



软件架构探索

The Fenix Project Alpha

Watch 13 Star 127 Follow 144

Release v1.0.20201004 License Apache 2.0 Doc License CC 4.0 Words 310,860 Author IcyFenix

周志明

icyfenix@gmail.com

发行日期：2020-10-04

这是什么？

这是一部以“架构师应该掌握哪些架构知识”为叙事主线的开源文档，是一幅帮助开发人员整理现代软件架构各条分支中繁多知识点的技能地图。文章《什么是“The Fenix Project”》详细阐述了这部文档的主旨、目标与名字的来由，文章《如何开始》简述了文档每章讨论的主要话题与内容详略分布，供阅前参考。

笔者出于以下目的，撰写这部文档：

- 笔者从事大型企业级软件的架构研发工作，借此机会，系统性地整理自己的知识，查缺补漏，将它们都融入既有的知识框架之中。
- 笔者正式出版过七本计算机技术书籍，遗憾的是暂时还没有一本与自己本职工作直接相关。能按照自己的兴趣去写作，还能获得不菲的经济报酬是一件很快乐的事情；撰写一部工作中能直接使用的、能随时更新、与人交流的在线文档，同样也是一件很实用、很有价值的事情。
- 笔者认为技术人员成长有一“捷径”，学技术不仅要去看、去读、去想、去用，更要去说、去写。将自己“认为掌握了的”知识叙述出来，能够说得有条理清晰，讲得理直气壮；能够让他人听得明白，释去心中疑惑；能够把自己的观点交予别人的审视，乃至质疑，在此过程之中，会挖掘出很多潜藏在“已知”背后的“未知”。未有知而不行者，知而不行，只是未知。

除文档部分外，笔者同时还建立了若干配套的代码工程，这是针对不同架构、技术方案（如单体架构、微服务、服务网格、无服务架构，等等）的演示程序。它们既是文档中所述知识的实践示例，亦可作为实际项目新创建时的可参考引用的基础代码。

如何使用？

根据“使用”的所指含义的不同，笔者列举以下几种情况：

- 在线阅读：本文档在线阅读地址为：<https://icyfenix.cn>。网站由 GitHub Pages 提供网站存储空间；由 Travis-CI 提供的持续集成服务实时把

Git 仓库的 Markdown 文档编译同步至网站；由[腾讯云 CDN](#) 提供国内访问的缓存支持。

- **离线阅读：**

- 部署离线站点：文档基于[Vuepress](#) 构建，如你希望在企业内部搭建文档站点，请使用如下命令：

```
# 克隆获取源码  
$ git clone https://github.com/fenixsoft/awesome-fenix.git && cd  
awesome-fenix  
  
# 安装工程依赖  
$ npm install  
  
# 运行网站，地址默认为http://localhost:8080  
$ npm run dev
```

sh

- 生成PDF文件：工程源码中已带有基于[vuepress-plugin-export](#) 改造（针对本文档定制过）的PDF导出插件，如你希望生成全文 PDF 文件，请在已进行上一步工程克隆和依赖安装的前提下使用如下命令：

```
# 编译PDF，结果将输出在网站根目录  
$ npm run export
```

sh

PDF 全文编译时间较长，在笔者机器上约耗时25分钟，在 Travis-CI 上约需要约8分钟。PDF 中文字体采用阿里巴巴普惠字体渲染，此字体被允许免费使用与传播。

- **二次演绎、传播和发行：**本文档中所有的内容，如引用其他资料，均在文档中明确列出资料来源，一切权利归属原作者。除此以外的所有内容，包括但不限于文字、图片、表格，等等，均属笔者原创，这些原创内容，笔者声明以[知识共享署名 4.0](#) 发行，只要遵循许可协议条款中**署名、非商业性使用、相同方式共享**的条件，你可以在任何地方、以任何形式、向任何人使用、修改、演绎、传播本文档中任何部分的内容。详细可见本文档的“协议”一节。
- **运行技术演示工程：**笔者专门在探索起步中的“[技术演示工程](#)”详细介绍了配套工程的使用方法，如果你对构建运行环境也有所疑问，在附录中的“[环境依赖](#)”部分也已包括了详

细的环境搭建步骤。此外，这些配套工程也均有使用 Travis-CI 提供的持续集成服务，自动输出到 Docker 镜像库，如果你只关心运行效果，或只想了解部分运维方面的知识，可以直接运行 Docker 镜像而无需关心代码部分。你可以通过下面所列的地址，查看到本文档所有工程代码和在线演示的地址：

- 文档工程：

- 软件架构探索：<https://icyfenix.cn>
- Vuepress 支持的文档工程：<https://github.com/fenixsoft/awesome-fenix>

- 前端工程：

- Mock.js 支持的纯前端演示：<https://bookstore.icyfenix.cn>
- Vue.js 2 实现前端工程：<https://github.com/fenixsoft/fenix-bookstore-frontend>

