，由IBM在1999年发布。MQTT最大优点在于，可以以极少的代码和有限的带宽，为连接远程设备提供实时可靠的消息服务。作为一种低开销、低带宽占用的即时通讯协议，使其在物联网、小型设备、移动应用等方面有较广泛的应用。

**1.2.2国内外应用研究现状**

从应用方面来说，消息系统相关的产品主要有以下几种类型：

1. 消息中间件厂商

目前大部分消息中间件都是基于JMS或者AMQP规范而实现的。比较著名的产品有Apache ActiveMQ，Apache Qpid，Apache RocketMQ，RabbitMQ等等。

1. 消息发送通道提供商

基于不同的消息类型，大致分为：电话语音通道，短信通道，彩信通道，邮件通道，微信通道，APP推送通道等等。

综上所述，消息系统的实际上并不是一套单一的统一的系统。不同类型的企业对其消息系统的定义也各不相同。因此，对以上各种平台和技术进行深入的研究和学习，设计出对企业业务发展有利用价值的消息系统，是具有非常重要的意义的。

* 1. **论文研究目标及主要内容**

结合上述理论与应用的现状，本课题主要基于JMS规范及其实现，研究并设计实现一个统一的，标准的，可靠的，易扩展且易维护的消息系统，为企业系统之间，企业与用户之间的消息传递提供安全的服务。

主要研究内容分为以下几个方面：

1. 消息系统的核心机制及关键组件

消息系统主要基于生产者-消费者设计模式，将企业系统与消息通道进行解耦，使得消息传递模型从原本通过直连方式进行消息传递变成统一经过消息系统进行消息传递，从而降低各个业务系统的复杂性和耦合性，提高系统对接效率，增强系统的可靠性和可维护性。

核心组件主要包括：消息任务管理模块，消息监控模块，消息存储模块，消息日志管理模块，配置管理模块，消息队列（Queue）和主题（Topic）。

1. 统一的对外服务接口API

对外服务接口位于消息系统的接口层，以一套标准的API接口提供给服务调用方，主要包括：创建消息，查询消息，更新消息，存储消息，消费消息，失败重试，批量消息，日志查询，状态监控等接口。

通过使用标准的对外服务接口，服务调用方只需要关注自己的业务逻辑，而不需要关注消息系统底层实现细节，进一步降低了业务开发的难度，更好的实现了系统间的解耦合。

1. 可维护的异常消息

通过标准的异常处理模块，将因为网络分区或者机器故障导致的传输异常消息保存到指定区域，通过可配置的重试策略将异常消息重新处理，最大限度保证消息的可靠传输。

* 1. **论文组织结构及章节安排**

本文以当前企业各种子系统中消息传递所遇到的实际问题和面临的挑战为出发点，通过对国内外相关理论和时间的研究和对比分析，提出了一个以消息系统作为中心的，业务系统围绕消息系统进行通信的架构。并且基于JMS规范，对其进行需求梳理，需求分析，总体架构，功能性设计和非功能性设计，关键技术实现，功能测试，性能测试等一系列工作。主要章节安排如下：

//TODO(未完待定~~~~~~先写下面的具体实现部分，在补充文章章节安排)

1. **消息系统的相关理论和技术研究**

随着企业信息化的不断发展，为了满足企业系统之间消息的可靠传输，简单的业务子系统之间进行消息交互已经不能满足企业系统建设过程中对成本控制，质量控制，可靠性保障和快速迭代的需求。因此，建设统一的，可靠的消息系统对于企业信息化建设来说迫在眉睫。通过统一的消息系统，将企业的业务系统进行解耦，以增强业务系统的通用性，降低其开发成本。本章节对消息系统所涉及的相关理论和技术进行综述，为了接下来基于JMS的消息系统的架构设计和实现奠定了理论基础和技术支持。

**2.1中间件相关简介**

中间件是一种独立于操作系统的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源。中间件位于客户机/服务器的操作系统之上，管理计算机资源和网络通讯。相连接的系统，即使它们具有不同的接口，但通过中间件相互之间仍能交换信息。执行中间件的一个关键途径是信息传递。通过中间件，应用程序可以工作于多平台或OS环境。

**2.1.1中间件的分类**

中间件大约分为以下几类：

1. 远程过程调用(Remote Procedure Call)中间件

远程过程调用是一种广泛使用于分布式系统中的处理方法。一个应用程序使用RPC来“远程”执行一个位于不同地址空间里的函数/过程，并且这种调用方式从效果上看和执行本地调用没有区别。事实上，任何RPC应用都至少可以分为两个部分：Server和Client。Server提供一个或多个远程计算函数/过程；Client向Server发出远程调用。Server和Client可以位于同一台计算机，也可以位于不同的计算机，甚至运行在不同的操作系统之上。Server和Client通过网络进行通讯。相应的stub和运行支持提供数据转换和通讯服务，从而屏蔽不同的操作系统和网络协议。

1. 对象请求代理(Object Request Broker)中间件

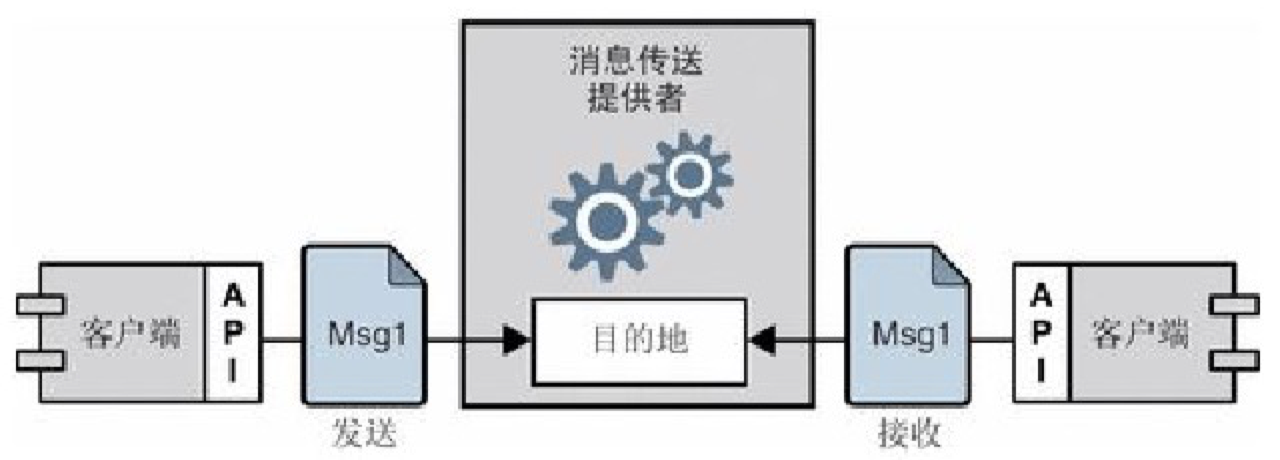
ORB使得对象可以透明地向其他对象发出请求或接受其他对象的响应，这些对象可以位于本地也可以位于远程机器。ORB与RPC的最主要区别在于，RPC是面向过程的远程调用，而ORB是面向对象的远程调用。ORB拦截请求调用，并负责找到可以实现请求的对象、传送参数、调用相应的方法、返回结果等。常见的ORB中间件有：CORBA，RMI/IIOP等等。

1. 事务处理监控（Transaction processing monitors）中间件

事务处理监控（Transaction processing monitors）最早出现在大型机上，为大型机提供支持大规模事务处理的可靠运行环境。随着分布计算技术的发展，分布式系统对大规模的事务处理提出了需求，比如商业活动中大量的关键事务处理。事务处理监控界于Client和Server之间，进行事务管理与协调、负载平衡、失败恢复等，以提高系统的整体性能。它可以被看作是事务处理应用程序的“操作系统”。

1. 面向消息(Message-Oriented Middleware)中间件

面向消息的中间件使得分布式系统可以不必直接互连，而是通过发送和接收消息来进行通信和数据交换。面向消息的中间件与RPC和ORB的主要区别在于，通信程序可以在不同的时间运行，对于通信程序没有特定的结构约束。典型的MOM系统架构模型如下图所示：



**2.1.2主流的面向消息中间件的标准和规范**

消息系统是基于MOM（Message-oriented middleware）构建的基础服务系统，目前，主要的MOM工业标准和规范如下表2-1所示。

表2-1面向消息的中间件的标准和规范对比

Table 2-1 The Difference Between MOM Standards

|  |  |  |
| --- | --- | --- |
|  | **JMS** | **AMQP** |
| 定义 | JMS（Java Message Service）即Java消息服务应用程序接口，是一个Java平台中关于面向消息中间件（MOM）的API，用于在多个应用程序之间，或分布式系统中发送消息，进行异步通信。从而达到系统接口的目的。Java消息服务是一个与具体平台无关的API，很多MOM提供商都对JMS规范提供支持。 | AMQP（Advanced Message Queuing Protocol）是一个提供统一消息服务的应用层标准高级消息队列协议。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同的开发语言等条件的限制。 |
| 开放性 | JavaEE平台的一部分 | 应用层开放协议 |
| 厂商支持 | 大部分国际厂商支持，并开发出工业界为人熟知的产品，如：Apache ActiveMQ，JBoss MQ等等 | 逐渐被国际厂商支持，并开发出一系列的产品，如：Apache Qpid，RabbitMQ |

**2.1.3ActiveMQ架构模型**

Java消息服务（Java Message Service，JMS）是一套可以跨平台的API，其是JavaEE规范的一部分。JMS与我们熟知的JDBC类似，是一套独立于厂商标准规范。ActiveMQ就是实现JMS规范的典型代表。下面介绍ActiveMQ的架构模型。

（1）消息传递模型

JMS提供2种消息传递模型，分别是：

**点对点（Point-to-Point，P2P）模型：**

在点对点模型下，ActiveMQ提供一个消息队列（Queue）的虚拟消息通道，让客户端以同步或者异步的方式来收发消息。每条消息只能被一个消费者接收。ActiveMQ点对点模型如下图2-1所示：



图2-1 ActiveMQ点对点模型

Figure 2-1 ActiveMQ Point-to-Point Model

**发布-订阅（Publish-Subscribe，Pub/Sub）模型：**

在发布-订阅模型下，ActiveMQ提供主题（Topic）的虚拟通道，讲消息发送给多个订阅该主题的消息接受者。一个消息可以被多个消费者接收。ActiveMQ发布-订阅模型如下图2-2所示：

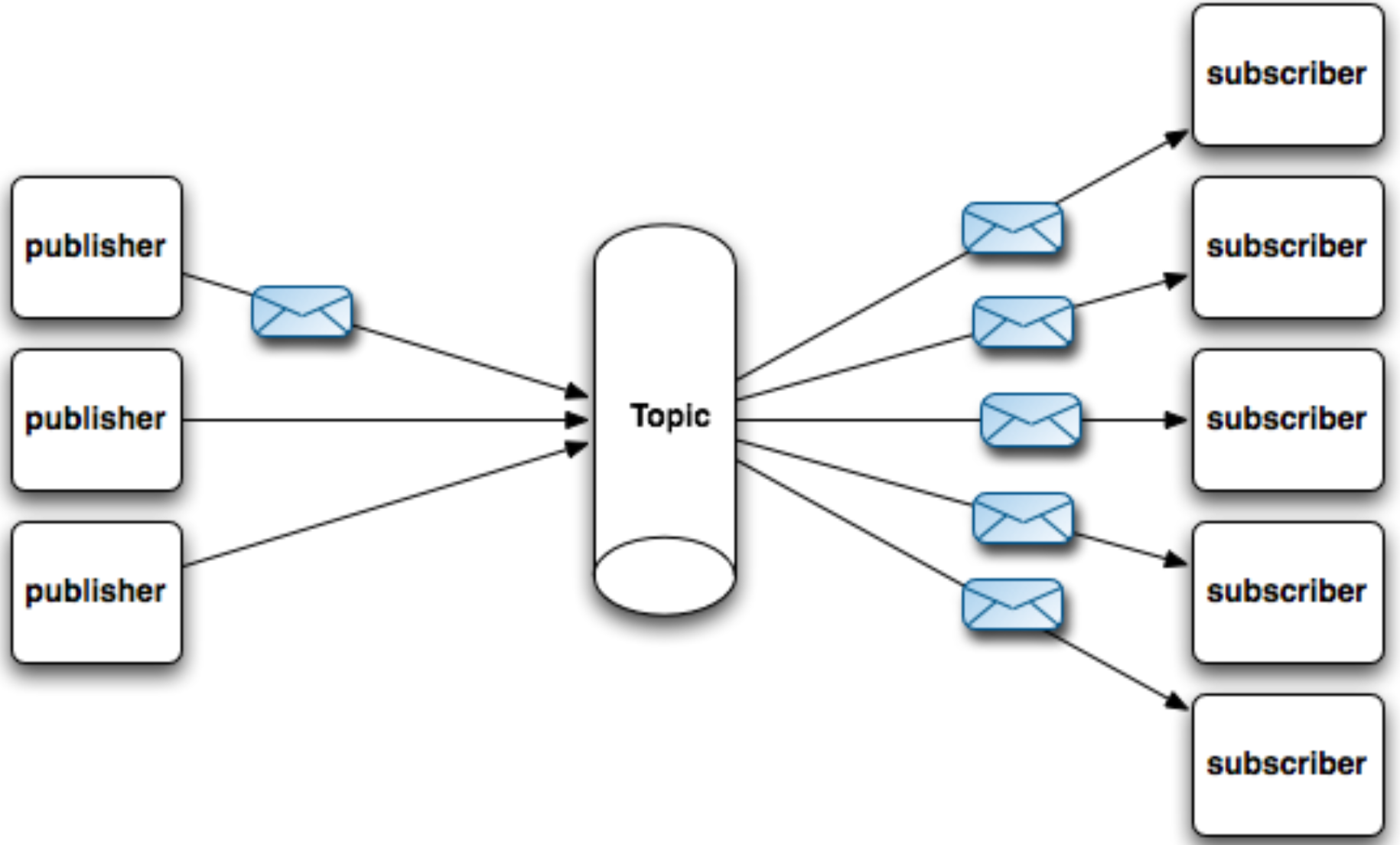


图2-2 ActiveMQ发布-订阅模型

Figure 2-2 ActiveMQ Pub/Sub Model

（2）Point-to-Point模型与Pub/Sub模型的区别

Point-to-Point模型与Pub/Sub模型虽然都是ActiveMQ支持的消息传递模型，但是这两者在许多方面存在不同，理解这两种模型的区别对于我们在实际应用中选择合适的模型解决实际问题有指导意义。两者主要区别归纳如下表2-2：

表2-2Point-to-Point模型与Pub/Sub区别

Tabel 2-2 The Difference Between Point-to-Point And Pub/Sub

|  |  |  |
| --- | --- | --- |
|  | Point-to-Point | Pub/Sub |
| 消息通道 | Queue | Topic |
| 消息发送模式 | 一条消息只能被一个消费者接收到 | 一条消息可以被多个消费者接收到 |
| 持久化 | Queue中的消息消费者不在线，消息会被持久化在Queue中，不会丢失。 | 分为持久订阅和非持久订阅  持久订阅是即使订阅此主题的消费者不在线，也会保存消息直到消费者上线。  非持久订阅指消费者不在线时，不会为其保存消息。 |
| 消息完整性 | 保证消息会收到 | 不保证消息一定被接收 |

**2.2分布式协调服务简介**

分布式协调技术主要用来解决分布式环境当中多个进程之间的同步控制，让进程间有序的访问某种临界资源，防止造成"脏数据"的后果。作为一个分布式系统，分区容错性是一个必须要考虑的关键点。一个分布式系统一旦丧失了分区容错性，也就表示放弃了扩展性。因为在分布式系统中，网络故障是经常出现的，一旦出现在这种问题就会导致整个系统不可用是绝对不能容忍的。所以，大部分分布式系统都会在保证分区容错性的前提下在一致性和可用性之间做权衡。

分布式协调服务是一个为分布式应用提供一致性协调服务的服务型软件。提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

**2.2.1分布式协调服务的分类**

目前工业界主要的分布式协调服务可以分为以下两种：

1. Zookeeper

ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，是Google的Chubby一个开源的实现，是Hadoop和Hbase的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。ZooKeeper的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

1. etcd

etcd是一个高可用的键值存储系统，主要用于共享配置和服务发现。etcd是由CoreOS开发并维护的，灵感来自于 ZooKeeper 和 Doozer，它使用Go语言编写，并通过Raft一致性算法处理日志复制以保证强一致性。Raft是一个来自Stanford的新的一致性算法，适用于分布式系统的日志复制，Raft通过选举的方式来实现一致性，在Raft中，任何一个节点都可能成为Leader。Google的容器集群管理系统Kubernetes、开源PaaS平台Cloud Foundry和CoreOS的Fleet都广泛使用了etcd。

1. Eureka

Eureka是一个基于REST的服务，主要用于定位运行在AWS域中的中间层服务，以达到负载均衡和中间层服务故障转移的目的。Spring Cloud将它集成在其子项目spring-cloud-netflix中，以实现Spring Cloud的服务发现功能。

1. Zookeeper与etcd的区别

虽然目前工业界主要的协调服务有Zookeeper，etcd和Eureka，但是这3种服务有着本质上的区别，二者具体区别如下表2-3所示：

表2-3 Zookeeper与etcd区别

Tabel 2-2 The Difference Between Zookeeper And etcd

|  |  |  |  |
| --- | --- | --- | --- |
|  | Zookeeper | etcd | Eureka |
| 服务的目的 | Zookeeper是一个具有高效率和高可靠性的分布式协同工作系统。Zookeeper更加关注的是CP（Consistency一致性，Partition tolerance分区容错性） | etcd是一个具有高可用性的 Key/Value 存储系统，主要用于分享配置和服务发现。 | Eureka是一个具有高可用性的服务发现框架。相比于Zookeeper，Eureka更加关注的是AP(Availability可用性，Partition tolerance分区容错性) |

