|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 分类号 |  | |  | | | 密 级 | |  |
| U D C |  | |  | | | 编 号 | | 10486 |
|  | | | | | | | | |
| 武汉大学logo  **硕 士 专 业 学 位 论 文** | | | | | | | | |
|  | | | | | | | | |
| 分布式服务治理框架的设计与实现 | | | | | | | | |
|  | | 研究生姓名 | | ： | 王伟 | |  | |
| 学号 | | ： | 2014212113103 | |
| 指导教师姓名、职称 | | ： | 龚奕利 副教授 | |
| 专业类别（领域） | | ： | 计算机技术 | |
|  | |  |  | |
|  | |  |  | |
|  | | | | | | | | |
| 二〇一八年十月 | | | | | | | | |

Dissertation Submitted to Wuhan University

For Degree of Master Engineering

The Design and Implementation of the Distributed Service Governance Framework

By

Wang Wei

Under the Guidance of

Professor Gong Yi Li

October 2018

**郑 重 声 明**

本人的学位论文是在导师指导下独立撰写并完成的，学位论文没有剽窃、抄袭、造假等违反学术道德、学术规范和侵权行为，否则，本人愿意承担由此而产生的法律责任和法律后果，特此郑重声明。

学位论文作者（签名）：

年 月 日

# 摘 要

应用业务是不断发展变化的，业务创新越来越成为企业核心竞争力的一部分。传统IT支撑业务发展的方式是为新业务构建新的业务系统，系统多是“竖井状”的孤岛形式。久而久之，这样的方式已经无法满足业务发展对IT带来的要求，IT成 为企业的严重短板。面向服务的架构的出现的主要目的就是彻底改变这种局面，使IT变得灵活，与业务对齐。甚至能够促进业务的发展，并创造新的业务价值。本文研究了国内外的服务治理框架框架，分析和总结了通用的框架特性和非功能特性，实现了一号店的服务治理框架Hedwig。

首先，在服务治理概述中论述了服务化架构的演进，描述了服务治理的需求，阐述了服务框架的整体架构和技术选型。

其次,框架的特性包括功能特性和非功能特性。功能特性表示框架是什么，有哪些功能模块和组件。模块是细粒度的功能，组件是粗粒度的功能。后续章节给出了Hedwig每个模块和组件实现的原理和设计方案。

最后，对于服务治理框架的容错和监控寻错功能进行了研究。故障是分布式系统要面对的主要困难之一，非功能特性中的容错模式为框架提出了提高服务可靠性和可用性的方式，在此基础上设计了Hedwig的容错方案。服务监控系统Detector对异常服务调用的排查手段加强了整个治理平台的查错排错能力。

从功能上看，Hedwig已然是功能完善的服务框架，能给团队提供统一和快速的服务层接入。从非功能特性上看，Hedwig为公司的业务团队提供了一个支持并发，吞吐量，容错可靠性，可用性都很高的分布式服务框架。通过将本文所研究和设计的系统Hedwig在一号店全公司的推行，在这5年间Hedwig已经满足了绝大部分团队的功能需求和性能需求，简单快速的接入方式也极大的节约了公司的成本。

关键词: 服务化治理；分布式服务化框架；分布式服务模式；容错模式；远程过程调用

# Abstract

The application business is constantly evolving, and business innovation is becoming a part of the core competence of the enterprise. The way that traditional IT supports business development is to build a new business system for the new business, which is mostly an isolated form of "silo". Over time, this approach has failed to meet the demands of business development for IT, which is a serious shortboard for businesses. The main purpose of a service-oriented architecture is to completely change the situation and make IT flexible and aligned with the business. It can even promote business development and create new business value. This paper studies the framework of service governance framework both at home and abroad, analyzes and summarizes the general features of the framework and the function characteristic, implements the Hedwig number one shop service governance framework.

First, this paper discusses the architecture of service in the service management overview evolution, describes the demand of service management, expounds the overall architecture of the service framework and technology selection.

Second, the features of the framework include both functional and non-functional features. The feature section describes what the framework is and what the functional modules and components are. Modules are fine-grained functions, and components are coarse-grained functions. The following sections give you the principles and designs for each module and component implementation.

Finally, the fault tolerance of service governance framework and the monitoring and fault finding function were studied. Fault is one of the main difficulties faced a distributed system, the features of fault tolerant model for the framework puts forward the ways that improve service reliability and availability, on the basis of the hedwig fault-tolerant scheme is designed. The monitoring system of the service monitoring system has enhanced the search and error capability of the whole governance platform.

In terms of functionality, Hedwig is already a fully functional service framework that provides a unified and rapid service layer access to the team. Look from the non-functional properties, the hedwig team provides a support for the company's business concurrently, throughput, reliability and fault tolerance of high availability are distributed service framework. Through this research and design of the system the hedwig in one shop all the implementation of the company, in the five years the hedwig already satisfied most of the team's functional requirements and performance requirements, simple and quick way of accessing and greatly saves the cost of the company.

Key word：Service governance**;** Distributed service framework**;** Distributed service pattern; Fault-tolerant pattern；RPC

**目 录**

[摘 要 1](#_Toc528168436)

[Abstract 2](#_Toc528168437)

[1 引言 7](#_Toc528168438)

[1.1 研究问题的背景 7](#_Toc528168439)

[1.2 国内外研究现状 8](#_Toc528168440)

[1.3 研究的内容与方法 8](#_Toc528168441)

[1.4 概念和名称 9](#_Toc528168442)

[1.5 论文组织结构 10](#_Toc528168443)

[2 服务治理框架的功能和组成 11](#_Toc528168444)

[2.1 服务化架构演进和RPC 11](#_Toc528168445)

[2.2 服务治理需求和分析 12](#_Toc528168446)

[2.1 服务注册和发现 14](#_Toc528168447)

[2.1.1 功能设计 14](#_Toc528168448)

[2.1.2注册中心订阅发布机制 15](#_Toc528168449)

[2.3 服务路由和负载均衡 16](#_Toc528168450)

[2.3.1 静态负载均衡算法 16](#_Toc528168451)

[2.3.2 动态负载均衡算法 16](#_Toc528168452)

[2.3.3 服务路由的种类 17](#_Toc528168453)

[2.4 序列化和反序列化 17](#_Toc528168454)

[2.5 服务调用 17](#_Toc528168455)

[2.5.1 同步服务调用 18](#_Toc528168456)

[2.5.2 异步服务调用 18](#_Toc528168457)

[2.5.3 并行服务调用 19](#_Toc528168458)

[2.6 RPC通信框架 19](#_Toc528168459)

[2.6.1 IO模型 19](#_Toc528168460)

[2.6.2 可靠性设计 20](#_Toc528168461)

[2.7 服务发布和引用 20](#_Toc528168462)

[2.7.1 本地接口封装成代理 20](#_Toc528168463)

[2.7.2 服务提供者信息注册 21](#_Toc528168464)

[2.8 服务框架部署方式 21](#_Toc528168465)

[2.8.1 框架作为应用三方库 21](#_Toc528168466)

[2.8.2 框架自身作为容器 21](#_Toc528168467)

[2.9 协议栈 22](#_Toc528168468)

[2.10 本章小结 22](#_Toc528168469)

[3 服务治理框架容错原理和模式 23](#_Toc528168470)

[3.1 集群容错 23](#_Toc528168471)

[3.1.1 容错策略 23](#_Toc528168472)

[3.2 服务降级 24](#_Toc528168473)

[3.2.2 人工开关降级 24](#_Toc528168474)

[3.2.3 系统自动降级 25](#_Toc528168475)

[3.3 服务可靠性 25](#_Toc528168476)

[3.3.1 服务状态监测 25](#_Toc528168477)

[3.3.2 服务健康检查 26](#_Toc528168478)

[3.4 服务隔离 26](#_Toc528168479)

[3.4.1 服务故障隔离 26](#_Toc528168480)

[3.5 流量控制 28](#_Toc528168481)

[3.5.1 静态流控 28](#_Toc528168482)

[3.5.2 动态流控 29](#_Toc528168483)

[3.5.3 服务接口限流 29](#_Toc528168484)

[3.5.4 限流算法 29](#_Toc528168485)

[3.6 超时与重试机制 31](#_Toc528168486)

[3.6.1 框架的超时与重试 31](#_Toc528168487)

[3.6.2 最佳实践 31](#_Toc528168488)

[3.7 服务容错模式 32](#_Toc528168489)

[3.7.1 熔断器模式 32](#_Toc528168490)

[3.7.2 限流模式 33](#_Toc528168491)

[3.7.3 超时和重试模式 34](#_Toc528168492)

[3.7.4 回退模式 34](#_Toc528168493)

[3.7.5 舱壁隔离模式 34](#_Toc528168494)

[3.8 本章小结 34](#_Toc528168495)

[4 Hedwig服务框架设计和实现 35](#_Toc528168496)

[4.1 通信框架和协议栈 35](#_Toc528168497)

[4.2逻辑模块和时序图 36](#_Toc528168498)

[4.3 功能模块设计和实现 40](#_Toc528168499)

[4.3.1 服务注册和发现功能设计 40](#_Toc528168500)

[4.3.2服务路由设计 41](#_Toc528168501)

[4.3.3 动态负载均衡算法设计 42](#_Toc528168502)

[4.3.4 序列化和反序列化实现 43](#_Toc528168503)

[4.3.5 服务调用设计 44](#_Toc528168504)

[4.3.6 可靠性设计 46](#_Toc528168505)

[4.4 功能组件设计 46](#_Toc528168506)

[4.5 容错设计和实现 48](#_Toc528168507)

[4.5.1 Failover实现 48](#_Toc528168508)

[4.5.2 Failcache实现 50](#_Toc528168509)

[4.5.3 人工降级设计 51](#_Toc528168510)

[4.5.4 集群隔离设计 51](#_Toc528168511)

[4.5.5 线程隔离设计 52](#_Toc528168512)

[4.5.6 限流接口的总请求数设计 52](#_Toc528168513)

[4.5.7 限流接口的时间窗请求数设计 53](#_Toc528168514)

[4.5.8 超时与重试机制设计 54](#_Toc528168515)

[4.6 Hedwig监控 54](#_Toc528168516)

[4.6.1 基础设施监控 55](#_Toc528168517)

[4.6.2 应用级别监控 55](#_Toc528168518)

[4.6.3 服务监控 56](#_Toc528168519)

[4.7 本章小结 58](#_Toc528168520)

[5 Hedwig功能与性能测试 59](#_Toc528168521)

[5.1 测试环境 59](#_Toc528168522)

[5.2 服务调用测试 59](#_Toc528168523)

[5.2.1服务调用成功测试用例 59](#_Toc528168524)

[5.2.2 服务提供者执行异常测试用例 60](#_Toc528168525)

[5.2.3 服务提供者处理超时测试用例 60](#_Toc528168526)

[5.3 集群与负载均衡测试 60](#_Toc528168527)

[5.3.1测试目的和过程 60](#_Toc528168528)

[5.3.2 测试结果 61](#_Toc528168529)

[5.4 性能测试 61](#_Toc528168530)

[5.4.1 测试目的和过程 61](#_Toc528168531)

[5.4.2 测试结果 62](#_Toc528168532)

[5.5 本章小结 63](#_Toc528168533)

[6 总结和展望 64](#_Toc528168534)

[参 考 文 献 65](#_Toc528168535)

[致 谢 68](#_Toc528168536)

# 1 引言

### 1.1 研究问题的背景

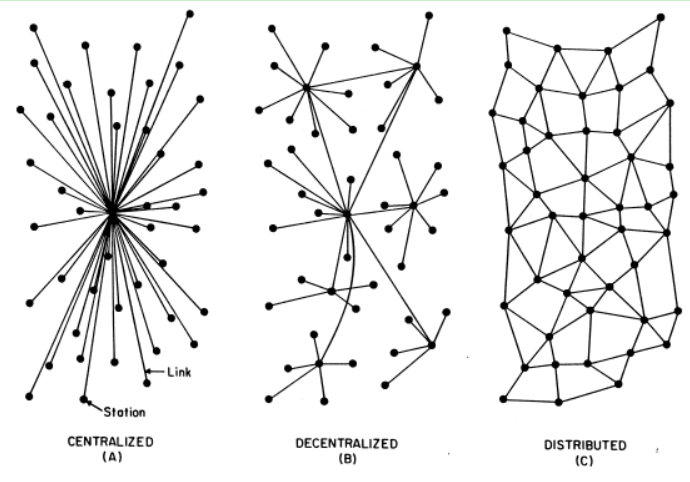
近年来，各大互联网公司都遇到了业务快速发展的情况,访问流量也是日益递增。因此，设计一个高可用、高性能且伸缩性强的网站架构逐步成为互联网领域关注的重点。经过研究发现，网站架构已经逐渐从集中式向分布式应用架构转变, 当今大型网站基本上是分布式设计的，如图1.1所示。

图 1.1 网站架构演进

分布式应用架构的核心思想之一是通过增加服务层来使远程服务访问透明化。服务之间通过抽象接口来交互，屏蔽底层实现的复杂度。如果把系统拆分成多个服务，分离服务边界和责任，也便于独立升级和维护系统。分布式服务化治理框架就是开发和管理服务层的重要方法和工具。

目前，在互联网公司中，成熟的服务治理框架非常多，大多没有开源。但也有少部分优秀的已经开源，并且被很多互联网公司直接使用。各家公司的服务框架都有自己的实现方式，不一而足,各有千秋。

服务层的上层是各种应用，承载了公司几乎所有的业务， 所以服务的稳定可靠性和性能，也会直接影响到互联网用户的用户体验。他山之石可以攻玉，借鉴优秀的开源框架并结合公司自身的特点，设计并实现一个通用分布式服务治理框架具有重要意义。

### 1.2 国内外研究现状

早在1996年Gartner第一次提出了SOA的概念，它认为未来SOA(Service-Oriented Architecture)将会成为软件工程的的重要方法[40]。曾经Web Service是企业应用系统开发领域构建分布式系统常用的技术。其中提出了一系列的标准如WSDL、UDDI、SOAP，并围绕这些规范开发了成熟的产品。但其缺点有很多，并不能满足大型应用高性能、高可用、易部署维护的要求。

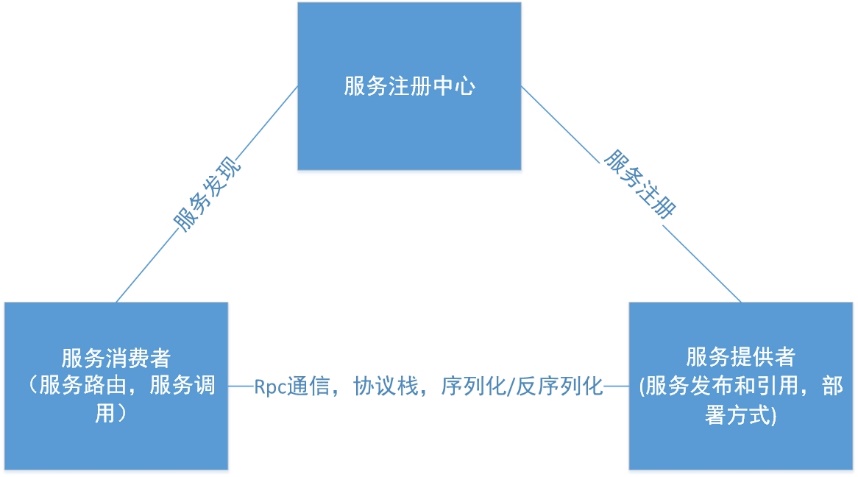
如今,在大规模的互联网公司内都会自己研发基于SOA理念的产品。从2010年开始， 国内各个互联网公司都开始进行了整体服务化的进程，其中沉淀了不少优秀的基于RPC(Remote Procedure Call)的服务化治理框架。其中包括淘宝的Dubbo[25]、大众点评的Pigeon、网易的Motan[35]、京东的JSF[17]，唯品会的OSP、一号店的Hedwig等；国外服务框架有Thrift、Finagle、gRPC等。服务框架主要有两个侧重方向，一个偏重于跨语言调用，另一个偏重于服务治理。

跨语言调用RPC[36]框架以Thrift为例，Thrift提供了IDL(Interface Description Language)中立语言，可以支持多种语言,有很好的平台中立性， 国内很多公司在Thrift框架之上开发了自己的服务框架，其中包括OSP和Pigeon。

服务治理型的服务框架的特点是功能丰富，适用于大型应用的服务化拆分和服务管理。淘宝Dubbo的有轻量的部署方式，并且功能全，扩展性极好，很方便二次开发，可运行在任何Java环境中。Dubbo使用微核加插件体系的方式来做扩展，即所有功能都可以在不修改原生代码的情况下在外围扩展。微博的 Motan功能和 Dubbo 类似，可以认为是一个轻量裁剪版的 Dubbo，并拥有服务健康检查和动态流量调整功能[15]。

### 1.3 研究的内容与方法

本文的主要研究目标是设计和实现服务框架Hedwig来为业务团队提供简洁、高效、灵活的服务接入方式及完善的服务治理功能，以提升业务开发的生产力。从整体上看，服务框架可以分为功能性特性和非功能性特性。功能特性表示框架是什么，是做什么的。组件是粗粒度的功能，模块是细粒度的功能。框架可以由几个粗粒度的组件组合在一起，并且可以把这些组件的职责抽象到模式。框架也可以由许多细粒度的模块组合在一起。组件是粗粒度的，组件会包括细粒度的模块。

图 1.2 服务框架功能

研究的Hedwig功能特性包括以下几点：服务注册和发现、服务路由(软负载均衡) 、序列化和反序列化、服务调用、RPC通信框架、服务发布和引用、部署方式、幂等。这些特性和服务消费者、服务提供者和服务注册中心三者的关系，如图1.2所示。其中服务路由、服务调用是消费端实现的功能;服务发布/引用、部署方式是服务提供者实现的功能;RPC通信框架、协议栈、序列化/反序列化、服务注册/发现是消费者和提供者两边共同实现的功能。

Hedwig非功能特性研究的是系统提供的容错可靠性、稳定性、性能方面的问题。网络本身是不可靠的，远程接口可能响应慢或者宕机。根据“Design for failure[11]”的理念，容错模式提供了通用的解决方案，在服务框架设计和实现时有良好的参考价值。