- 后端工程：

- Spring Boot 实现单体架构：https://github.com/fenixsoft/monolithic_arch_springboot
- Spring Cloud 实现微服务架构：https://github.com/fenixsoft/microservice_arch_springcloud
- Kubernetes 为基础设施的微服务架构：https://github.com/fenixsoft/microservice_arch_kubernetes
- Istio 为基础设施的服务网格架构：https://github.com/fenixsoft/servicemesh_arch_istio
- AWS Lambda 为基础的无服务架构：https://github.com/fenixsoft/serverless_arch_awslambda

协议

- 本作品代码部分采用 [Apache 2.0 协议](#) 进行许可。遵循许可的前提下，你可以自由地对代码进行修改，再发布，可以将代码用作商业用途。但要求你：
 - 署名：在原有代码和衍生代码中，保留原作者署名及代码来源信息。
 - 保留许可证：在原有代码和衍生代码中，保留 Apache 2.0 协议文件。
- 本作品文档部分采用 [知识共享署名 4.0 国际许可协议](#) 进行许可。遵循许可的前提下，你可以自由地共享，包括在任何媒介上以任何形式复制、发行本作品，亦可以自由地演绎、修改、转换或以本作品为基础进行二次创作。但要求你：

- **署名**：应在使用本文档的全部或部分内容时候，注明原作者及来源信息。
- **非商业性使用**：不得用于商业出版或其他任何带有商业性质的行为。如需商业使用，请联系作者。
- **相同方式共享的条件**：在本文档基础上演绎、修改的作品，应当继续以知识共享署名4.0国际许可协议进行许可。

目录

数据统计

列入目录文章 100 篇，目前已完成 87 篇，合计总字数 310,860 字，最后更新日期 2020-10-04。

1. 目录	57 字
2. 前言	
2.1. 关于作者	651 字
2.2. 什么是“The Fenix Project”	3,257 字
3. 探索起步	
3.1. 阅读指引	
3.1.1. 更新日志	1,046 字
3.1.2. 如何开始	3,081 字
3.2. 技术演示工程	324 字
3.2.1. 前端工程	2,252 字
3.2.2. 单体架构：Spring Boot	2,546 字
3.2.3. 微服务：Spring Cloud	3,475 字
3.2.4. 微服务：Kubernetes	3,541 字
3.2.5. 服务网格：Istio	3,848 字
3.2.6. 无服务：AWS Lambda	1,372 字
4. 演进中的架构	
4.1. 服务架构演进史	579 字
4.1.1. 原始分布式时代	3,101 字

4.1.2. 单体系统时代	3,136 字
4.1.3. SOA时代	3,136 字
4.1.4. 微服务时代	4,426 字
4.1.5. 后微服务时代	3,227 字
4.1.6. 无服务时代	2,723 字
5. 架构师的视角	
5.1. 远程访问	649 字
5.1.1. 远程服务调用	8,281 字
5.1.2. REST服务设计风格	11,891 字
5.2. 事务处理	1,032 字
5.2.1. 本地事务	6,725 字
5.2.2. 全局事务	3,523 字
5.2.3. 共享事务	1,328 字
5.2.4. 分布式事务	8,579 字
5.3. 透明多级分流系统	1,741 字
5.3.1. 客户端缓存	3,522 字
5.3.2. 域名解析	1,893 字
5.3.3. 传输链路	4,751 字
5.3.4. 内容分发网络	3,652 字
5.3.5. 负载均衡	7,387 字
5.3.6. 缓存	10,985 字
5.4. 安全架构	633 字
5.4.1. 认证	6,741 字
5.4.2. 授权	7,811 字
5.4.3. 凭证	5,370 字
5.4.4. 保密	3,775 字
5.4.5. 传输	7,689 字
5.4.6. 验证	2,678 字

6. 分布式的基石

6.1. 分布式共识算法	1,997 字
6.1.1. Paxos	4,401 字
6.1.2. Multi Paxos与Raft	3,216 字
6.1.3. Gossip协议	1,840 字
6.2. 从类库到服务	276 字
6.2.1. 服务发现	4,800 字
6.2.2. 网关路由	4,699 字
6.2.3. 客户端负载均衡	4,778 字
6.3. 服务与流量治理	618 字
6.3.1. 服务容错	8,671 字
6.3.2. 流量控制	6,759 字
6.4. 可靠通讯	236 字
6.4.1. 零信任网络	3,600 字
6.4.2. 服务安全	5,343 字
6.5. 可观测性	2,165 字
6.5.1. 事件日志	4,970 字
6.5.2. 链路追踪	3,969 字
6.5.3. 聚合度量	5,574 字