续表2-3

|  |  |  |  |
| --- | --- | --- | --- |
|  | Zookeeper | Etcd | Eureka |
| 接口 | Zookeeper基于TCP的自协议，需要安装相应客户端程序。 | etcd提供了REST接口（HTTP+JSON）方便集群中每一个主机访问 | Eureka提供REST接口，方便第三方程序集成Eureka的功能 |
| 功能 | 提供了key，value存储服务，集群中建立临时节点，观察key值变化等。 | key，value存储服务，集群队列同步服务，观察一个key的数值变化，以及查询历史key值信息等。 | Eureka服务注册发现功能，客户端缓存功能 |
| 分布式协议 | 基于Paxos一致性协议，改进的Zab协议。提供强一致性保证。 | 基于Raft一致性协议。提供强一致性保证。 | 允许一段时间内数据不一致 |
| 服务健康检查 | (弱)长连接，keepalive | 连接心跳 | 可配支持 |
| 实现语言 | java，实现代码量要多于go，在小规模集群中性能一般，但是在大规模情况下，使用对多线程的优化后，也和go相差不大。 | go 拥有几乎不输于C的效率，特别是go语言本身就是面向多线程，进程通信的语言。在小规模集群中性能非常突出。 | Java Servlet |
| KV存储 | 支持 | 支持 | 不支持 |
| 自身监控 | 不支持 | metrics | Metrics |

**2.2.2 Zookeeper的功能和特性分析**

ZooKeeper是一种为分布式应用所设计的高可用、高性能且一致的开源协调服务，它提供了一项基本服务：分布式锁服务。由于ZooKeeper的开源特性，后来我们的开发者在分布式锁的基础上，摸索了出了其他的使用方法：配置维护、组服务、分布式消息队列、分布式通知/协调等。

ZooKeeper性能上的特点决定了它能够用在大型的、分布式的系统当中。从可靠性方面来说，它并不会因为一个节点的错误而崩溃。除此之外，它严格的序列访问控制意味着复杂的控制原语可以应用在客户端上。ZooKeeper在一致性、可用性、容错性的保证，也是ZooKeeper的成功之处。

1. 命名服务

在Zookeeper的文件系统里创建一个目录，即有唯一的path。在我们无法确定上游程序的部署机器时即可与下游程序约定好path，通过path即能互相探索发现，不见不散了。典型的RPC框架dubbo就是通过将注册的服务通过path暴露给调用方的。

1. 配置管理

分布式系统中总是需要许多配置的，如果程序分散部署在多台机器上，要逐个改变配置就变得困难。如果把这些配置全部放到Zookeeper上去，保存在 Zookeeper 的某个目录节点中，然后所有相关应用程序对这个目录节点进行监听，一旦配置信息发生变化，每个应用程序就会收到 Zookeeper 的通知，然后从 Zookeeper 获取最新的配置信息应用到系统中就可以了。Zookeeper配置管理如下图2-3所示：

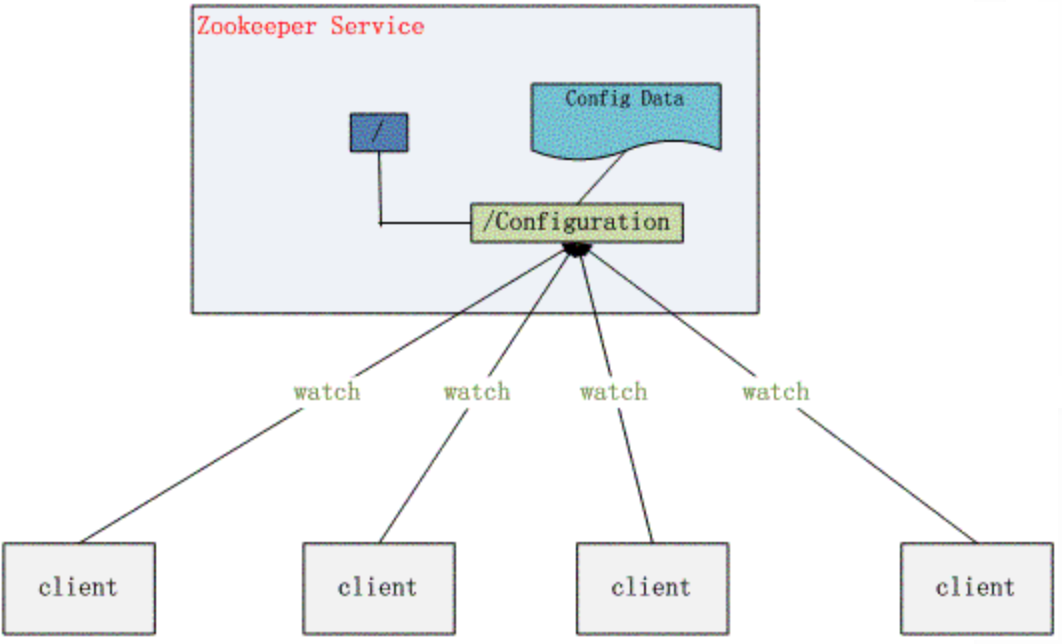


图2-3 Zookeeper配置管理

Figure 2-3 Zookeeper Configuration Management

1. 集群管理

集群管理主要在于两点：是否有机器退出和加入、选举master。

对于第一点，所有机器约定在父目录GroupMembers下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器宕机，该机器与 zookeeper的连接断开，其所创建的临时目录节点被删除，所有其他机器都收到通知：某个兄弟目录被删除，于是所有子节点都会知道。当有新机器加入的场景也是类似，所有机器收到通知：有新兄弟目录加入了。

对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为master就好。

Zookeeper分布式集群管理如下图2-4所示：

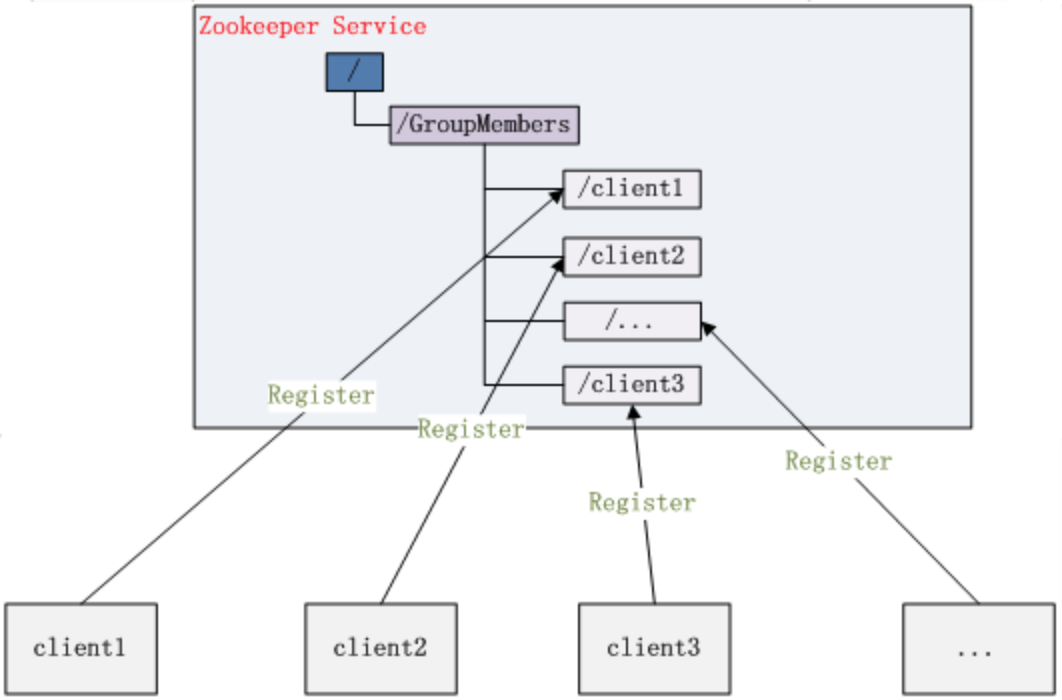


图2-4 Zookeeper集群管理

Figure 2-3 Zookeeper Cluster Management

1. 分布式锁

Zookeeper锁服务可以分为两类，一类是保持独占的独占锁，另一个是控制时序时序锁。

对于第一类，我们将Zookeeper上的一个znode看作是一把锁，通过createznode的方式来实现。所有客户端都去创建 /distribute\_lock 节点，最终成功创建的那个客户端也即拥有了这把锁，用完删除掉自己创建的distribute\_lock 节点就释放出锁。所有客户端都依照这个规则依次进行，从而就达到了独占的目的。

对于第二类， /distribute\_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选master一样，每次只有编号最小的才能获得锁，用完之后将改临时编号目录节点删除，让出控制权给改节点序号之后的下一个节点获得控制权。

Zookeeper分布式锁如下图2-5所示：

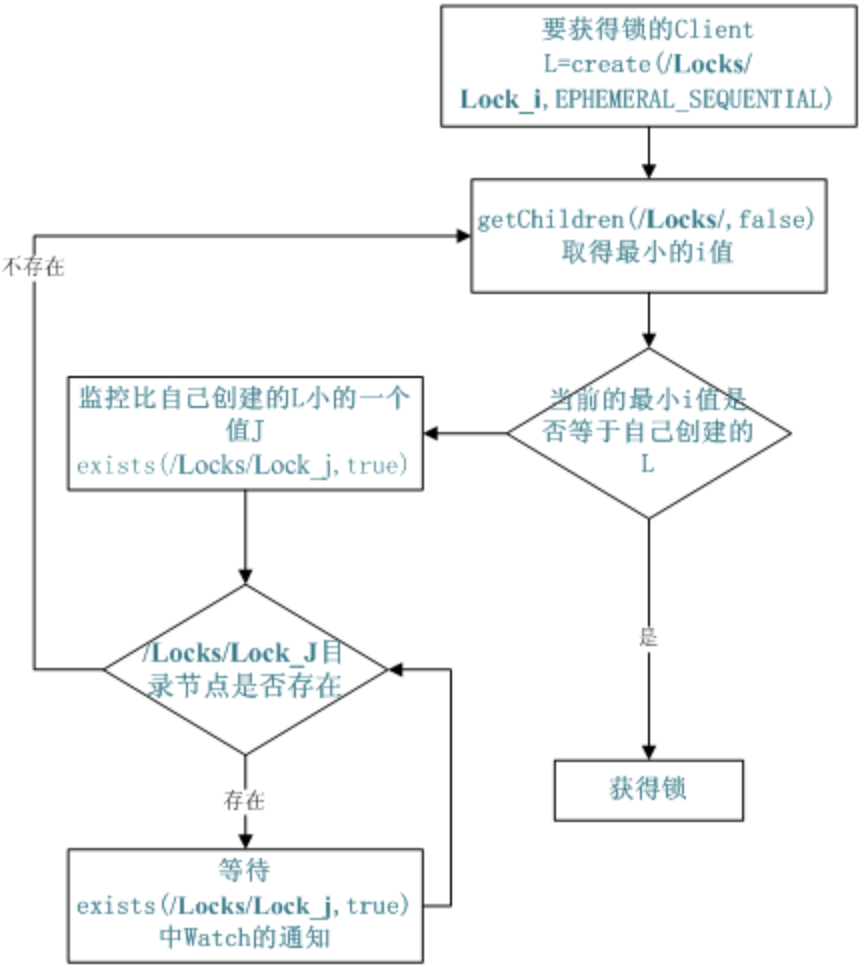


图2-5 Zookeeper分布式锁管理

Figure 2-3 Zookeeper Distributed Lock Management

1. 队列管理

Zookeeper队列管理可以维护两种类型的队列：

第一类，同步队列，当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。

第二类，按照 FIFO 方式进行入队和出队操作的队列。和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。

了解了Zookeeper的功能和特性后，对于我们选择合适的分布式协调服务具有很好的指导意义。并且对消息系统的构建和技术选型提供了一定的理论基础和技术支持。

**2.3缓存服务简介**

**2.3.1 缓存服务的分类**

硬件缓存就是数据交换的缓冲区（称作Cache），当某一硬件要读取数据时，会首先从缓存中查找需要的数据，如果找到了则直接执行，找不到的话则从内存中找。由于缓存的运行速度比内存快得多，故缓存的作用就是帮助硬件更快地运行。因为缓存往往使用的是RAM（断电即掉的非永久储存），所以在用完后还是会把文件送到硬盘等存储器里永久存储。电脑里最大的缓存就是内存条了，最快的是CPU上镶的L1和L2缓存，显卡的显存是给显卡运算芯片用的缓存，硬盘上也有16M或者32M的缓存。

在软件领域也有许多缓存服务，其目的是为了加快访问速度，保证数据一致性。比如Oracle、MySQL等数据库，数据都是存放在磁盘中。那么在高并发场景下频繁对数据库产生的增、删、查、改的操作会给服务器带来庞大IO压力。所以缓存技术在此诞生，实现热点数据的高速缓存，提高应用的响应速度，极大缓解后端数据库的压力。

工业领域目前常见的缓存服务有：

1. Ehcache

Ehcache是在Java应用中广泛使用到的一个开源的缓存服务。Ehcache配置简单，依赖的服务非常少（核心程序仅仅依赖slf4j.jar），具有很强的扩展性。它是一个开源的、设计于提高在数据从RDBMS中取出来的高花费、高延迟采取的一种缓存方案。正因为Ehcache具有健壮性（基于java开发）、被认证（具有apache 2.0 license）所以被用于大型复杂分布式系统的各个节点中。

1. Memcache

Memcache 是一种高性能、分布式对象缓存系统，最初设计于缓解动态网站数据库加载数据的延迟性，其本质上就是一个key-value键值缓存，所有数据都保存在内存中，重启Memcached或者重启操作系统会导致其中保存的全部数据丢失。Danga Interactive为了LiveJournal所发展的，以BSD license释放的一套开放源代码软件。

Memcache是C语言所编写，依赖于最近版本的GCC和libevent。GCC是它的编译器，同事基于libevent做socket io。在安装Memcache时需要保证系统同时具备有这两个环境。

Memcache支持多个cpu同时工作，支持多线程。通过libevent完成socket的通讯，理论上性能的瓶颈落在网卡上。

1. Redis

Redis是一个开源的使用ANSI C语言编写、支持网络、可基于内存也可以持久化的日志型、Key-Value数据库，并提供多种语言的API。

Redis 支持 String 、List、Set、Sorted Set、Hash 多种数据类型。丰富的数据类型深受开发者欢迎。

Redis的本地持久化支持两种方式：RDB和AOF。RDB 以内存快照的方式持久化数据。AOF是将Redis每一条更新操作都记录下来（保存的是这条记录的生成命令），如果不使用Redis做内存数据库的话，一般不需要开启AOF功能。

Redis提供主从复制方案。主从通过增量复制，复制的是新增记录命令，主库新增记录将新增脚本发送给从库，从库根据脚本生成记录，这个过程非常快，一般主从都是在同一个局域网，所以可以说Redis的主从近似及时同步，同时Redis还支持一主多从，动态添加从库，从库数量没有限制。

1. MongoDB

MongoDB是一个基于分布式文件存储的数据库。由C++语言编写。旨在为应用提供可扩展的高性能数据存储解决方案。

MongoDB做为内存型数据库，数据操作会先写入内存，然后再会持久化到硬盘中去。MongoDB在启动时，专门初始化一个线程不断循环（除非应用crash掉），用于在一定时间周期内来从defer队列中获取要持久化的数据并写入到磁盘的journal(日志)和mongofile(数据)处。因为持久化不是在用户添加记录时就写到磁盘上，当进行更新操作操作时，记录(Record类型)都被放入到defer队列中以供延时批量（groupcommit）提交写入。所以从MongoDB使用者角度来看，持久化不会造成性能上的损耗。

MongoDB是面向文档的非关系型数据库，不是现在使用最普遍的关系型数据库，其放弃关系模型的原因就是为了获得更加方便的扩展、稳定容错等特性。面向文档的基本思路就是：将关系模型中的“行”的概念换成“文档（document）”模型。面向文档的模型可以将文档和数组内嵌到文档中。因此，实际中可以用一条数据表示非常复杂的结构。

**2.3.2 几种缓存服务对比**

本节将Ehcache、Memcache、Redis和MongoDB进行对比，以明确各种缓存服务的优缺点。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Ehcache | Memcache | Redis | MongoDB |
| 性能 | 最好，由于是直接读取内存数据，性能可以保证，大约在0.5ms的性能级别。 | 单核上Redis在存储小数据时比  Memcached性能更高 | 100k以上的数据中，Memcached性能要高于Redis | 读写性能都低于前三者 |
| 数据类型 | 由于是保存在JVM中，支持所有Java数据结构 | Memcache数据结构单一 | Redis相比于Memcache而言，支持更丰富的数据结构 | 使用BSON类型，BSON是二进制序列化的形式。类似JSON，同样支持内嵌各种类型。 |
| 内存空间 | 内存回收策略支持LRU、LFU、FIFO等算法 | Memcache可以修改最大可用内存, 内存回收策略支持LRU算法 | Redis在2.0版本后增加了自己的VM特性，突破物理内存的限制 | MongoDB适合大数据量的存储，依赖操作系统VM做内存管理 |
| 可用性 | 保存在JVM中，一旦JVM重启或宕机，数据丢失。可选策略是持久化到磁盘上。 | Memcache本身没有数据冗余机制。依赖成熟的环状hash的算法，解决单点故障引起的抖动问题 | 支持主从复制。每次从节点重新连接主节点都要依赖整个快照,无增量复制。 | MongoDB支持主从，replicasetauto sharding等都中机制。对客户端屏蔽了故障转移 |
| 可靠性 | 支持将持久化到磁盘防止数据丢失 | Memcache不支持持久化，通常用在做缓存,提升性能 | Redis支持依赖快照进行持久化，增强了可靠性的同时，对性能有所影响 | MongoDB从1.8版本开始采用binlog方式支持持久化的可靠性 |

1. **消息系统的架构设计**

本章将从企业消息推送方面涉及的实际问题入手，详细阐述了基于JMS的消息系统的在需求分析方法，系统总体架构模型，系统功能性设计以及非功能性设计方面所采用的模型和方法，为基于JMS的消息系统的设计与实现提供需求参考和设计原型。