在Hedwig提供服务的时候，同时Detector作为服务治理中心负载监控服务。Detector本身也是个平台，利益相关人员通过Detector对服务调用的历史、当前的运行状态、健康度和服务依赖关系等各个指标维护和分析，能够在符合当前高质量运行的情况下持续优化服务,或者能在服务调用有异常的时候排查问题。

### 1.4 概念和名称

定义：

Domain表示一号店的部门;

Pool表示一号店的子部门;

App是一号店包含业务的应用;

Hedwig 是一号店分布式服务框架;

Detector是一号店分布式服务监控和治理平台;

TPS是每秒事务处理量(Transaction Per Second);

QPS是每秒查询处理量(Queries Per Second);

Netty是异步化、事件驱动的网络应用框架;

Tomcat是Web应用的容器,服务可以在容器中运行;

RPC是远程过程调用(Remote Procedure Call),是一种进程间通信方式。

### 1.5 论文组织结构

本文余下章节的组织结构如下：

第二章节主要是介绍服务治理框架的功能和组成。第三章节介绍服务框架非功能特性,包括容错原理和容错模式。第四章介绍Hedwig服务框架详细设计和实现。第五章介绍Hedwig功能和性能测试。第六章是总结与展望，总结现阶段工作的内容和不足，展望未来研究的重点。

# 2 分布式服务的关键技术

### 2.1 服务化架构演进和RPC

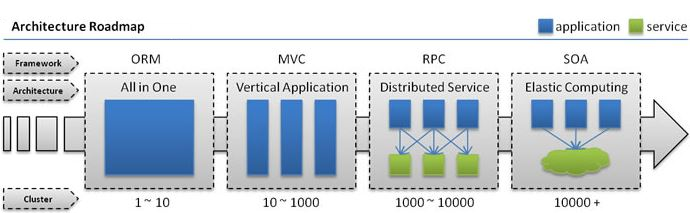
随着互联网的发展，常规的烟囱式垂直应用架构已无法应对网站应用规模不断扩大的情况，所以分布式服务架构以及弹性计算架构势在必行，亟需一个治理系统确保应用架构有条不紊的演进。

图2.1 面向服务架构的演进路线图

根据服务的个数从小到大和复杂度从简到繁来分，服务化大致可以分成如下4种架构，如图2.1所示：

### 2.1.1 单一应用架构

网站初期流量较小时，只需一个应用将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架是关键。

### 2.1.2 垂直应用架构

当网站访问量逐渐增大，单一应用访问速度会越来慢，将应用拆成互不相干的几个应用以提升效率。此时，用于加速前端页面开发的Web框架是关键。

### 2.1.3 分布式服务架构

当垂直应用越来越多，应用之间交互和依赖就不可避免，所以将核心业务抽取出来作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的需求。此时，用于提高业务复用的分布式服务框架(RPC)是关键。

### 2.1.4 弹性计算架构

当服务数量越来越多，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度平台和服务治理中心(SOA)是关键。

基于分布式服务架构和弹性计算架构的服务实例数一般在千到万的数量级别。

分布式服务框架提供了高性能和透明化的RPC(远程服务调用)方案，分布式服务框架亦称做分布式服务中间件。RPC[36]概念术语在上世纪 80 年代提出，Bruce Jay Nelson指出一个最简单的RPC框架应该包含以下几个部分。如图2.2，当 Client 想发起一个远程调用时，实际是通过本地调用 Client-stub，而 Client-stub 负责将调用的接口、方法和参数通过约定的协议规范进行编码并通过本地的 RPC-Runtime 实例传输到远端的实例。远端 RPC-Runtime 实例收到请求后交给 Server-stub 进行解码后发起本地端反射调用，调用结果再返回给 Client 端。

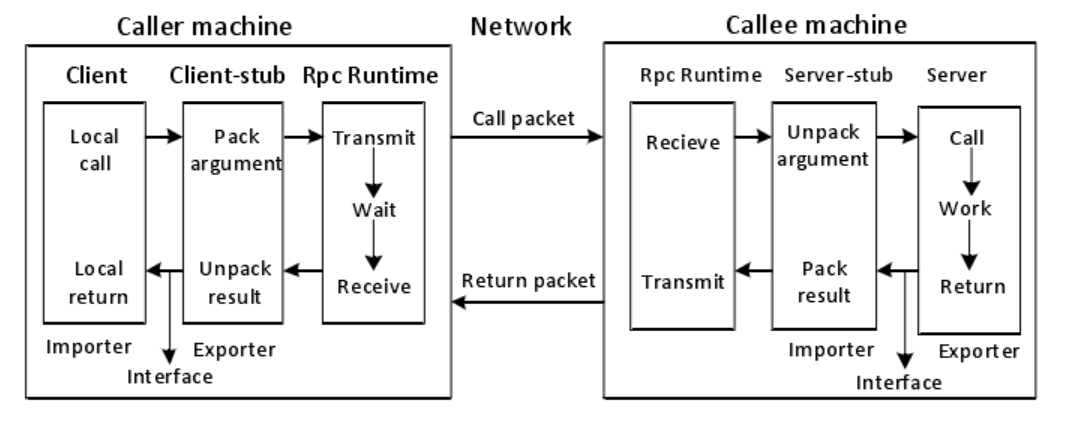


图2.2 RPC原理

### 2.2 服务治理需求和分析

1.在大规模服务化之前，应用可能只是通过RMI等工具简单的暴露和引用远程服务，通过配置服务的URL地址进行调用，通过硬件进行负载均衡。

当服务量越来越多时，服务URL配置管理变得非常困难，硬件负载均衡器的单点压力也越来越大。此时需要一个服务注册中心，动态的注册和发现服务，使服务的位置透明。然后通过在消费方获取服务提供方地址列表，实现软负载均衡和Failover功能。所以，服务框架中需要服务注册与发现和软负载均衡模块。

2.当服务的调用量越来越大，服务的扩容问题就暴露出来，这时需要确定使用的机器数量和扩容的时间点。

为了解决这些问题，第一步，要将服务现在每天的调用量、响应时间统计出来作为容量规划的参考指标。其次，要可以动态调整权重，在生产环境将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阀值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

所以，为了做服务容量评估，依次需要服务监控与统计，然后作服务使用情况报告，最后就可以把服务权重动态调整。

3.服务应用一般是一个后台Daemon程序，不需要Web容器。如果以Web应用为主，就不得不把服务运行在Web容器里。所以服务部署方式也是一个要考虑的功能点，需要实现一个非Web的容器。

4.确保服务运行时质量是关键问题。当关键应用和非关键应用同时调用同一个服务，这个服务就应该向关键应用倾斜，当服务支撑不住时，应限制非关键应用的访问，保障关键应用的可用性。这时就需要服务路由，控制不同应用访问不同机器。所以，服务框架中需要服务路由的功能模块。

5．在实际生产中会遇到如下的场景，当服务突然多了一个消费者，这个消费者的请求量直接把整个服务给拖跨，其它消费者跟着一起出现故障。

所以服务提供方需要流控，当流量超标时，服务提供方能拒绝部分请求来进行自我保护。 所以，服务流控是个容错特性，能提高服务的稳定性。

6.在服务的不停升级中故障不可避免，核心服务宕机影响面很大，如何控制故障的影响面，服务是否可以功能降级是需要考虑的问题。

通过应用间声明依赖强度，然后基于依赖强度来计算出影响面，并定期测试复查，加强关键路径上的服务的优化和容错，清理不该在关键路径上的服务。

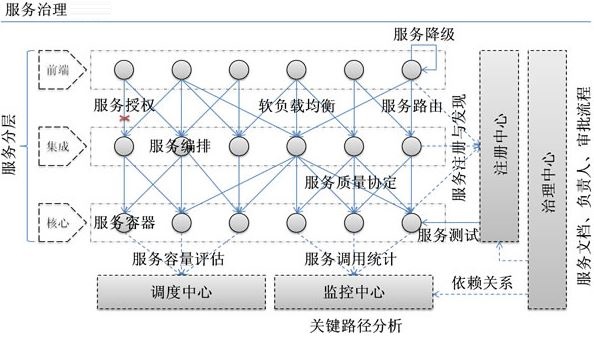
所以，服务框架需要服务降级的功能，具体可以实现为服务伪装(Mock)容错。

7.有些情况下并不是所有服务的访问量都大，很多的服务都只有很小的访问量，却需要部署两台提供服务的机器来进行高可用互备。如何减少浪费的机器，此时可能需要让服务容器支持在一台机器上部署多个应用。隔离的机制可以保护服务实例之间不相互影响。所以，隔离是个容错特性，并能提高服务的稳定性。

第1-4点为典型的服务治理功能性需求，对应为服务框架的功能特性，主要在本章论述介绍。第5-7点为典型的服务治理非功能性需求，对应为服务框架的非功能特性，主要在第三章论述介绍。

如图2.3所示为服务治理的分层结构，服务框架前端层包括服务消费者，服务框架核心层包括服务提供者。服务治理的重要外围系统包括服务注册中心，监控中心和调度中心。服务注册中心主要存储服务提供者的元数据，监控中心监控服务运行时的状态，调度中心在容量评估后做服务的缩容或者扩容。

服务治理以服务框架为主，服务监控为辅。监控服务后，可以对运行时的服务做出动态变更，比如权重的变更。也就是说服务监控后的结果可以反作用于运行时的服务，服务的运行时状态再在监控中展示，如此形成闭环。

图2.3 服务治理的分层结构

分布式服务框架可以分为功能特性和非功能性特性，功能特性如表2.1所示，下面就逐一分析各个功能和模块。

表2.1 分布式服务框架功能特性

|  |  |
| --- | --- |
| 角色/关系 | 功能 |
| 客户端 | 1. 服务路由   2. 服务负载均衡  3. 服务调用  4. 服务发现 |
| 服务端 | 1. 服务发布  2. 服务注册  3. 服务部署方式 |
| 客户端和服务端的联系 | 1. 通信框架RPC  2. 序列化/反序列化  3. 通信协议 |

### 2.1 服务注册和发现

服务提供者发布服务后，服务消费者最关心的是如何获取到它所需要的服务。

如何有效的管理服务的订阅/发布，避免在服务消费者中硬编码服务提供者地址信息是分布式服务框架需要解决的一个问题。 通过将服务统一管理起来，可以有效的优化内部应用对服务发布/使用的流程和管理，服务注册中心就是专门用来管理服务订阅/发布的配置管理平台。

### 2.1.1 功能设计

如图2.4所示，服务框架中可以分为以下几种角色:

Provider:提供服务方称之为“服务提供者”, 服务提供者提供业务逻辑的远程接口。

Consumer:调用远程服务方称之为“服务消费者”，调用服务提供者提供的远程接口。

Registry:服务注册与发现的目录服务称为“服务注册中心”，存储服务提供者列表。

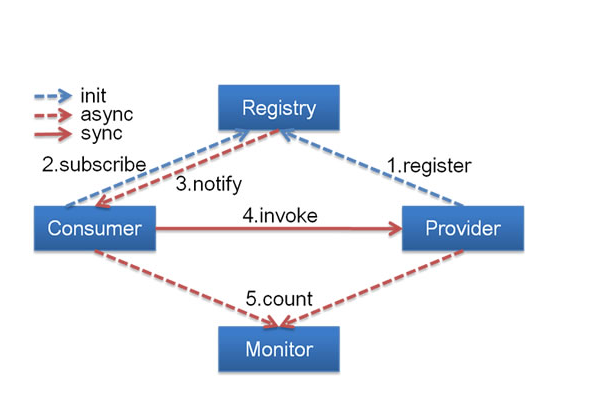
Monitor:统计服务的调用次数和调用时间等日志服务称为“服务监控中心”。

图2.4 分布式服务框架中的角色

注册中心、服务提供者、服务消费者三者之间均为长连接。注册机制原理如图2.4所示：

1. 服务提供者启动时，依据配置的服务发布信息向注册中心注册自身提供的服务。

2. 服务消费者启动时，根据消费者配置的服务消费信息向注册中心订阅自己所需的服务，消费者在本地保存路由表。

3. 注册中心返回服务端地址列表给服务消费者。如果有变更，注册中心主动推送变更给服务消费者，并刷新本地的路由表。

4. 服务消费者从本地的服务端地址列表中，基于某种负载均衡算法选择一个服务端进行调用。

### 2.1.2注册中心订阅发布机制

消费者可以监听一个或者多个服务目录，当目录名称或内容发生变更时，消费者可以实时的获得变更的数据。服务提供者可以发布一个或多个服务，动态修改服务名称或服务内容等，可以主动将修改后的数据推送给所有监听此服务目录的消费者。如果提供者宕机，由服务注册中心实时通知消费者。

注册中心宕掉后仍能通过缓存提供服务列表查询，但不能注册新服务。如果注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯。

### 2.3 服务路由和负载均衡

服务路由是指将服务实例部署在集群中，当服务请求到达时，根据配置的路由规则选出合适的请求处理者，这个过程即为服务路由。分布式服务框架要能够满足用户灵活的路由需求，负载均衡策略是服务路由的关键部分。分布式服务框架通常会提供多种负载均衡策略供用户使用。如下是几种常见的负载均衡算法：

### 2.3.1 静态负载均衡算法

1. 一致性哈希算法:

相同请求总是发到同一提供者。

2. 顺序轮询算法(Round Robin):

顺序循环访问服务提供者列表，到达上限后，重新归零，继续顺序循环访问。

3. 基于权重的轮询算法(Weight Round Robin):

基于权重的轮询算法,按公约后的权重设置轮循比率。

4. 广播算法(Broadcast)：

调用幂等的多个方法,如果其中有一个返回值，放弃调用其他方法的返回值。

### 2.3.2 动态负载均衡算法

1. 服务调用时延算法

客户端缓存所有服务端的服务调用时延, 周期性的计算服务调用平均时延，

然后计算每个服务端服务调用时延与平均时延的差值，根据差值大小动态调整权重，保证服务时延大的服务端接收更少的消息。自动的动态权重调整使服务调用时延接近平均值，实现负载均衡。

1. 最少连接数算法(Least Connection)

首先找到当前连接数最小的提供者，然后把新的连接请求分配给它。

### 2.3.3 服务路由的种类

如果机器配置低权重就低，请求量应该相应少，相反，机器配置高权重就高，请求量应该相应多，这时可以使用基于权重的路由；如果某个服务只能调用某几个机器，类似白名单访问机制，这称作基于IP的路由；基于分组的路由会自动根据配置调用某个服务分组内的服务，做到与其他分组的隔离；基于参数路由会根据接口参数访问不同的节点或者根据方法名作读写分离；如果使用基于机房的路由，同机房因为服务响应延迟低而被优先调用。

### 2.4 序列化和反序列化

进行RPC调用时，需要使用特定的序列化[21]技术，对网络传输的对象做编码或者解码。序列化指将对象编码为字节数组；反序列化会把从网路磁盘等读取的字节数组还原成原始对象。

序列化与通信框架不是强耦合的关系，通信框架提供的编解码框架可以非常方便的支持用户通过扩展实现自定义的序列化格式，用户可以在应用程序及其他位置实现对象的序列化和反序列化。

序列化与通信协议是解耦的，同一种通信协议可能由多种序列化方式承载，同一种序列化方式也可以用在不同协议里。比如，HTTP协议可以用XML/JSON来承载消息体，也可以是图片附件等二进制流媒体协议。序列化/反序列化一般分为两种，基于文本或者二进制。

在设计序列化/反序列化框架的时候，需要从功能、多语言支持、兼容性、性能等多个角度进行综合考量。框架功能包括支持的数据结构种类，接口是否友好简洁。一个好的序列化框架应该支持数据结构的前向兼容性，例如能支持新增/删除字段，调整字段顺序等。还有，衡量序列化框架通用性的一个重要指标就是对多语言的支持。

### 2.5 服务调用

服务消费者调用服务提供者的方式主要有同步和异步调用，同步调用往往是默认的调用方式。相比于同步调用，异步调用把串行调用化为并行调用，缩短业务线程阻塞的时间，能减少服务的整体响应时间。

当几个服务组合成一个业务服务时，并行服务调用比串行服务调用更能减少业务端到端的响应时间，并能提升整个系统吞吐量。

### 2.5.1 同步服务调用

分布式服务框架默认都需要支持同步服务调用。同步方式调用服务时，客户端发起远程服务调用，用户线程做完消息序列化后，将消息投递到RPC框架，然后同步阻塞，等待通信线程发送请求并接收到应答后，唤醒同步等待的用户线程，用户线程获取到应答后返回。

### 2.5.2 异步服务调用

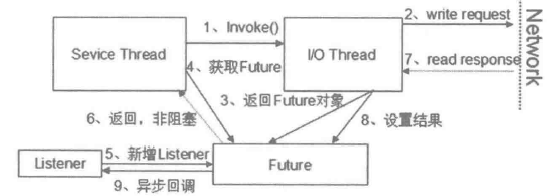
1. 单向(One-way)调用方式

单向调用是只管发送请求却不关心返回结果的方式，是单方向的通知。如果在NIO方式下使用，会比同步调用简单。

2. Future方式

Future方式包括被动异步回调通知和主动获取，适用于不同的业务场景。在工程中会提供Future-Listener机制，Future-Listener机制扩展了Future，例如Guava ListenableFuture实现异步调用。

被动异步回调通知方式，如图2.5所示

图2.5 Future-Listener被动调用机制

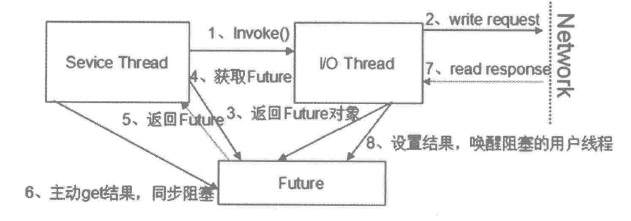
主动获取方式不添加监听器，如图2.6所示。 业务线程会阻塞，但相比同步服务调用，这种方式的阻塞时间更短。用户连续发起多次服务调用，然后顺序从RPC Context中获取Future对象，最终再主动获取结果。

图2.6 异步服务调用主动get结果原理图

1. 回调方式

请求方发送请求后会继续执行自己的操作，等到服务方有响应后进行一次回调。因为回调本身的代码执行时间久了会影响IO线程或者定时任务，所以还是建议用新的线程来执行回调。

