一 传统垂直mvc项目

1.垂直架构图

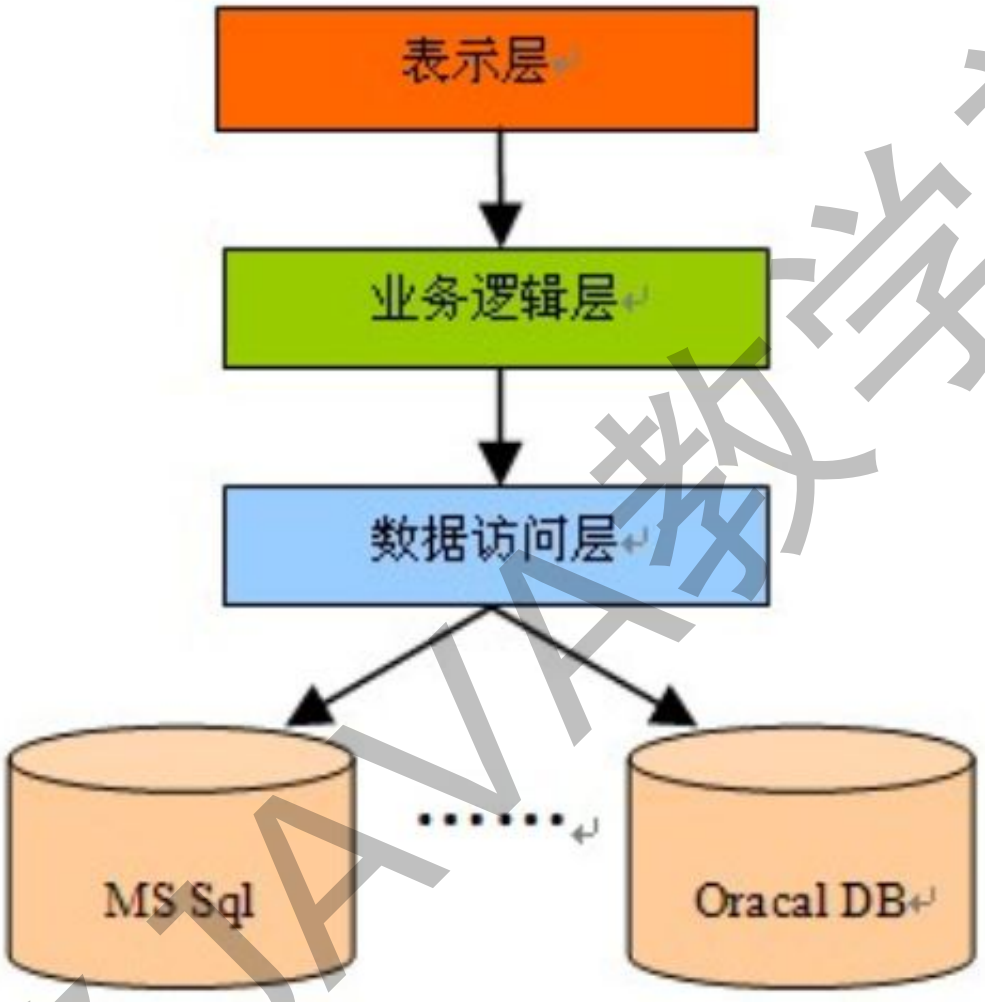


图 1 三层系统的分层式结构

> 通常mvc并不包括数据访问层,运行也比较简单,直接运行在一个tomcat等web容器中即可,适合小型项目

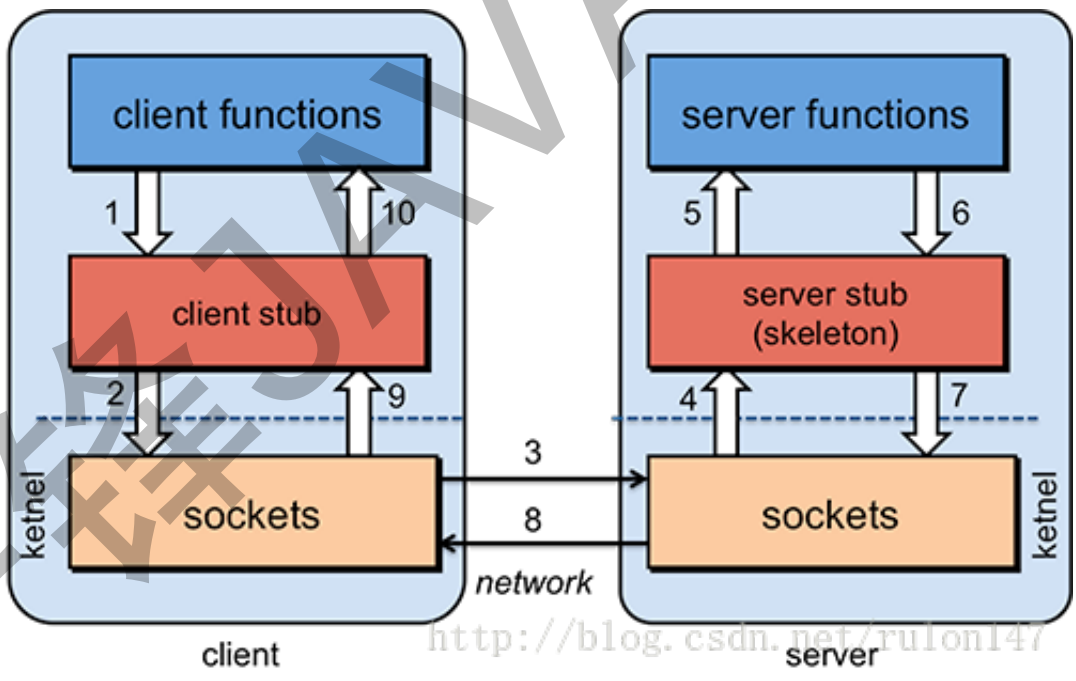
垂直架构的缺点

> 随着业务的不断发展,应用规模越来越大,问题越来越凸显,比如:

1)复杂应用的开发维护成本变高,部署效率逐渐降低,一个功能出问题,整个系统就得重新打包
2)团队协作效率变差,公共功能重复开发,代码重复率太高
3)系统可靠性变差,流量,负载均衡,数据库压力变大,因为在一个进程中,如果出现内存溢出等故障,将导致整个节点崩溃,然后集群中的其它节点也会如此
4)维护和定制困难,无法随时拆分,修改一处,牵一发动全身
5)新功能上线周期变长,因为公共功能的变更导致测试工作量激增,因为重复代码多,一个地方修改需要同时修改多个地方,然后修改后继续测试
新功能无法独立打包测试,需要和整个系统进行一起打包测试,出现bug会导致整个系统重新部署,强耦合导致效率低下