7. 不可变基础设施

7.1. 从微服务到云原生	575 字
7.2. 虚拟化容器	1,194 字
7.2.1. 容器的崛起	7,857 字
7.2.2. 以容器构建系统	7,285 字
7.2.3. 应用为中心的封装	8,190 字
7.3. 容器间网络	459 字
7.3.1. Linux网络虚拟化	13,294 字

7.3.2. 容器网络与生态	5,808 字
7.4. 容器持久化存储	447 字
7.4.1. Kubernetes存储设计	6,288 字
7.4.2. 容器存储与生态	8 字
7.5. 调度硬件资源	7 字
7.6. 服务网格	5 字
7.6.1. 网格化服务	6 字
7.6.2. xDS协议	4 字
7.6.3. Envoy代理	4 字
8. 技术方法论	
8.1. 向微服务迈进	604 字
8.1.1. 目的：微服务的驱动力	2,366 字
8.1.2. 前提：微服务需要的条件	3,752 字
8.1.3. 边界：微服务的粒度	2,723 字
8.1.4. 治理：理解系统复杂性	4,403 字
8.2. 架构设计模式	
8.2.1. 事件驱动架构	591 字
8.2.1.1. 事件溯源	5 字
8.2.1.2. 查询职责分离	7 字
8.2.1.3. 复杂事件处理	7 字
8.2.2. 扩展性立方体	10 字
8.2.3. 编排与协同	6 字
9. 专题随笔	
9.1. Graal VM	1,358 字
9.1.1. 新一代即时编译器	851 字
9.1.2. 向原生迈进	1,729 字
9.1.3. 没有虚拟机的Java	1,846 字

9.1.4. Spring over Graal	2,678 字
--------------------------	---------

10. 附录

10.1. 部署环境	143 字
10.1.1. 部署Docker CE容器环境	2,123 字
10.1.2. 部署Kubernetes集群	447 字
10.1.2.1. 使用Kubeadm部署	4,243 字
10.1.2.2. 使用Rancher部署	1,647 字
10.1.2.3. 使用Minikube部署	634 字
10.2. 部署Istio	4 字
10.3. 部署Elastic Stack	5 字
10.4. 部署Prometheus	4 字

关于作者

周志明 [github](#)、[weibo](#)、[email](#)

- 80后程序员

职业是上市公司的高级管理人员；兴趣永远是当一名纯粹的程序员。

工作中主要从事大型企业级软件的研发；业余里对计算机科学相关的多个领域都有持续跟进。

- 远光研究院 院长

博士，[远光软件](#)研究院院长、澳门科大-远光人工智能联合实验室主任。研究方向为机器学习特征选择自动化。

- 计算机技术作家

正式出版过七部计算机技术书籍，撰写过两部开源文档，口碑和销量均得到读者的认可。其中四本书在[豆瓣](#)上获得了9.0分或以上的评价，“深入理解Java虚拟机”系列重印超过40次，总销量逾30万册。

- 2020年 《软件架构探索：The Fenix Project》（开源文档，进行中）
- 2019年 《深入理解Java虚拟机：JVM高级特性与最佳实践（第三版）》（豆瓣 9.6）
- 2018年 《智慧的疆界：从图灵机到人工智能》（豆瓣 9.0）
- 2016年 《深入理解Java虚拟机：JVM高级特性与最佳实践（第二版）》（豆瓣 9.0）
- 2015年 《Java虚拟机规范（Java SE 8中文版）》（官方授权翻译，豆瓣 8.0）
- 2014年 《Java虚拟机规范（Java SE 7中文版）》（官方授权翻译，豆瓣 9.0）
- 2013年 《深入理解OSGi：Equinox原理、应用与最佳实践》（豆瓣 7.7）
- 2011年 《深入理解Java虚拟机：JVM高级特性与最佳实践（第一版）》（豆瓣 8.6）
- 2011年 《Java虚拟机规范（Java SE 7中文版）》（开源文档）

- 技术布道师

开源技术的积极倡导者和推动者，国内主流云计算厂商技术专家，新媒体撰稿人。

- [腾讯云最有价值技术专家（TVP）](#)
- [阿里云最有价值技术专家（MVP）](#)
- [华为云最有价值技术专家（MVP）](#)
- [InfoQ.CN专栏撰稿人](#)、[极客时间布道师](#)

什么是“The Fenix Project”

“Phoenix”这个词东方人不常用，但在西方的软件工程读物——尤其是关于Agile、DevOps话题的作品中时常出现。软件工程小说《The Phoenix Project》讲述了徘徊在死亡边缘的Phoenix项目在精益方法下浴火重生的故事；马丁·福勒（Martin Fowler）对《Continuous Delivery》的诠释里，曾多次提到“Phoenix Server”（取其能够“涅槃重生”之意）与“Snowflake Server”（取其“世界上没有相同的两片雪花”之意）的优劣比对。也许是东西方的文化的差异，尽管有“失败是成功之母”这样的谚语，但我们东方人的骨子里更注重的还是一次把事做对做好，尽量别出乱子；而西方人则要“更看得开”一些，把出错看做正常甚至是必须的发展过程，只要出了问题能够兜底使其重回正轨便好。