回调方式是被动的，是真正异步的方式，回调的执行不是在原请求线程中；而Future方式能主动控制超时和获取结果，它的执行还是在原请求线程中。

### 2.5.3 并行服务调用

可并行调用的条件有如下2点：一个长流程业务中调用几个服务，并且对时延比较敏感，其中有些逻辑无上下文关联。多个服务没有相互依赖关系，没有严格要求的执行先后顺序。

并行服务调用的原理如图2.7所示：同时发起多个调用，流程fork后，再利用Future主动等待获取结果，进行结果汇集。实现并行服务调用的常用方案是批量串行服务调用。

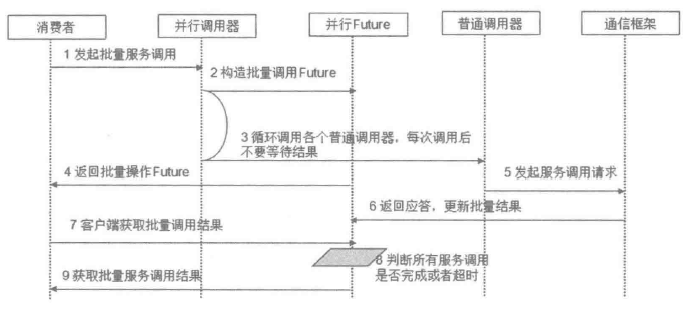


图2.7 批量服务调用原理图

### 2.6 RPC通信框架

本地方法调用变成远程服务调用之后，一个高性能的通用通信框架将成为服务框架必不可少的组成部分。IO模型是通信框架性能是否好的关键点。

### 2.6.1 IO模型

1. BIO：同步阻塞模式的IO模型，其性能和可靠性有着巨大的瓶颈。服务端的线程数和客户端数是1比1的关系,所以BIO缺乏弹性伸缩能力。

2. NIO：I/O多路复用技术把多个I/O的阻塞复用到同一个select的阻塞上。系统开销小是I/O多路复用技术的优势。NIO的非阻塞模式可以开发高负载，高并发的网络应用。

3. Actor：全异步化事件驱动。

通用并发编程模型中Actor模式是一种适用性很好的模型。Actor能提供很好的容错性，适合解决分布式的问题，也可以应用于共享内存架构和分布式内存架构。Actor模型保留了可变状态，但不共享可变状态。Actor封装了状态，并通过消息与其他Actor通信。所有Actor可以同时运行，Actor之间的消息传递是真实的传递消息。

### 2.6.2 可靠性设计

远程调用往往是不可靠的，网络丢包和阻塞也不是鲜见之事，所以设计出可靠的底层通信框架才能保证服务框架的可靠性。一般会采取以下两种方式提高可靠性。

1. 链路有效性检测

周期性的对链路进行有效性的检测可以解决链路的可靠性问题。目前通用的实现方式是用心跳检测。心跳检测目的是确认对端存活并且能够正常收发消息。心跳检测机制有请求-响应型和双向心跳型。

心跳超时后要关闭链路，然后由客户端发起重新连接的操作，保证链路能恢复到正常的状态。

2. 重连机制

客户端检测到链路中断后，等待t时间，会发起重连操作，如果重连后还是失败，间隔周期t后再发起重连，直到重连成功。

在首次断连时客户端需要等待t时间后再重连，而不是失败后就马上重连，这是为了保证服务端有足够的时间释放资源。任何情况下的重连失败，客户端都要保证自身的资源被及时释放，以防止内存泄漏的情况。

### 2.7 服务发布和引用

服务发布和引用是指业务定义的抽象接口发布（Export）为服务的过程。服务提供者需要通过XML配置、注释、API调用等方式，把本地接口发布成远程服务；对于消费者，可以通过对等的方式引用远程服务提供者。

XML配置化对业务代码零侵入，扩展和修改方便，修改配置不需要重新编译代码。采用XML配置化的方式发布服务是最佳选择，所以大多数服务框架推荐采用配置化方式发布服务。

### 2.7.1 本地接口封装成代理

框架根据导入的服务端接口API和服务应用信息，生成远程服务的本地动态代理对象；它负责将本地的API调用转换成远程服务调用，然后将结果返回给调用者，见图3.1中的Client-stub。

### 2.7.2 服务提供者信息注册

服务注册要在服务发布的最后环节进行。一旦服务地址信息注册到服务注册中心，消费者就能够获得地址信息并发起请求调用；如果此时提供者还没有完成初始化，就会导致服务调用失败。

服务注册的结构有多种方式，例如按照主机地址、按照服务名或者URL。

### 2.8 服务框架部署方式

在运行期服务框架与应用和容器的关系也称为服务框架的部署方式。一般有以下两种主流的部署方式：

### 2.8.1 框架作为应用三方库

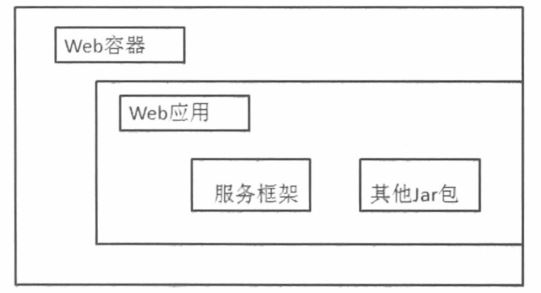
如图2.8所示：分布式服务框架与应用一起打包,它作为应用的一个依赖包。这种情况下，服务框架变成了应用的一个三方或者二方库，并和应用一起启动。

图2.8 框架作为应用三方库

### 2.8.2 框架自身作为容器

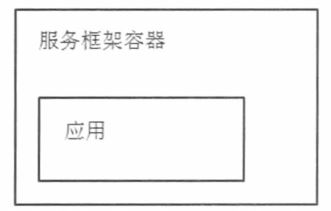
****如图2.9所示：有的情况下应用不是Web应用，或者不使用现有容器，那么服务框架自身需要变为一个容器来承载远程服务。

图2.9 服务框架本身作为容器

### 2.9 协议栈

大部分服务框架都支持多协议，不同服务在性能上适用不同协议进行传输。比如在公网对接异构第三方服务时，通常会选择HTTP/REST等共有协议；对于公司内部不同应用之间的服务调用，往往会选择性能较高的二进制私有协议。私有协议往往可以根据业务的具体需求进行针对性优化，因此性能更优。

私有协议定义了通信模型和消息模型，并支持点对点长连接通信等功能。通常私有协议栈的消息模型分为消息头和消息体。消息头存放协议公共字段、服务调用相关的公共参数和用户扩展字段，消息体则用于承载消息内容。如果没有特殊需求，分布式服务框架默认使用性能更高、扩展性更好的私有协议(二进制)进行通信。

如果是开源的通用分布式服务框架，考虑到通用性往往会提供一些常用的协议，例如Web Services和HTTP。对于Web Services，HTTP协议缺少服务描述文件、服务订阅发布机制，不太适合作为SOA服务化的标准协议。公有协议在序列化性能和连接性能方面普遍都比私有协议差。

需要指出的是，服务框架需要具备通过扩展的方式支持多协议的能力，协议栈应该作为一个架构扩展点开放出来。

### 2.10 本章小结

本章讲述了分布式服务框架各个功能模块实现的原理。

# 3 服务治理框架容错原理和模式

在互联网公司，一旦服务规模超过1万台服务器和5万个磁盘后，一天内都会发生多次故障。James Hamilton在2007年提出了Design for failure[11]的观点，指出大规模网络中的各种服务和硬件的不可靠事件都会发生。国内外宕机事件层出不穷，网络闪断、网关升级、访问量过大、误删数据库、挖断电缆等原因都可能造成服务不可用或者宕机。所以，设计高可靠性、可容错的服务框架是重要的考虑点。

### 3.1 集群容错

集群中服务调用失败后，服务框架需要能够在底层自动容错，上层应用对下层错误是无感知的，这样就提高了框架的可用性。容错策略很多，分别适用于不同场景。

### 3.1.1 容错策略

1. 失效转移(Failover)

Failover是指当发生远程调用异常时，重新选择查找路由到下一个可用的节点。

Failover策略的设计思路如下:消费者路由操作完成后， 获得目标地址，调用通信框架的消息发送接口发送请求，监听服务端应答。 如果返回的结果是RPC调用异常（超时，流控，解码失败等系统异常），根据消费者集群容错的策略进行容错路由，如果是Failover，则重新返回到路由handler的入口，从路由节点继续执行。 重新路由选择后，对目标地址进行比对， 防止重新路由到故障服务节点，过滤掉上次的故障服务端之后，调用通信框架的消息发送接口发送请求消息。

2. 失败缓存(Failcache)

Failcache是失败自动恢复的一种策略。对时间延迟要求不是很高的服务、必须发送到指定的服务端、通知类型的服务的这样3种场景适合Failcache机制。

服务调用失败通常是链路暂时不可用、服务流控、fullgc导致服务进程暂时夯住。当遇到以上这种情况，服务框架将消息缓存，等待周期t后，重新发送，直到服务端正常处理该消息为止。

Failcache中缓存对象要设置上限来防止内存溢出。Failcache中要限制定时重试的周期t，重试的最大次数等，并支持用户指定。重试达到最大上限任失败时，需要丢弃消息并记录异常日志。

3. 快速失败(Failfast)

快速失败是指业务高峰期如大促，对于非核心的服务只调用一次，失败后不再次调用，为核心服务节约了宝贵的资源。快速失败的设计很简单，拿到调用异常后，忽略异常，只记录异常日志。

### 3.2 服务降级

当访问量激增、非核心服务影响到核心流程的性能、服务出现问题时，仍然需要保证服务还是可用的，即使是有损服务。这时，可以根据一些关键指标进行自动降级，或者配置开关作人工降级。降级通常会放弃关闭非核心服务，最终目的是保证核心服务和关键主流程可用。

### 3.2.2 人工开关降级

流量高峰时如果通过监控发现某些服务存在问题，这时需要将这些服务暂时关闭。如果服务之间存在激烈竞争，导致核心服务不稳定，可能会影响系统的用户体验。弱依赖[23][24]的服务在大流量下停止调用.这时要对非核心服务做强制降级，不发起远程服务调用直接返回空值，减少自身对公共资源的消费，把资源释放出来供核心服务使用。

有时发现突然调用量太大，可能需要同步转换为异步调用，此时可以使用开关来完成降级。另外，有时候不太明确该服务的逻辑是否正确，但又想让新开发的服务在线上作灰度测试，这时，可以设置降级开关，当新服务有问题时可以通过开关切换回老的正常的服务。还有多机房的情况，如果某个IDC机房断电不可用了，则需要将这个机房的服务流量整体切到另一个机房，此时，可以通过开关完成切换。功能降级时也可以设置开关，暂时屏蔽掉某些功能。

人工降级操作是可逆的，当系统压力恢复正常水平，可以对已经人工降级的服务恢复正常。恢复之后，客户端重新调用远程的服务端，同时服务状态被修改为正常状态。

我们需要通过配置方式来动态开启/关闭降级开关。降级开关可以存放到配置文件、数据库或者Zookeeper[34].如果涉及的服务器/系统较少，则初期可以考虑使用配置文件进行配置。如果涉及的服务器/系统较多，则应该使用配置中心进行配置，这样可以集中管理配置开关。实现时要做到不需要修改代码，不需要重启应用即可动态配置开关。

### 3.2.3 系统自动降级

系统自动降级也称为容错降级.容错降级可用于业务放通，也常用于服务端在客户端执行容错逻辑，容错逻辑主要有RPC异常和Service异常。RPC异常一般指超时异常，消息编解码异常，流控异常等。Service异常指一般的业务异常。

系统自动降级大致可以分为以下4类：

1.自动开关降级

自动降级是根据系统负载等资源使用情况进行降级。

2.超时降级

当访问的远程服务调用响应慢，且该服务是非核心服务的话，在超时后可以自动降级。在实际应用中，超时时间和超时重试次数一定要配置好。

3. 统计失败次数降级

当依赖某些外部服务时，可能会碰到不稳定的API，当失败调用次数到达某个阀值时会自动降级。然后，通过异步线程去检测服务是否已经恢复，如果恢复则取消降级。此类降级也被称为熔断器[6]。

4. 限流降级

在电商秒杀的时间段，可能会因为访问量剧增而导致后端系统崩溃，如果超过限流阀值，后续请求会被丢弃。

### 3.3 服务可靠性

相对于本地调用，跨进程的分布式服务调用面临的故障风险更高。对于应用而言，分布式服务框架需要具备足够的健壮性，在框架底层能够拦截并向上屏蔽故障，业务只需要配置容错策略，即可实现高可靠性。

### 3.3.1 服务状态监测

在分布式服务调用时，某个服务提供者可能已经宕机，如果采用随机路由策略，消息会继续发送给已经宕机的服务提供者，导致消息发送失败。为了保证路由的正确性，消费者需要能够实时获取服务提供者的状态，当某个服务提供者不可用时，将它从缓存的路由表中删除掉，不再向其发送消息，直到对方恢复正常。

工程中通常会使用注册中心对服务端进行状态监测，当监测到服务端不可用时，会将故障状态广播到所有的客户端，客户端被通知到后，根据故障信息把故障节点排除在路由表之外。

服务消费者和提供者之间默认往往采用长链接，并且通过双向心跳监测保障链路的可靠性。在某些特殊场景中，提供者和注册中心之间，消费者和注册中心之间网络可达，但是提供者和消费者之间网络不可达，或者提供者和消费者之间链路已经断连。此时，如果消费者任旧向链路中断的提供者发送消息，写操作将会失败。注册中心并不能检测到服务端异常。

因此，一般要使用注册中心检测加上提供者和消费者之间的链路有效性检测来保障系统的可靠性。

当消费者通过双向心跳检测发现链路故障之后，会主动释放链路，并将对应的服务端从路由表中删除。当链路恢复之后，重新将恢复的故障服务端地址信息加入地址列表中。

### 3.3.2 服务健康检查

在集群组网环境下，由于硬件性能差异、各服务提供者的负载不均等原因，如果采用随机路由分发策略，会导致负载较重的服务提供者不堪重负被压垮。

使用健康度监测机制，可以对集群的服务实例进行检测，根据检测结果对健康度打分，得分较低的亚健康服务节点，路由权重会被自动调低，发送到对应节点的消息相比于其他健康节点会少很多，这样就能够实现“能者多劳，按需分配”，实现更合理的资源分配和路由调度。

服务的健康度检测通常需要采集如下性能指标：

1.服务调用延迟,QPS,调用成功率

2.基础资源使用情况，例如堆内存，CPU使用率等

健康度评分通常由独立的监控节点负责。服务端将性能数据周期性汇报到监控节点，性能汇总和监控节点根据内置的健康度算法对各个服务端做健康度打分，然后周期性的将服务端健康度信息下发给客户端。客户端根据健康度评分动态调整路由权重，向处理能力强的节点发送更多的消息，保护亚健康服务节点。

### 3.4 服务隔离

隔离是指将系统或资源分割开，系统隔离是为了在系统发生故障时，能限定故障传播范围和影响范围，即发生故障后不会出现滚雪球效应，从而保证只有出问题的服务不可用，其他服务还是可用的。资源隔离能减少资源竞争，保障服务间互相不影响。

### 3.4.1 服务故障隔离

1. 线程隔离

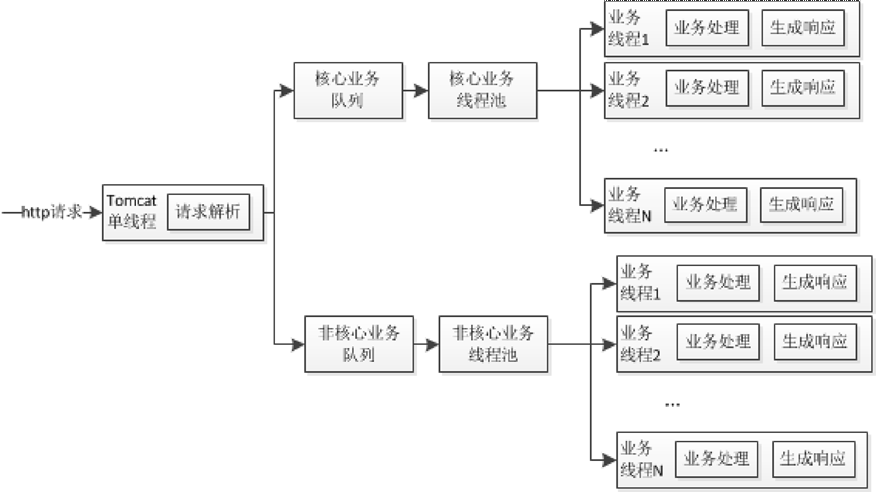
线程隔离指线程池隔离。把分类过的请求交给不同的线程池处理。当某业务的请求处理线程膨胀消耗过多的资源时，不会将故障扩散到其他线程池，从而确保其他服务可用。

图3.1 Tomcat中的线程隔离

如图3.1所示，Tomcat中可以配置多个线程池，核心业务分配核心业务线程池，线程池可以大一点，优先级高一点；非核心业务分配非核心业务线程池，线程池相对小一点，优先级低一点。

使用Tomcat时，多个服务会共享一个线程池，默认线程池大小是150。假如其中一个服务访问的数据库响应很慢，这会使得服务时延增加，多数线程会阻塞等待响应返回，导致整个Web服务器线程池都被该服务占用。线程或者信号量隔离可以来解决这个问题，使得某个服务的线程池满了也不会对其他服务造成严重的故障。

线程隔离包括三种粒度的隔离实现, 如下所示

粗粒度实现:服务分组加上线程池，每个服务分组配置一个隔离线程池，相同分组的情况配置共用一个线程池。

细粒度实现:服务和线程池，服务分组中每个服务配置一个隔离线程池。

混合实现:对核心服务单独设置线程池，一个服务分组配置一个线程池。

2. 集群隔离

当某个服务不稳定并可能影响到其他的服务分组时，要考虑为服务提供独立的服务集群(服务分组)。

其他还包括虚拟机(Virtual Machine)故障隔离、物理机故障隔离、机房故障隔离、基于接口方法参数的路由隔离等。

### 3.5 流量控制

限流的目的是通过对并发请求或者一个时间窗口内的请求进行限速来保证系统的稳定性。如果达到限制速率就可以降级、排队和等待、拒绝服务。在压测时能找出每个系统的处理峰值，然后通过设定阀值来拒绝处理过载的请求以此保障系统可用。另外，也应根据系统的吞吐量、响应时间、可用率来动态调整限流阀值。