二 RPC架构

RPC（Remote Procedure Call Protocol）——远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。
RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易
RPC采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器。
首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息到达为止。
当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行。



1 RPC架构分为三部分：

> 服务提供者，运行在服务器端，提供服务接口定义与服务实现类。

> 服务中心，运行在服务器端，负责将本地服务发布成远程服务，管理远程服务，提供给服务消费者使用。

> 服务消费者，运行在客户端，通过远程代理对象调用远程服务

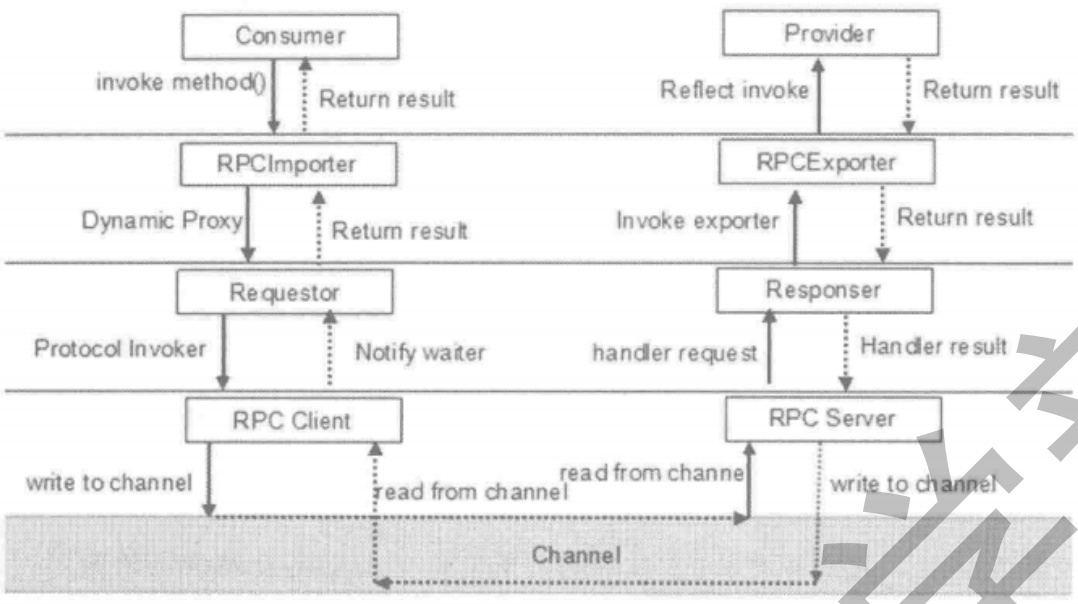


图 1-4 RPC 框架原理图

2 RPC框架的核心技术点

1)远程服务提供者需要以某种形式提供服务的调用相关信息,包括但是不限于服务接口定义,数据结构,或者中间态的服务定义文件
服务调用者需要通过一定的途径获取远程服务的调用相关信息,例如接口的定义jar包等
2) 远程代理对象:服务调用者调用的服务实际是远程服务的本地代理,对于java而言,他的实现就是jdk动态代理,通过动态代理拦截机制,将本地调用封装成远程服务调用
3)通信:RPC框架与具体的协议无关,只要双方遵从约定好的即可,比如可以是http invoke,可以是rmi invoke,也可以是其他任意的二进制压缩协议
4)序列化:远程通信,需要将对象转换成二进制数据进行网络传说,不同的所以需要将数据序列化,不同的序列化框架支持的数据类型,数据包大小,异常类型或者性能都不同
不同的RPC框架针对的场景不同,因此技术选择也各不相同,一些框架支持多种序列化框架,甚至支持用户自定义序列化框架

3 RPC框架的问题

在大规模服务化之前,应用可能只通过RPC框架 简单的暴露和引用远程服务,通过配置的url地址进行远程调用,路由通过F5硬件负载均衡进行简单的负载均衡
当服务越来越多,服务的url越来越多,管理越来越困难,负载均衡单点压力变大,此时需要一个服务的注册中心,动态的注册和发现服务,使服务位置透明,消费者在本地缓存服务提供者列表,实现软负载均衡,可以降低对F5硬件负载的依赖,降低硬件成本
随着业务的发展,服务间的依赖关系变的错综复杂,甚至分不清哪个应用需要在哪个应用之前启动,需要一个分布式消息跟踪系统可视化战士服务调用链,用于以来分析,业务调用路径梳理,防止服务架构腐化
服务的调用量越来越大,服务的容量问题就出现了,某个服务需要多少机器支撑,什么时候该加机器
服务上线容易下线难,上线的审批,下线的通知,需要统一的服务生命周期管理流程进行管控,不同的服务安全权限不同,如何保证敏感数据服务不被误用,服务的访问安全策略如何定制
服务化后随之而来的就是服务治理问题,纯粹的RPC框架服务治理能力都不强悍,需要通过服务框架+服务治理来完成

4常用的rpc框架

1. Thrift；
2. Hadoop的Avro-RPC;
3. Hessian；
4. gRPC;

> 单论rpc的话，没太多可说的，可是如果加上服务治理，那复杂度就几何倍数增长了。服务治理里面东西太多了，动态注册，动态发现，服务管控，调用链分析等等问题这些问题，单凭rpc框架解决不了，所以现在常用的说的服务化框架，通常指的是rpc+服务治理2个点。

SOA 服务化架构

微服务

MSA也是一种服务化架构风格，正流行ing，服务划分

1. 原子服务，粒度细；
2. 独立部署，主要是容器；

MSA与SOA的对比：

服务拆分粒度：soa首要解决的是异构系统的服务化，微服务专注服务的拆分，原子服务；
服务依赖：soa主要处理已有系统，重用已有的资产，存在大量服务间依赖，微服务强调服务自治，原子性，避免依赖耦合的产生；
服务规模：soa服务粒度大，大多数将多个服务合并打包，因此服务实例数有限，微服务强调自治，服务独立部署，导致规模膨胀，对服务治理有挑战；
架构差异：微服务通常是去中心化的，soa通常是基于ESB的；
服务治理：微服务的动态治理，实时管控，而soa通常是静态配置治理；
交付：微服务的小团队作战。
感觉在有了docker后，微服务这个概念突然火了起来，总结就是微服务+容器+DevOps。