The Phoenix Project

在软件工程里，任何产品的研发，只要时间尺度足够长，人就总会疏忽犯错，代码就总会携有缺陷，电脑就总会宕机崩溃，网络就总会堵塞中断……如果一项工程需要大量的人员，共同去研发某个大规模的软件产品，并使其分布在网络中大量的服务器节点中同时运行，随着项目规模的增大、运作时间变长，其必然会受到墨菲定律的无情打击。

Murphy's Law :

Anything that can go wrong will go wrong —— 如果事情可能出错就总会出错

为了得到高质量的软件产品，我们是应该把精力更多地集中在提升其中每一个人员、过程、产出物的能力和质量上，还是该把更多精力放在整体流程和架构上？

笔者先给这个问题一个“合稀泥”式的回答：这两者都重要。前者重术，后者重道；前者更多与编码能力相关，后者更多与软件架构相关；前者主要由开发者个体水平决定，后者主要由技术决策者水平决定；

然而，笔者也必须强调此问题的另外一面：这两者的理解路径和抽象程度是不一样的。如何学习一项具体的语言、框架，工具，譬如Java、Spring、Vue.js……都是相对具象的，不论其蕴含的内容多少，复杂程度高低，它是至少能看得见摸得着。而如何学习某一种风格的架构方法，譬如单体、微服务、服务网格、无服务、云原生……则是相对抽象的，谈论它们可能要面临则“一百个人眼中有一百个哈姆雷特”的困境。谈这方面的话题，若要言之有物，就不能是单纯的经验陈述。笔者想来，回到这些架构根本的出发点和问题上，真正去使用这些不同风格的架构方法来实现某些需求，解决某些问题，然后在实践中观察它们的异同优劣，会是一种很好的，也许是最好的讲述方式。笔者想说一下这些架构，而且还想说得透彻明白，这需要代码与文字的配合，于是便有了这个项目。

可靠的系统

让我们再来思考一个问题，构建一个大规模但依然可靠的软件系统，是否是可行的？

这个问题令人听起来的第一感觉也许会有点荒谬：废话。如果这个事情从理论上来说就是根本不可能的话，那我们这些软件开发从业人员现在还在瞎忙活些什么？但你再仔细想想，前面才提到的“墨菲定律”和在“大规模”这个前提下必然会遇到的各种不靠谱的人员、代码、硬件、网络等因素，从中能得出的一个听起来颇为符合逻辑直觉的推论：如果一项工作要经过多个“不靠谱”的过程相互协作来完成，其中的误差应会不断地累积叠加，导致最终结果必然不能收敛稳定才对。

这个问题也并非杞人忧天庸人自扰式的瞎操心，计算机之父冯·诺依曼（John von Neumann）在1940年代末期，曾经花费了大约两年时间，研究这个问题并且得出了一门理论《[自复制自动机](#)》（Theory of Self-Reproducing Automata），这个理论以机器应该如何从基本的部件中构造出与自身相同的另一台机器引出，其目的并不是想单纯地模拟或者理解生物体的自我复制，也并不是简单想制造自我复制的计算机，他的最终目的就是想回答一个理论问题：如何用一些不可靠的部件来构造出一个可靠的系统。



当时自复制机的艺术表示（[图片来自维基百科](#)）

自复制机恰好就是一个最好的用不可靠部件构造的可靠的系统例子。这里，“不可靠部件”可以理解为构成生命的大量细胞、甚至是分子。由于热力学扰动、生物复制差错等因素干扰，这些分子本身并不可靠。但是生命系统之所以可靠的本质，恰是因为它可以使用不可靠的部件来完成遗传迭代。这其中的关键点便是承认细胞等这些零部件可能会出错，某个具体的零部件可能会崩溃消亡，但在存续生命的微生态系统中一定会有其后代的出现，重新代替该零部件的作用，以维持系统的整体稳定。在这个微生态里，每一个部件都可以看作一只不死鸟（Phoenix），它会老迈，而之后又能涅槃重生。

架构的演进

软件架构风格从大型机（Mainframe），到[原始分布式](#)（Distributed），到[大型单体](#)（Monolithic），到[面向服务](#)（Service-Oriented），到[微服务](#)（Microservices），到[服务网格](#)（Service Mesh），到[无服务](#)（Serverless）……技术架构上确实呈现出“从大到小”的发展趋势。当近年来微服务兴起以后，涌现出各类文章去总结、赞美微服务带来的种种好处，诸如简化部署、逻辑拆分更清晰、便于技术异构、易于伸缩拓展应对更高的性能等等，这些当然都是重要优点和动力。可是，如果不拘泥于特定系统或特定某个问题，以更宏观的角度来看，前面所列这种种好处却都只能算是“锦上添花”、是属于让系统“活得更好”的动因，肯定比不上系统如何“确保生存”的需求来得关键、本质。笔者看来，架构演变最重要的驱动力，或者说这种“从大到小”趋势的最根本的驱动力，始终都是为了方便某个服务能够顺利地“死去”与“重生”而设计的，个体服务的生死更迭，是关系到整个系统能否可靠续存的关键因素。