当资源成为瓶颈时，服务框架需要对消费者或者提供者做限流，启动流控保护机制。对系统的整体流控，可分为针对访问速率的静态流控和针对资源占用的动态流控。对单一接口的限流称做服务接口限流。限流算法主要有计数器算法、漏桶算法、令牌桶算法。

### 3.5.1 静态流控

静态流控是针对消费端访问速率进行控制，一般会根据约定的TPS做全局流量控制。

1.静态流控设计方案

静态流控设计采用预分配方案，根据静态流控阀值和集群服务节点个数，计算每个服务节点分摊的TPS阀值。系统运行时，每个服务按照自己的阀值进行流控，如果超出阀值就拒绝请求访问。

服务框架启动时，将节点的静态流控阀值加载到内存中，服务框架通过handler拦截器在服务调用前做拦截计数，当计数器在指定周期t达到QPS上限时，启动流控，拒绝新的请求消息接入。

要注意的是，到达流控阀值之后拒绝新的请求消息接入，不能拒绝后续的应答消息，否则会导致客户端超时或者触发Failover，增加服务端的负载。

分布式服务框架的一个特性是服务的动态上下线和自动发现机制，在运行期间服务节点数会随业务量的变化而变化，在这种场景下静态分配方案显然无法满足需求。

2. 动态配额申请制

动态配额申请制在系统部署时，根据服务节点数和静态流控QPS阀值，预先分配一定比例的配额做为初始分配，剩余的额度放在配额资源池。如果服务使用完了配额，会主动向注册中心申请。如果总配额被使用完，各个申请配额的服务节点只能拿到零配额，服务节点对后续的请求进行流量控制。

动态配额申请制的实时性高,负载情况在各个节点内计算，性能数据在本地内存中计算获取。动态配额申请制流控精准,各个服务节点根据自身负载情况去申请配额，保证性能高的节点有更高的额度，性能差的自然配额就少，这样能够更合理地调配资源。

实践验证表明，采用动态配额申请制的静态流控更精确，在实战中效果也更好。

### 3.5.2 动态流控

动态流控不是对流量或者访问速度做精确控制，而是拒绝过大的负载，目标是为了保证系统的正常运行。

动态流控分为多个级别，每个级别被拒绝掉的消息比例不同，流控系数也就不同。每个级别都有自身的流控阀值，一般会支持在线动态调整。

### 3.5.3 服务接口限流

1. 限制并发量和连接数

限制并发量是对线程并发数进行控制。并发控制有针对服务端的全局控制和针对客户端的局部控制两种形式。限制连接数是为了防止客户端连接数过多导致服务端压力过大，需要针对连接数进行流控。

2. 限流总并发/请求数

如果接口会有突发流量，又担心负载太大造成系统崩溃，这个时候需要限制接口的总并发/请求数。这种方式适合对需要过载保护或者可降级业务的服务进行限流。一些开放平台会限制用户调用某个接口的请求量，这样就可以用计数器来实现。这种方式没有作平滑处理，也是简单的限流，具体根据实际情况按需实现。

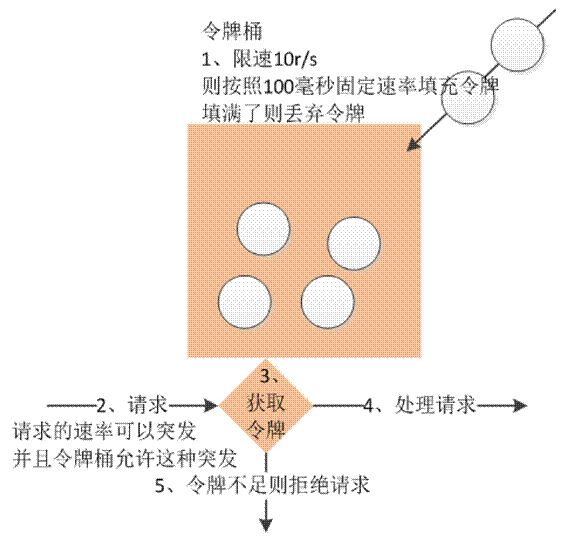
3. 限流时间窗内请求数

比如一些基础服务会被很多其他上层应用调用，所以要对每秒/每分钟的调用量进行限速。

### 3.5.4 限流算法

限流算法有计数器算法、漏桶算法、令牌桶算法等。

令牌桶算法如图3.2所示：往桶中添加令牌有固定速率，如果桶里有令牌则请求能被处理，当令牌数减到零时，就拒绝新的请求。

图3.2 令牌桶算法

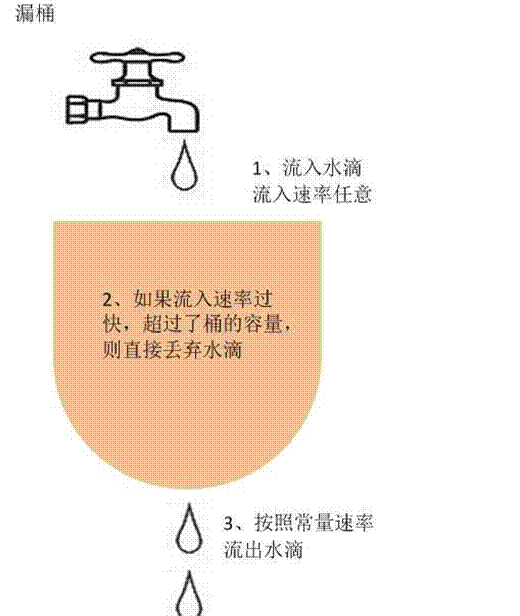
漏桶算法如图3.3所示：请求流入速率任意，请求流出有固定速率。流入请求数超过漏桶容量时，后续请求被拒绝。

图3.3 漏桶算法

令牌桶限制的是平均流入速率，并允许一定程度的突发流量。漏桶限制的是常量流出速率，从而平滑突发流入速率。

一些场景中需要把突发请求整形为平均速率请求。令牌桶算法和漏桶算法可以满足这个场景。Guava RateLimiter提供的令牌桶算法可用于实现平滑预热限流和平滑突发限流。

### 3.6 超时与重试机制

在实际开发过程中，太多故障是因为没有设置超时或者设置的不对而造成的。

如果应用不设置超时，则可能会导致请求响应慢，慢请求会累积导致连锁反应，甚至造成应用雪崩。而有些框架在超时后会进行重试（如设置超时重试两次），读服务天然适合重试，但写服务大多不能重试（如果写服务是幂等的，则重试是允许的），重试次数太多会导致多倍请求流量，即模拟了DDos攻击，后果可能是灾难性的。所以务必要设置合理的重试机制，并且应该和熔断、快速失败机制配合使用。

在系统的各个分层都有超时机制，这里论述服务框架相关的超时。其中最重要的超时设置是网络连接/读/写的超时时间设置。

### 3.6.1 框架的超时与重试

服务框架客户端超时与重试需要设置客户端的网络连接/读/写超时时间与失败重试机制。

提供者/消费者与注册中心之间进行服务注册发现时需要配置超时时间，提供者服务运行需要配置超时时间，消费端调用也需要配置超时时间。超时时间都有默认值，一般为2秒到5秒。

提供者服务运行超时也称作业务超时，可以分任务型和服务调用型两类。任务型超时比如订单超时未支付取消超时，活动自动关闭等。服务调用型比如某个服务的全局超时时间是500ms，服务内部有多处服务调用，每处服务调用的超时时间可能不一样，此时可以简单的使用Future[37]来解决问题。

### 3.6.2 最佳实践

超时之后应该有相应的策略来处理超时，常见的策略有重试，比如等待一段时间再重试、尝试调用其他分组服务、尝试调用其他机房服务，重试算法可考虑使用如指数退避算法。其他的策略还有在负载均衡场景下从服务列表中摘掉不存活节点，回退返回历史数据/静态数据/缓存数据等。

对于非幂等写服务应避免重试，或者可以考虑提前生成唯一流水号来保证写服务操作，通过判断流水号来实现幂等操作。

超时重试必然导致请求响应时间增加，最坏情况下的响应时间等于重试次数\*单次超时时间。这很可能严重影响用户体验，导致用户不断刷新页面来重复请求，最后导致服务接收的请求太多而宕机。因此除了控制单次超时时间，也要控制好用户能忍受的最长超时时间。

超时时间设置太短会导致服务调用成功率降低，超时时间太长又会导致本应成功的调用却失败了，这就要根据实际场景来选择最适合当前业务的超时时间，甚至是程序自动计算超时时间。根据经验值，超时时间要设置的稍微大于TP99的响应时间。

客户端和服务端都应该设置超时时间，而且客户端根据场景可以设置比服务端更长的超时时间。如果存在多级依赖关系，如A调用B，B调用C，则超时设置应该是A>B>C,否则可能会一直重试，引起DDos攻击效果。不过最终如何选择还是要看场景，有时候客户端设置的超时时间就是要比服务端的短，可以通过在服务端实施限流/降级等手段防止DDos攻击。

在实际开发中要重视超时时间，很多重大事故都是因为超时时间不合理导致的，设置超时时间只有好处没有坏处。

### 3.7 服务容错模式

服务容错的设计的基本原则是"Design for Failure"。为了避免出现"千里之堤溃于蚁穴"的情况，在设计上需要考虑到各种边界场景和对于服务间调用出现的异常或响应延迟的情况。这些都是为了让系统具备自我恢复能力，并使得在故障出现时不会严重破坏用户的体验。本节将要介绍的几个模式基于这个原则和目标，能够解决分布式服务调用中的一些问题，提高系统在故障发生时的可用性。

模式是某种场景下一类问题及其解决方案的归纳总结，往往可以重用。设计模式[38]是一套经过分类编目的可反复使用的代码设计经验总结。而在服务容错这个方向，工程中也已经有了不少实践总结出来的设计模式和解决方案。

超时与重试模式、限流模式和舱壁隔离模式[6]可以作为独立的模式。熔断器模式可以解决超时、限流和隔离的问题，保护服务自身和其下游服务，熔断之后可以使用回退模式做最后的工作。

### 3.7.1 熔断器模式

保险丝会在电流异常升高到一定的高度和热度的时候，自身熔断切断电流，从而起到保护电路安全运行的作用。同样，在服务集群中，如果调用的远程服务或者资源由于某种原因无法使用时，如果没有某种过载保护，就会导致请求阻塞在服务器上等待从而耗尽系统或者服务器资源。很多时候刚开始可能只是系统出现了局部的、小规模的故障，然而由于种种原因，故障影响的范围越来越大，最终导致了全局性的后果。系统中的这种过载保护就是将要谈到的熔断器[6]模式(Circuit Breaker)。

熔断器(Circuit Breaker)模式中有三种可相互迁移的状态，如图3.4所示。

闭合状态: 如果请求失败率没有超过阀值或者关闭了熔断开关，则熔断开关处于closed状态，此时不进行降级处理。

打开状态：如果失败率超过阀值或者打开了熔断开关，则启动熔断机制，调用降级处理方法进行降级处理。有异常后并不会立即熔断，等失败率到了一定程度后，闭合状态才会迁移到开启的状态。

半打开状态：熔断开关打开后并不会一直熔断下去，会在一个时间窗口后进行重试。如果请求对服务的调用成功，那么可以认为之前导致调用失败的错误已经修复，此时熔断器切换到闭合状态并且将错误计数器重置;如果一定数量的请求有调用失败的情况，则认为导致之前调用失败的问题仍然存在，熔断器切回到打开方式，然后开始重置计时器来给系统一定的时间来修复错误。半断开状态能够有效防止正在恢复中的服务被突然而来的大量请求再次拖垮。

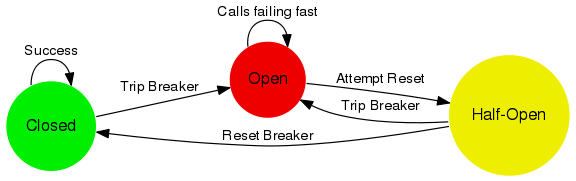


图3.4 熔断器状态迁移图

Hystrix实现了熔断器模式，它会在一段时间内进行一次调用重试，重试成功就关闭熔断开关，否则还会处于打开状态，这样可以做到快速失败和快速恢复。以下几种错误会进入熔断状态：即信号量拒绝、线程池拒绝、异常、超时。

### 3.7.2 限流模式

限流(Rate Limite)也可以叫流量控制。节流(Throttling) 也是限流的一种方式，就是简单丢弃流量。 服务负载很高的时候，处理请求很慢时就处理一部分请求，丢弃余下请求流量。至于处理哪些和丢弃哪些请求，有throttleFirst、throttleLast和throttleWithTimeout策略来决定处理和丢弃哪些流量。

反压(Backpressure) 也是限流的一种方式，后端消费者对前端生产者流量的反向控制。消费者需要多少流量生产者就生产多少流量。工程实现中Akka ,Storm,Spark Streaming,RxJava等框架都实现了反压机制。

### 3.7.3 超时和重试模式

超时（Timeout）可以避免线程的阻塞，从而可以规避级联的失效问题。慢响应有很多种可能，有时候并不需要知道确切原因。有时只需要放弃，然后继续往下执行。有时客户端的超时因为远端服务端出现的问题，不能立即解决，如再重试还会是超时。所以，先缓存操作再事后重试是一个好主意。

### 3.7.4 回退模式

在发生限流、熔断、超时重试失败的时候，为了马上恢复服务需要提供回退(Fallback)的机制。失败沉默、快速失败、自定义方式是常见的回退策略。

失败沉默(Fail-silent)一般用在可降级功能的场景下，数据为空也不会影响用户体验，直接返回缺省值或空值。快速失败(Fail-fast)一般用在非强依赖的场景，如非核心服务超时的处理可以直接抛出异常。自定义方式是如果业务是关键流程并且严重影响用户体验的场景，可以使用备用服务获取数据,也可以将请求放入队列待后续处理，也可以使用缓存中的默认数据来临时支撑服务调用。

### 3.7.5 舱壁隔离模式

舱壁 (Bulkhead Isolation)是工程中通用的故障隔离模式，可以选择隔离服务器的CPU、集群中的一个服务器或者是应用中的线程池。

### 3.8 本章小结

本章各个章节讲了分布式服务框架的非功能特性中容错的原理、算法或最佳实践。最后总结了工程中常见的容错模式。

# 4 Hedwig服务框架设计和实现

Hedwig的一次服务调用中，客户端将请求封装成协议消息，按照指定路由策略将请求发送到服务端，业务线程注册监听器回调或者同步等待。客户端反序列化响应消息，然后唤醒等待的业务线程。业务线程获得服务调用结果返回。如果客户端调用超时，则执行容错策略（重试策略），以此保证服务调用成功。

接下来就依次介绍Hedwig内部是如何设计和实现功能模块和容错模块的。

### 4.1 通信框架和协议栈

性能是服务框架要考虑的重要特性，高性能的服务框架才能提供应用高性能的业务服务。框架中的远程通信模块是性能的关键模块，所以Hedwig两个版本的选型主要考虑业界成熟的通信框架。Hessian和Akka是Hedwig考虑使用的两个通信框架。

Hessian是老牌的二进制Web服务协议，基于 HTTP协议进行传输。Hessian 通过其自定义的串行化机制将请求信息进行序列化，产生二进制流。

Hessian性能属于中上，瓶颈在IO模型和线程模型上。Hessian没有自身的线程模型，在Web容器中部署的时候用的是容器的线程模型，比如在Tomcat容器中部署的时候，使用的是Tomcat的基于BIO同步阻塞线程模型，客户端发起请求时要同步阻塞，等返回后才能执行后面的逻辑，所以吞吐量上不去.

Hessian客户端使用基于请求/响应的协议来与服务通信。在交互的时间段里，客户端和服务端必须可用，这样就减少了可用性。

Akka是JAVA虚拟机平台上构建高并发、分布式和有弹性的消息驱动应用的工具和运行时环境。Akka的Supervisor（监督者）的继承树有“let-it-crash”的语意，适合于稳定性要求高的系统，适合写容错自愈从不会宕机的应用。Akka被设计成在分布式的环境里工作,所有Actor之间的交互使用真正的异步消息来发送，所有调用都是异步的。Akka本身实现了一个分布式轻量级的消息系统。

因为Akka是全异步化和并行化的，相对于Hessian吞吐和延迟上性能高了很多。Akka是原生基于网络层TCP的，相对于原生使用HTTP的Hessian协议有更高的性能。

Akka使用基于消息的通信风格。服务间通信使用异步消息。通过在消息通道之间交换消息来做服务通信。这种方式是松耦合，解耦了客户端和服务端。消息Broker缓存消息直到消费者有能力处理他们，这样增强了可用性。并且提供了大量的通信模式，其中包括请求/响应，通知，请求/异步响应，发布/订阅，发布/异步响应等。

Hedwig 0.1.x版本选用了Hessian作为通信框架。为了更高的性能提升，0.2.x选用了Akka作为通信框架。Akka的Actor和Actor之间用Mailbox队列来收发消息，某个Actor失效重启后，能继续消费Mailbox，这样保证了不会丢消息，所以框架的可用性也大大提高了。

Hedwig支持基于Hessian的HTTP共有协议和基于Akka的TCP私有协议。HTTP共有协议是Hedwig0.1版本使用的，TCP私有协议是Hedwig0.2版本使用的。HTTP协议适合在公有网络场景时使用，比如调用别的公司提供的公有服务。更高性能的TCP私有协议适合在公司内部服务之间调用的场景。

Hedwig 0.1版本中, 分布式服务框架与应用一起打包,它作为应用的一个依赖包，Web容器是Tomcat，详见图3.5。Hedwig 0.2版本把服务框架自身变为一个容器来承载远程服务，详见图3.6。

Akka有在Intel和亚马逊使用，国内用户相对少，据笔者了解在服务框架中几乎没有用过。一号店Hedwig使用Akka构建服务框架是相对有创新意义和有特色的。

### 4.2逻辑模块和时序图

如图4.1,4.2所示为Hedwig0.1, 0.2版本的整体模块图。hedwig-zk是Zookeeper客户端，存储服务提供者的元数据到Zookeeper中。hedwig-common是一些通用模块，包含一些框架底层抽象的接口和功能。hedwig-client模块是服务消费者，包括服务发现、流控、负载均衡等。hedwig-provider模块是服务提供者，包括服务注册、流控等。hedwig-client和hedwig-provider模块依赖hedwig-common模块，hedwig-common模块依赖hedwig-zk。photon是0.2版本新增的一个模块，剥离出了通用的包括路由、负载均衡、连接池管理、序列化/反序列化等功能。