下面通过服务化架构演进图，总结一下这 4 代架构的演进历史，如图 1-6 所示。

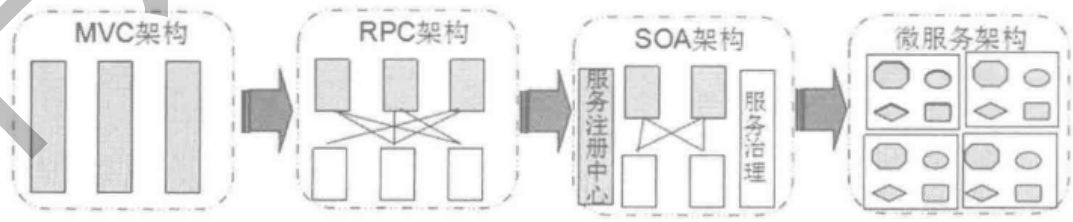


图 1-6　应用架构演进历史

第一章架构演进

分布式服务框架入门

背景

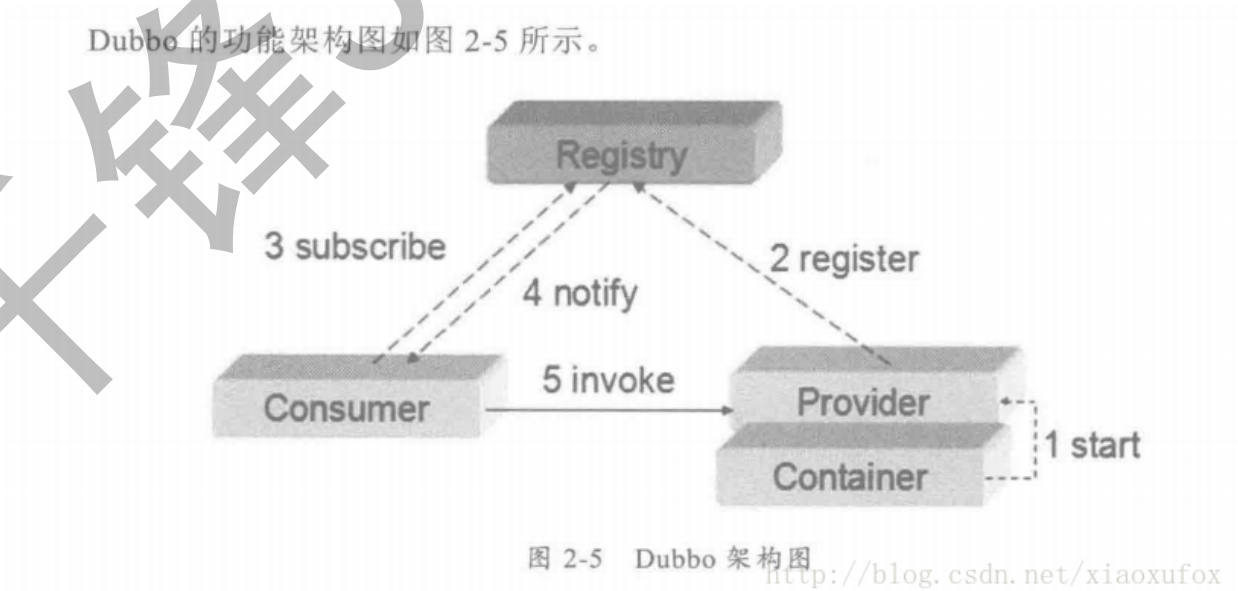
应用从集中式走向分布式

随着业务的发展导致功能的增多，传统的架构模式开发，测试，部署整个流程变长，效率变低，后台服务的压力变大，只能通过硬件扩容来暂时缓解压力，但解决不了根本性问题：
应用规模变大，开发维护成本变高，部署效率降低；
代码复用：原来是本地api调用，导致一些公用功能可能是按需开发，不统一，随意等问题；
交付面临困难：主要是业务变得复杂，新增修改测试变得困难，拉长整个流程。
通用法宝：拆分，大系统拆小系统，独立扩展和伸缩。
纵向：分业务模块；
横向：提炼核心功能，公共业务；
需要服务治理

大拆小，核心服务提炼后，服务的数量变多，而且需要一些运行态的管控，这时候就需要服务治理：
服务生命周期管理；
服务容量规划；
运行期治理；
服务安全。
服务框架介绍

Dubbo

阿里开源的Dubbo应该是业界分布式服务框架最出名的了吧，看过公司的rpc框架，Dubbo的扩展性比我们的好的多了，我们的框架每次升级，改动都很多，改天要看下Dubbo的源码了解了解扩展性。
Dubbo架构图