举个例子，譬如某企业中应用的单体架构的Java系统，其更新、升级都必须要有固定的停机计划，必须在特定的时间窗口内才能按时开始，必须按时结束。如果出现了非计划的宕

机，那便是生产事故。但是软件的缺陷不会遵循领导定下的停机计划来“安排时间出错”，为了应对缺陷与变化，做到不停机地检修，Java曾经搞出了OSGi和JVMTI Instrumentation等这样复杂的HotSwap方案，以实现给奔跑中的汽车更换轮胎这种匪夷所思却又无可奈何的需求；而在微服务架构的视角下，所谓系统检修，不过只是一次在线服务更新而已，先停掉1/3的机器，升级新的软件版本，再有条不紊地导流、测试、做金丝雀发布，一切都是显得如此理所当然、平淡寻常；而在无服务架构的视角下，我们甚至都不可能去关心服务所运行的基础设施，连机器是哪台都不必知道，停机升级什么的就根本无从谈起了。

流水不腐，有老朽，有消亡，有重生，有更迭才是生态运行的合理规律。请设想一下，如果你的系统中每个部件都符合“Phoenix”的特性，哪怕其中某些部件采用了由极不靠谱的人员所开发的极不靠谱程序代码，哪怕存有严重的内存泄漏问题，最多只能服务三分钟就一定会崩溃。而即便这样，只要在整体架构设计有恰当的、自动化的错误熔断、服务淘汰和重建的机制，在系统外部来观察，整体上仍然有可能表现出稳定和健壮的服务能力。

The Fenix Project

在企业软件开发的历史中，一项新技术发布时，常有伴以该技术开发的“宠物店（PetStore）”作为演示的传统（如J2EE PetStore、.NET PetShop、Spring PetClinic等）。作为不同架构风格的演示时，笔者本也希望能遵循此传统，却无奈从来没养过宠物，遂改行开了书店（Fenix's Bookstore），里面出售了几本笔者撰写过的书籍，算是夹带一点私货，同时也避免了使用素材时可能的版权问题。

尽管相信没有人会误解，但笔者最后还是多强调一句，Oracle、Microsoft、Pivotal等公司设计宠物店的目的绝不是为了日后能在网上贩卖小猫小狗，只是纯粹的演示技术。所以也请勿以“实现这种学生毕业设计复杂度的需求，引入如此规模的架构或框架，纯属大炮打苍蝇，肯定是过度设计”的眼光来看待接下来的“Fenix's Bookstore”项目。相反，如果可能的话，笔者会在有新的技术、框架发布出来时，持续更新，以恰当的形式添加到项目的不同版本中，可能使其技术栈越来越复杂。笔者希望把这些新的、不断发展的知识，融合进已有的知识框架之中，让自己学习、理解、思考，然后将这些技术连同自己的观点看法，传播给感兴趣的人。

也算是缘分，网名“IcyFenix”在二十多年前我的中学时代开始使用，最初它是来源于暴雪公司的即时战略游戏《星际争霸》的Protoss英雄Fenix——如名字所预示的那样，他曾经是Zealot，牺牲后以Dragoon的形式重生，带领Protoss与刀锋女王Kerrigan继续抗争。尽