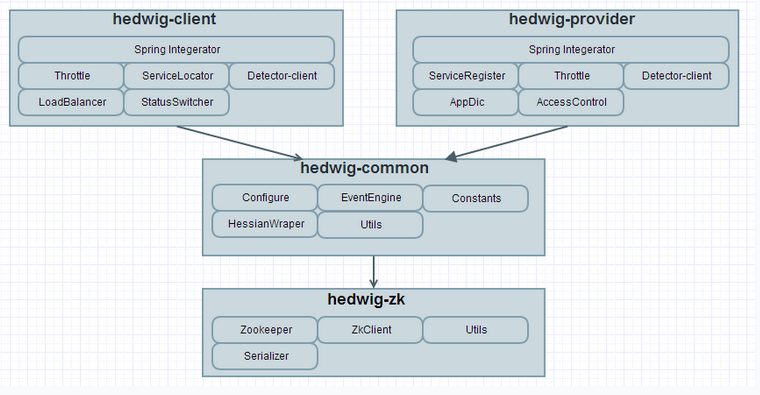
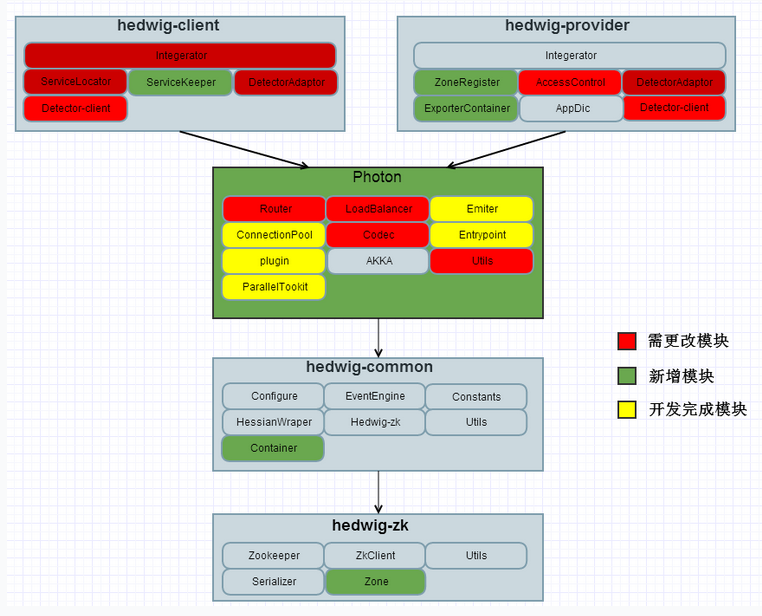


图4.1 0.1.x版本整体模块

图4.2 0.2.x版本整体模块

如图4.3是Hedwig0.2版本的客户端的请求时序图, ActorInvoker是客户端的发起方，提供了同步、Future、Oneway的方式服务调用方式。

ActorInvoker发消息到到MethodActor，如果限流开关打开，则会初始化生成限流器，限流器在大流量时会限制发送的流量。LogedAckActor会处理服务端的响应，如果响应超时，先把有问题的路由节点置为临时不可用，然后做重试的策略。

实现ILoadBalancer负载均衡策略的接口包括一致性hash的策略、基于权重的轮询(Weight Round Robin)等， 实现方式是路由内部实现一个环，路由器（IRouter）会按不同的策略在环中取一个可用节点来发送请求到服务提供者。

EndPointActor会发送请求到服务提供者，用户可以使用默认或者自定义的序列化方式来发送二进制的请求。

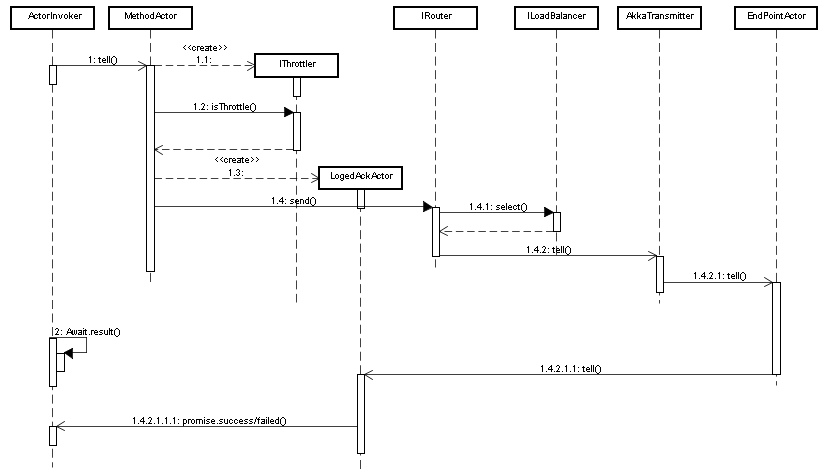


图4.3 Hedwig0.2客户端请求时序图

如图4.4是Hedwig0.2版本的服务端的响应时序图，EndpointRepository用来初始化服务提供者，一个服务的方法默认会对应16个workerActor，每一个workerActor都有唯一的名字uNameKey，并把uNameKey和方法名映射起来。LogedWorkerActor是workerActor，是真正执行业务代码的类，拿到方法之后通过反射的方式调用真正的业务方法，最后会把响应结果发送回LogedAckActor。

通过图4.3、4.4这些类和方法完成了整个服务调用的请求/响应过程。

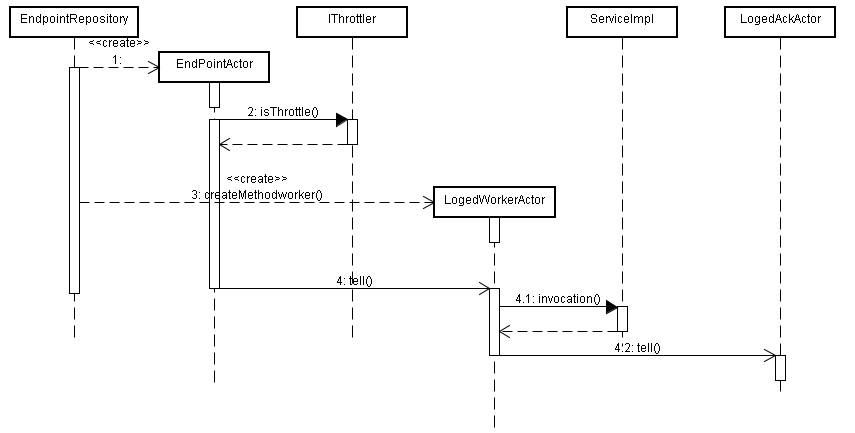


图4.4 Hedwig0.2服务端响应时序图

### 4.3 功能模块设计和实现

### 4.3.1 服务注册和发现功能设计

Hedwig的客户端和服务端有服务自动注册/发现机制, 如图4.5所示。Hedwig利用工具类解析服务提供者信息。服务提供者在Zookeeper中注册服务时，往Zookeeper的目录里写元数据，存储使用的是Zookeeper的临时节点。每个服务提供者正常启动之后会到Zookeeper集群进行注册，所谓注册就是在Zookeeper的树形结构中生成一个znode来标明服务可用。如果服务下线链接断开，Zookeeper临时节点会自动消失。服务提供者的元数据包括该服务接口隶属于哪个系统、服务的 IP 以及端口号、服务的请求 URL 和服务的权重等等。

服务消费者在Zookeeper中查询到服务的元数据，然后调用服务,并且同时监视（watch）住了Zookeeper的临时节点，订阅了这个服务。在消费端添加watcher来监听子节点的变化，任何服务提供者节点的增减都会触发Locator模块重新获取可用服务器列表。并且将新列表传递给负载均衡(Load balance)模块，为下次发送请求做准备。这样就可以感知到服务的下线和上线，做到了服务上线注册和下线反注册的功能。如果单机故障某个服务不可用了，这时和人工下线服务的情况是一样的，客户端也能收到下线通知。

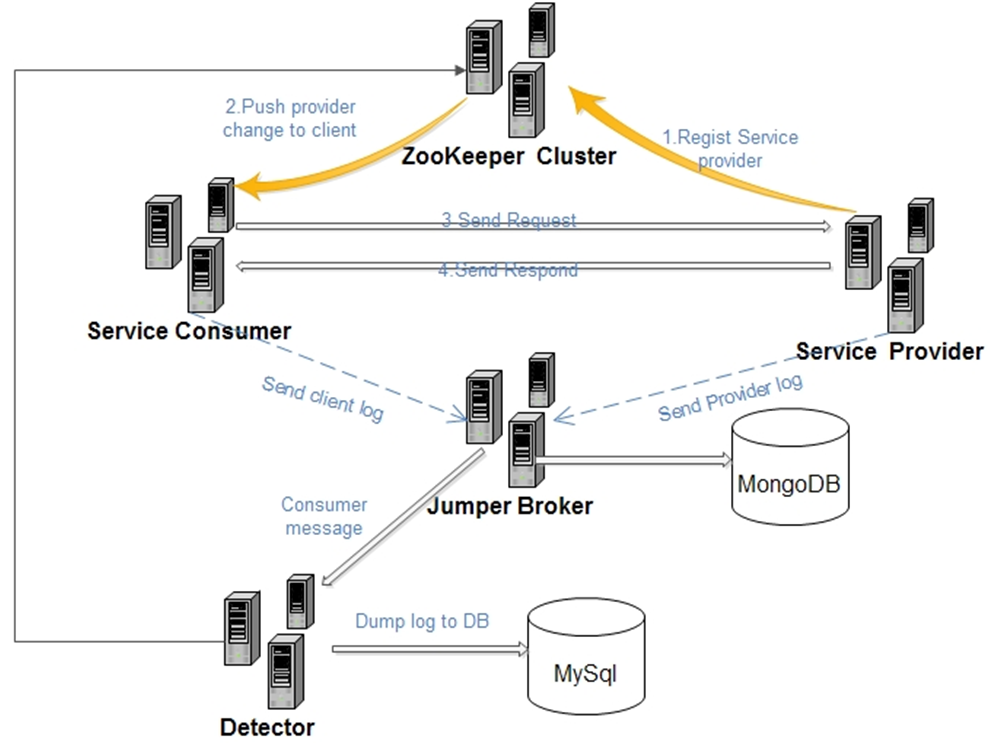
通过Zookeeper的协调，减少了客户端的配置，可以按需要添加和减少服务提供者的数量，极大的增加整体架构的弹性，并能提供统一的服务治理能力。

图4.5 Hedwig 服务注册和发现

### 4.3.2服务路由设计

1. 直接地址路由

Hedwig客户端里可以配置target属性，表示指定访问机器IP，这样就可以不需要经过注册中心直接路由直连到某个地址, 这个功能可以用在本地研发的场景。

2. 基于分组路由

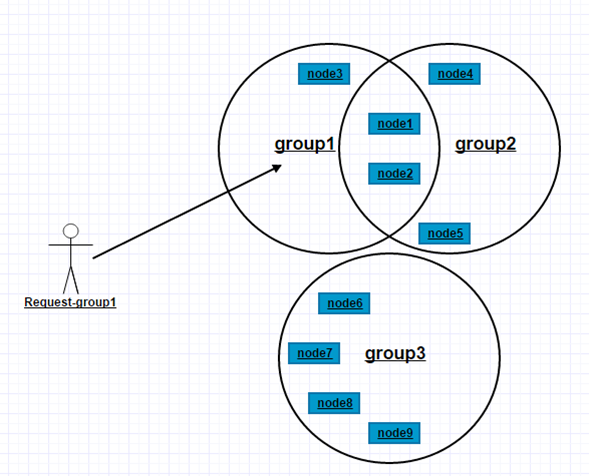
如图4.6所示, 每个Pool的分组方式由服务提供方在发布客户端时定义，可以根据调用方需要的功能或重要级别加以区分。服务发布后把服务端节点分入预先定义的分组中，同一个节点可以属于多个分组。服务请求只会路由到指定分组中的节点。

图4.6 客户端和分组的关系

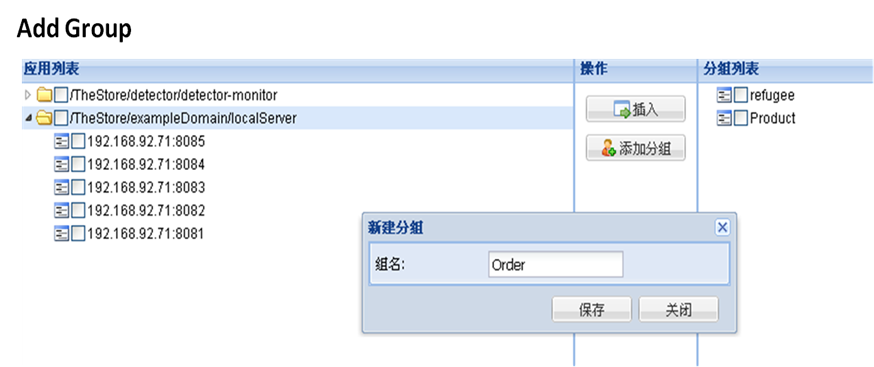
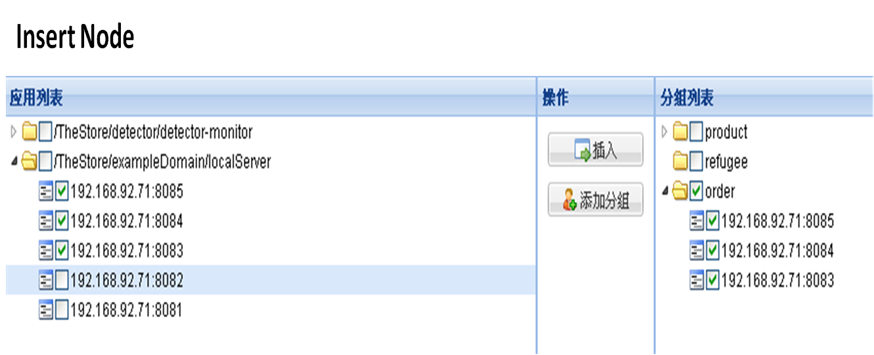
如图4.7、4.8所示，Hedwig的分组可以在Detector里管理，并且可以在分组里添加机器。

图4.7 Detector中添加分组

图4.8 Detector分组中添加机器

### 4.3.3 动态负载均衡算法设计

通过注册中心分组或者直连指定的方式，经过路由的选择可以拿到了一组地址，这一组地址如果选出一个地址进行本次的调用就涉及到负载均衡的选择。负载均衡的计算全部是在客户端实现的。

case ROUNDROBIN:

balancer = new RRBalancer(routees);

break;

case CONSISTENT\_HASH:

balancer = new CHBalancer(routees);

break;

case BROADCAST:

balancer = new BCBalancer(routees);

break;

case LEAST\_USAGE:

balancer = new LUBalancer(routees);

break;

case WEIGHT\_ROUNDROBIN:

default:

balancer = new WRRBalancer(routees);

break;

图4.9 Hedwig 负载均衡算法

如图4.9所示, Hedwig实现了Round Robin、基于权重的Round Robin、一致性hash、广播算法。默认使用基于权重的轮询算法。

所有路由算法都以可用节点生成的环为基础，每次通过适当的方式从环中取一个或多个节点来发送请求。

RoundRobin负载均衡算法在实现时，不关心权重，按照方法级进行轮询，相互不会影响。可用节点列表是一个环状，每次都取环中的下一个节点，轮询完了就再重新开始。主要用来调用量比较少的情况下。基于权重的Round Robin算法中，在构造环时根据节点的权重来决定环中节点的冗余个数。一致性 hash算法保证了同样参数的请求能够负载均衡到同样的节点上。广播算法中，一次给环中的每个服务提供者都发送请求。

### 4.3.4 序列化和反序列化实现

Hedwig分自身的序列化和可扩展的序列化，自身的序列化是默认的序列方式，用户也可以继承HedwigCompatibleCodecProcessor自定义序列方式。默认的Hessian序列化方式适合大多数的应用场景，在某些高性能的场景下用户可以自定义更高效的序列方式。具体如图4.10、4.11所示。

public class HessianCodecProcessor implements ICodecProcessor<RemoteRequest<InvocationContext>, RemoteResponse>{

public HessianCodecProcessor() {

byte[] bytes = this.toBinary("init");

Object s = this.fromBinary(bytes);

}

@Override

public byte[] reqeust2binary(RemoteRequest<InvocationContext> request) {

return toBinary(request);

}

@Override

public RemoteResponse binary2respones(byte[] array) throws Throwable {

Object o = fromBinary(array);

return o == null ? null : (RemoteResponse) o;

}

@Override

public byte[] respones2binary(RemoteResponse response) {

return toBinary(response);

}

@Override

public RemoteRequest<InvocationContext> binary2request(byte[] array) {

Object o = fromBinary(array);

return o == null ? null : (RemoteRequest<InvocationContext>) o;

}

}

图4.10 hessian序列化

public class HedwigCompatibleCodecProcessor implements ICodecProcessor<RemoteRequest<InvocationContext>, RemoteResponse> {

@Override

public byte[] reqeust2binary(RemoteRequest<InvocationContext> request) {

...

out.call(request.getMangleName(), request.getParameters(), request.getContext());

return array;

}

@Override

public RemoteRequest<InvocationContext> binary2request(byte[] array) {

...