Dubbo服务治理

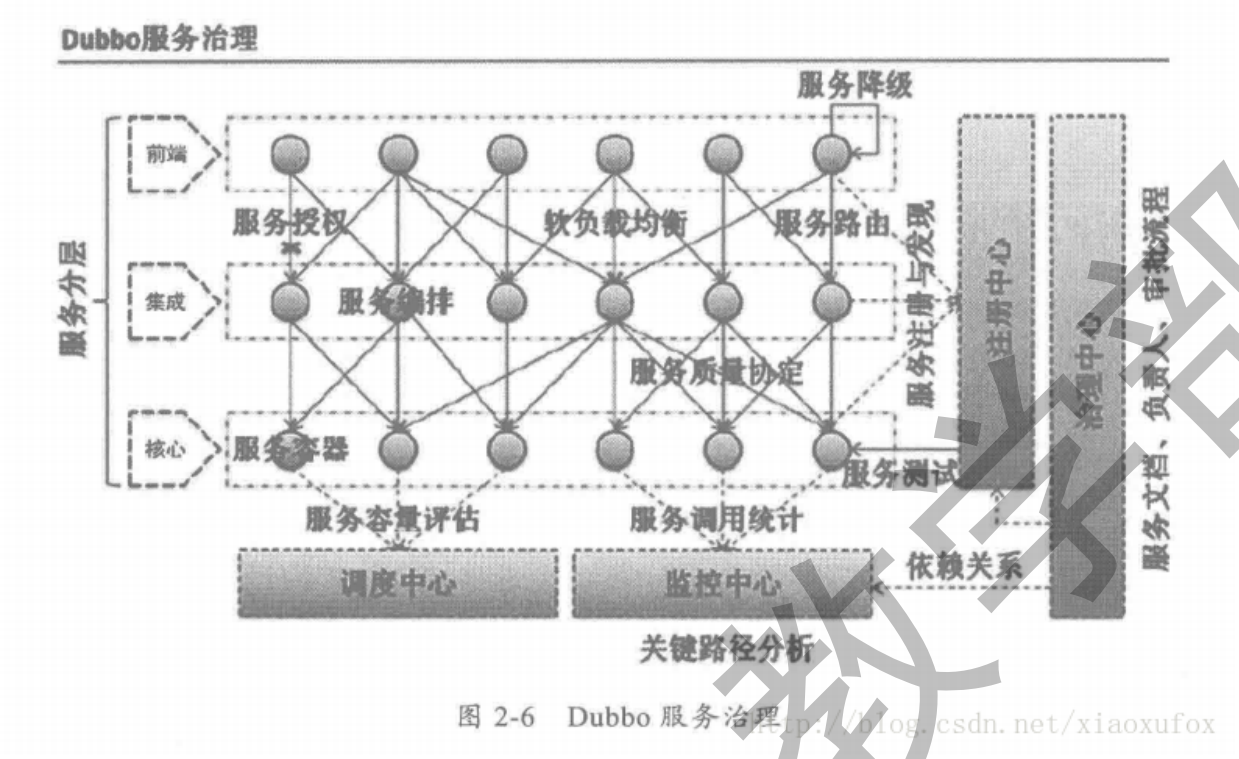


图 2-6 Dubbo 服务治理 http://blog.csdn.net/xiaoxufox

HSF

淘宝的体量决定了他对极致性能的追求，HSF跨机房特性挺牛。

HSF架构图

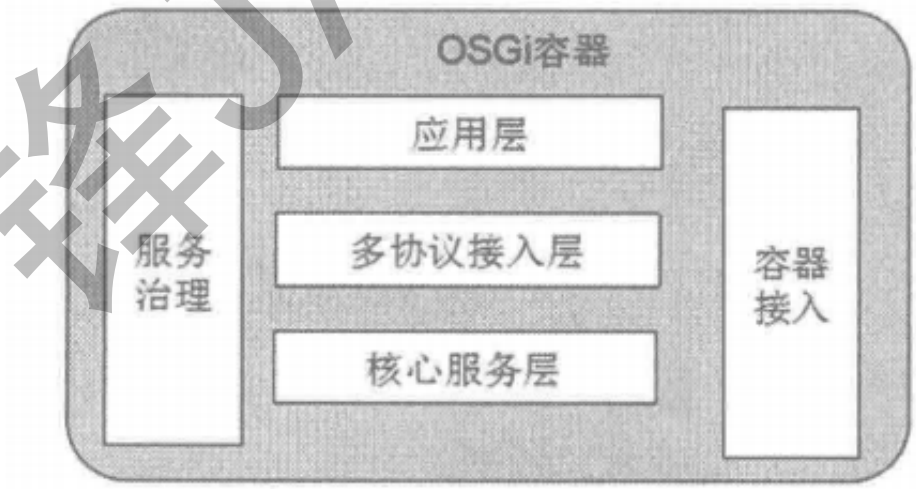


图 2-7 HSF 架构图 http://blog.csdn.net/xiaoxufox

HSF服务治理

HSF 服务治理整体结构图如图 2-8 所示。