**3.1当前消息推送系统所面临的实际问题**

当前消息系统采用的是每个生产者和消费者独立独立监听消息通道的方式进行交互，如下图3-1所示：

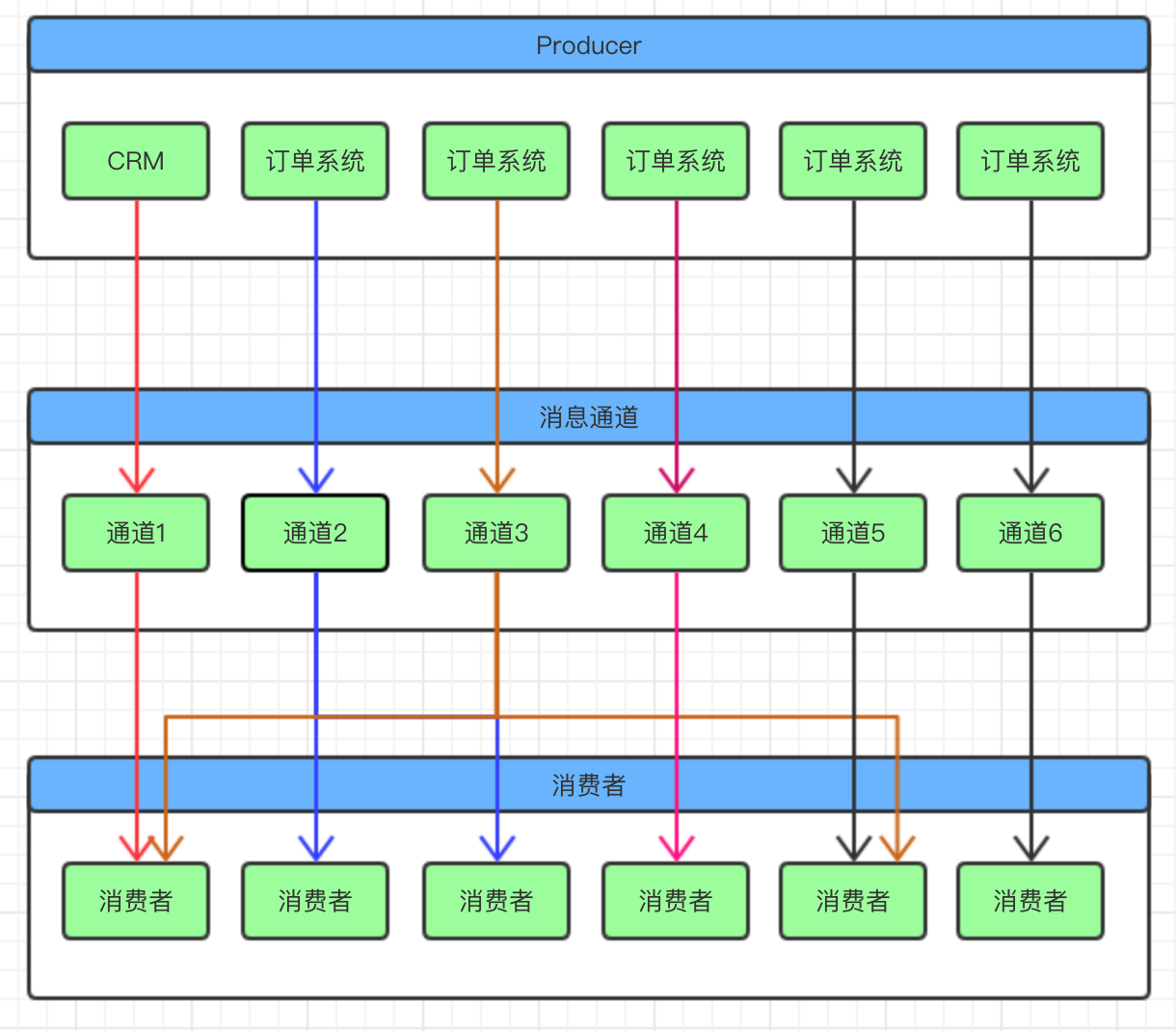


图3-1 当前消息推送系统概览

Figure 3-1 Overview Of Current Message Push System

在现有消息推送系统使用过程中，遇到主要的问题如下：

1. 可扩展性差

对于企业的业务系统而言，如果要想新增一种不同类型的消息，则需要新增一个消息通道，生产者系统和消费者系统都需要同步修改代码，支持对新的消息通道的监听，然后生产者系统和消费者系统都要修改完代码之后，发布上线。如果还想实现消息调度等功能，那么工作量将增加很多。这就导致了系统的扩展性较差。

1. 可维护差

如果消息通道负载过高，通道将抛弃生产者继续发送到通道的消息，这种情况下会造成消息的丢失。又或者生产者系统因为逻辑错误，导致发送了错误的消息到消息通道中，那么现在的系统架构将没有办法处理，只能紧急将消费者系统进行回滚，等待消费者把错误的消息全部消费完。这就造成了系统的可维护性较差。

1. 可用性较差

当前系统架构中，如果某个消息通道发生异常，生产者系统将没法继续向消息通道发送消息，消费者系统也没法从消息通道继续接受消息，这种情况下只能重新新增一个消息通道，修改生产者系统和消费者系统的代码，对新的消息通道进行监听，并且将消息通道异常期间丢失的消息重新发送到新的消息通道，以保证业务的正常进行。这种架构导致系统可用性降低，影响企业系统正常运转。

1. 可靠性差

在这样的消息推送系统中，生产者系统只关注生产消息，消费者系统只需要关注消费消息，整个系统没有办法感知到系统的异常，等到异常情况发生时，已经是没有办法挽回的局面了。系统中没有主动的异常情况汇报，异常情况只能是被动的由生产者或者消费者自己感知，系统可靠性非常差。

目前的消息推送系统可以基本完成消息在各个企业系统间进行交互的需求，但是这套系统仍旧有很多非功能性需求没法保障。

综上所述，如何保证在完成企业系统间正常消息交互的情况下，解决系统的可扩展性差、可维护性差、可用性差和可靠性差的问题，是十分值得研究和考量的，也是本课题研究的初衷，更是企业目前迫在眉睫需要解决的问题。下面的章节将详细讲解基于JMS的消息系统的架构设计，功能性需求和非功能需求。

**3.2消息系统的总体架构设计**

**3.2.1功能性需求分析**

就目前企业应用的情况分析，企业系统间交互的消息可以分为两类：

1. 面向终端用户的消息

指的是通过短信，彩信，邮件，微信推送消息和APP推送消息的形式直接发送到用户端的消息。这类消息具有高度的敏感性，如果消息发送的不及时或者消息内容不准确，可能会造成很大的影响。

1. 面向内部系统的消息

这类消息是企业系统间交互的消息。比如，订单系统定期会把已支付的订单信息同步给物流系统。这类消息不会直接面向用户。企业系统间会有大量的这类的消息交互。

因此在结合了企业消息系统的消息推送方面的实际需求与所遇到的问题，消息系统至少应该支持以下的功能：

1. 发送即时消息

指的是生产者系统发送消息之后，消息应该立刻被转发给消费者，不会将消息进行延迟处理。因此对消息发送的实时性和到达率要求比较高。这类消息的主要用于发送服务类消息的场景，如：发送注册邮件或者发送用户短信验证码等。

1. 发送定时消息

定时消息是指生产者发送消息时，设定了一个消息被转发给消费者的时间，当时间到达指定的时间后，触发消息发送给消费者系统。这类消息主要应用于实现被安排好的任务消息任务场景，如信用卡系统在规定的时间将用户上个月的消费账单发送到用户邮箱。

1. 发送批量消息

除了可以支持基本的单条消息的发送，还应该支持一次性发送多条消息的功能，比如群发功能。这类消息的主要的使用场景是发送营销类消息的场景。如对某一地区的用户批量发送促销活动短信，或者对某一类用户批量发送系统升级通知等。

1. 消息状态查询

可以对消息发送状态及时间进行查询，对积压的消息进行查询，一遍开发人员对数据状态进行评估降级或者扩容策略。

1. 消息日志查询

可以对消息发送的历史及其消费路径进行跟踪和导出，以便方便对消息进行相关的统计分析，为优化消息传输路径提供参考。

1. 默认消息通道

消息客户端可以选择性的配置通道名，如果不愿意配置通道，那么将通过配置下发可用的通道，而不会出现生产者系统和消费者系统互相不知道消息通道的情况。生产者系统和消费者系统可以都采用默认的消息通道进行通信，而不会出现异常。

1. 错误消息手动处理

可以对生产者已经生产的消息追踪，展示给开发者，提供手动的方式实现对消息的处理。例如：消息订单系统生成的订单消息逻辑有误，需要紧急将已经产生的错误消息删除，以保证损失最小。这时候在控制台提供一键清除功能，然后对生产者进行回滚操作，使消费者立刻停止对错误消息的消费。

1. 失败消息记录

对于高并发场景或者网络故障导致的消息无法正常发送的情况，将异常的消息进行记录，以方便查询损失

1. 失败消息重试

对于记录下的失败消息，可以在一段时间后，对消息进行重新尝试。知道失败消息处理完或者通知开发人员失败次数过多的消息。

1. 开发人员自定义重试策略

开发人员可以根据自己的业务场景，自定义失败消息重试策略，根据当前业务并发性，紧急性，严重性等特征，定制个性化的失败消息重试策略。

1. 死信队列处理

对于失败过多的消息，可以将其转移到死信队列这种专有的队列，这种队列消费的速度非常缓慢，不会造成业务系统频繁的出现异常，导致业务系统稳定性下降。例如：在午饭时间，点餐高峰期，某一消息由于格式不正确，始终不能被消费者正常消费，可以将其转移到死信队列中，保证业务系统可以正常处理大多数消息，同事通知开发人员死信队列中有异常消息，尽快处理。

消息系统的用例图如图3-2所示：

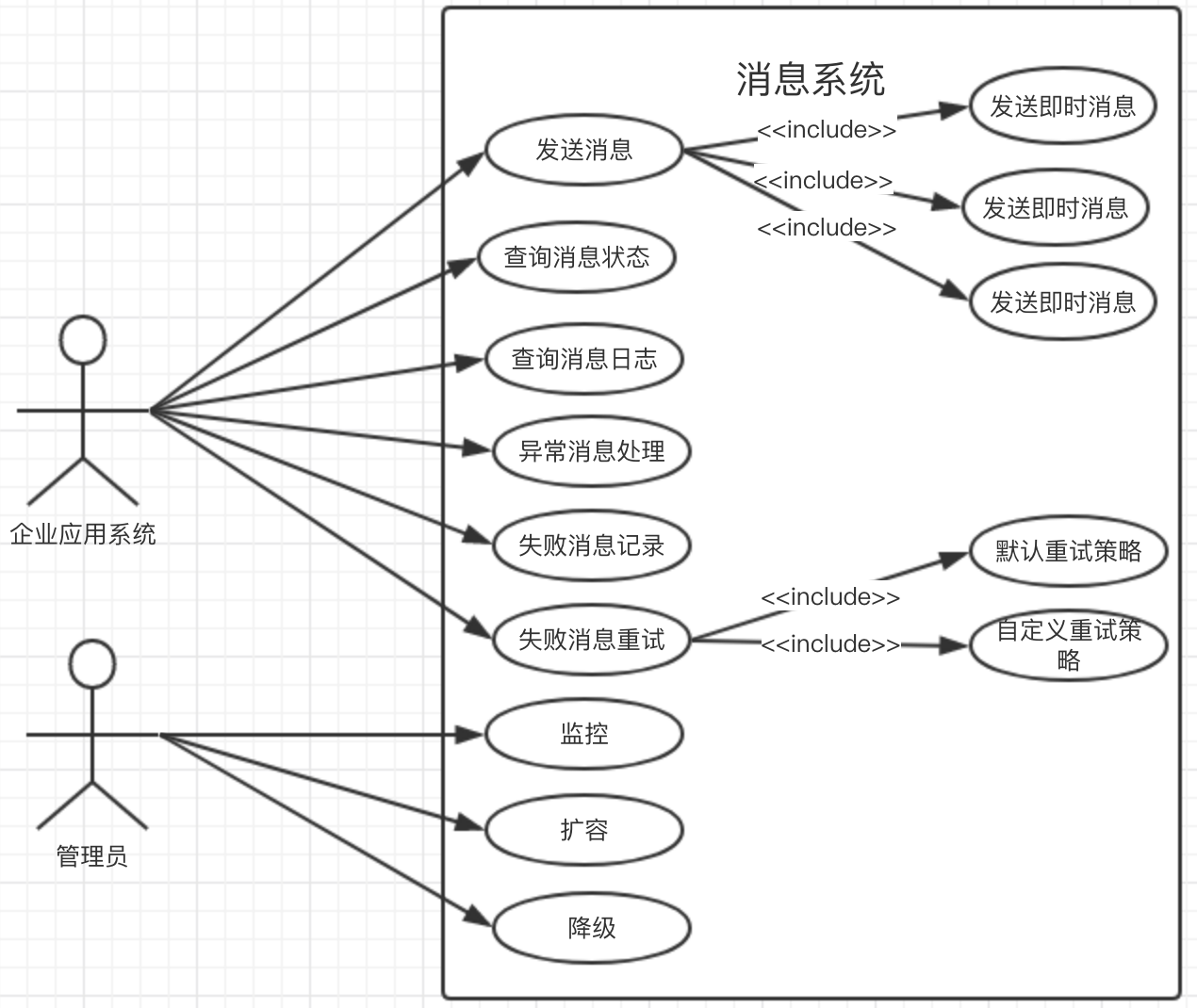


图3-2消息系统用例图

Figure 3-2 Message System Use Case

**3.2.2总体架构设计**

通过对以上消息系统的需求进行分析，并结合了当前企业系统面临的难题，我们引入“中介者（Mediator）”，即基于JMS的消息系统，来隔离企业各个应用系统之间的通信，降低企业业务系统之间的耦合性，并且借助消息系统实现消息路由，消息存储，消息日志等多种手段，确保企业应用系统之间消息交互的高可靠性，提高了消息系统的可扩展性，可维护性和可用性。消息系统的总体架构设计如下图3-3所示：

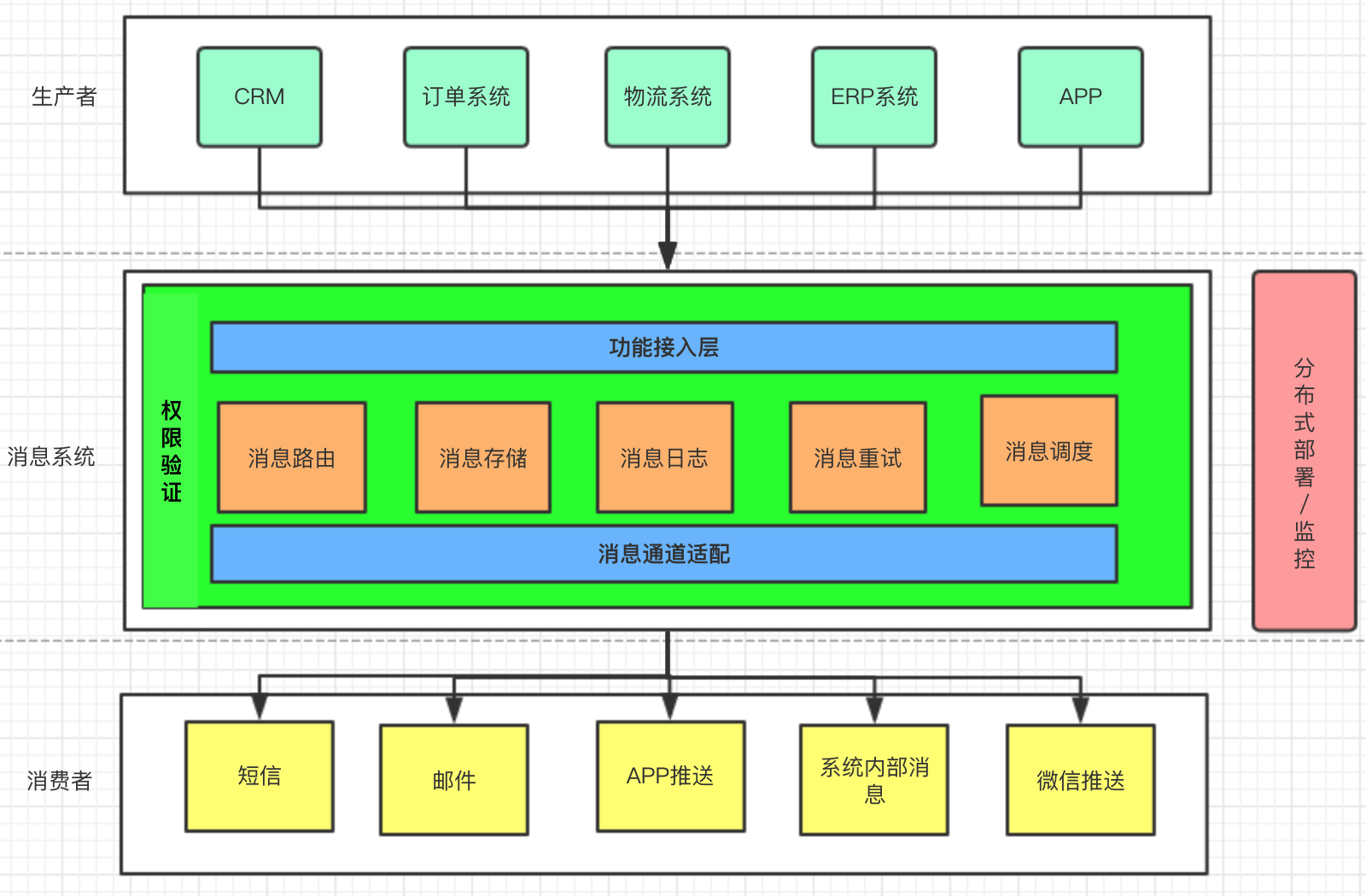


图3-3消息系统架构图

Figure 3-3 Message System Architecture

消息系统采用分层和模块化的设计思想，以消息系统为中心，企业应用系统与消息通道连接，将企业应用中生产者和消费者进行隔离，降低了企业应用系统的耦合性，提高了企业应用系统的可扩展性和可维护性。