HedwigHessianInput in = new HedwigHessianInput(bis);

o = in.readObject();

return (RemoteRequest<InvocationContext>) o;

}

@Override

public byte[] respones2binary(RemoteResponse obj) {

ByteArrayOutputStream baos = new ByteArrayOutputStream();

HedwigHessianOutput out = new HedwigHessianOutput(baos);

array = baos.toByteArray();

return array;

}

@Override

public RemoteResponse binary2respones(byte[] array) throws Throwable {

...

resp = o != null ? ResponseFactory.createSuccessResponse(o, 0) : null;

...

return resp;

}

}

图4.11 可扩展的序列化

### 4.3.5 服务调用设计

Hedwig 0.2(Akka) 版本有三种调用方式。

CALL\_SYNC是同步的方式，因为Akka天然是基于消息的全异步化的通信方式，所以使用Await.result把异步调用转为同步调用，Await.result可以同步获取结果、超时或者异常。当前线程发起调用后阻塞请求线程，需要在指定的超时时间内等到响应结果才能完成本次调用。如果超时时间内没有得到响应结果，那么会抛出超时异常。同步调用模式最常用，注意要根据服务的处理能力合理设置消费端超时时间。

CALL\_FUTURE是异步Future的调用方式。客户端发起调用后不会同步等待服务端的结果，而是获取到Hedwig给到的一个 Future 对象，调用过程不会阻塞线程，然后继续执行后面的业务逻辑。服务端返回响应结果被 Hedwig缓存，当客户端需要响应结果的时候需要主动获取结果，获取结果的过程阻塞线程。Hedwig内部使用Promise来设置获取结果的状态，如果能获取结果,设置Promise为success，如果获取结果超时或者失败，则设置Promise为failure。

CALL\_ONEWAY是异步单向的调用方式，不关心响应结果，请求线程不会被阻塞。Oneway 调用不保证成功，消费者无法知道调用结果，使用 Oneway 调用需要注意控制调用节奏防止压垮服务提供者。

Hedwig并行调用API如图4.12所示：批量服务调用加上Future机制。并行调用比串行调用效率高,能并行返回多个响应结果。实现中没有产生新的线程，使用者不用担心依赖线程上下文的功能会抛出异常。

final IQueryService queryService = (IQueryService) context.getBean("queryService2");  
Map<String, List<Future<RemoteResponse>>> fMap = ParallelToolkit.*createScalaFuture*(new MultiCall<RemoteResponse>() {  
 @Override  
 public void call() throws Throwable {  
 for (int i = 0; i < 10; i++) {  
 queryService.queryStrings(Long.*valueOf*(i), *s*);  
 }  
 }  
});

//do other;  
for (int i = 0; i < 10; i++) {  
 Result r = (Result) ParallelToolkit.*getResult*(fMap.get("queryService2.queryStrings\_long\_string").get(i), 1000);  
}

图4.12 Hedwig并行接口

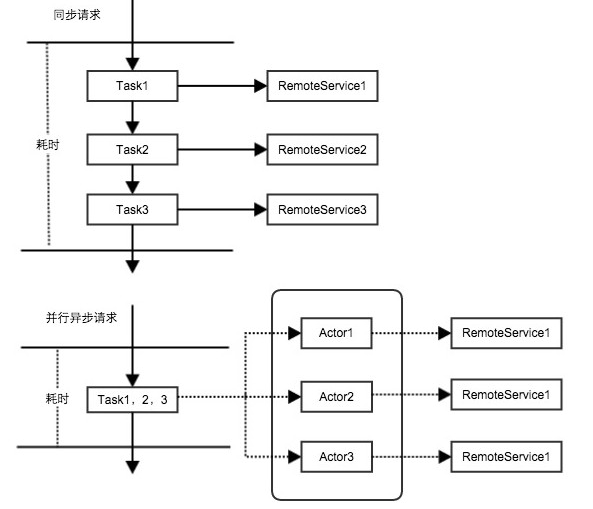
并行调用比串行调用减少了更多的响应时间。如图4.13所示，如果Task1耗时5ms，Task2耗时7ms，Task3耗时8ms，串行调用需耗时5+7+8=20ms，并行调用只需要消耗Max（5,7,8）=8ms。

图4.13 服务串行和并行调用比较

### 4.3.6 可靠性设计

长连接是在建立连接后发送数据、接收数据，但是不主动断开连接，并且主动通过心跳等机制来维持这个连接可用，当再次有数据发送请求时，不需要进行建立连接的过程。TCP keepalive 机制可以在连接无活动一段时间后发送一个空 Ack，使 TCP 连接不会被防火墙关闭。TCP keepalive会遇到客户端非正常关系但是服务端还保持连接的状态，而且TCP keepalive在真实网络中也容易失效。所以，大多数情况下，应用会采用业务心跳加TCP keepalive 一起使用的方案，互相作为补充。业务心跳具有更大的灵活性，可以自己控制检测的间隔、检测的方式，并且同时适用于 TCP 和 UDP。

在用Netty做底层RPC通信框架时，会依靠Netty的心跳机制，心跳机制由空闲链路检测机制实现，链路关闭后会触发事件，然后重新发起连接操作。整个可靠性是依赖Netty实现的。

Hedwig0.1服务发布的容器是Tomcat。Tomcat默认是开启keep-alive的，有2个相关的参数可以配置。keepAliveTimeout表示在复用一个连接时，两次请求之间的最大间隔时间，超过这个间隔服务器会主动关闭连接，默认值为20秒。maxKeepAliveRequests表示一个连接最多可复用多少次请求，默认是100。

而Hedwig0.2的通信框架是Akka，Akka在Actor 模型中使用监督（supervision）来增加容错性。 监督[27]的核心思想就是把对于失败的响应和可能引起失败的组件分隔开，并且把可能发生错误的组件通过层级结构来组织，以便管理。当创建Actor 时，新建的Actor都是作为另一个Actor的子Actor，父Actor负责监督子Actor，Actor的路径结构就展示了它的层级结构。

### 4.4 功能组件设计

分布式服务框架一般由几个粗粒度的组件组合在一起。组件化的好处是组件之间是松耦合的，组件易于替换和扩展。如果要扩展新的Hedwig RPC协议也就是写一个Client Request Handler和Server Request Handler，其他的组件不需要变更。

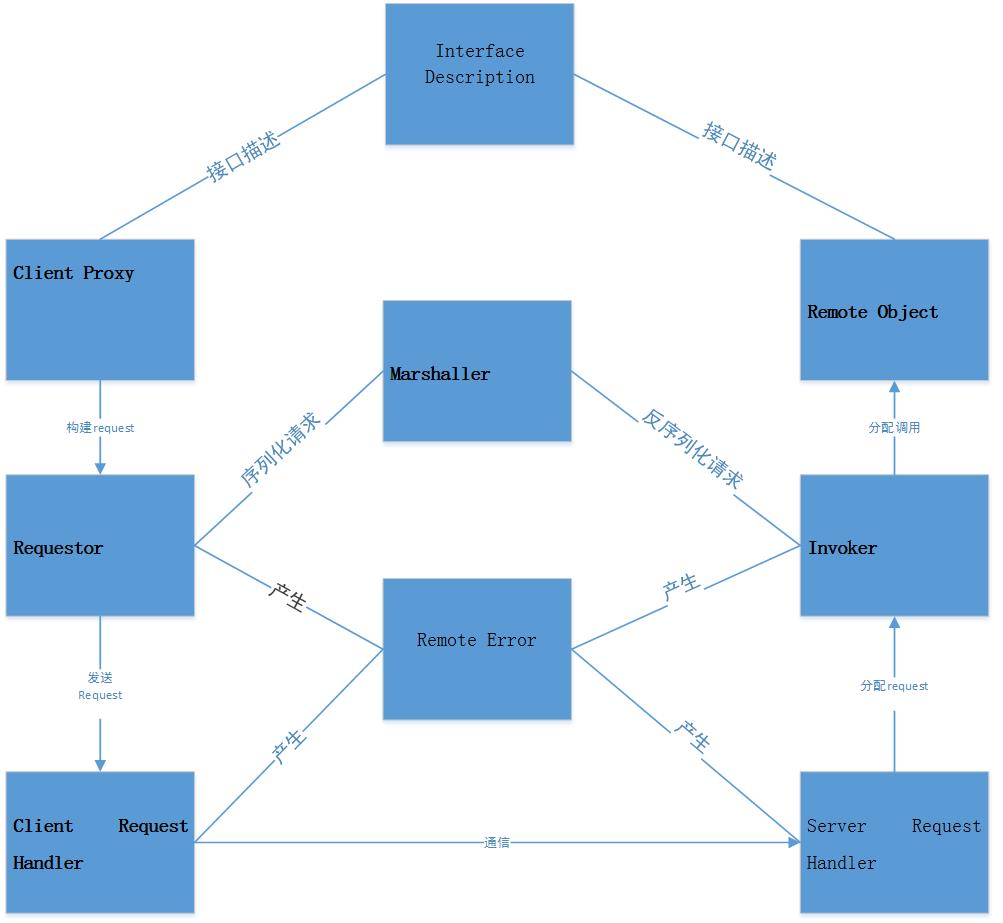
如图4.14所示，服务框架组件模式[1][3]主要由以下几个过程实现，左面3个是客户端的组件，右边是服务端的组件，接口描述（Interface Description）和远程错误(Remote Error)是客户端和服务端的公用组件。由Client Proxy构建请求发送到Requestor，Requestor传递请求到Client Request Handler，Client Request Handler和Server Request Handler远程通信并发送请求。远程Server Request Handler分配请求给Invoker，Invoker再分配调用给Remote Object。Client proxy和Remote Object之间使用统一的接口描述即契约。

图4.14 服务框架组件模式

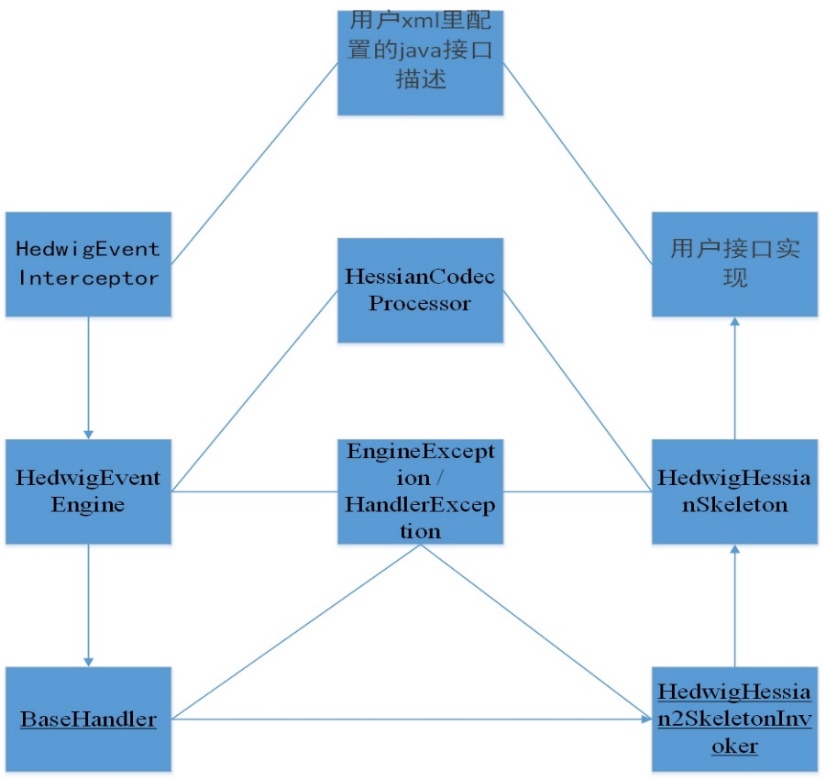
Hedwig和其他主流服务框架的功能点和模块有共通点，抽象出设计模式，按模块的职责分，可以分成以下几个组件，如图4.15所示。Client Proxy由HedwigEventInterceptor实现，Requestor称为Engine层，BaseHandler称为handler层。HedwigEventInterceptor调用Engine层，Engine层调用handler层。Invoker由HedwigHessian2SkeletonInvoker实现，Server Request Handler由HedwigHessianSkeleton实现。Remote Error分别由EngineException和HandlerException实现。

图4.15 Hedwig组件关系

### 4.5 容错设计和实现

### 4.5.1 Failover实现

当一个业务请求调用一个故障节点时会报错，我们把故障节点剔除，并直接返回异常给上层业务方。这种场景下为了减少业务系统报错并增强系统可用性，需要做一种优化，需在上层业务所能接受的时间内，进行重试另外一个可用节点，并将正确结果返回上层业务系统。

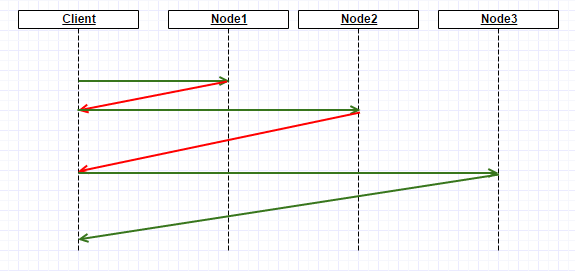
在重试另外一个可用节点时需要设置重试开关redoable，可重试服务必须为幂等服务，触发条件为非业务异常。在业务方配置的超时时间内进行重试，最多重试3次，重试时不能选择当前出错节点。如图4.16为Hedwig Failover流程。

图4.16 Hedwig Failover流程

如图4.17所示， 如果客户端中收到的响应是ReceiveTimeout的时候会调用retryMessage方法重试其他可以的节点。也就是说服务端响应慢了超时了，会采用Failover机制。

} else if (message.equals(ReceiveTimeout.getInstance())) {

Channel channel = \_msg.getSenderChannel();

if (channel != null) {

ClientNettySystem.getInstance().getSyncConnPool().destory(channel);

}

RouteeRef routeeRef = \_request.getRoutee(\_seq);

RouteeWraper routee = routeeRef != null ? routeeRef.getRoutee() : null;

if (routee != null) {

PhotonStatus status = routeeRef.getStatus();

if (PhotonStatus.ENABLE.equals(status)) {

routee.setTmpDisable();

logger.error("Request read timeout on:" + routee.getHostUrl() + ",kick out node and msg going to be retry");

}

} else {

logger.error("Request don't assigned a remote host yet,sequence:" + \_seq);

}

HedwigMonitorClientUtil.setException(cbLog, RemoteExceptionFactory.createRequestTimeoutException(\_request));

cbLog.setInParamObjects(\_request.getParameters());

retryMessage(cbLog);

图4.17 Hedwig failover实现

### 4.5.2 Failcache实现

如图4.18所示，当服务端有网络异常并且客户端有限流的时候，客户端会设置服务为TEMPORARY\_DISENABLE状态，表示服务暂时不可用.

if (HandlerUtil.*isNetworkException*(e) && context.getClientProfile().isClientThrottle()

&& !sp.getCurStatus().equals(ServiceStatus.*TEMPORARY\_DISENABLE*)) {

sp.setCurStatus(ServiceStatus.*TEMPORARY\_DISENABLE*);

}

图4.18 服务状态迁移

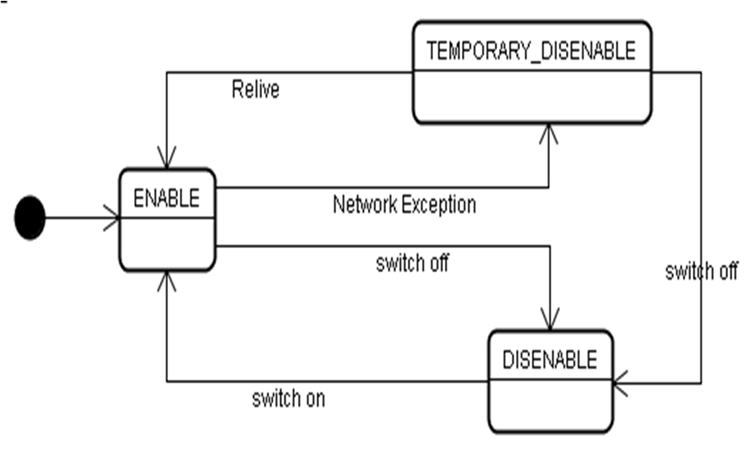
如图4.19所示，服务在正常运行时，服务在Enable状态。如果这个时候有网络的异常，客户端设置服务进入TEMPORARY\_DISENABLE表示临时不可用的状态。经过relive的策略后， 服务可以从TEMPORARY\_DISENABLE状态迁移到Enable状态。触发临时不可用(TEMPORARY\_DISENABLE)状态的网络异常包括链接的Connect异常,Socket异常,I/O异常,Socket超时异常四种。

图4.19 Hedwig服务状态迁移图

在自动复活的时候，实际这是一个Failcache的容错策略，会重试有问题的节点。如表4.1所示，自动复活(relive)的策略有三种。时间策略是服务节点第一次被踢出会在500ms后恢复，如果复活后的请求仍然失败，下次复活时间乘以2,时间上限为60s。请求次数策略是被踢出服务节点被请求次数达到50次后恢复，如果复活后的请求仍失败，下次复活需要的检索次数乘以2，次数上限为60000。混合策略(TPS策略)是当TPS小于100时会触发时间策略，大于100会触发请求次数策略。复活时间乘以2和检索次数乘以2来保证服务端有一段时间来恢复到正常的状态，如果大量客户端一直重试会对服务端有很大的压力，反而对恢复流程不利。

表4.1 Relive参数

|  |  |
| --- | --- |
| Relive参数 | 参数描述 |
| photon.emmitor.relive.interval=100 | First relive interval,interval\*2 next fail |
| photon.emmitor.relive.interval.max=60 | max relive interval |
| photon.emmitor.relive.count=10 | first relive count,count\*2 next fail |
| photon.emmitor.relive.count.max=60000 | max relive count |

### 4.5.3 人工降级设计

如图4.20所示，在Detector里实现了对Hedwig服务的人工降级开关。如果服务已经开启,可以在Detector里关闭服务,客户端会自动感知到服务状态。如果服务已经关闭,可以在Detector里开启服务,客户端会自动感知到服务状态。

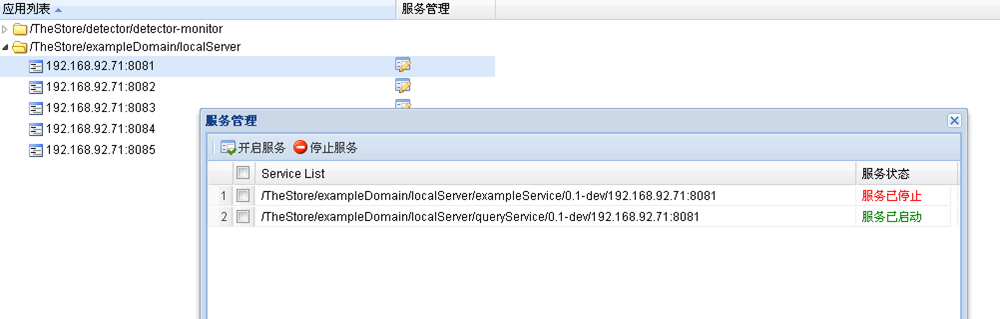


图4.20 在Detector里控制服务的启停

### 4.5.4 集群隔离设计

当一个服务集群供众多Pool使用时，因为服务资源是有限的，由于业务特点，生产环境下不同Pool对服务集群资源消耗是不一样的，有的非常均衡，有的很不平衡。有时一个使用方由于bug或其它原因，突然大量占用服务资源，在整个服务集群造成影响，所以需要将这种影响做隔离，或者说将影响缩小在局部范围，不要扩散到整个集群。可以根据不同调用方业务特点，对服务进行分组。

如图4.21所示，实现了服务集群的分组，客户端可以调用分组内的机器。如果一个分组内机器不稳定，不会影响到别的分组。

<bean id="baseHessianService"

class="com.yihaodian.architecture.hedwig.client.HedwigClientFactoryBean">

...