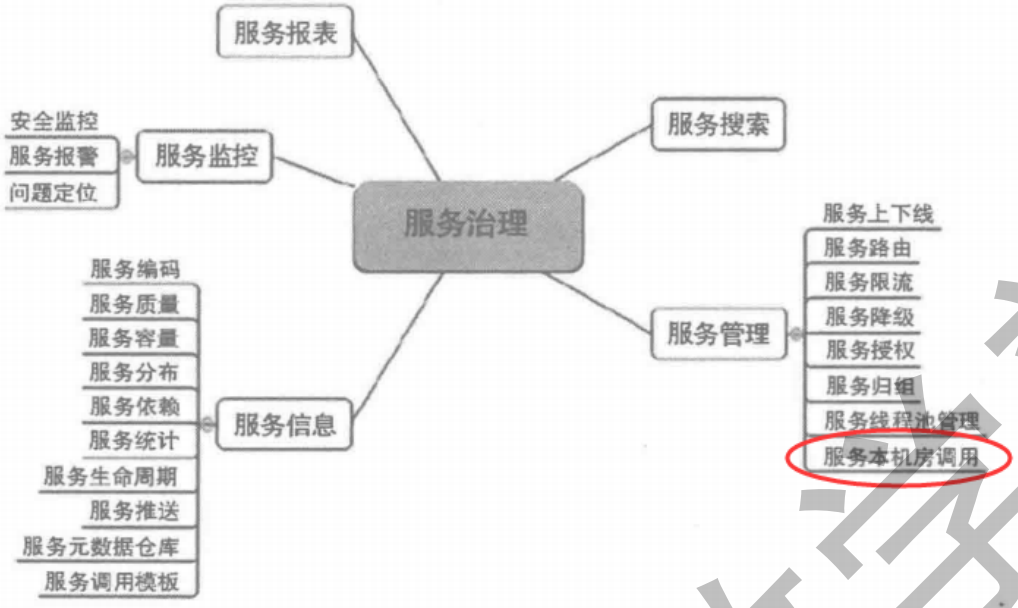


图 2-8 HSF 服务治理结构图

Coral Service

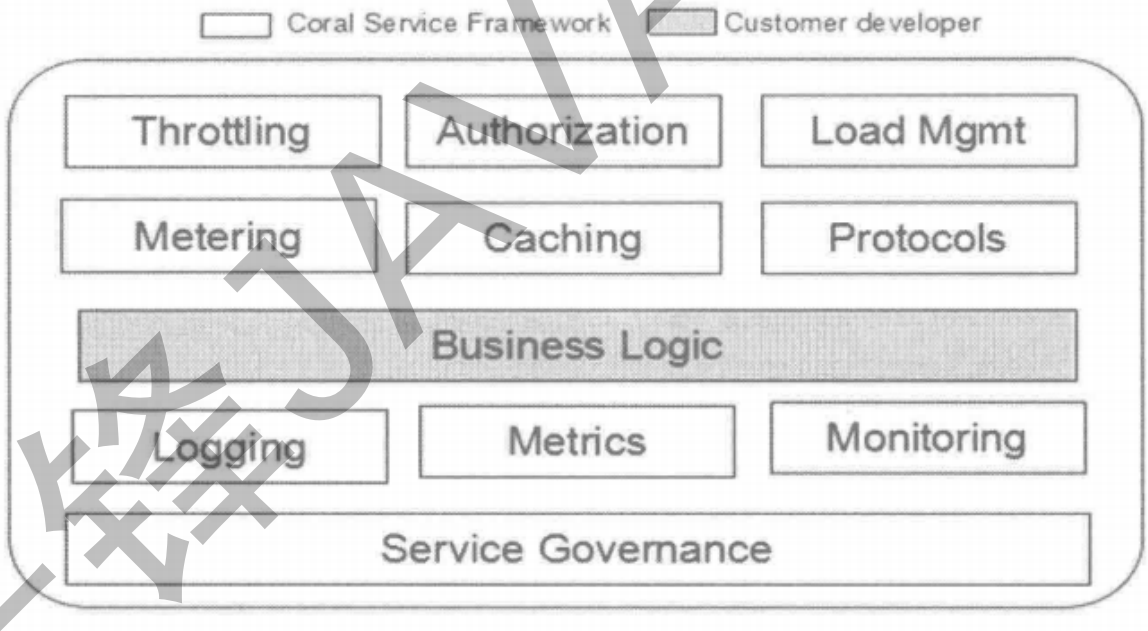


图 2-9 Coral Service Framework

框架设计

架构原理

万变不离其中，这张图可以概括rpc的一些通用原理：

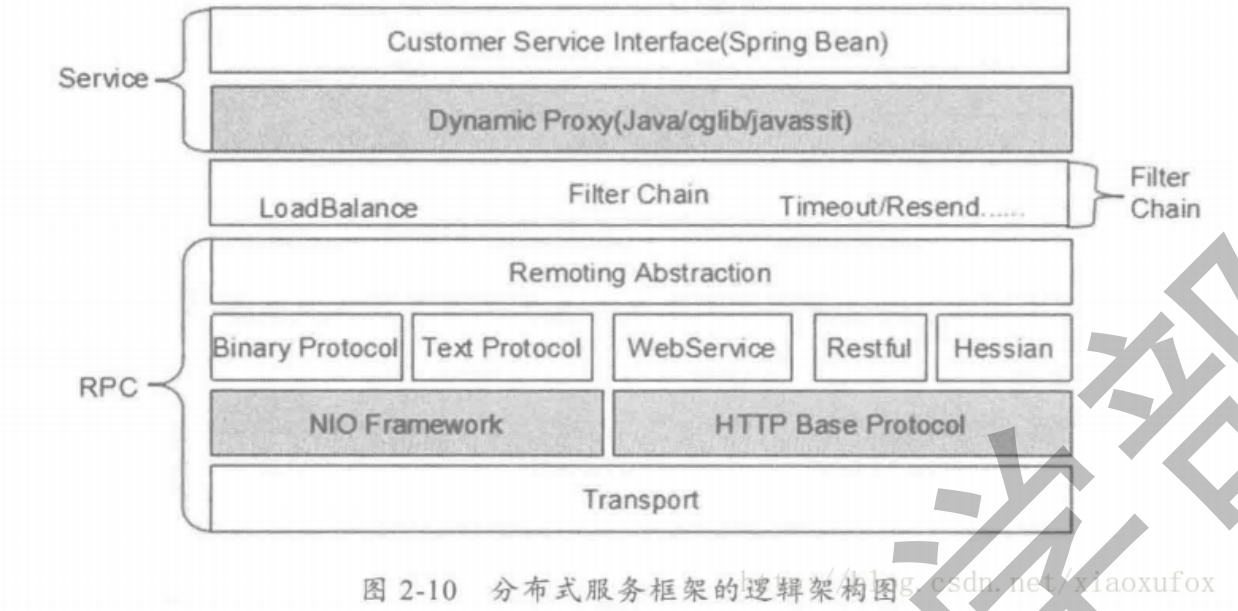
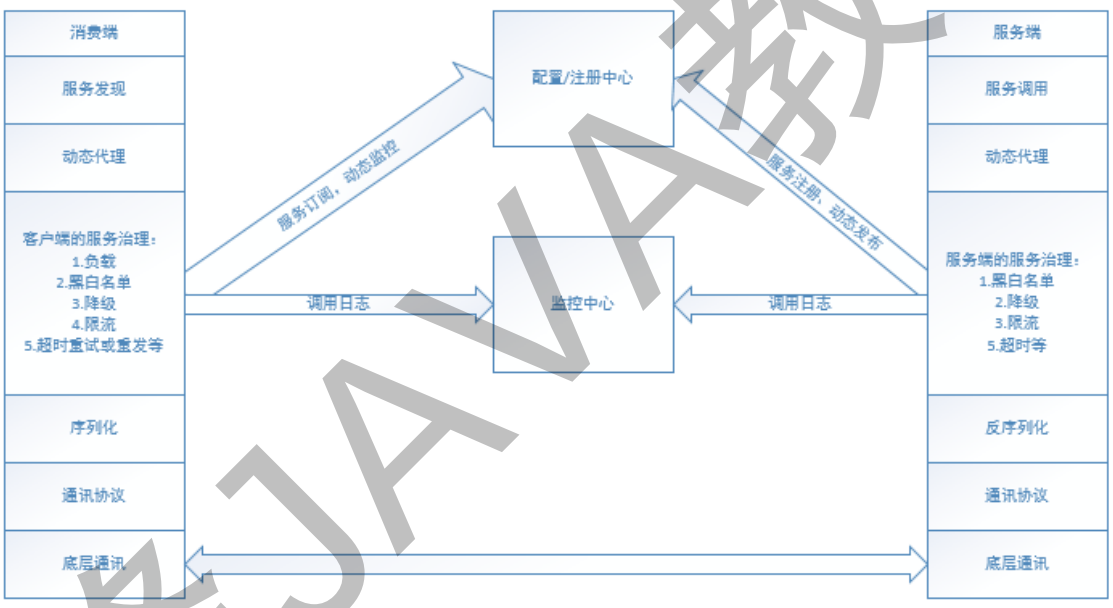