根据以上对消息系统的总体架构分析，从消息系统的层次结构上来说，消息系统主要可以分为以下几层：

1. 功能接入层

功能接入层包含了消息系统的统一接入组件。主要包括：生产者权限验证，消费者权限验证，生产者接入接口，消费者接入接口。功能接入层将接入的生产者、消费者进行权限管理，并提供统一的API接口供生产者和消费者进行接入。功能接入层提高了消息系统的安全性和易用性，提高了生产者和消费者接入的效率。

1. 核心功能层

核心功能层包含了消息系统的核心通用组件，主要包括：消息任务调度模块，消息存储模块，消息日志模块，消息重试模块，消息调度模块，消息队列模块和消息主题模块等。消息系统的核心功能基于生产者-消费者设计模式，利用消息通道，将需要发送消息的生产者和需要接受消息的消费者进行解耦，并通过异步的消息模式，生产者和消费者无需关注消息发送和传输的底层实现细节，增加了生产者和消费者系统的可扩展性和可维护性。

1. 消息通道适配层

消息通道适配层统一管理了消息发送通道，使用适配器模式，将不同类型的消息（如短信，邮件，微信，APP推送，内部系统交互消息）发送到特定的通道中，并可以可扩展的方式支持新的消息类型的接入。消息通道适配层提高了消息通道的灵活性和扩展性，并实现了方便的进行消息通道的“插拔”。

**3.2.3主要特点**

根据以上的消息系统中体架构设计图，我们可以看出消息系统具有如下的一些特点：

1. 安全

消息系统权限验模块对生产者、消费者和通道进行安全性验证，确保验证不通过的消息不会被发送到消息通道，或者验证不通过消费者不会收到消息，保证了消息系统中消息的安全性。

1. 解耦

消息系统以其自身为中心，将企业应用系统中的生产者和消费者进行解耦，使得各个企业应用系统只需要关注自己的业务逻辑，无需更多关注消息应该发送到哪些通道，消息应该被哪些消费者消费等。这样就可以让企业应用系统和消息系统相互独立演化和迭代开发，复杂软件工程中“知道最少原则”，降低了企业应用系统的耦合性，增强了消息系统的可维护性和易用性。

1. 异步

消息系统以消息中间件作为基础，支持点对点的消息传输和发布-订阅的消息传输这两种消息传递模型，消息生产者只需要通过统一的API接口，将消息发送后立刻返回。消息消费者也只需实现消息系统的API接口，等待消息的到来即可。消息系统通过自己维护的消息路由策略将消息推送到指定的消费者。这种异步的通信方式，提升了整个消息系统的效率和性能。

1. 可扩展性

消息系统的可扩展性表现在两方面。其一，对于生产者新增的消息类型可以动态支持，生产者无需关注新增消息类型的工作。其二，对于消息发送通道是可以动态扩展的。消息系统的可扩展性使得在需要的时候，消息系统管理员可以比较容易的扩展对不同类型的消息的发送和接收功能，也可以很容易的支持消息通道的动态替换和调整。消息系统管理员可以在必要的时候消息系统进行动态扩容，保证消息系统在高并发场景下更加稳定的运行。

1. 业务无关性

消息系统连接着生产者和消费者，其主要的职责就是将企业所需要发送的不同类型的消息发送到指定的消费者。消息系统最关注的问题是如何将消息生产者发送的消息何时以合同形式发送给哪些消费者，不必关心企业应用系统的任何逻辑。这样就实现了消息发送和企业应用系统进行了隔离，符合“开闭原则”，提高了系统的复用性和可维护性。

**3.3消息系统的功能性设计**

根据以上对消息系统的总体总体架构分析，从系统集成的角度来看，消息系统主要可以分为如图3-4所示的几个子模块/子系统：

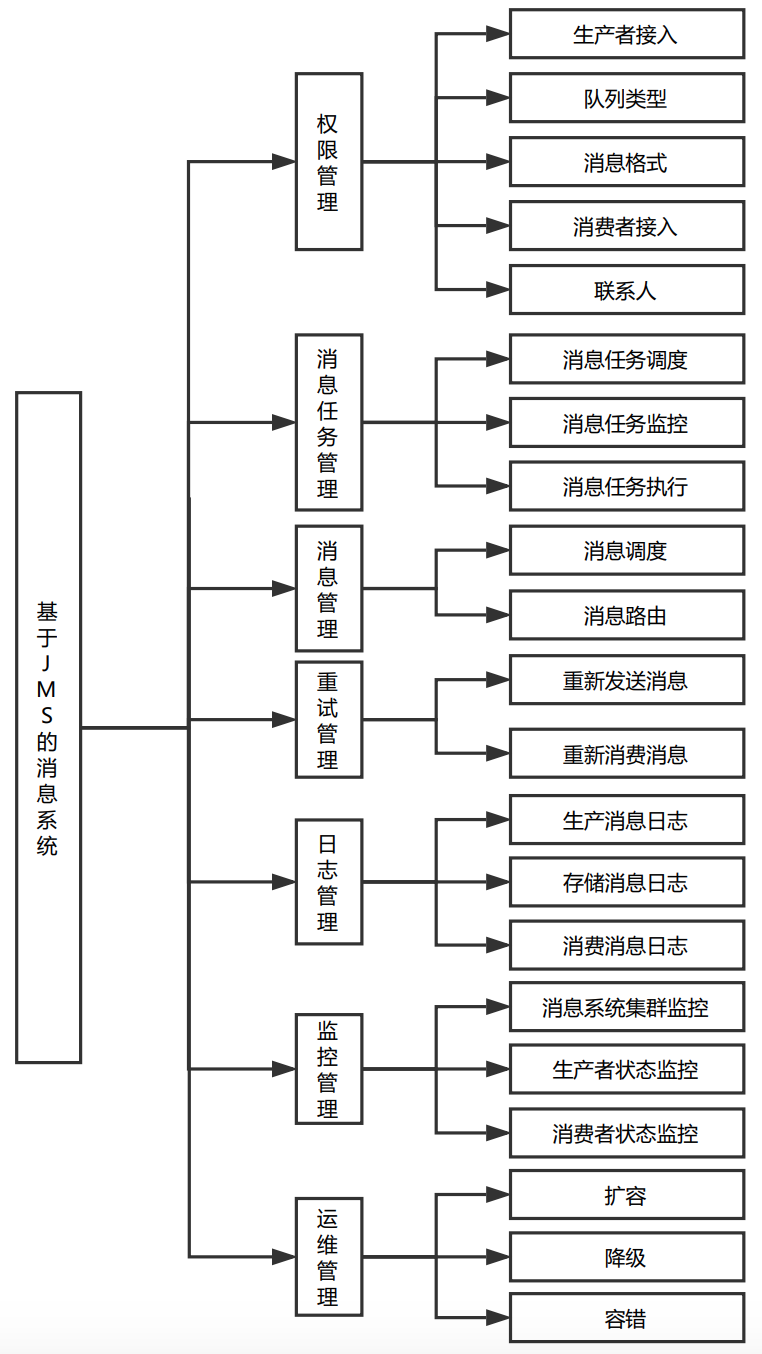


图3-4消息系统功能架构图

Figure 3-4 Message System Functional Architecture

**3.3.1 权限模块**

权限模块位于整个消息系统的最外层，是消息系统最基础的功能之一。权限模块将验证生产者权限，队列权限，生产者权限等。三者中有一者权限异常，将无法完成消息整个消息传输的过程。通过权限模块，可以找到生产者、队列、消费者对应的联系人，在系统负载过高或者系统频繁故障的时候，及时通知到对应的联系人，对当前的消息系统故障进行及时的扩容降级或者容错等。权限模块的主要流程如图3-5所示：

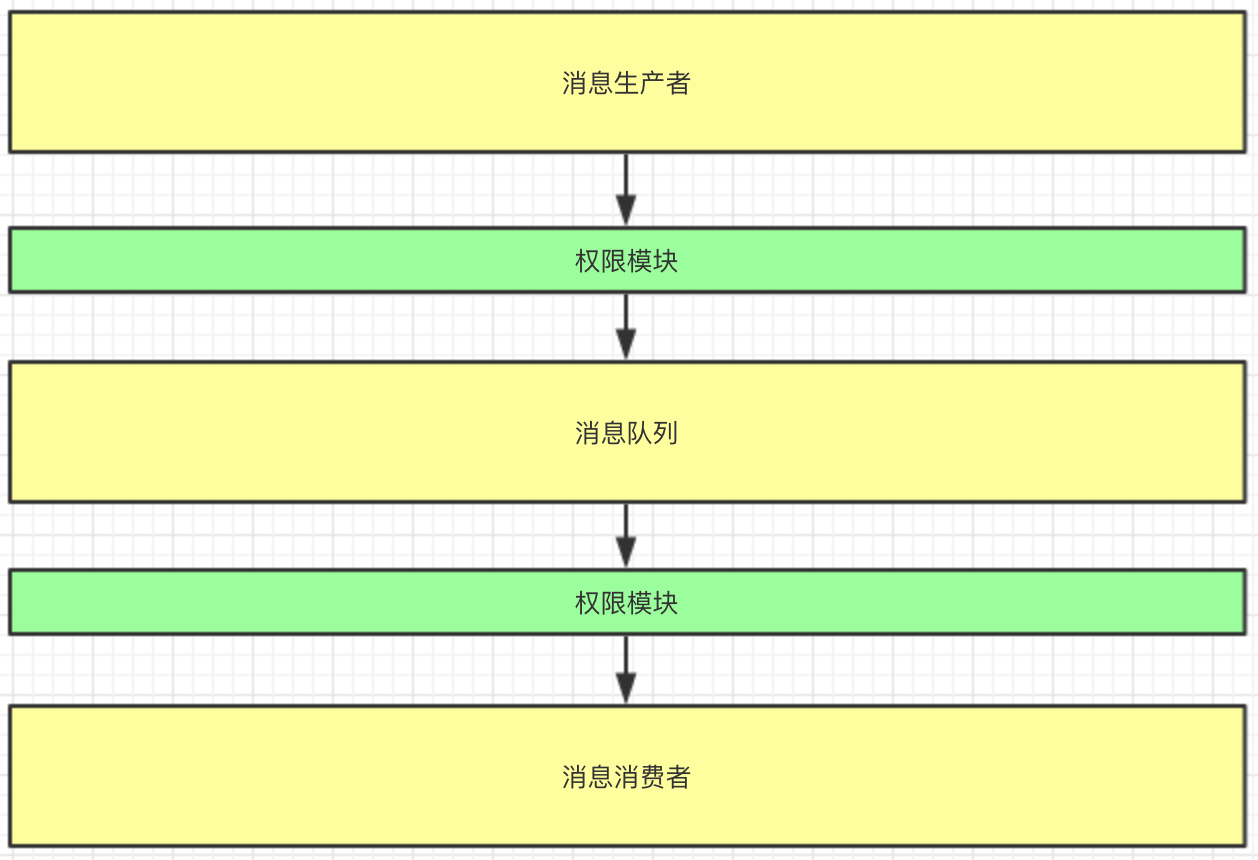


图3-5消息系统权限模块

Figure 3-5 Message System Permission Module

消息生产者、消息队列、消息消费者都会在消息系统注册权限，在消息运行过程中，实时对三者的权限进行验证。如果三者中任一者权限异常，整个消息传输流程都是失败。

一个消息生产者可以向一个消息队列发送消息，一个消息队列可以被多个消息消费者消费。三者的关系如下图3-6所示：

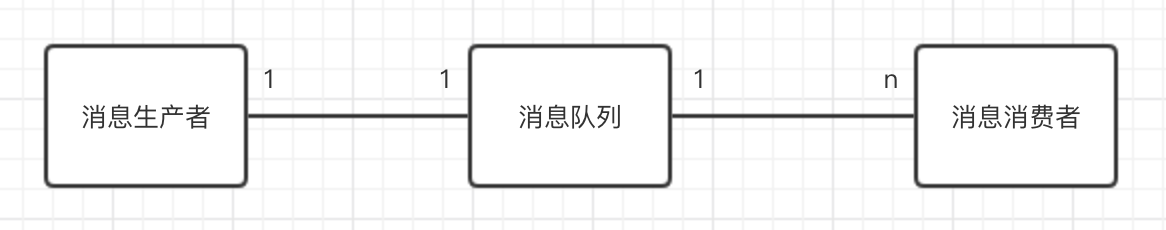


图3-5消息系统权限模块类图

Figure 3-5 Message System Permission Module Class Diagram

**3.3.2消息任务管理**

消息任务管理是这个消息系统的核心功能之一。消息任务是面向业务系统的，对于特定的业务场景选择对应的消息推送模式。对推送消息的进行逻辑组织，既可以实现实时消息推动，又可以延迟的消息推动场景。比如，当用户注册时，需要输入手机验证码，这时就需要使用实时的消息推送模式，毕竟如果一个手机验证码耗时太久，对用户体验极其不好。又例如，在银行引用卡账单日，定期对用户发送其消费账单，此时就可以通过延迟消息的推送模式，可以再账单日之前将用户账单算出，待到账单日统一发送给所有用户。消息任务管理流程图如下图3-7所示：

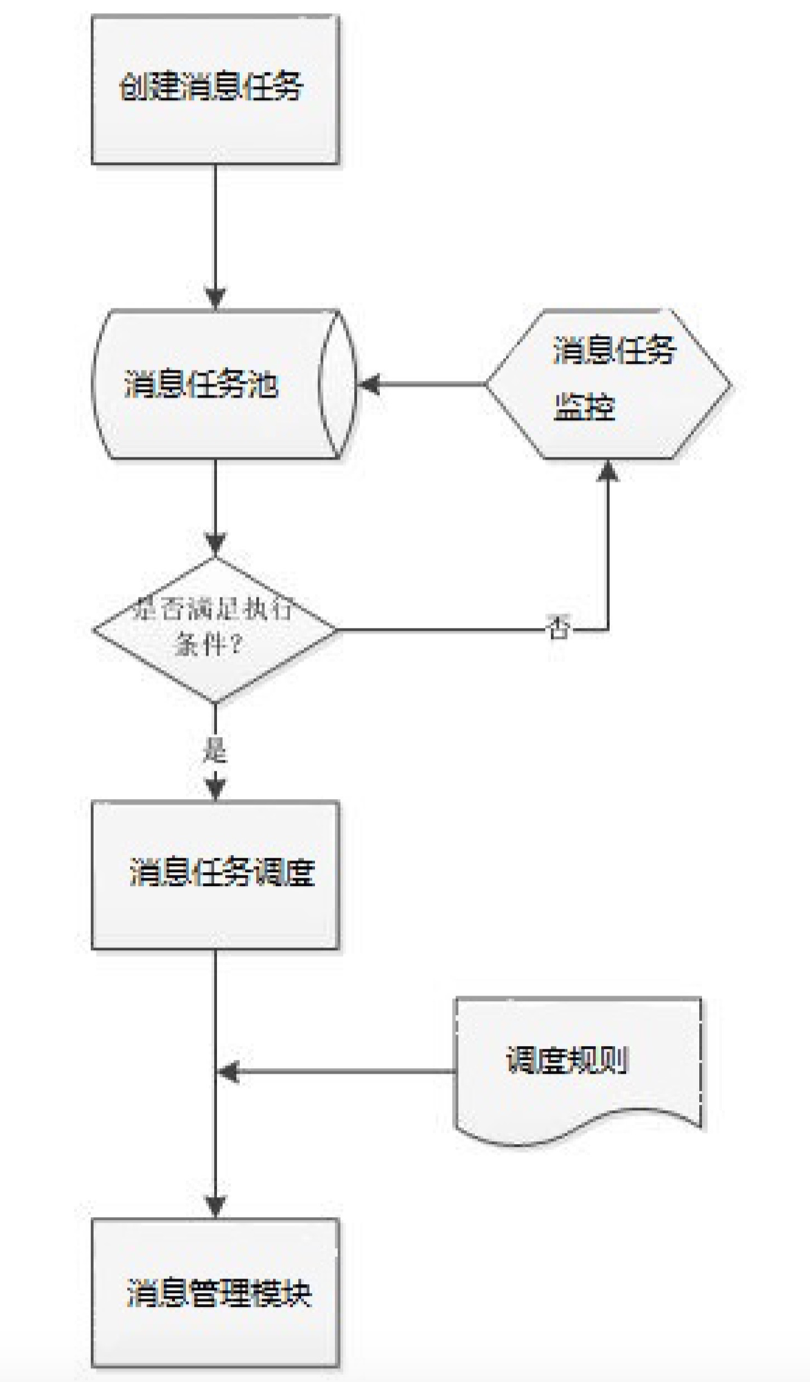


图3-6消息任务管理模块主要流程

Figure 3-6 Main Process Of Message Task Management

首先是创建消息任务，创建成功后，此消息任务会进入消息任务池。当消息任务监控发现某条消息任务满足执行条件时，会将消息任务取出来，并交给消息任务调度器，调度器会根据任务调度规则，将该任务转交给消息管理模块去执行。当消息任务执行完毕，就可以根据实际情况，对消息日志和消息执行结果进行追踪和记录。

**3.3.3消息管理**

消息管理模块是消息系统最重要最核心的功能模块，它复杂对消息的整个生命周期进行管理，以及决定消息在什么时间以什么样的方式被消费者消费。消息管理模块主要功能有：消息调度，消息路由，消息存储和消息模型。

1. 消息调度

消息调度主要包含两块：一是根据消息对象本身的优先级决定消息发送的时机。优先级高的消息将会比优先级低的消息优先发送和处理。二是消息通道的繁忙程度及发送速率，对消息通道进行负载均衡。合理的调度消息可以保障消息的时效性和完整性，同时也保障了消息系统的可靠性和稳定性。