<!--当有分组需求时配置-->

<property name="groupName" value="testGroup2"></property>

...

</bean>

图4.21 服务分组配置

### 4.5.5 线程隔离设计

如图4.22所示，Hedwig Akka是一种细粒度的隔离, 服务分组中每个服务有一个独立的线程池。methodWorkerCount表示线程池中工作线程的数量。 用户可以自己设置，默认值是16。

Akka中一个Actor对应一个Mailbox。Worker线程池管理多个Actor的Mailbox, 拿一些消息之后, 会处理这个消息。如果消息有积压，可以伸缩这个线程池，产生或释放更多线程。

<bean name="readService" class="com.yhd.arch.laserbeak.provider.SpringServiceExporter">

<property name="appMeta" ref="appMeta"></property>

<property name="service" ref="rs" />

<property name="serviceInterface" value="com.yhd.arch.spi.IReadService" />

<property name="serviceVersion" value="0.1-jl"></property>

<property name="methodWorkerCount" value="50"></property>

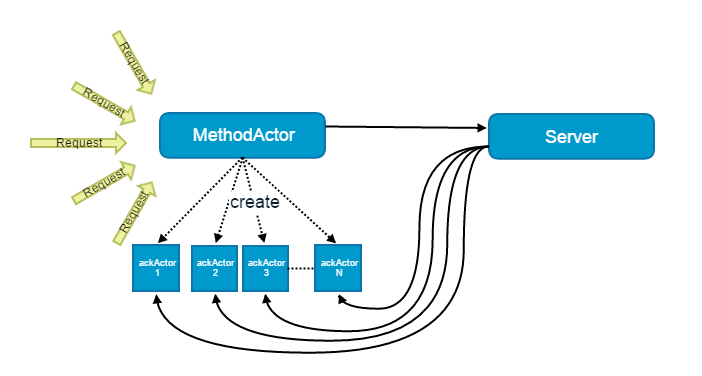
<property name="initStart" value="true"></property>

</bean>

图4.22 服务工作线程池的配置

### 4.5.6 限流接口的总请求数设计

Hedwig Akka客户端流控会限制某个方法的总请求数，如图4.23所示。

 图4.23 Hedwig methodActor和AckActor的关系（客户端流控）

一个服务由多个方法方法构成，一个方法会对应一个MethodActor，发送一个请求后，MethodActor会对应产生一个AckActor，AckActor表示对请求的发送/接收处理。一个接口发送了多少消息等待处理，就会有多少个AckActor实例。所以，Hedwig客户端根据MethodActor下AckActor的个数也就是总请求数做流控，如果请求量大于阀值，就拒绝后续请求，配置如表4.2所示。

表4.2 Hedwig akka客户端配置

|  |  |
| --- | --- |
| Option and Default Value | description |
| photon.emmitor.channel.count | core size of connection pool for upward compatible |
| photon.emmitor.channel.max | max size of connection pool for upward compatible |
| photon.emitter.throttler=  com.yhd.arch.photon.core.MethodActorThrottler | method throttler class name |
| photon.emmitor.method.limit=800 | pending request count in client |

### 4.5.7 限流接口的时间窗请求数设计

Hedwig Akka服务端流控如表4.3所示，根据某个时间窗口中请求数TPS的阈值做流控，这个阈值默认值为3000TPS。

表4.3 Hedwig akka 服务端配置

|  |  |
| --- | --- |
| Option and Default Value | Description |
| photon.remote.endpoint.receiver.count=100 | server entry actor count |
| photon.endpoint.throttler=  com.yhd.arch.photon.plugin.EndpointThrottler | throttler class name |
| photon.endpoint.throttle.enable | switcher of server throttle, value is disable/enable |
| photon.endpoint.throttle.tps=3000 | server throttle tps |
| photon.endpoint.throttle.interval=3000 | throttle calcuate interval |

### 4.5.8 超时与重试机制设计

如图4.24所示，Hedwig客户端可以设置两个超时相关的参数(timeout和readTimeout)

1.readTimeout的值为服务端处理请求的时间加上网络传输时间; 创建并维持网络连接所允许的最大等待时间, 单位为毫秒.

2. timeout是从发起请求到收到响应所允许的最大等待时间,单位为毫秒.timeout的值根据准许的重试次数可以设置为readtiemout的3-5倍;timeout等于重试次数 乘以readTimeout;

<bean id="queryService\_AKKA\_2" parent="laserBeakParent\_2">

...

<property name="readTimeout" value="50000"></property>

<property name=" timeout " value="200000"></property>

<!--当服务调用出错时, 是否允许重试.-->

<property name="autoRedo" value="true"></property>

...

</bean>

图4.24 Hedwig 超时时间

### 4.6 Hedwig监控

及时发现问题是解决问题的重要保障。及时发现问题的手段是完善监控与报警，包括从硬件、网络到应用业务的监控。服务监控从底层硬件到上层应用服务一般有以下的三个层次：

### 4.6.1 基础设施监控

监控会探测硬件瓶颈相关的早期迹象并捕获硬件故障信号，在出现更大的问题之前对其进行应对。基础设施监控的范围包括机器的健康状况、CPU使用、内存消耗以及网络带宽，基于这些监控信息，能够判断基础设施的当前状态，从而进行报警或者必要的升级扩展。

Hedwig异常报警如图4.25所示,表示某个服务节点的网络出流量异常。

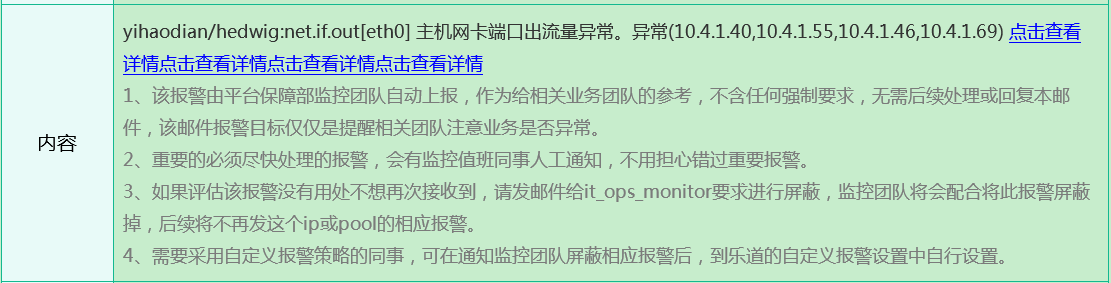
****

图4.25 Hedwig机器网络出口流量异常报警

### 4.6.2 应用级别监控

应用级别的监控涉及到监控各种服务器的状态，如应用服务器、数据库服务器等，要监控的参数则是与应用或工具相关的。要监控的指标包括JVM内存，JVM GC情况，线程数，磁盘IO，索引的读取/写入操作。

一号店乐道平台和一号店Zabbix监控系统的应用级别监控包括以下这些指标，如图4.26所示：JVM(gc,线程)，CPU，内存，网卡/磁盘读写，Tomcat线程池，连接数等。

可用性监控主要观察系统是否存活，服务可用率是否低于阀值。如图4.27所示，Zookeeper节点、Hedwig节点、Detector节点宕机或者资源不足，会有报警邮件。

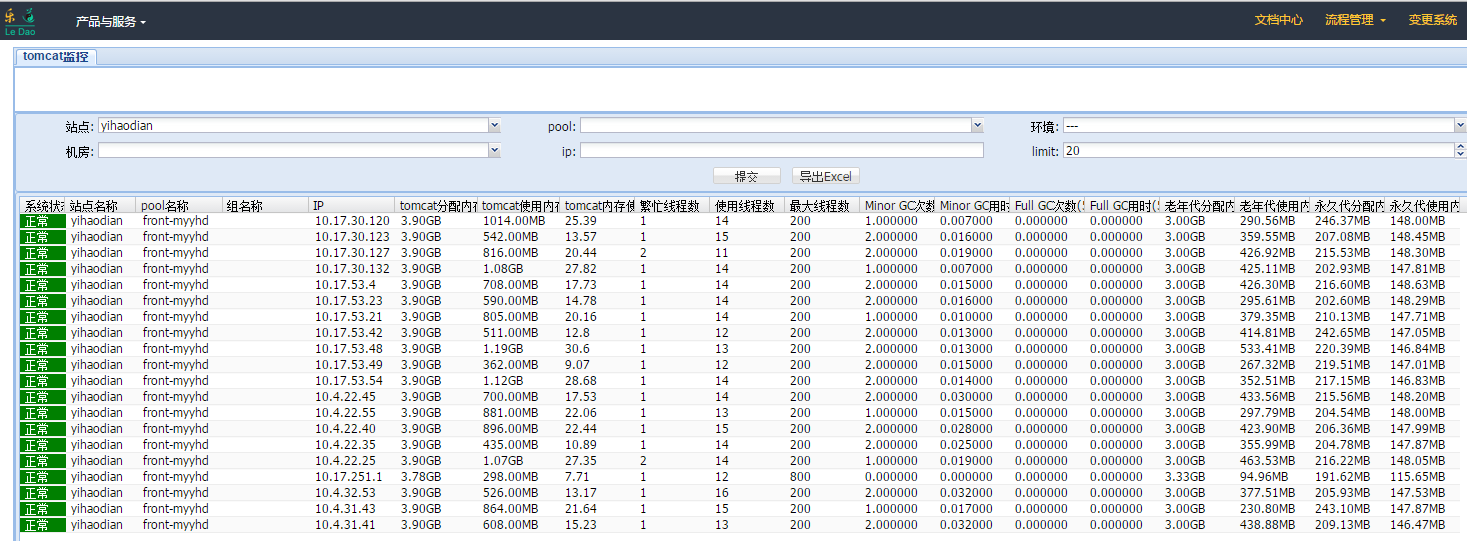
****

图4.26 乐道里Hedwig应用监控

cid:image001.png@01D2BDCB.4A4544B0图4.27 Hedwig服务 zookeeper集群报警

### 4.6.3 服务监控

服务监控就是要监控每个服务的吞吐量和性能，进而确保在任何时间都能满足SLA的需求。这种类型的监控一般都需要对应用进行Instrument操作，让Instrumentation是可配置的，通过收集器（collector）收集应用的状态，并阶段性地将这些状态发送到永久存储、分析器和预警系统中。此类监控往往会产生大量的数据，因此有可能会影响到性能。

服务监控可以分为对所有服务调用的整体指标的监控和对单个服务\方法处理能力的监控。

1. 处理能力的监控

如图4.28、4.29、4.30、4.31所示，Detector对应用的性能和健康度监控, 监控的指标有服务调用的成功率/失败率/响应时间/调用量。问题排查时可以查异常调用的错误日志和明细日志，当大促错误量比较大的时候会做错误日志的抽样，保证日志量不会太多。服务调用链路图能看到整个链路的全貌和服务调用的情况，也能看到单个链路上细粒度(Pool名/服务名/方法名/分组名)的调用情况。



图4.28 Detector服务治理

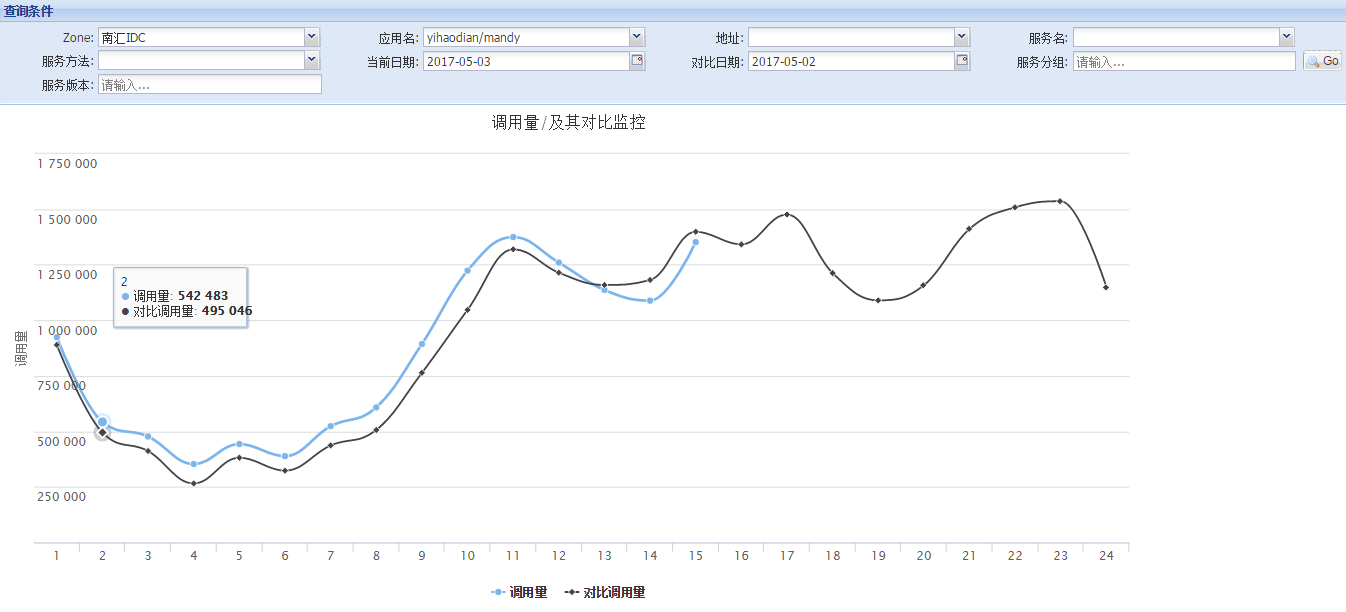


图4.29 服务每小时调用量

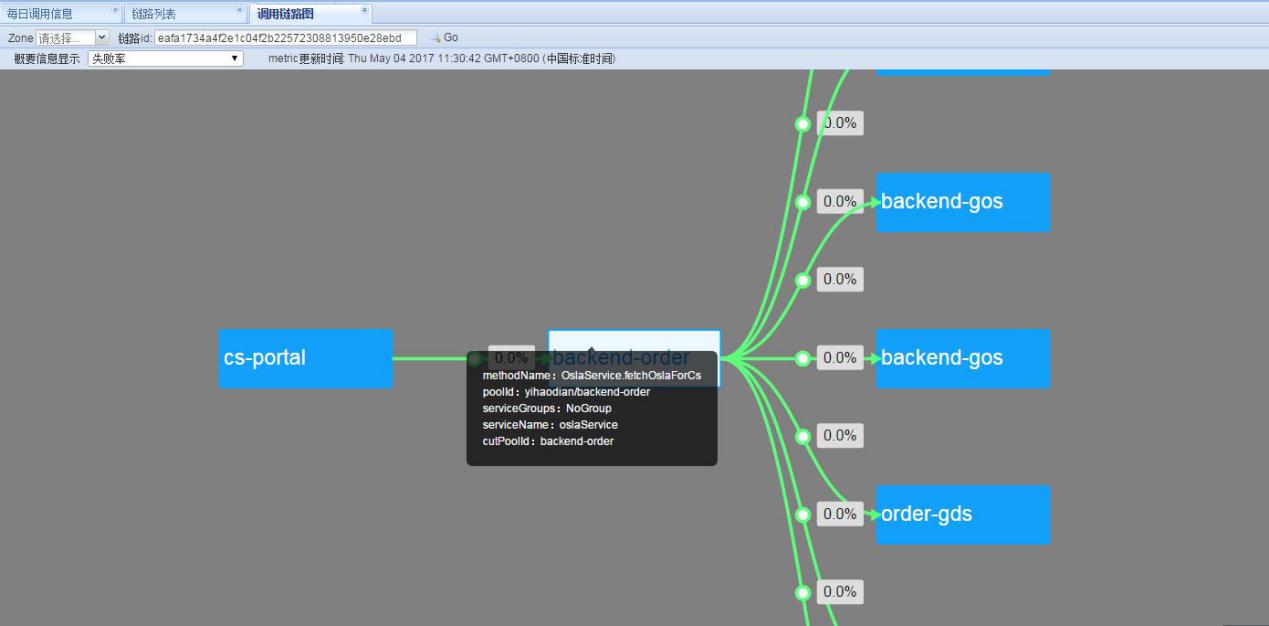


图4.30 Detector链路调用关系

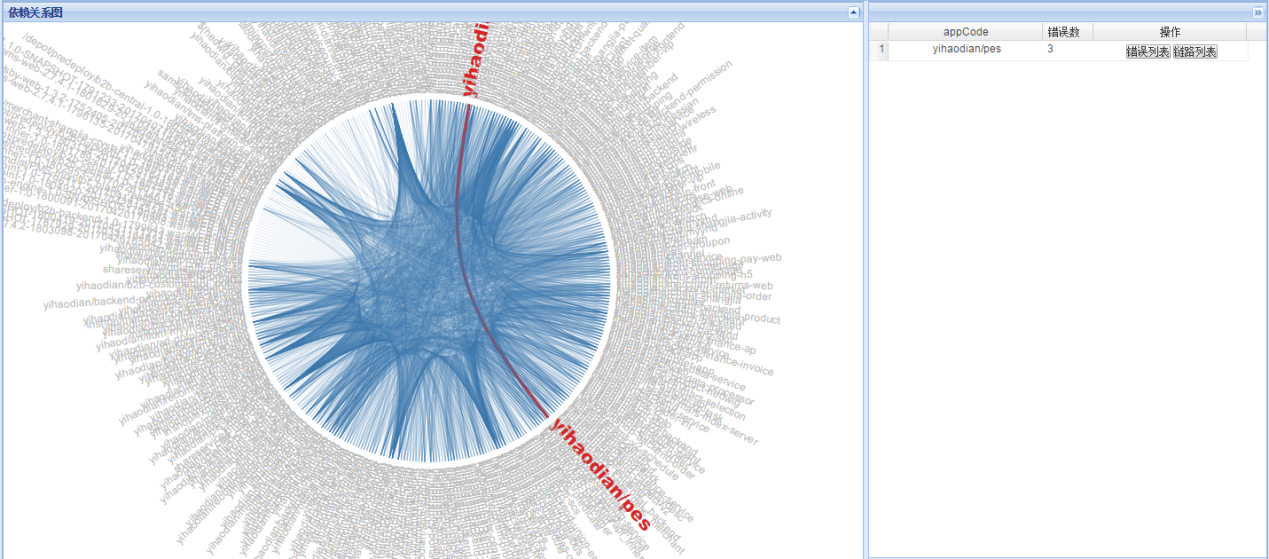


图4.31 服务Pool之间的依赖关系图

1. 整体指标监控

如图4.32所示，现在,一号店的Hedwig南汇IDC每天有1200个服务的31亿多次的调用，成功率为99.99%以上，平均耗时在19ms左右，包括170个左右的Pool。