管中学时期我已经笃定自己未来肯定会从事信息技术相关的工作，但显然不可能预计到二十年后我会写下这些文字。

所以，既然我们要开始一段关于“Phoenix”的代码与故事，那便叫它“The Fenix Project”，如何？

更新日志

2020年10月4日

- 完成了“[Kubernetes存储设计](#)”章节

2020年9月28日

- 完成了“[容器网络与生态](#)”章节
- 随着这篇文章的更新，整部文档超过了30万字，按计划应该在35万字以内结束。

2020年9月23日

- 完成了“[Linux网络虚拟化](#)”章节

2020年9月14日

- 完成了“[应用为中心的封装](#)”章节

2020年9月10日

- 完成了“[以容器构建系统](#)”章节

2020年9月8日

- 完成了“[虚拟化容器](#)”与“[容器的崛起](#)”章节

2020年9月4日

- 完成了“[聚合度量](#)”章节

- 目前到了26万字，整部文档所规划的框架里，只剩下介绍云原生的“不可变基础设施”这一最后一大块了，希望今年内能全部写完。

2020年9月1日

- 完成了“事件日志”章节
- 完成了“链路追踪”章节

2020年8月29日

- 完成了“可观测性”章节

2020年8月27日

写了几篇谈理论的务虚文章：

- 向微服务迈进
 - 目的：微服务的驱动力
 - 前提：微服务需要的条件
 - 边界：微服务的粒度
 - 治理：理解系统复杂性

2020年8月14日

- 重写了事务处理中的“本地事务”一节。

2020年8月11日

- 重写了安全架构中的“认证”一节。

2020年8月7日

- 提供了新的架构演示“AWS Lambda 为基础的无服务架构”。

2020年8月5日

- 完成了“[缓存](#)”章节。

2020年7月23日

- 完成了“[服务安全](#)”章节。

2020年7月20日

- 更新了[基于Spring Cloud](#)和[基于Istio](#)的Fenix's Bookstore的代码，提供了RSA SHA256的JWT令牌实现，以配合后面两节的主题。
- 完成了“[零信任网络](#)”章节。
- 这部文档的总字数在今天突破了20万字，留个纪念。

2020年7月13日

- 完成“[流量控制](#)”章节。

2020年7月8日

- 完成“[服务容错](#)”章节。

2020年7月2日

- 完成“[客户端负载均衡](#)”章节。

2020年6月29日

- 完成“[网关路由](#)”章节。

2020年6月18日

- 重写了“[远程服务调用](#)”章节。

2020年6月18日

- [GraalVM](#)增加了视频PPT讲解：[GraalVM——云原生时代的Java](#)。
正在与某知识服务商合作，未来本文档的主要内容会提供成音频稿。并且仍然会以公开课的性质免费提供。
- 更新了[服务架构演进史](#)，大概增加了40%的内容。

2020年6月13日

- 提供了新的架构演示“[基于Istio实现的后端工程](#)”。

2020年5月25日

- 提供了新的架构演示“[基于Kubernetes实现的后端工程](#)”。

2020年5月15日

- 完成“[服务发现](#)”章节。

2020年5月9日

- 完成“[分布式共识算法](#)”章节。

2020年5月5日

- 创建更新日志页面。
- 在[目录](#)中增加根据Git提交时间生成的内容更新标识。

2020年5月2日

- 完成“[服务架构演进史](#)”章节。
- 查了Git文档是在2019年12月23日创建的，今天在[微博](#)上开始小范围公开。

如何开始

《软件架构探索》这个开源文档项目的是笔者对自己在软件架构方面知识的总结，它是完全免费开放的，但免费的、开源的文档并不意味着你使用它时就没有成本，也不见得这个文档中所有的内容对每一个开发人员来说都是必要的。鲁迅说浪费别人的时间等于谋财害命，为了避免浪费阅读者的时间和精力，笔者除了自身力求在知识点准确性和叙述流畅性方面保证质量之外，同时也在本文中简要介绍每一章的主题和所面向的读者类型，本文档各章节之间并没有明显的前后依赖关系，阅读时有针对性的查阅是完全可行的，无需一篇不漏地顺序阅读，在此目录中列出了各文章的明细及字数，希望有助于你制定阅读计划。

第一章 探索起步

字数: 20,439 字

本章面向于准备对文档介绍的内容亲身实践的探索者。

本章没有知识性的内容，是整部文档的引导，也可以视为这个文档附带示例工程的说明书，以及相应运行环境的部署手册。之所以将它安排在目录的第一位，是因为笔者相信如果你是一名驾驶初学者，最合理的学习路径应该是先把汽车发动，然后慢慢行驶起来，而不是马上从“引擎动力原理”、“变速箱构造”入手去设法深刻地了解一台汽车。相信计算机技术也是同理，先从运行程序，看看效果，搭建好开发、调试环境，对即将进行的工作有一个整体的认知开始是很有好处的。

工程提示

本文档所涉及到的工程均在GitHub上存有独立的项目，以方便构建、阅读、运行和fork。

本章中的部分内容，是由这些工程的README.md文件人工同步而来，并没有通过持续集成工具自动处理，所以可能有偶尔更新不一致的情况，如可能，建议到这些项目的GitHub页面上查看最新情况，在右上角有相应的超链接。如这些工程对你有用，望请不吝给个

☆ Star 127。

本章所提供的工程是作为后面所述知识的演示样例，由于数量确实不少，并无必要一次性地把上面所有的工程都运行起来。因为它们是采用不同的技术来解决同一个问题，所以每个工程执行后，最终看到的界面效果均是一样的，只是实现的架构不同。而通过不同的架构、技术去解决同一个问题，这也正是这批工程的最大价值所在。

如果你本身对某些架构风格已经熟练掌握，那笔者的建议是不妨选择一种你目前关注的架构风格去运行起来，然后与你熟悉的技术方案进行比对（第一章 探索起步）、了解它解决的问题与背景（第二章 演进中的架构）、思考这种架构涉及到哪些标准方案（第三章 架构师的视角）、理清分布式系统中新的挑战与应对（第四章 分布式的基石），以及如何将这些纯粹的技术问题隐藏起来，使其不会干扰业务代码（第五章 不可变基础设施）。