1. 消息路由

与消息调度不同的是，消息调度侧重于将读取消息，并将消息推送给消息通道，消息路由侧重于决定哪一类消息通过哪一个消息通道到达消息消费者。

消息路由是指将不同类型的消息根据不同的路由规则推送到不同的消息通道上。路由规则的基础是JMS规范中定义的两种消息传递模型来设计的，消息路由如下图所示：

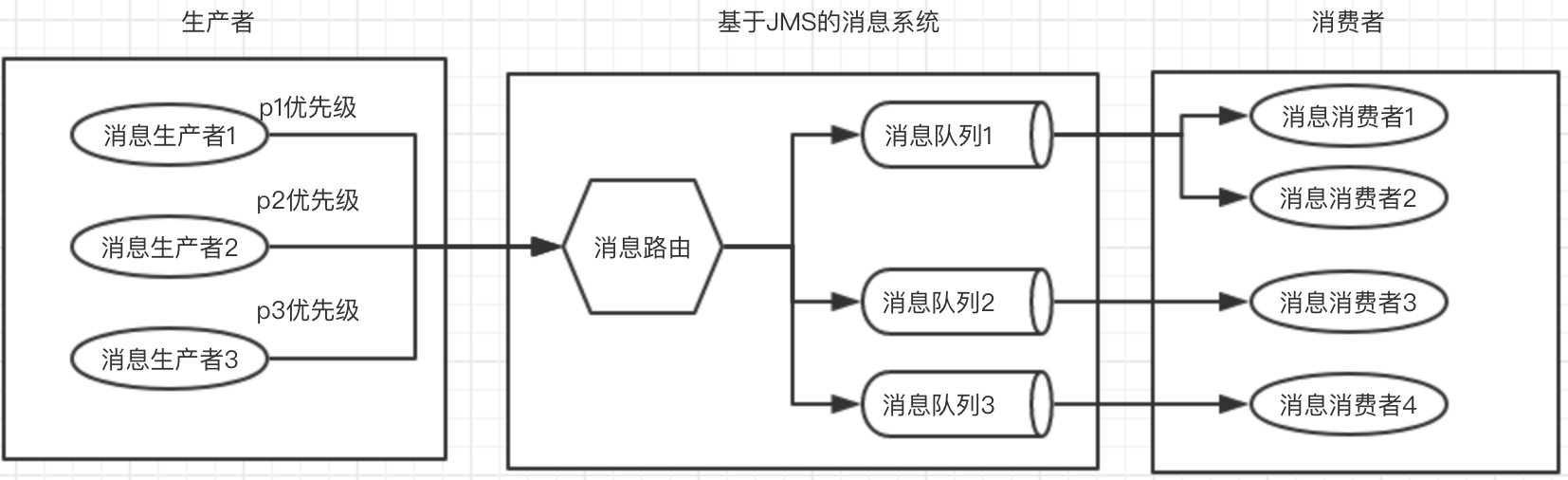


图3-7消息管理模块示意图

Figure 3-7 Overview Of Message Management

JMS支持两种类型的消息传递模式，分别是点对点传递模式和发布订阅传递模式。

点对点的传递模式如上图中生产者2、生产者3所示，它们生产的消息只能被一个消费者所消费（消费者3消费生产者2的消息，消费者4消费生产者3的消息），如果有多个消费者同时对一个生产者生产的消息进行消费，只能被一个消费者消费，确保了消息生产者产生的消息不会被重复消费。常见的场景如，用户只能收到自己的话费清单而不会重复的收到其他用户的话费清单。

发布订阅传递模式最大的特点就是同一个消息会被多个消费者所接收到。如上图3-7中的生产者1所示，它生产的消息会被消费者1和消费者2接收到，发布订阅传递模式主要用于群发或者广播的场景。常见的场景有：双十一优惠群发，所有的用户都会收到同样的关于双十一的营销。如果这种中场景使用的点对点的传递模式显然不合适，因为如果这样，优惠发布需要重复的发送同样的营销短信给所有的用户，操作成本大幅度提高。

1. 消息模型

在JMS规范中，将文字、图片、邮件等消息封装成一种“消息信封”的形式来进行传输，本消息系统也沿用了这种消息模型。“消息信封”主要分为三个部分：消息头，消息扩展属性，消息主体。

消息头包含了所有消息的所有共有属性，这些属性可以被消息生产者和消息消费者共同识别和使用的。如消息路由信息，消息提交模式，消息ID，消息发送时间，消息过期时间，消息优先级等。

消息扩展属性以“key-value”的形式对消息信息进行自定义扩展，可以灵活地满足不同消息类型的特定需要。一般通过setXXXProperty方法来定义消息属性，XXX取值为：Boolean、Byte、Double、Float、Int、Long、Object、Short及String。可以通过消息选择器（Message Selector）根据这些自定义的属性，将消息消费者感兴趣的特定的消息筛选出来。

消息主体包含了消息的核心数据。JMS定义了5中类型的消息主体：TextMessage、MapMessage、BytesMessage、StreamMessage、ObjectMessage。选择最合适的的消息主体类型可以使得JMS最有效的处理消息。

文本消息TextMessage将数据作为简单字符串存放在主体中(XML/JSON就可以作为字符串发送)。

映射表消息MapMessage使用一张映射表来存放其消息主体内容。

字节消息BytesMessage将字节流存放在消息主体中。适合于下列情况：必须压缩发送的大量数据、需要与现有消息格式保持一致等情况下适合使用字节类型的消息主体。

流消息StreamMessage用于处理原语类型。使用这种消息格式时，收发双发事先协商好字段的顺序，以保证写读顺序相同。

对象消息 ObjectMessage用于往消息主体中写入可序列化的对象。消息主体中可以存放一个对象，如果要存放多个对象，需要建立一个对象集合，然后把这个集合写入消息主体。

通过“消息信封”这种模型，将不同类型的消息以一种统一的格式进行表述，提高了消息的封装性和一致性，也提高了消息系统对于可支持的消息类型的扩展性。

1. 消息存储

当消息的消费者处于不可用状态或者消息费消费速率跟不上消息生产者的速率时，消息的存储就显得至关重要了。通过消息存储，保证了消息的送达率和消息的可靠性。

本消息系统采用了ActiveMQ框架（ActiveMQ是实现了JMS规范的开源的消息中间件框架）基于文件的消息数据库来实现消息的存储。ActiveMQ除了支持基于文件的消息存储方式外，也提供了基于关系数据库和基于内存的消息存储。在综合了这几种消息存储的优劣和特性后，本消息系统采用基于文件的消息存储方式在性能和稳定性上比较合适。下图3-8主要阐述了基于文件的消息存储的几个组成部分：

① cache部分

在内存（cache）中的部分是B-Tree结构，存储的是索引元数据。通过将索引缓存到内存中，可以加快查询速度(quick retrival of message data)。但是需要定时将 Metadata Cache 与 Metadata Store同步。这个同步过程就称为：check point。由checkpointInterval选项决定每隔多久时间进行一次checkpoint操作。

② B-Tree Indexes

B-Tree Indexes是保存在磁盘上的，称为Metadata Store，它对应于文件db.data，它就是对Data Logs以B树的形式建立的索引。有了它，Broker（消息服务器）可以快速地重启恢复，因为它是消息的索引，根据它就能恢复出每条消息的location。如果Metadata Store被损坏，则只能扫描整个Data Logs来重建B树了，这个过程是很复杂且缓慢的。

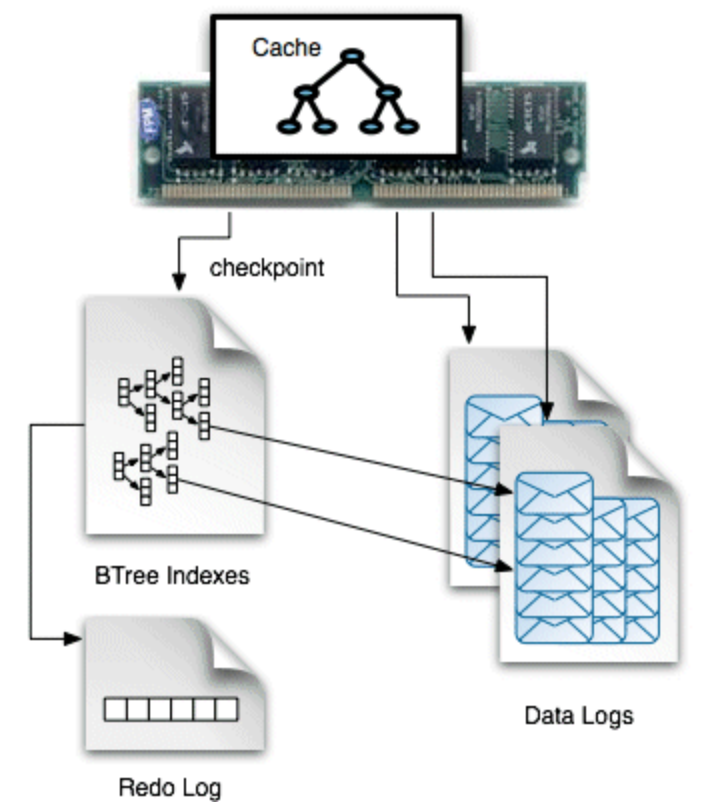


图3-8消息系统的消息存储机制

Figure 3-8 Message Storage Mechanism Of Message System

③ Data Logs

Data Logs对应于文件 db-\*.log，默认是32MB。Data Logs以日志形式存储消息，它是生产者生产的数据的真正载体。消息会按顺序被追加到文件的末尾，当文件写满后，会新建一个新的文件继续存储。

④ Redo Log

Redo Log原理用到了“Double Write”，它是为了保证B-Tree Indexes的完整性和当系统奔溃后重建索引而使用。如果在写入过程中出现故障（突然断电）就会导致数据只写入了一部分(partial page write)，而采用了“Double Write”之后，将数据写入磁盘时，先写到一个Recovery Buffer中，然后再写到真正的目的文件中。

**3.3.4消息日志管理**

消息日志管理模块是整个消息系统的核心功能之一。消息日志记录了消息在整个消息系统中的生命周期变化，以方便企业系统开发人员对消息进行监控和统计分析，定位每个阶段中消息的状态。消息状态的变化可以如下图3-9所示：

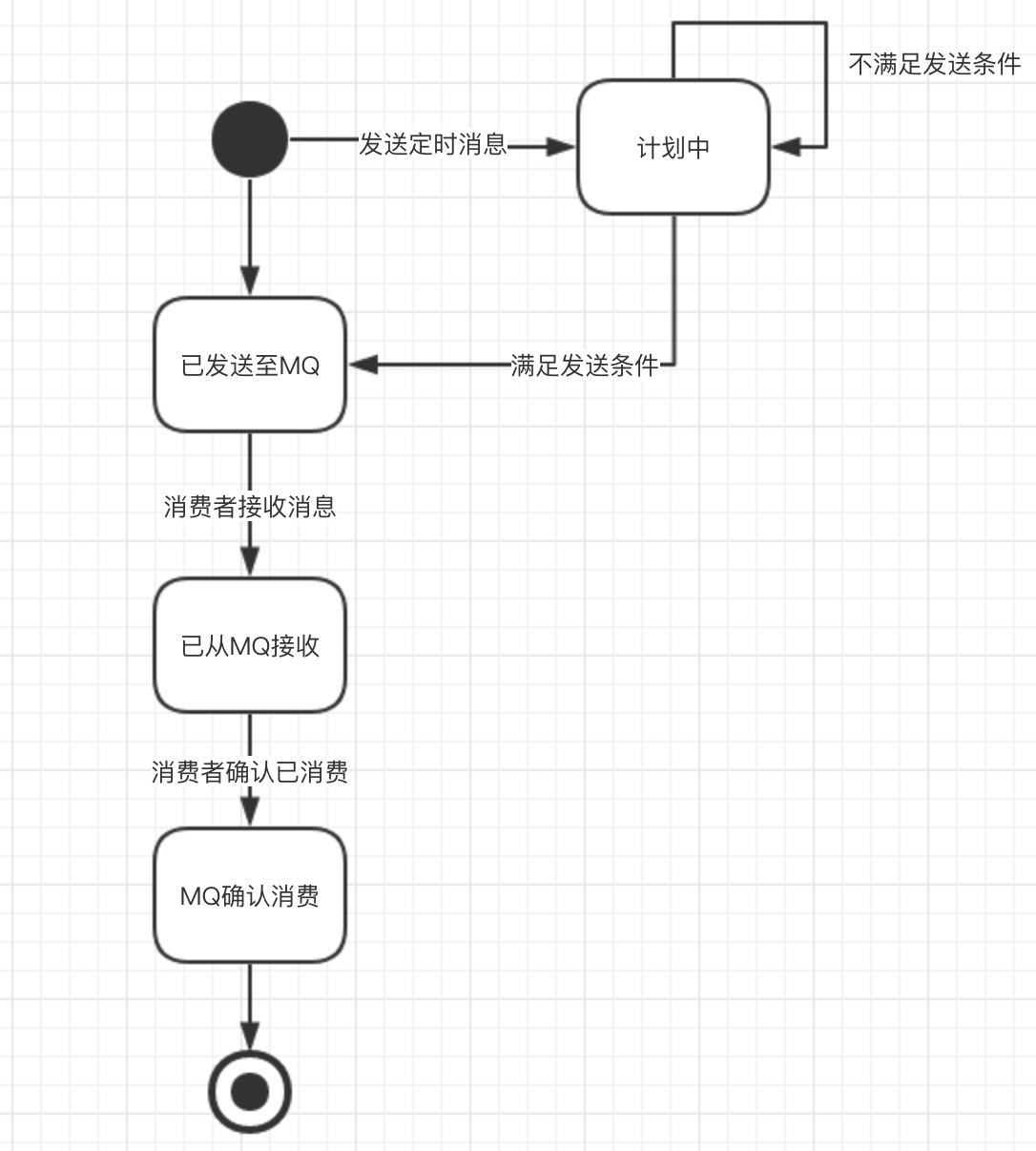


图3-9消息系统的消息状态转换

Figure 3-9 Message Status Transition Of Message System

如上文所述，消息系统既可以支持实时的消息发送，也支持定时的消息发送。当生产者通过统一的API接口创建发送实时的消息时，消息会根据根据消息系统的已有路由机制，发送到对应的队列中，此时消息状态转换为“已发送是MQ”；当生产者发送定时消息的时候，消息在消息系统中的状态变成“计划中”，此时消息系统会轮询消息的发送时间，当满足消息的发送条件时（如当前时间已经到达定时发送的时间），消息才会发送到MQ，此时消息状态转换为“已发送至MQ”。当消费者从消息队列中接收消息时，消息状态转换为“已从MQ接收”。当消息消费者确认消息已经消费时，消息状态转换为“MQ确认消费”。

在消息的整个生命周期过程中，“MQ确认消费”即消息应答，这一点是比较重要的。它是保证不会被重发，确认消息可靠性的重要手段。如果没有消息应答机制，很有可能造成消息系统不能确定消息是够已经发送给消费者，从而可能导致消息被错误的重复发送多次。这可能导致非常严重的后果，如：消息系统发送扣费消息到用户账户，如果用户账户不确认消息已经消费，就会导致重复的对用户进行扣费。可以通过消息队列的确认特性（AUTO\_ACKNOWLEDGE）来避免上述情况的发生。

**3.3.5配置管理**

配置管理模块主要面向的是消息系统的管理员。当企业系统接入消息系统的时候，消息系统的管理员可以根据接入的企业系统的重要性和企业系统的消息的敏感性决定是否需要业务系统实现相关功能。如，生产者权限验证，消费者权限验证等。权限验证包括身份验证（Authentication）和授权管理（Authorization）。通过身份验证可以识别对应消息系统的企业实体是否是合法的。授权管理是用来判断企业系统是否有权限对消息系统进行生产/消费行为。

通过配置管理，增加了消息系统的扩展性和灵活性，使得消息系统的各个模块实现了“热插拔”，赋予了接入消息系统的所有企业系统更多的选择性。

**3.3.6功能接口层的设计**

消息系统的对外接口都是基于SOA的思想而设计的，将消息系统的功能以一套标准的，统一的，无状态的服务接口暴露给企业系统，隐藏了消息推送和存储等相关的实现细节。消息系统的功能接口主要包括如下几部分：

消息权限接口，对应着消息系统的权限管理模块，如：生产者对消息系统的接入权限管理，生产者对消息系统的消息队列的接入权限管理，消费者对消息系统的接入权限管理等等。

消息管理类接口，对应着消息系统的消息管理模块，如：发送实时消息，发送定时消息，发送批量消息，消费消息，消息确认，查询消息状态，取消消息任务等等。

消息日志接口，对应着消息日志管理模块，如日志查询，告警分析等。

通过消息系统提供统一的功能服务接口，接入的企业系统以一致的方式接入消息系统，企业系统只需关注各个接口的功能，而不必关注接口的具体实现细节，增强了企业系统与消息系统的互操作性，进一步促进了消息系统与企业系统之间的解耦。

**3.4消息系统的非功能性设计**

消息系统使得原本互连的企业系统之间的点到点的架构演变成了以消息系统为中心，企业系统与消息系统直连的架构形式。随着企业应用架构的演进，消息系统将在企业系统中扮演越来越重要的角色。所以消息系统在实现了基本的功能的基础上，对于非功能性的设计也至关重要。

**3.4.1消息系统通道扩展性设计**

消息系统将消息以“消息信封”的形式进行封装，消息通道使用的适配器模式，将不同类型的消息（如，短信，彩信，邮件，用户订单，用户评价）发送到对应的消息通道中。而对应消息系统来说，消息通道是以插件的方式进行集成和扩展的，这增加了消息系统的灵活性和扩展性。下面以用户评价为例，分析消息通道的扩展性实现。