Hedwig IDC中服务整体指标包括整体的调用量，成功率，平均耗时。

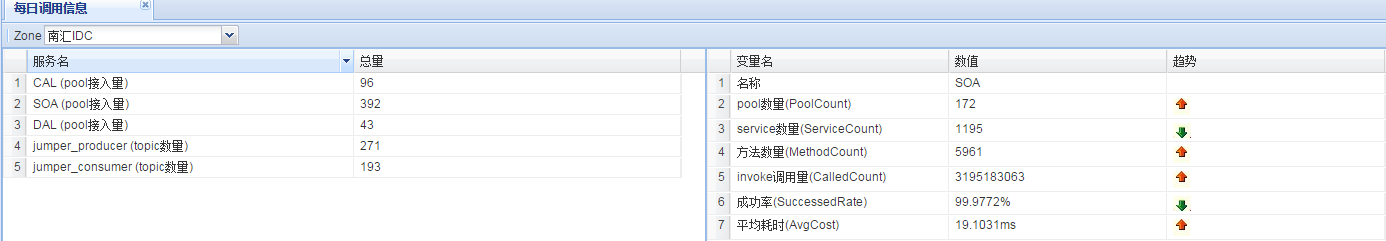


图4.32 一号店南汇IDC中服务整体的每天调用量

### 4.7 本章小结

本章论述了Hedwig的技术选型、功能模块、功能组件、容错和服务监控的设计和实现。

# 5 Hedwig功能与性能测试

### 5.1 测试环境

测试环境所有服务器使用同一规格。服务提供者部署有15台服务器，服务消费者部署有20台服务器。注册中心使用1台服务器，开启3个Zookeeper进程。监控中心使用1台服务器，它使用的流式计算使用共享集群。

1. 硬件环境

表5.1 硬件配置

|  |  |
| --- | --- |
| 参数 | 配置 |
| 机型 | PowerEdge R730 |
| CPU | 主频2.10GHz，  24核，64位 |
| 内存 | 32G |
| 网络 | 带宽1000Mbps |
| 系统环境 | CentOS 6 |

1. 软件环境

表5.2 软件配置

|  |  |
| --- | --- |
| JDK版本 | JDK-8u191 |
| 开发工具 | IntelliJ IDEA 15 |

### 5.2 服务调用测试

RPC框架最核心的功能是服务调用，服务消费者与提供者应当可以基于契约进行远程交互。在测试阶段需要验证的是框架的准确性与可靠性，所以在测试用例的设计上要覆盖执行正常与异常的场景。

### 5.2.1服务调用成功测试用例

1. 测试目的和过程

在TestForService接口中增加方法methodA(),并直接返回TestMsg对象。TestForService各提供一个消费者和提供者实例。提供者接收并处理来自消费者的methodA()调用。消费者获取返回结果并验证。

1. 测试结果

测试通过，提供者成功响应了消费者实例的调用，返回的对象经比对与预期的一致。

### 5.2.2 服务提供者执行异常测试用例

1. 测试目的和过程

在TestForService接口中增加方法methodError(),提供者实现methodError方法，实现逻辑是直接抛出RuntimeException异常。TestForService各提供一个消费者和提供者实例。消费者实例调用methodError方法。

1. 测试结果

测试通过，提供者执行调用时产生的异常正确地返回到消费者。消费者实例对于methodError方法的调用抛出RuntimeException异常。

### 5.2.3 服务提供者处理超时测试用例

1. 测试目的和过程

在TestForService接口中增加方法methodTimeout(),提供者实现methodTimeout方法，实现逻辑是在方法执行过程中休眠当前线程5秒钟。TestForService各提供一个消费者和提供者实例。消费者实例设置调用超时时限为3秒钟。消费者实例调用methodTimeout方法

1. 测试结果

测试通过，消费者在时限内没有获取返回结果并抛出超时异常。消费者调用methodTimeout方法时抛出RpcTimeoutException异常对象。

### 5.3 集群与负载均衡测试

### 5.3.1测试目的和过程

服务框架通过集群来实现服务吞吐量的水平扩展，服务消费者按照服务提供者列表进行软负载均衡。在这一测试用例中，将安排设置了不同负载均衡策略的消费者实例对提供者实例集群发起集中调用，通过统计消费者实例本地调用日志，验证流量分配是否符合负载均衡算法。

此测试用例中部署20个消费实例，15个服务端实例，使用基于权重的轮询算法(Weight Round Robin)，其中模拟两个节点处理能力比较弱，权重为1/7。

### 5.3.2 测试结果

如图5.1所示，能看到2个服务节点的调用量比较小，调用量差不多是1/7的比例，说明使用的基于权重的轮询算法已生效并能正常工作。

图5.1 负载均衡监控

### 5.4 性能测试

### 5.4.1 测试目的和过程

1. 服务提供者测试：并发启动多个客户端，配置连接数60，并发数120，分别请求1k到10M的字符串，测试服务端的平均TPS和平均响应时间。

2. 服务消费者测试：

启动一个消费者，配置连接数，在并发线程数分别为1、10、30、50的情况下，分别进行如下场景测试：传入1k-10M长度的字符串，提供者不做任何处理，直接返回调用结果。测试维持1个小时，主要测试服务框架的序列化和网络I/O的性能。

### 5.4.2 测试结果

从表5.3所示，系统的最佳请求数在1.5万左右，此时的瓶颈主要是网卡带宽，这个结果对生产环境中的服务部署有较大的参考价值。

表5.3 Hedwig0.2服务端测试结果

|  |  |  |
| --- | --- | --- |
| 场景测试 | 平均TPS | 平均响应时间 |
| 请求1k长度的字符串 | 约1.5万 | 0.6ms |
| 请求50k长度的字符串 | 约2千 | 7.8ms |
| 请求200k长度的字符串 | 约6百 | 17.6ms |

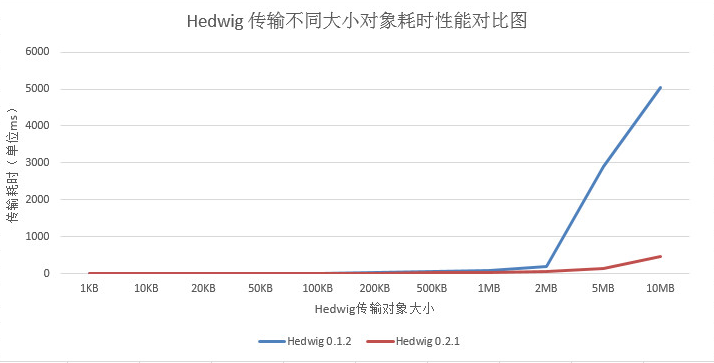
Hedwig0.1版本基于Hessian协议， 0.2版本基于Akka框架。Hedwig0.2选用了Akka框架提供的Actor模型，它是普遍的并发模型[9]之一，它提供了全异步化的功能，有更高的吞吐，更容易的编程模型。Hedwig 在Akka的类库上研发了一个photon项目， 其中序列化是并行的， 相对于Hessian有更高的性能。经过测试，Hedwig传输对象和响应时间对比见图5.2。

图5.2 Hedwig传输对象和响应时间对比

当传输对象大小在100KB以内，0.2版本和0.1版本性能基本相当。当传输对象大小在100KB以上时，0.2版本比0.1版本性能要优越。当传输对象大小在2MB左右，0.2版本性能优势十分明显。

Hedwig的设计目的是为了满足高并发小数据量的RPC调用,在大数据量下的性能表现并不好,建议使用RMI或HTTP协议。在相同大小对象传输上，0.2版本性能要优于0.1版.在吞吐持续变大的情况下，Hedwig0.2的响应时间没有变的太大，也就是说在单位时间内，Hedwig0.2能承受更大的TPS并发量，能承受的系统吞吐量更大。所以，Hedwig0.2为公司业务团队提供了一个高并发、高吞吐量、高性能的分布式服务框架。

### 5.5 本章小结

论述了Hedwig的负载均衡测试和性能测试，验证了Hedwig是一个高并发、高吞吐的分布式服务框架。

# 6 总结和展望

从12年开始一号店就开始了公司整体服务化的进程，其中最主要的产品之一是服务治理框架Hedwig。Hedwig几乎承载了公司商城网站和公司内部所有的业务。Hedwig在这几年的发展中也是在各种大促和大流量的情况下经受过大考，一致得到了公司的认可。

Hedwig的核心的功能和非功能特性已经实现了，但和阿里的Dubbo，唯品会的OSP，点评的Pegion相比，还是有可以借鉴的地方，有些算法和细节还有精雕细琢的空间。

服务调用错误每天都在发生，错误模式是很好的一个总结，因此服务框架设计和实现需要参考这部分的内容。

Hedwig监控方面的功能基本都已经实现了，但在APM相关的全链路服务监控还比较薄弱，这是以后要加强的一个方向，能够更快速响应错误的服务调用。

# 参 考 文 献

1. Volter M. Remoting Patterns - Foundations of Enterprise, Internet and Realtime Distributed Object Middleware[J]. IEEE Computer Society, 2006, 8(6):60-68.
2. Brewer E. CAP twelve years later: How the "rules" have changed[J]. Computer, 2012, 45(2):23-29.
3. Frank Buschmann ， Keclin Henney, Douglas C.Schmidt (作者) 肖鹏 陈立 (译者)．面向模式的软件架构：分布式计算的模式语言(卷4)［Ｍ］.北京：人民邮电出版社，2010：348．
4. 李林锋．分布式服务框架:原理与实践［Ｍ］.北京：电子工业出版社，2016：312．
5. 曾宪杰．大型网站系统与Java中间件实践［Ｍ］.北京：电子工业出版社，2014：340．
6. Michael T. Nygard．Release It! Design and Deploy Production-Ready Software［Ｍ］.美国：The Pragmatic Programmers，2007：300．
7. Sam Newman．Building Microservices：Designing Fine-Grained Systems 微服务设计［Ｍ］.北京：人民邮电出版社，2016：228．
8. 张开涛. 亿级流量网站架构核心技术——跟开涛学搭建高可用高并发系统[M]. 北京:电子工业出版社, 2017.
9. 布彻, (Paul, Butcher), (作者), 黄炎, (译者). 七周七并发模型[M]. 北京:人民邮电出版社, 2015.
10. 宸存, 巴斯, 克莱门茨,等. 软件构架实践(第二版)[M]. 清华大学出版社, 2004.
11. Hamilton J. On designing and deploying internet-scale services[C] Conference on Large Installation System Administration Conference. USENIX Association, 2007:18.
12. Mcheick H, Mohammed Z R, Lakiss A. Evaluation of Load Balance Algorithms[C] International Conference on Software Engineering Research, Management and Applications. IEEE, 2011:104-109.
13. 郭正敏. 基于SOA架构的分布式服务化治理方案的研究[D]. 南京邮电大学, 2016.
14. 陶明. 一种分布式服务框架的设计与实现[D]. 北京邮电大学, 2013.
15. 张羽. 一种分布式服务治理框架的设计与实现[D].北京交通大学,2016.
16. 肖仲垚. 微服务通信框架的设计与实现[D].北京交通大学,2017.
17. 章耿. 服务化框架技术选型与京东JSF解密[EB/OL]. http://mt.sohu.com/20161222/n476595246.shtml, 2016-12-22
18. 臧秀涛. 专访阿里巴巴毕玄：异地多活数据中心项目的来龙去脉[EB/OL]. http://www.infoq.com/cn/articles/interview-alibaba-bixuan, 2015-04-07
19. Chris, Richardson. Microservices patterns[EB/OL]. http://microservices.io.
20. Tony, Mauro. Adopting Microservices at Netflix: Lessons for Architectural Design[EB/OL]. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, 2015-02-19
21. 刘丁. 序列化和反序列化[EB/OL]. http://tech.meituan.com/serialization\_vs\_deserialization.html, 2015-02-26
22. 林世洪. 京东大促备战思路和方法2.0解密[EB/OL].http://zhuanlan.51cto.com/art/201612/525311.htm, 2016-12-19
23. 阿里小邪. 稳定性思考-强弱依赖 [EB/OL]. http://jm.taobao.org/2012/10/31/stability-considerations-dependent-strength/.
24. 阿里小邪. 稳定性思考-强弱依赖2[EB/OL]. http://jm.taobao.org/2012/10/31/stability-considerations-dependent-strength-2/.
25. Apache Dubbo - home[EB/OL]. http://dubbo.apache.org/en-us/index.html.
26. 美团绿麟. 服务容错模式[EB/OL]. <http://tech.meituan.com/service-fault-tolerant-pattern.html>.
27. Jason, Goodwin. Akka入门与实践 [M]. 北京:人民邮电出版社 , 2017.
28. Shailesh, Mangal. Monitoring Connect applications[EB/OL]. <https://developer.atlassian.com/blog/2015/08/monitoring-connect-applications/>.
29. blogdevteam. Dubbo 团队成员梁飞专访 [EB/OL].http://blog.csdn.net/blogdevteam/article/details/8173233.
30. 梁宇鹏. 单元化架构，为什么要用以及我们如何做到[EB/OL]. <http://www.infoq.com/cn/articles/how-weibo-do-unit-architecture>.
31. 邱小侠. 微服务（Microservice）那点事[EB/OL]. <https://yq.aliyun.com/articles/2764>.
32. Jonas, Boner. Scalability,Availability & Stability Patterns[EB/OL]. <http://www.mif.vu.lt/~donatas/Vadovavimas/Temos/Reactive%20systems/Scalability/2010%20Scalability%20patterns%28slides%29.pdf>.
33. 点评. RPC framework of DIANPING[EB/OL]. <https://github.com/dianping/pigeon>.
34. zookeeper官网[EB/OL]. <http://zookeeper.apache.org/>.
35. 微博Motan[EB/OL]. <https://github.com/weibocom/motan>.
36. Birrell A D. Implementing remote procedure calls[J]. Acm Transactions on Computer Systems, 1984, 2(5):3.
37. 童云兰. Java并发编程实战[M]. 机械工业出版社华章公司, 2012.
38. Erich, Gamma, Richard, Helm, Ralph, Johnson, John, Vlissides, 李英军(译者). 设计模式 可复用面向对象软件的基础[M]. 机械工业出版社, 2007
39. 庄鹏. 基于ZooKeeper的分布式服务中间件设计与实现[D].深圳大学,2016.
40. Gartner. Service-oriented Architecture Scenario[J].Gartner group,2003:04-16
41. 徐培德.基于Java的分布式远程过程调用框架的研究与实现[D].上海交通大学,2016.DOI:"".
42. 李艳鹏、, 杨彪. 分布式服务架构:原理、设计与实战[M]. 北京:电子工业出版社, 2017.
43. 倪超. 从Paxos到Zookeeper:分布式一致性原理与实践[M]. 北京:电子工业出版社, 2015.
44. 纽曼. 微服务设计[M]. 北京:人民邮电出版社, 2016.
45. SOFARPCLab. 【剖析 | SOFARPC 框架】系列之 SOFARPC 同步异步实现剖析 [EB/OL]. <https://mp.weixin.qq.com/s/VJhABLzBs9h0_pLfIJNnHQ>.
46. SOFARPCLab明不二. 【剖析 | SOFARPC 框架】系列之 SOFARPC 路由实现剖析[EB/OL]. <https://mp.weixin.qq.com/s/kjAYRxLwiSgFAjqJU-z41Q>.
47. SOFARPCLab. 【剖析 | SOFARPC 框架】系列之 SOFARPC 单机故障剔除剖析[EB/OL]. <https://mp.weixin.qq.com/s/WusXmhMnsvQ1tQh5wiCyDw>.
48. SOFARPCLab. 【剖析 | SOFARPC 框架】系列之连接管理与心跳剖析[EB/OL]. <https://mp.weixin.qq.com/s/JErQfMtWf2x2g0gbfxMWbA>.
49. 碧远. 剖析 | SOFARPC 框架之总体设计与扩展机制[EB/OL]. <https://mp.weixin.qq.com/s/ZKUmmFT0NWEAvba2MJiJfA>.
50. SOFARPCLab, 水寒. 【剖析 | SOFARPC 框架】系列之 SOFARPC 数据透传剖析[EB/OL]. <https://mp.weixin.qq.com/s/WzhzJtawjRf-jC1jWMrVuw>.

# 致 谢

本文是在我的导师龚奕利教授的悉心指导下完成的。从课程的学习、论文的选题及最终定稿都得到了导师的帮助。导师严谨的治学态度，平易近人、诲人不倦的品格给我留下了深刻的印象，使我终生受益。在此，谨向导师表示最深深的敬意和由衷的感谢!

同时感谢武大计算机学院，感谢所有的授课和指导老师，对我在学业上的帮助和生活上的关心!正是你们孜孜不倦的传道、授业、解惑，使我在思想上和学业上都取得了巨大的进步。

感谢在这两年多的学习和生活中帮助、关心我的同学以及给我的论文提供许多宝贵素材的朋友们。有了这么多人的帮助，使我顺利的完成了我的研究生阶段学习，在今后的工作中，我会将这些人的帮助化为自己前进的动力。

最后，我要特别感谢我的家人长期以来对我默默的奉献和支持,是他们为我提供了坚忍不拔和不断进取的强大精神动力，使我能顺利完成学业。家是我终生的精 神寄托和心灵归宿，也是我今后人生道路上扬帆远航为理想打拼的起点。感谢我的父母为我营造了一个的温馨的港湾，当我身心疲弊时能得以憩息和修养。

在毕业之际衷心的祝愿你们生活美满、前程似锦!