图 2-10　分布式服务框架的逻辑架构图

细化了下：



1 rpc层：底层的通讯框架，通讯协议，序列化和反序列化；
2 服务发布订阅；
3 服务治理；

功能

表 2-1　分布式服务框架的功能特性

特 性 名	功 能 名	说　明
服务订阅发布	配置化发布和引用服务	支持通过 XML 配置的方式发布和导入服务，降低对业务代码的侵入
	服务自动发现机制	支持服务实时自动发现，由注册中心推送服务地址，消费者不需要配置服务提供者地址，服务地址透明化
	服务在线注册和去注册	支持运行态注册新服务，也支持运行态取消某个服务的注册
服务路由	默认提供随机路由、轮循、基于权重的路由策略等	默认提供常用的路由策略，避免每个框架使用者都重复开发
	粘滞连接	总是向同一个提供方发起请求，除非此提供方挂掉，再切换到另一台
	路由定制	支持用户自定义路由策略，扩展平台的功能
集群容错	Failover	失败自动切换，当出现失败，重试其他服务器，通常用于读操作；也可用于幂等性写操作
	Failback	失败自动恢复，后台记录失败请求，定时重发，通常用于消息通知操作
	Failfast	快速失败，只发起一次调用，失败立即报错，通常用于非幂等性的写操作
服务调用	同步调用	消费者发起服务调用之后，同步阻塞等待服务端响应
	异步调用	消费者发起服务调用之后，不阻塞立即返回，由服务端返回应答后异步通知消费者
	并行调用	消费者同时对多个服务提供者批量发起服务调用请求，批量发起请求后，集中等待应答
多协议	私有协议	支持二进制等私有协议，支持私有协议定制和扩展
	公有协议	提供 Web Service 等公有协议，用于外部服务对接
序列化方式	二进制类序列化	支持 Thrift、Protocol buffer 等二进制协议，提升序列化性能
	文本类序列化	支持 JSON、XML 等文本类型的序列化方式，提升通用性和可读性
统一配置	本地静态配置	安装部署修改一次，运行态不修改的配置，可以存放到本地配置文件中
	基于配置中心的动态配置	运行态需要调整的参数，统一放到配置中心（服务注册中心），修改之后统一下发，实时生效

性能

表 2-2　分布式服务框架的性能特性

线性特性	说　明
高性能	在同等资源占用情况下，单服务提供者的 TPS 要尽量高
低时延	在同等资源占用情况下，服务调用时延要尽量低
性能线性增长	扩展服务提供者，性能要能够线性增长

可靠性

表 2-3　可靠性功能列表

特　性　名	功　能　名	说　　明
服务注册中心	服务健康状态检测	注册中心通过心跳检测服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者
	故障切换	注册中心对等集群，任意一台宕掉后，将自动切换到另一台
	高 HA	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信
消除单点故障	服务无状态	服务提供者无状态，任意一台宕掉后，不影响使用
	服务集群容错	只要集群中有一台机器的服务提供者可用，业务就不会中断
链路健壮性	心跳检测	链路空闲时没有业务消息，通过定时心跳检测链路是否可用
	断连重连机制	链路断连之后，根据客户端配置的重连策略定时重连，重连成功之前消息不会路由到断连的服务提供者上

服务治理

表 2-4　服务治理功能列表

特　性　名	功　能　名	说　　明
服务运行态管控	服务路由	业务高峰期，通过动态修改路由策略实现导流
	服务限流	资源成为瓶颈时，服务端和消费者的动态流控
	服务迁入迁出	实现资源的动态分配
	服务降级	服务提供者故障时或者业务高峰期，进行服务强制或者容错降级，执行本地降级逻辑，保证系统平稳运行
	服务超时控制	动态调整超时时间，在业务高峰期保障业务调用成功率
服务监控	性能统计	统计项包括服务调用时延、成功率、调用次数等
	统计报表	提供多维度、实时和历史数据报表，同比、环比等性能对比数据，供运维、运营等使用
	告警	指标异常，根据告警策略发送告警，包括但不限于短信、E-mail、记录日志等
服务生命周期管理	上线审批	服务提供者不能随意线上发布服务，需要通过正规的审批流程批准之后才能上线
	下线通知	服务提供者在下线某个服务之前一段时间，需要根据 SLA 策略，通知消费者
	服务灰度发布	灰度环境划分原则、接口前向兼容性策略，以及消费者如何路由，都需要灰度发布引擎负责管理
故障快速定界定位	分布式日志采集	支持在大规模分布式环境中实时采集容器、中间件和应用的各种日志
	海量日志在线检索	支持分布式环境海量日志的在线检索，支持多维度索引和模糊查询
	调用链可视化展示	通过分布式消息跟踪系统输出调用链，可视化、快速地进行故障定界
	运行日志故障定位	通过调用链的故障关键字，在日志检索界面快速检索故障日志，用于故障的精确定位
服务安全	敏感服务的授权策略	敏感服务如何授权，防止恶意调用
	链路的安全防护	消费者和服务提供者之间的长连接，需要增加安全防护，例如基于 Token 的安全认证机制

通讯框架

技术点

1. 长连接：主要是链路的创建过程到最后的关闭，耗时耗资源；每次调用都要创建的话，调用时延的问题，很可能链路创建的耗时比代码真正执行时长还多；
2. BIO还是NIO：主要是线程模型的选择，推荐篇文章 IO - 同步，异步，阻塞，非阻塞 （亡羊补牢篇）；
3. 自研还是使用开源NIO框架：一般来说还是使用开源吧，技术成熟，社区支持，现在netty和mina使用较多了吧。