第二章 演进中的架构

字数: 20,405 字

本章适合所有开发者，但尤其推荐刚刚从单体架构向微服务架构转型的开发者去阅读。

架构并不是“发明”出来的，是持续进化的结果。“服务架构演进史”这部分，笔者假借讨论历史之名，来梳理微服务发展里程中出现的大量名词、概念，借着微服务的演变过程，我们将从这些概念起源的最初，去分析它们是什么、它们取代了什么、以及它们为什么能够在斗争中取得成功，为什么变得不可或缺的支撑，又或者它们为什么会失败，在竞争中被淘汰，或逐渐湮灭于历史的烟尘当中。

第三章 架构师的视角

字数: 111,307 字

本章讨论与风格无关的架构知识，适合所有技术架构师、系统设计、开发人员。

“架构师”这个词的外延非常宽泛，不同语境中有不同所指，这部文档中的技术架构师特指的是企业架构[\[1\]](#)中面向技术模型的系统设计者，这意味着讨论范围不会涉及到贴近于企业战略、业务流程的系统分析、信息战略设计等内容，而是聚焦于贴近一线研发人员的技术方案设计者。本章将介绍作为一个架构师，你应该在做架构设计时思考哪些问题，有哪些主流的解决方案和行业标准做法，各种方案有什么优点、缺点，不同的解决方法会带来什么不同的影响，等等。以达到将“架构设计”这种听起来抽象的工作具体化、具象化的目的。

本章介绍的内容与具体哪一种架构风格无关，作为后续实践的基础，讨论的是普适的架构技术与技巧，无论你是否关注微服务、云原生这些概念，无论你是从事架构设计还是从事编码开发，了解这里所列的基础知识，对每一个技术人员都是有价值的。

第四章 分布式的基石

字数: 67,939 字

本章面向于使用分布式架构的开发人员。

只要选择了分布式架构，无论是SOA、微服务、服务网格或者其他架构风格，涉及与远程服务交互时，服务的注册发现、跟踪治理、负载均衡、故障隔离、认证授权、伸缩扩展、传输通讯、事务处理，等等，这一系列问题都是无可避免的。不同的架构风格，其区别是到底要在技术规范上提供统一的解决方案，还是由应用系统自行去解决，又或者在基础设施层面将一类问题隔离掉，本章将会讨论这类问题的解决思路、方法和常见工具。

第五章 不可变基础设施

字数: 51,495 字

本章面向于基础设施运维人员、技术平台的开发者。

“不可变基础设施”这个概念由来已久。2012年Martin Fowler设想的“[凤凰服务器](#)”与2013年Chad Fowler正式提出的“[不可变基础设施](#)”，都阐明了基础设施不变性所能带来的益处。在[CNCF](#)定义的“云原生”概念中，“不可变基础设施”提升到了与微服务平级的重要程度，此时它的内涵已不再局限于方便运维、程序升级和部署的手段，而是升华为向应用代码隐藏分布式架构复杂度、让分布式架构得以成为一种可普遍推广的普适架构风格的必要前提。在[云原生时代](#)、[后微服务时代](#)中，软件与硬件之间的界线已经彻底模糊，无论是基础设施的运维人员，抑或是技术平台的开发人员，都有必要深入理解基础设施不变性的目的、原理与实现途径。

第六章 技术方法论

字数: 14,485 字

本章面向于在企业中能对重要技术决策进行拍板的决策者。

这部文档的主体内容是务实的，多谈具体技术，少谈方向理论。只在本章中会集中讨论几点与分布式、微服务、架构等相关的相对务虚的话题。

笔者认为对于一个技术人员，成长主要的驱动力是实践，在开发程序、解决问题中增长自身的知识，再将知识归纳、总结、升华成为理论的，所以笔者将本章安排到了整部文档的末尾，也是希望大家能先去实践，再谈理论。同时，笔者也认为对于一名研究人员，或者企业中真正能决定技术方向的决策者，理论与实践都不可缺少，涉及决策的场景中，成体系的理论知识甚至比实践经验还要关键，因为执行力再强也必须用在正确的方向上才有价值。如果你对自己的规划是有朝一日要从一名技术人员发展成研究或者管理角色，补充这部分知识是必不可少的。

第七章 专题随笔

字数: 8,508 字

本章无特定读者对象，内容全凭笔者心情。

这部分是一些笔者所了解的开发、设计中常见技巧和编程模式的集合，由于它们还不具备足够的系统性，没有安排入前面的知识框架之中。但有一些或精彩，或有价值，或实用的技巧，笔者不想错过，所以安排了这一章相对独立的内容。