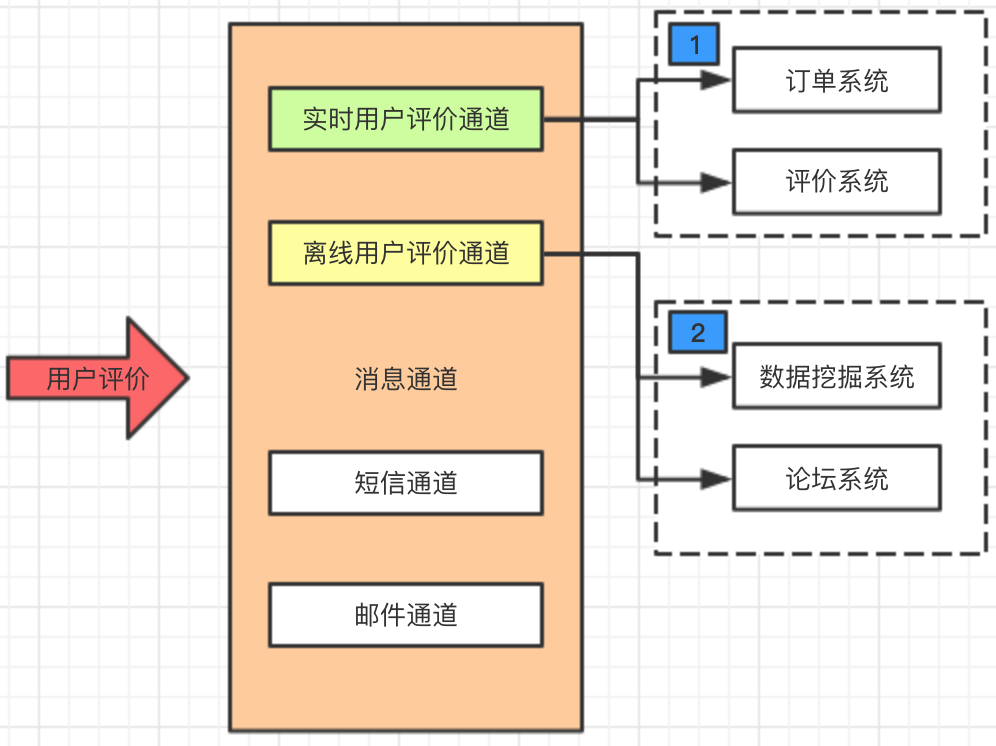


图3-10通道适配层的扩展性

Figure 3-10 Extension Of Message Channel Adapter

上图3-10所示，消息通道中的实时用户评价通道对接订单系统和评价系统。假设有如下场景，当用户购买了某件商品，确认收货后，评价其购买的商品。评价结束后，订单系统和评价系统都会记录该商品的用户评价。用户刷新对应的页面后，可以看到自己的评价内容。如果此时还想增加让商家知道自己的商品被评价了，可以再商家系统中监听用户评价信息，简单的做法是在图中1号实例系统中添加商家系统，就可以达到扩展的需求。

当用户评价积累了一定的数量后，如果数据挖掘系统想要挖掘用户评价的内容，此时如果直接按照上述方法直接监听用户评价信息，则不能满足需求。原因如下：一是实时用户评价通道不会记录全量的历史用户评价信息；二是，实时用户评价通道负载不足以满足数据挖掘系统的要求。此时可以在消息通道中新增一个消息通道—离线用户评价通道，该通道的配置要优于实时用户评价通道。这时候订单系统将全量的用户评价数据发送到离线用户评价通道，数据挖掘系统就可以拿到订单系统的全量的评价信息了。

通过以上分析，消息通道是以“插件”的方式新增了消息通道实例，对于企业系统来说，这个过程是完全透明的。其扩展性降低了企业系统的开发成本。

**3.4.1消息系统高可用性设计**

消息系统的高可用性的实现主要借助于消息系统的服务器的主-从（master-slave）架构来保证的。如下图3-11所示：

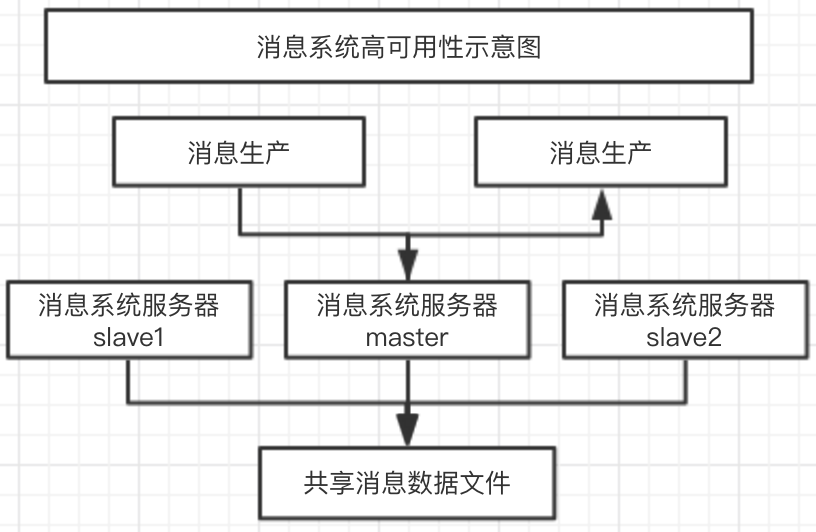


图3-11消息系统高可用方案

Figure 3-10 High Availability Solution Of Message System

通过主-从（master-slave）架构，当消息系统主服务器不可用的时候，消息系统从服务器中的其中一个会接替主服务器的角色（取决于从服务器中谁先获得共享消息数据文件的排他锁），同时，消息生产者和消息消费者也会断开与不可用的主服务器的连接，与新的主服务器重新建立连接，从而达到了消息系统的的高可用性。

**3.5本章小结**

本章从消息系统的功能性设计和非功能性设计两方面入手，详细阐述了消息系统的设计思想和整体架构。消息系统采用分模块和分层的设计思想，这个消息系统分为权限模块，任务管理模块，消息管理模块，消息日志管理模块，配置管理模块等，实现了系统的可维护性和可扩展性。

1. **消息系统的关键技术与实现**

在上一章中已经分析了消息系统的功能性设计和非功能性设计及架构，本章将以邮件消息为例，对消息系统的关键技术和实现进行详细的阐述，包括消息权限管理，消息路由与调度，消息存储，消息日志管理，配置管理及消息系统的高可用性实现方案。并通过UML图等方式，阐述消息系统关键技术的实现细节，为基于JMS的消息系统提供更详细的理论和实践基础。

**4.1消息系统技术架构概览**

消息系统主要基于Spring，ActiveMQ，MySQL，Zookeeper等技术来实现的。消息系统的技术架构图如下图4-1所示：

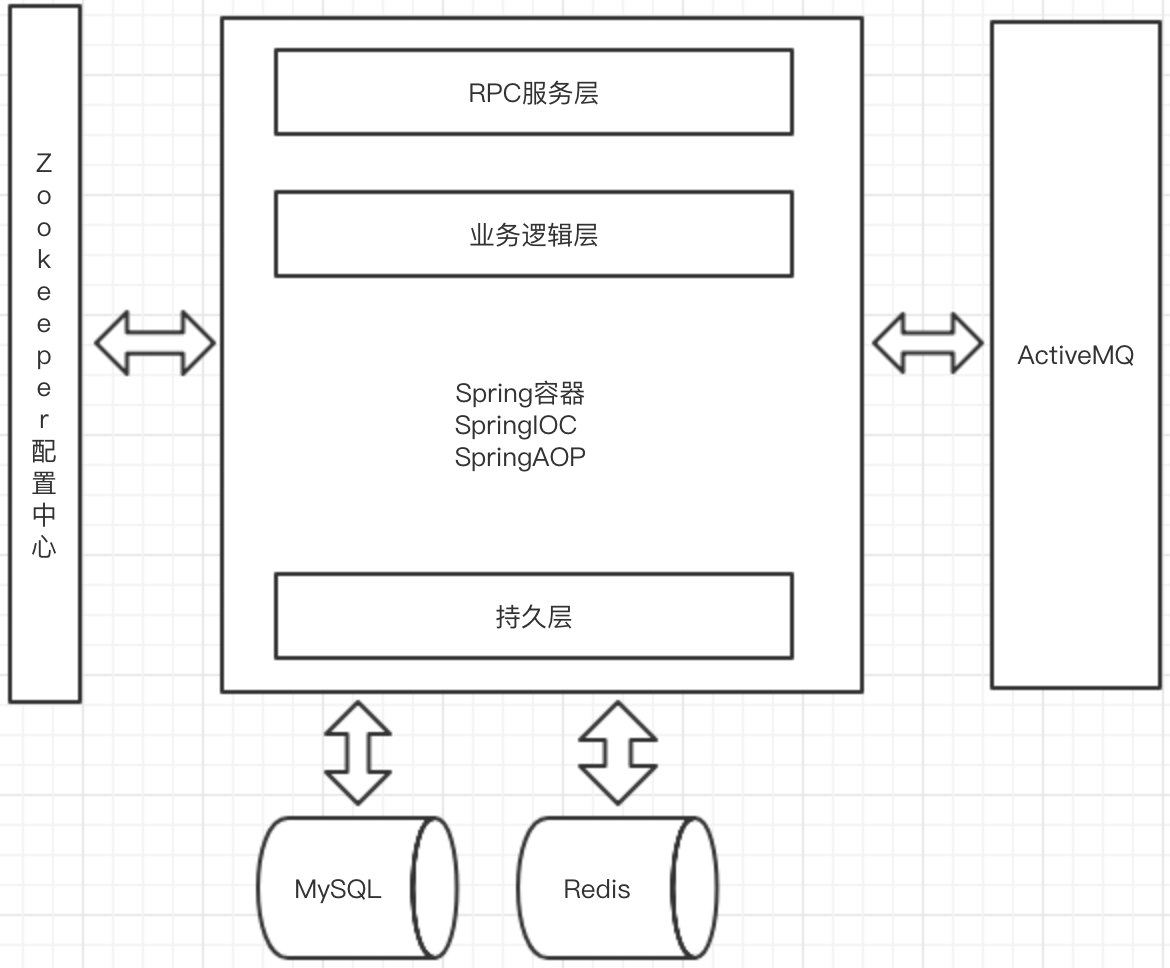


图4-1消息系统技术架构概览

Figure 4-1 Overview Of Message System Technical Architecture

如上图4-1所示，Spring IOC主要作为容器将相关的JavaBean置于容器中进行统一管理，并统一管理这些JavaBean对象的生命周期。与传统的Java编程不同，Spring IOC重点是在系统运行中，动态的向某个对象提供它所需要的其他对象，而不是由开发人员手动创建这些对象。Spring是通过依赖注入来实现的（Dependency Injection）。这样做的好处是，降低了开发人员代码的复杂度，并且由于Spring IOC都是单例模式设计的（Singleton Pattern），降低了系统运行时内存中对象的数量，节约了系统内存。

**4.2消息传递与调度的核心功能实现**

消息传递和消息调度是消息系统的核心功能，也是整个系统中最重要，最复杂的部分。消息系统主要是基于生产者-消费者设计模式而设计的，通过消息队列来存储消息，以及在消息生产者和消息消费者之间传递消息。下面将对消息系统的消息路由和传递等机制的实现细节进行阐述。

**4.2.1消息生产者核心类及其关系**

消息系统的生产者将消息发送到消息队列，发送消息的过程分为两步骤：第一步是通过MessageConverter将不同类型的Java对象转换成标准的JMS消息信封格式的消息。我们可以实现MessageConverter接口来自定义一个消息转换器，将Java对象转换成JMS规范支持的消息类型。第二步是使用Spring集成的ActiveMQ的客户端spring-jms-4.3.14.jar中的JmsTemplate将消息发送到执行的队列中。通过Message-Converter和JmsTemplate二者的配合，可以将符合JMS规范的消息发送到指定位置。下图4-2所示是MessageConverter的相关类图：

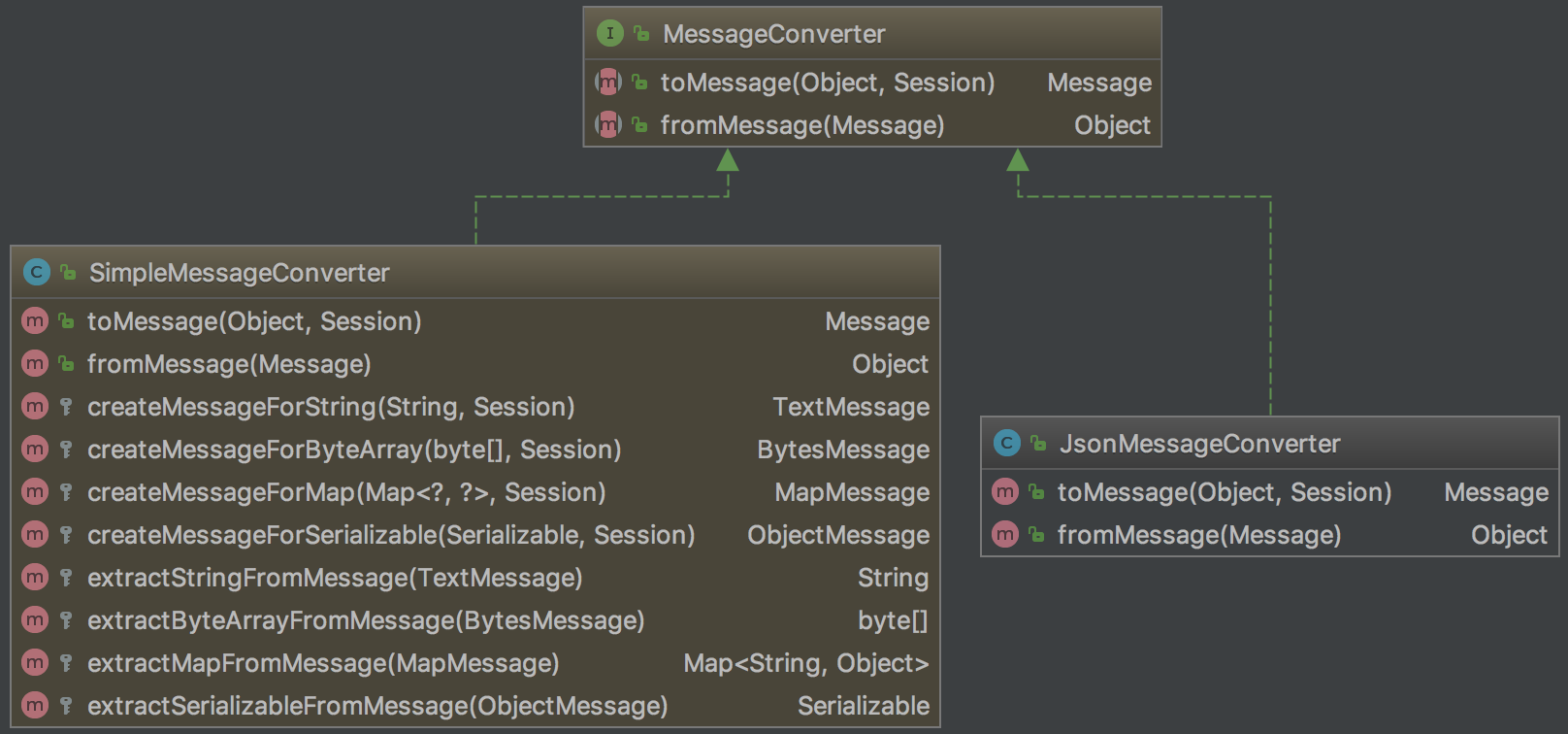


图4-2 MessageConverter相关类图

Figure 4-2 MessageConverter Related Class Diagram

上图中展示了两种类型的消息转换器，SimpleMessageConverter是默认的一种消息转换器，如果不指定任何消息类型转换器，那么JmsTemplate将会默认使用SimpleMessageConverter作为消息转换器。JsonMessageConverter是自定义的一种消息转换器，它可以将Java对象序列化成JSON串，传递到消息队列中。此外，用户还可以通过实现MessageConverter接口自定义其他的消息转换器，并通过JmsTemplate提供的setMessageConverter(MessageConverter messageConverter)方法设置JmsTemplate的消息转换器。

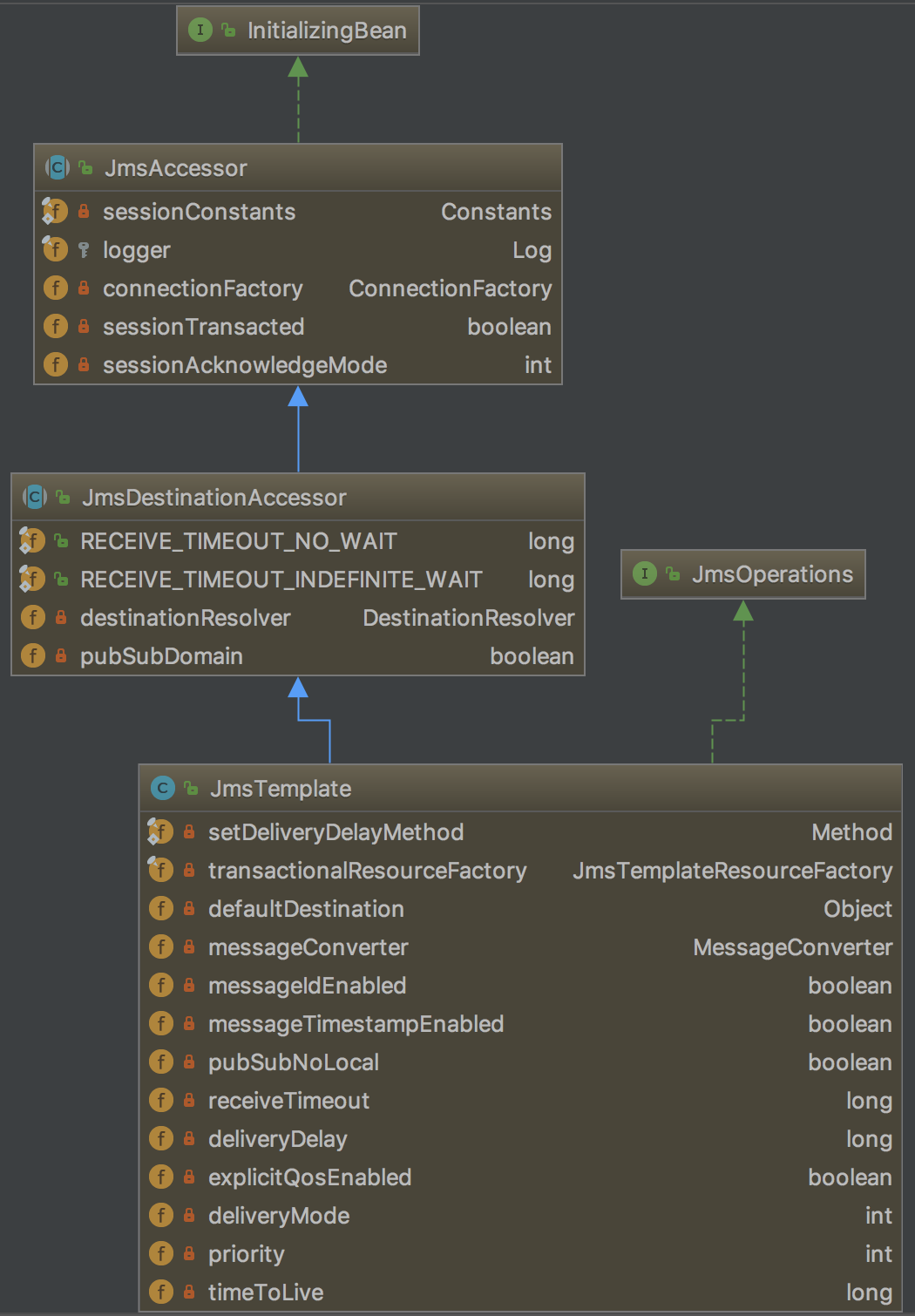


图4-3 JmsTemplate相关类图

Figure 4-3 JmsTemplate Related Class Diagram

如上图4-3所示，消息系统的消息生产者可以通过JmsTemplate将消息发送到指定的消息队列中。

**4.2.2消息消费者核心类及其关系**

消息消费者从消息队列接收消息，同消息生产者类似，消息消费者从消息队列获取消息也是分为两个步骤，第一步是通过Spring集成的ActiveMQ客户端中的MessageListener接口监听消息，第二步是在监听到消息后，通过MessageConverter将标准的JMS消息转换成用户自定义的Java对象。消息系统提供了通过MessageListenerAdapter监听消息，用户也可以通过自定义类实现MessageListener接口来实现对消息的监听。下图4-3所示是MessageListener相关的类图：