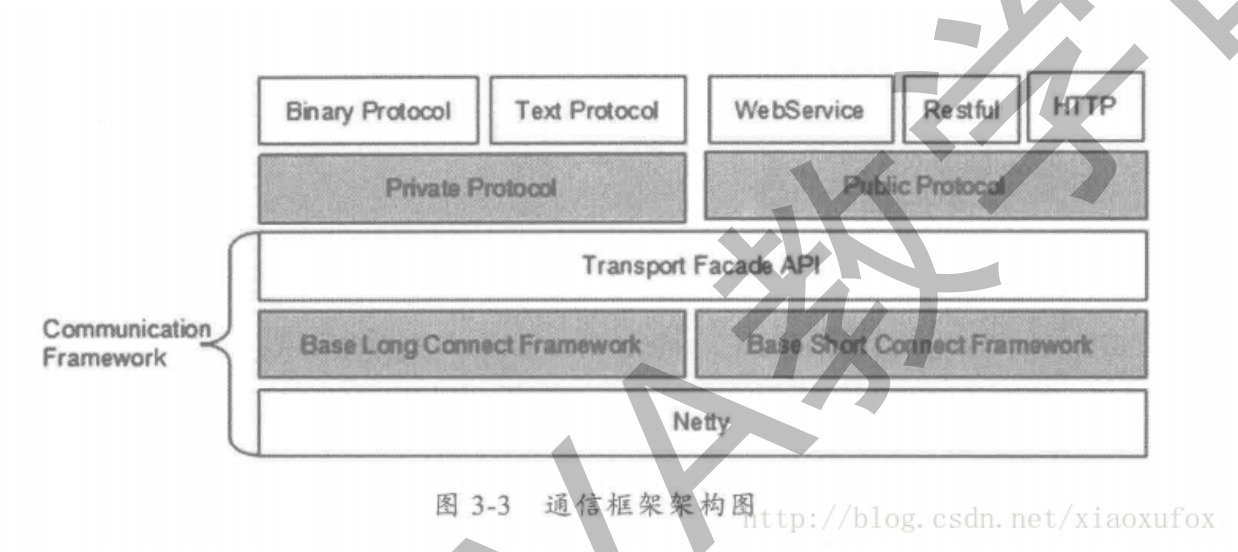


图 3-3　通信框架架构图

> 在功能设计方面，作者基于netty给了demo服务端和客户端的代码，个人理解：

1. 通用性api；
2. 扩展性，封装底层，提供上层接口，隔离协议和底层通讯；

可靠性设计

谈分布式系统必谈可靠性。

链路有效性

> 通过心跳来确认双方c、s存活，保证链路可用，心跳检测机制分为3个层面：

```
1. tcp层面，即tcp的keep-alive，作用于整个tcp协议栈；
2. 协议层的心跳检测，主要存在于长连接协议中，例如smpp协议；
3. 应用层的心跳，业务双方的定时发送心跳消息；
```

第2个没听说过，常用的是1,3。一般使用netty的话用的是netty的读写空闲来实现心跳。

断连

> 不管因为网络挂了还是服务端宕机，还是心跳超时什么的，导致链路不可用关闭，这时候就需要链路重连，需要注意的一点就是短连后，不要立即重连，留时间给系统释放资源，可以scheduler处理。

消息缓存重发

> 底层消息不会立即发送（也会导致半包粘包），断链后，导致消息丢失，看有无业务需求，有就支持断链后消息重发。

资源释放

> 主要是断链后，一定要保证资源销毁和释放，当然也包括一些线程池，内存等的释放。

性能设计

性能差的三宗罪

> 对于底层通讯框架来说，主要是下面几个：

```
1．通讯模型的选择，主要是阻塞非阻塞那些东西；
2．序列化反序列化（后面有章单讲序列化）；
3．线程模型，主要是服务端选择什么样的线程模型来处理消息。
```

通信性能三原则

既然有上面的3个问题，那就针对这些做优化了：

1. 传输：BIO\NIO\AIO的选择；
2. 选择自定义协议栈，便于优化；
3. 服务端线程模型，单线程处理还是线程池，线程池是一个，还是分优先级，Reactor还是其他什么的。
   高性能之道这节作者讲了netty的优势。

序列化与反序列化

> 也就是通常所说的编码、解码。通常的通讯框架会提供编解码的接口，也会内置一些常用的序列化反序列化工具支持。 与通讯框架和协议的关系，感觉可以理解为：通讯框架是通道，其上跑的码流数据是利用各种序列化编码后的各种协议。

功能设计

> 各种序列化框架需要考虑的主要有：

- 序列化框架本身的功能的丰富，支持的数据类型；
- 多语言的支持；

- 兼容性，往大了说：
    1. 服务接口的前后兼容；
    2. 协议的兼容；
    3. 支持的数据类型的兼容。
- 性能，目的是最少的资源，最快的速度，最大的压缩：
    1. 序列化后码流大小；
    2. 序列化的速度；
    3. 序列化的资源占用。

实际开发中，一般不太会使用这些东西，都会提供序列化反序列接口，自行扩展定义，所以扩展性特重要。 常用的序列化，xml，json，hessian，kryo，pb，ps，看需求需要支持那种，具体可以搜索各序列化的性能和压缩后大小。

协议栈

> 这一章最主要的是讲了自定义协议栈的设计，以及交互的过程，其他讲的可靠性设计什么的跟之前通讯框架一章有重复。

通信模型

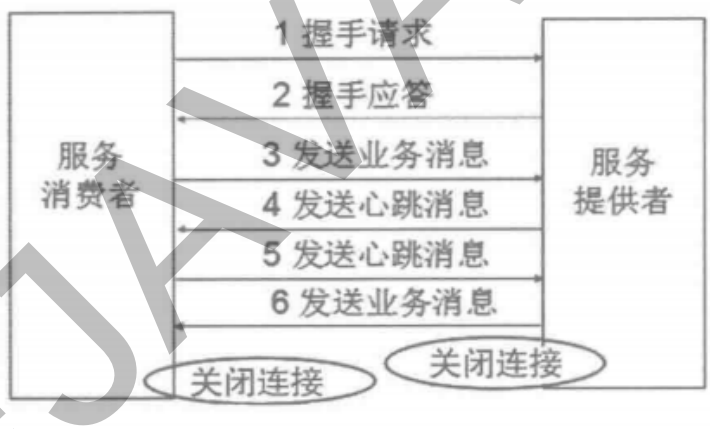


图 5-1　私有协议栈通信模型

> 服务提供者和消费者之间采用单链路，长连接通信，链路创建流程：

1. 客户端发送握手请求，携带节点ID等认证信息；
2. 服务端校验：节点ID有效性，重复登录，ip地址黑白名单等，通过后，返回握手应答信息；
3. 链路建立后，客户端发送业务消息；
4. 客户端服务端心跳维持链路；
5. 服务端退出时，关闭连接，客户端感知连接关闭，关闭客户端连接。

协议消息定义

表 5-1　消息定义模型

名　称	类　型	长　度	描　述
header	Header	变长	消息头定义
body	byte []	变长	字节数组：对于请求消息，它是方法的参数；对于响应消息，它是返回值

消息头通常包含服务调用相关的公共参数，参考定义如表 5-2 所示。

表 5-2　消息头定义

名　称	类　型	长　度	描　述
crcCode	整型 int	32	协议栈校验码，它由三部分组成。 1）0xAFBA：固定值，表明该消息是私有协议消息，2 个字节 2）主版本号：1～255，1 个字节 3）次版本号：1～255，1 个字节 crcCode = 0xAFBA + 主版本号 + 次版本号
length	整型 int	32	消息长度，整个消息，包括消息头和消息体

名　称	类　型	长　度	描　述
type	byte	8	0：业务请求消息 1：业务响应消息 2：业务 ONE WAY 消息（既是请求又是响应消息） 3：握手请求消息 4：握手应答消息 5：心跳请求消息 6：心跳应答消息
priority	byte	8	消息优先级：0～255
interfaceName	string	变长	接口名
methodName	string	变长	方法名
attachment	Map<String, String>	变长	可选字段，用于扩展消息头

通过attachment兼容了扩展性。作者还讲了将消息头的通用序列化和消息体的自定义序列化，看需求吧，我们公司的框架没做这部分支持，做了简化，将消息头和消息体统一封装，然后再加一个序列化方式组成一条消息发送。

安全性设计

1. 内部的，不一定需要认证，也有基于系统，域名，ip的黑白名单，安全认证的；
2. 外部开发平台的话，基于秘钥认证；

服务路由

> 服务路由指的是服务提供者集群部署，消费端如何从服务列表中选择合适的服务提供者提供服务进行调用。

透明化路由

1. 基于zk的服务注册中心的发布订阅；
    2. 消费者本地缓存服务提供者列表，注册中心宕机后，不影响已有的使用，只是影响新服务的注册和老服务的下线。

负载均衡

- 随机
- 轮循
- 服务调用时延
- 一致性Hash
    1. 有个一致性hash算法，挺有意思的，redis的客户端shard用的

图 6-4　一致性Hash环原理

- 黏滞连接
    1. 这个应该不太常用，服务提供者多数无状态，一旦有状态，不利于扩展
    这些都是点对点的连接，负载均衡大多会在客户端执行，有种场景会取决于服务端负载，就是服务端服务配置的是域名。

本地路由优先策略

- injvm：jvm也提供了消费端的服务，可以改成优先本jvm，对于消费端来说，不需关注提供者；
- innative：injvm比较少，多得是可能是这种，一个物理机部署多个虚拟机，或者一个容器部署多个服务提供者，消费者不需远程调用，本机，本地或本机房优先。

路由规则

除了上面提供的各种路由负载均衡，还容许自定义路由规则:

- 条件路由：主要是通过条件表达式来实现；
- 脚本路由：通过脚本解析实现。

其实应该还有一种客户端通过代码自定义路由选择。这些主要是为了扩展性。

路由策略定制

自定义路由场景：

1. 灰度；
2. 引流；

路由策略：

1. 框架提供接口扩展；
2. 配置平台提供路由脚本配置；

配置化路由

1. 本地配置：包括服务提供者和消费者，全局配置3种；
2. 注册中心：路由策略统一注册到服务注册中心，集中化管理；
3. 动态下发：配置后动态下发各服务消费端。

集群容错

指的是服务调用失败后，根据容错策略进行自动容错处理。

集群容错场景

- 通信链路故障：
    1. 通信过程中，对方宕机导致链路中断；
    2. 解码失败等原因Rest掉链接；
    3. 消费者read-write socketchannel发生IOException导致链路中断；
    4. 网络闪断故障；
    5. 交换机异常导致链路中断；
    6. 长时间Full GC导致；

- 服务端超时：
    1. 服务端没有及时从网络读取客户端请求消息，导致消息阻塞；
    2. 服务端业务处理超时；
    3. 服务端长时间Full GC；
- 服务端调用失败：
    1. 服务端解码失败；
    2. 服务端流控；
    3. 服务端队列积压；
    4. 访问权限校验失败；
    5. 违反SLA策略；
    6. 其他系统异常；
       业务执行异常不属于服务端异常。

容错策略

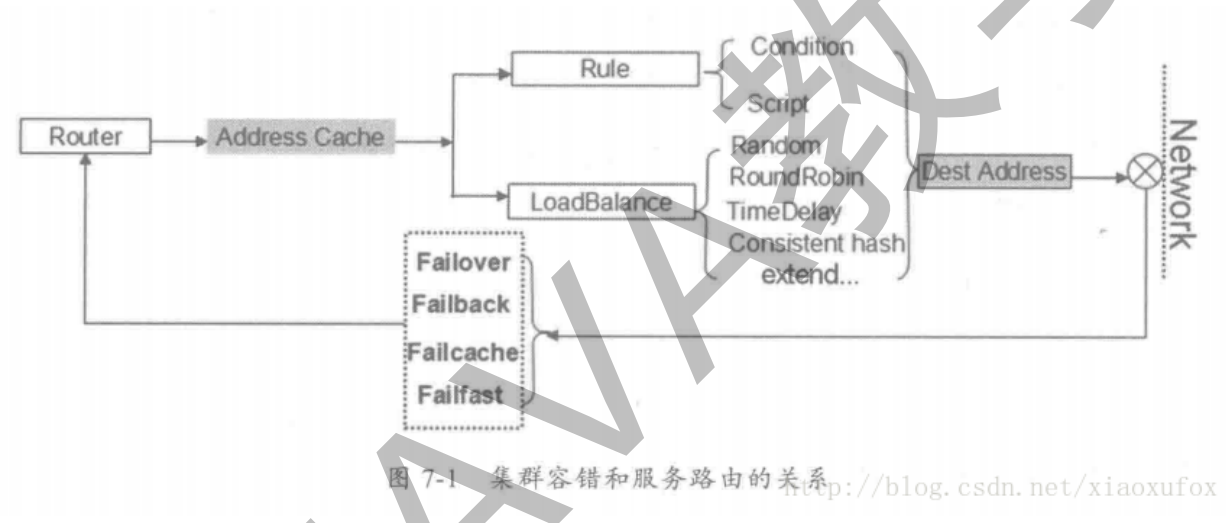


图 7-1 集群容错和服务路由的关系

- 失败自动切换（Failover）：
    1. 调用失败后切换链路调用；
    2. 服务提供者的防重；
    3. 重试次数和超时时间的设置。
- 失败通知（FailBack）：失败后直接返回，由消费端自行处理；
- 失败缓存（Failcache）：主要是失败后，缓存重试重发，注意：
    1. 缓存时间、缓存数量；
    2. 缓存淘汰算法；
    3. 定时重试的周期T、重试次数；
- 快速失败（Failfast）：失败不处理，记录日志分析，可用于大促期间，对非核心业务的容错。

容错策略扩展

1. 容错接口的开放；
2. 屏蔽底层细节，用户自定义；
3. 支持扩展。
   其实还有一点，感觉也挺重要，就是支持容错后本地mcok。调用失败后的链路切换和快速失败

肯定要支持，缓存重发可以不用