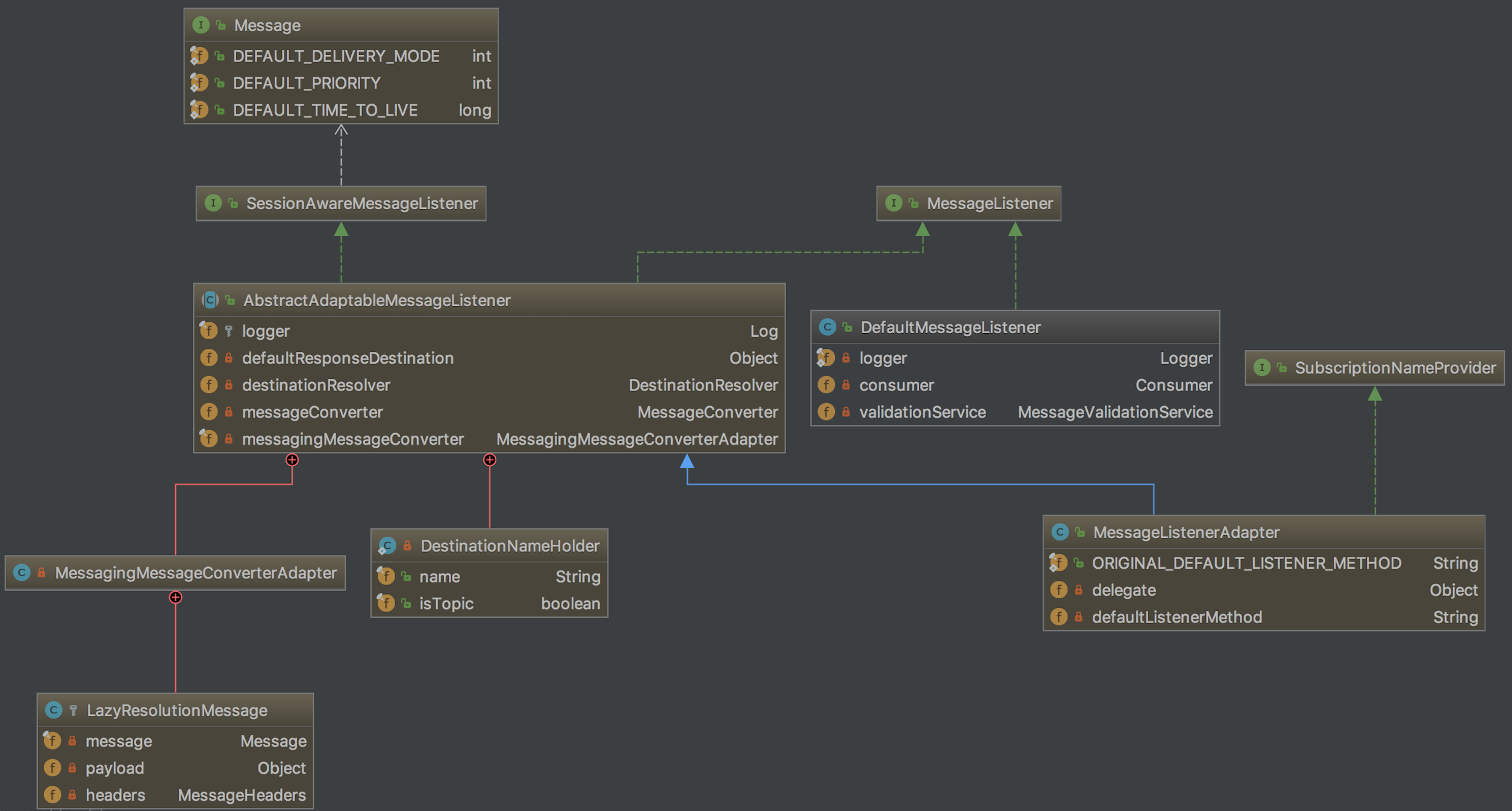


图4-4 MessageListener相关类图

Figure 4-4 MessageListener Related Class Diagram

如上图4-4所示，用户可以使用MessageListenerAdapter来监听消息，也可以使用DefaultMessageListener来监听消息。如果以上两个类都不能满足需求，可以实现MessageListener接口来自定义消息消费者的逻辑。需要注意的是，ActiveMQ消费者使用的push模式，消费者不需要发起消费请求，消息队列会把消息自动发送到消息消费者。这是ActiveMQ与kafka最大的不同之处。

**4.2.3消息传输机制和可靠性保证**

消息生产者想消息系统发送消息，消息消费者从消息系统消费消息；消息系统是将生产者和消费者解耦的工具。消息系统需要解决各种场景下的用户需求，如支持异步调用，支持流控，可靠性等保证。

下图4-5所示是一条消息的生命周期：

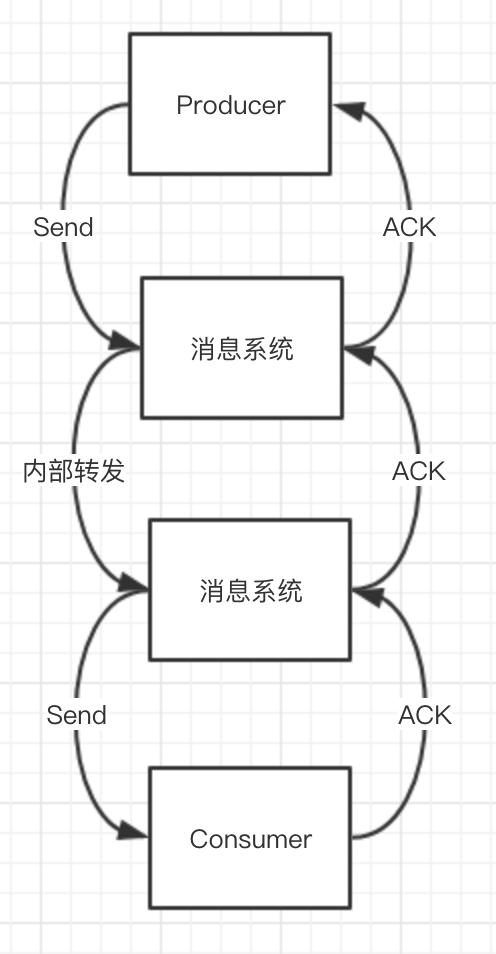


图4-5消息的生命周期

Figure 4-5 Lifecycle Of A Message

一条消息从Producer发出后，到达了消息系统，消息系统将其写入存储介质（JDBC/kahadb/leveldb）后，返回一个ACK信号给生产者，表明此时消息已经被消息系统接收到，可以进行下一条消息的发送。如果消息系统迟迟不返回ACK给Producer，则会触发Producer的从发机制，对消息进行重新发送。

当Consumer从消息系统消费了一条消息后，此时消费者确认消息处理接收到后，也需要向消息系统返回一个ACK信号，表明此消息已经成功接收，消息系统接收到Consumer发来的ACK信号后，才会将消息删除，否则消息系统会重复向Consumer发送该条消息。

ACK模式与类型，消息系统支持以下4种ACK模式，分别如下：

（1）AUTO\_ACKNOWLEDGE = 1自动确认

如果消费者正常消费，消息将会正常确认（STANDARD\_ACK\_TYPE）。如果消费者发生异常，将导致client端补充发送一个ACK\_TYPE为REDELIVERED\_ACK\_TYPE确认指令，该指令将会触发消息的重新发送。这样就可以保证消息不会因为消费者异常而丢失，但是缺点是消费者必须保证对消息的幂等处理，因为会重复接收到之前已经发送过的消息。

（2）CLIENT\_ACKNOWLEDGE = 2客户端手动确认

客户端手动确认，这就意味着消息系统将不会自动的为使用者发出任何ACK消息，使用者需要自己择机确认。如果消费者因为某种原因导致acknowledge方法未被执行，将导致大量消息不能被确认，broker端将不会继续push消息，事实上client端将处于“假死”状态，而无法继续消费消息。我们要求消费者在消费一定数量的消息之后，必须进行一次ACK发送；通常我们总是每消费一个消息调用一次，这是一种良好的设计。

（3）DUPS\_OK\_ACKNOWLEDGE = 3自动批量确认

"消息可重复"确认，意思是此模式下，可能会出现重复消息，并不是一条消息需要发送多次ACK才行。它是一种潜在的"AUTO\_ACK"确认机制，为批量确认而生。当消费者故障重启后，那些尚未ACK的消息会重新发送到消费者。

（4）SESSION\_TRANSACTED = 0事务提交并确认

当session使用事务时，就是使用此模式。在事务开启之后，和session.commit()之前，所有消费的消息，要么全部正常确认，要么全部redelivery。

当开发者决定事务可以提交时，必须调用session.commit()方法，commit方法将会导致当前session的事务中所有消息立即被确认；事务的确认过程中，首先把本地的deliveredMessage队列中尚未确认的消息全部确认(STANDARD\_ACK\_TYPE)；此后向broker发送事物提交指令并等待broker反馈，如果消息系统事务操作成功，那么将会把本地deliveredMessage队列清空，新的事务开始；如果消息系统事务操作失败，那么对于session而言，将执行inner-rollback，这个rollback所做的事情，就是将当前事务中的消息清空并要求对未提交的消息进行重发(REDELIVERED\_ACK\_TYPE)，同时commit方法将抛出异常。

1. INDIVIDUAL\_ACKNOWLEDGE = 4单条消息确认

它的确认时机和CLIENT\_ACKNOWLEDGE几乎一样，当消息消费成功之后，需要调用确认方法来确认此消息(单条)，而CLIENT\_ACKNOWLEDGE模式则是将整个session中所有消息被确认(批量确认)。

**4.2.4消息系统消费模型的优化**

如果消费者每接每消费一条消息就发送一个ACK信号，那么就会极大限制了消费者的消费速率。消息系统对消息确认机制做出的优化。

prefetchSize指定消费者可以预获取一定数量的消息，如果prefetchACK为true，那么prefetch必须大于0；当prefetchACK为false时，使用者可以指定prefetch为0以及任意大小的正数。不过，当prefetch=0是，表示消费者将使用PULL(拉取)的方式从broker端获取消息，broker端将不会主动push消息给消费者，直到消费者发送PullCommand时；当prefetch>0时，就开启了push模式。

prefetch表达了“预获取”消息的语义，消息系统主动的批量push多条消息给消费者，总比消费者多次发送PULL指令然后消息系统返回一条消息的方式要优秀很多，它不仅减少了消费者在获取消息时阻塞的次数和阻塞的时间，还能够大大的减少网络开销。optimizeACK表达了“延迟确认”的语义(ACK时机)，消费者在消费消息后暂且不发送ACK，而是把它缓存下来(pendingACK)，等到这些消息的条数达到一定阈值时，只需要通过一个ACK指令把它们全部确认；这比对每条消息都逐个确认，在性能上要提高很多。由此可见，prefetch优化了消息传送的性能，optimizeACK优化了消息确认的性能。

以下配置信息表示一个优化的消息消费模型：

String brokerUrl = "tcp://localhost:61616?" +

// optimizeAcknowledge表示可以延迟对消息的确认

"jms.optimizeAcknowledge=true" +

"&jms.optimizeAcknowledgeTimeOut=30000" +

"&jms.redeliveryPolicy.maximumRedeliveries=6";

// prefetchSize表示可以预获取100条消息

String queueName = "test-queue?customer.prefetchSize=100";

**4.3消息类型扩展**

由于消息系统受众是企业所有的业务系统，因此消息系统支持的消息类型必须是可扩展的，这也是消息系统设计时重点考虑的问题之一。消息系统的消息统一继承自Message接口，不同的企业系统在实现了Message接口之后便可以定义出自己的消息对象，配合实现MessageConverter接口，就可以将企业系统的消息按照其自身需求进行序列化，然后发送到消息系统中，消息系统的消费者只需要知道生产者的消息类型和序列化方法就可以反解出生产者发送的消息。消息系统消息对象模块的类图如下图4-6所示：

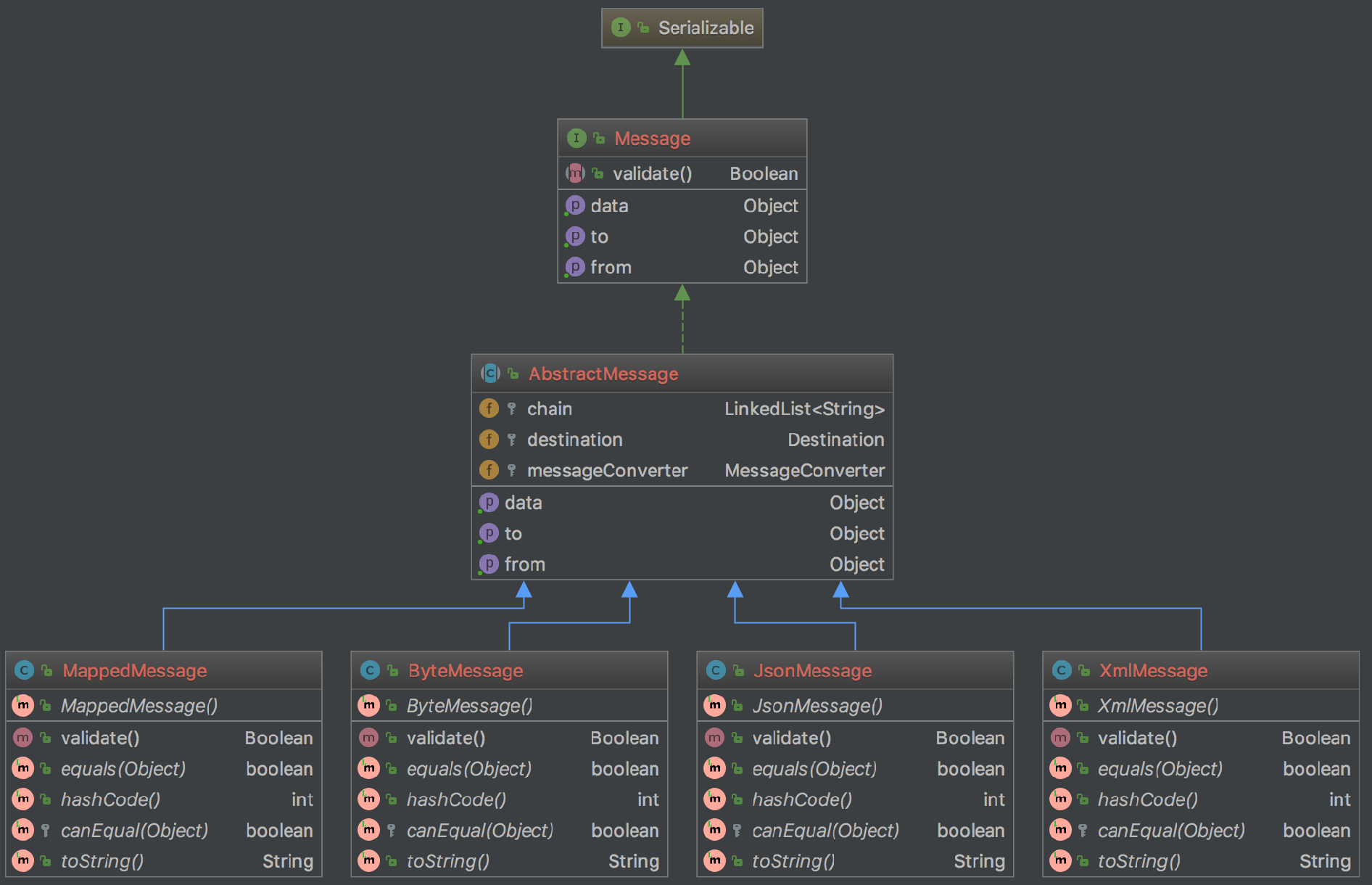


图4-6 Message相关类图

Figure 4-6 Message Related Class Diagram

上图中可以看到，消息系统现支持的消息类型有：支持字典对象的MappedMessage，支持MXL格式的XMLMessage，支持格式的ByteMessage，支持JSON格式的JSONMessage。假设有一个企业系统需要一个可以加密的消息对象，那么以上这些消息对象可能都不能满足这个需求，这时候企业系统可以实现自己的加密消息对象EncryptMessage，并将实现MessageConverter即可满足其对加密消息的需求。

Message接口对消息系统中所有的消息对象进行了抽象，在消息系统上层需要使用到消息的时候（如MessageConsumerFilter通过对象的validate方法对消息进行校验和过滤），可以通过Message进行统一的处理，屏蔽了不同消息的差异性，切实履行了软件工程中的的开闭原则和里氏替换原则，使得消息系统具有更好的扩展性和兼容性。

**4.4消息系统日志技术——日志中心**

**4.4.1 消息系统总体架构**

消息系统运行过程中会产生大量的日志，日志记录的消息系统实时的运行情况，对消息系统的日志进行处理时，不仅仅要考虑到消息日志的功能需求，同时还需要考虑消息日志查询的效率。消息日志模块有助于提高消息系统的可维护性。

消息系统的日志接入了日志中心。日志中心的实现如下：日志中心提供通用的对外接口，消息系统和企业系统通过将系统实时产生的日志作为消息发送到kafka集群，kakfa集群的对应的消费者消费到消息系统和企业系统产生的日志后，将消息保存的Elasticsearch集群中，最后开发人员可以根据Elasticsearch语法及其索引快速定位到想要查找的日志。消息系统日志技术如图4-7所示：

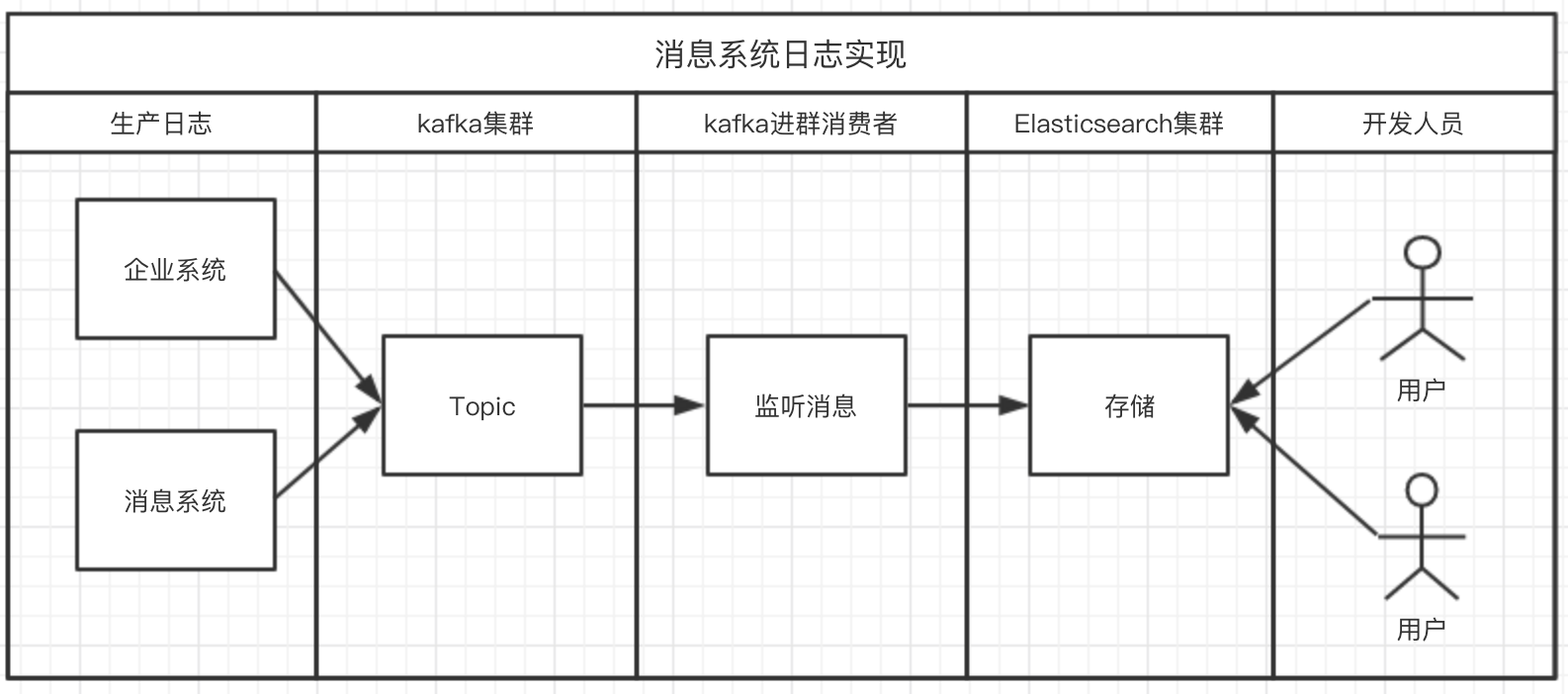


图4-7 消息系统日志实现

Figure 4-7 The Implementation Of Message System Logging

消息系统的日志模块作为一个单独的消息系统而独立处理，这样实现的优点有：

1. 将消息日志与消息解耦，降低消息系统复杂度
2. 消息系统可以作为一个单独的系统而存在，面向更多的其他企业系统，扩展性更好
3. 消息系统接入方式简单，只需在log4j2中配置KafkaAppender即可实现。

<!-- 输出info日志到Kafka -->

<Kafka name="KAFKA" topic="test">

<ThresholdFilter level="info" onMatch="ACCEPT" onMismatch="DENY" />

<PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss,SSS}:%4p %t (%F:%L) - %m%n" />

<Property name="bootstrap.servers">127.0.0.1:9092</Property>

</Kafka>

1. 不同系统产生的日志便于区分，在消息系统管理台申请对应系统的Topic就可以实现不听系统日志的分离
2. 消息日志的存储基于全文搜索引擎 Elasticsearch，可以实现快速的存储、搜索和海量数据分析。

日志中心中也使用了一个消息中间件—kafka，消息系统采用的ActiveMQ与kafka的区别如下表4-1所示：

表4-1 ActiveMQ与Kafka对比

Table 4-1 The Difference Between ActiveMQ And Kafka

|  |  |  |
| --- | --- | --- |
| 对比项 | ActiveMQ | Kafka |
| 规范 | 遵守JMS规范 | 未遵守JMS规范 |
| 性能 | 一般 | 高性能 |
| 吞吐量 | 万级 | 十万级 |
| 可用性 | 高（主从） | 非常高（分布式） |
| 消息丢失 | 低 | 理论上不会丢失 |
| 部署难易度 | 低 | 高 |
| 持久化 | 内存、文件、数据库 | 文件 |
| 消息顺序 | 保证发送时顺序 | 无序 |
| 事物 | 支持 | 不支持 |

如上表4-1所示，ActiveMQ支持的持久化类型较多，并且ActiveMQ可以支持事物，保证消息的有序性，基于以上这几点，消息系统采用的是ActiveMQ作为基础；而消息中心，往往面临的是大量的日志信息（毕竟要生成一个消息之前，可能要经过复杂的逻辑处理，这些处理过程都需要进行日志记录，以便故障的时快速定位问题），所以日志中心选择kafka作为基础，就是考虑到kafka的高吞吐量的特性，在企业级开发中，面对每天上百G甚至是T级别的日志时，kafka有较好的表现。

既然日志中心是面向企业系统运行日志的，那么对于每天产生的大量日志，Elasticsearch就可以发挥其作用了。

**4.4.2 Elasticsearch简介**

Elasticsearch之所以具有如此高的全文搜索的性能得益于其底层的数据存储结构——倒排索引（Inverted Index）。下面将对比倒排索引与正排索引的区别。

1. 正排索引（正向索引）

正排索引（forward index）是值当用户搜索某个关键词的时候，需要扫扫描索引库中的所有文档，找到包含关键词的文档后，将根据模型对文档的关联度进行打分，最终把数据展现给用户。正排索引如下图4-8所示：

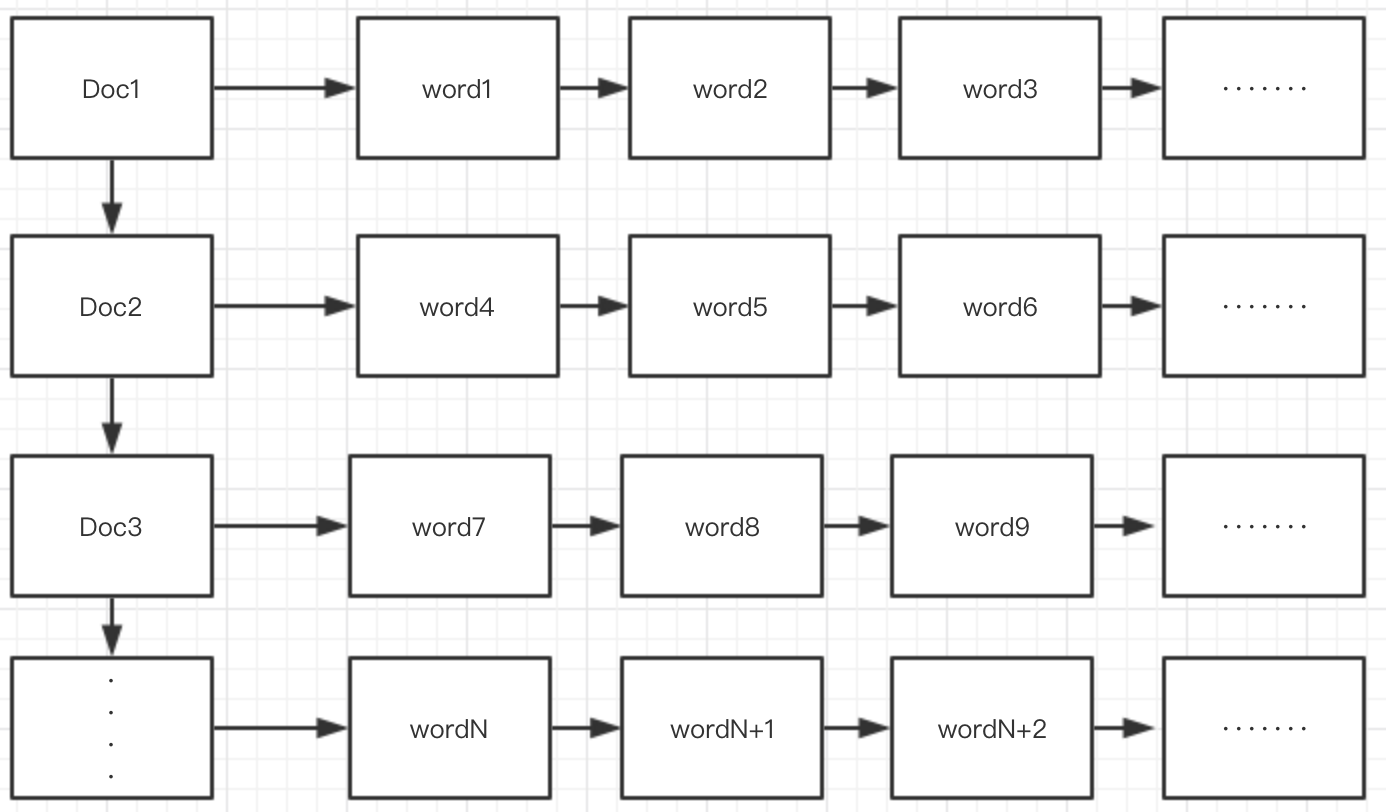


图4-8 正排索引图

Figure 4-8 Forward Index Diagram

正向索引的结构是：“文档1”的ID > 单词1：出现次数，出现位置列表；单词2：出现次数，出现位置列表；…………。

正向索引的弊端：当用户在主页上搜索关键词“四大名著”时，假设只存在正向索引（forward index），那么就需要扫描索引库中的所有文档，找出所有包含关键词“四大名著”的文档，再根据打分模型进行打分，排出名次后呈现给用户。因为互联网上收录在搜索引擎中的文档的数目是个天文数字，而且每个文档的内容也可能非常大（比如一部书，一篇论文等），这样的索引结构根本无法满足实时搜索的要求。

1. 倒排索引

倒排索引（Inverted Index）会按照指定语法（官网提供有各种语言的分词器）对每一个文档进行分词，然后维护一张表，列举所有文档中出现的分词以及它们出现的文档ID和出现频率。搜索时同样会对关键词进行同样的分词分析，然后查表得到结果。倒排索引如下图4-9所示：

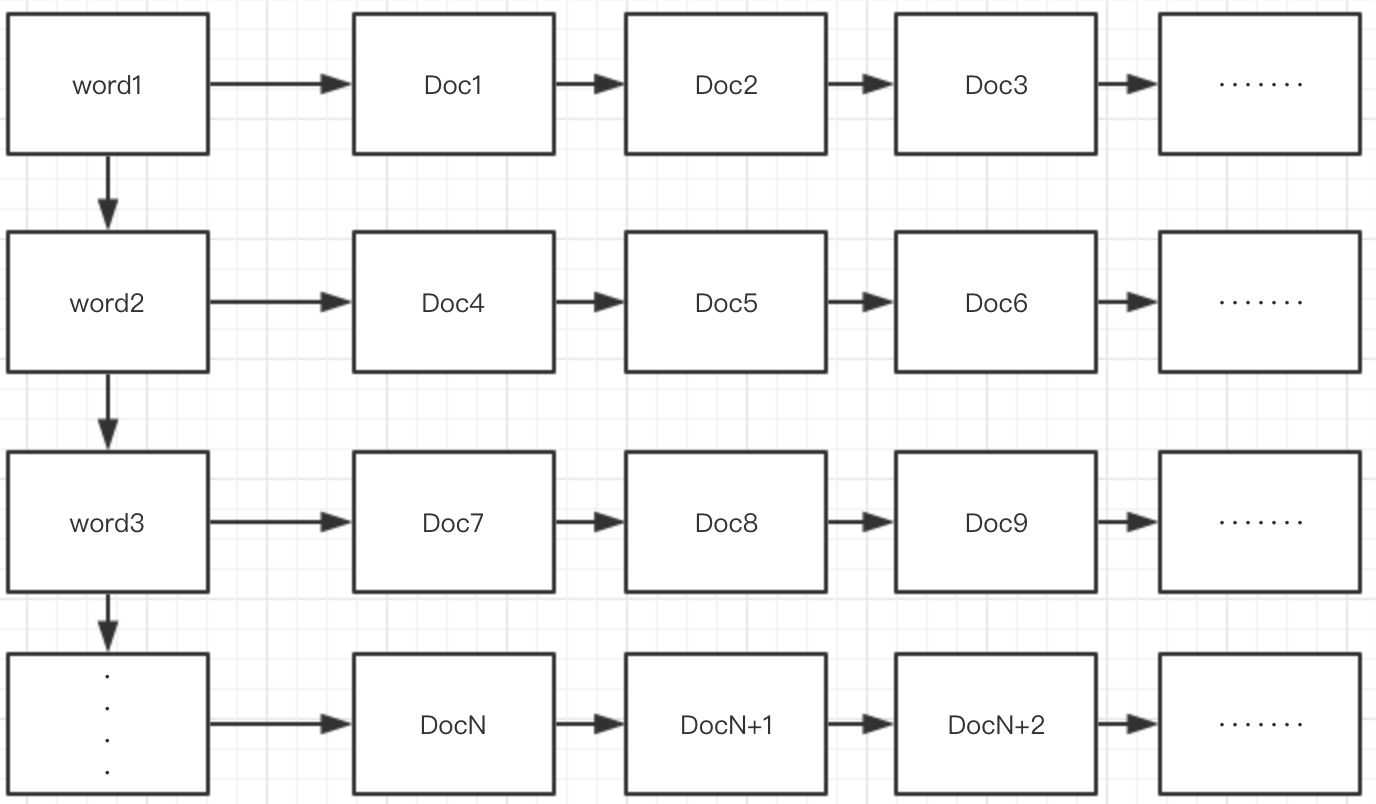


图4-9 倒排索引图

Figure 4-9 Inverted Index Diagram

Elasticsearch使用倒排索引，当输入某个关键词的时候，可以快速搜索到包含该关键词的文档，避免了像正排索引只扫描到了很多不必要的文件，从而达到了高效的查询。一个最简单的倒排索引的实例如下：

下面我们通过具体实例来进行说明，倒排索引的具体操作。假设文档列表含五个文档，每个文档内容如图4-10所示，在图中最左端一栏是每个文档对应的文档编号。我们的任务就是对这个文档集合建立倒排索引。



图4-10 示例文档列表

Figure 4-9 Demo Document List

文和英文等语言不同，单词之间没有明确分隔符号，所以首先要用分词系统将文档自动切分成单词序列。这样每个文档就转换为由单词序列构成的数据流，为了系统后续处理方便，需要对每个不同的单词赋予唯一的单词编号，同时记录下哪些文档包含这个单词，在如此处理结束后，我们可以得到最简单的倒排索引。如图4-11所示，“单词ID”一栏记录了每个单词的单词编号，第二栏是对应的单词，第三栏即每个单词对应的倒排列表。比如单词“谷歌”，其单词编号为1，倒排列表为{1,2,3,4,5}，说明文档集合中每个文档都包含了这个单词。



图4-10 示例文档倒排索引

Figure 4-10 Demo Documents Inverted Index

以上就是一个最简单的倒排索引的一个示范，Elasticsearch就是通过这样的存储方式，从而实现了高效的全文搜索。

消息系统的日志系统采用Elasticsearch作为存储，方便的存储大量的日志信息，同时也使得日志中心具有很好的扩展性，可实现高效的全文搜索。在企业级开发过程中，采用这样的架构的消息中心，可以轻松应对每天百G以上的日志存储，也方便开发在日志中心中根据关键词搜索日志，如想搜索发生异常的日志，可能只需要在消息中心中输入“Exception”或者“Error”这样的关键词就可以轻松的定位到对应的日志信息，更好的辅助开发人员快速定位到问题，节约了企业系统开发的成本。