另外，这部分内容类似于笔者的随笔博客，将不会出现在文档的音频版与传统纸质书版本中。

第八章 附录

字数: 9,267 字

本章面向刚刚开始接触云原生环境的设计者、开发者。

这一章内容主要是云原生环境搭建和程序发布过程，原本它们并不属于笔者准备讨论的重点话题，至少没有到单独开一章的必要程度。但由于容器化的服务编排环境本身构建、管理和运维都有一定的复杂性，尤其是在国内特殊的网络环境下，无法直接访问到Google等国外的代码仓库，以至于不得不通过手工预载镜像或者代理的方式来完成环境搭建。为了避免刚刚接触这一领域的读者在入门第一步就受到不必要的心理打击，笔者专门设置了这个目录章节。这章与其他几章讨论设计思想、实现原理的风格差异很大，它是整部文档唯一的讨论具体如何操作的内容。

市面上介绍如何安装环境的书籍、资料已经不计其数，肯定有相当一部分读者这章的内容本身就是了解的，已掌握的读者建议无需仔细阅读，在有需要的时候，可当作工具查阅。

技术演示工程

除文档部分外，笔者同时还建立了若干配套的代码工程，这是针对不同架构、技术方案（如单体架构、微服务、服务网格、无服务架构，等等）的演示程序（[PetStore-Like-Project](#)）。它们即是文档中所述知识的实践示例，亦可作为实际项目新创建时的可参考引用的基础代码。

本小节内容是由这些工程的README.md文件同步而来，由于未经过持续集成工具自动处理，所以可能有偶尔更新不一致的情况，如可能，建议到这些项目的GitHub页面上查看最新情况。

- 文档工程：
 - 软件架构探索：<https://icyfenix.cn>
 - Vuepress支持的文档工程：<https://github.com/fenixsoft/awesome-fenix>
- 前端工程：
 - Mock.js支持的纯前端演示：<https://bookstore.icyfenix.cn>
 - Vue.js 2实现前端工程：<https://github.com/fenixsoft/fenix-bookstore-frontend>
- 后端工程：
 - Spring Boot 实现单体架构：https://github.com/fenixsoft/monolithic_arch_springboot
 - Spring Cloud 实现微服务架构：https://github.com/fenixsoft/microservice_arch_springcloud
 - Kubernetes 为基础设施的微服务架构：https://github.com/fenixsoft/microservice_arch_kubernetes
 - Istio 为基础设施的服务网格架构：https://github.com/fenixsoft/servicemesh_arch_istio
 - AWS Lambda 为基础的无服务架构：https://github.com/fenixsoft/serverless_arch_awslambda

前端工程



Release v1.0 build passing Doc License CC 4.0 License Apache 2.0 Author IcyFenix

如果你此时并不曾了解过什么是“The Fenix Project”，建议先阅读[这部分内容](#)。

Fenix's Bookstore的主要目的是展示不同的后端技术架构，相对而言，前端并非其重点。不过，前端的页面是比起后端各种服务来要直观得多，能让使用者更容易理解我们将要做的是一件什么事情。假设你是一名驾驶初学者，合理的学习路径肯定应该是把汽车发动，然后慢慢行驶起来，而不是马上从“引擎动力原理”、“变速箱构造”入手去设法深刻地了解一台汽车。所以，先来运行程序，看看最终的效果是什么样子吧。

运行程序

以下几种途径，可以马上浏览最终的效果：

- 从互联网已部署（由提供Travis-CI支持）的网站（由GitHub Pages提供主机，由腾讯云CDN提供国内加速）访问：

直接在浏览器访问：<http://bookstore.icyfenix.cn>

- 通过Docker容器方式运行：

```
$ docker run -d -p 80:80 --name bookstore icyfenix/bookstore:frontend
```

sh

然后在浏览器访问：<http://localhost>

- 通过Git上的源码，以开发模式运行：

```
# 克隆获取源码  
$ git clone https://github.com/fenixsoft/fenix-bookstore-frontend.git  
  
# 进入工程根目录  
$ cd fenix-bookstore-frontend  
  
# 安装工程依赖  
$ npm install  
  
# 以开发模式运行，地址为localhost:8080  
$ npm run dev
```

然后在浏览器访问：<http://localhost:8080>



也许你已注意到，以上这些运行方式，均没有涉及到任何的服务端、数据库的部署。现代软件工程里，基于MVVM的工程结构使得前、后端的开发可以完全分离，只要互相约定好服务的位置及模型即可。Fenix's Bookstore以开发模式运行时，会自动使用Mock.js拦截住所有的远程服务请求，并以事项准备好的数据来完成对这些请求的响应。

同时，你也应当注意到，以纯前端方式运行的时候，所有对数据的修改请求实际都是无效的。譬如用户注册，无论你输入何种用户名、密码，由于请求的响应是静态预置的，所以最终都会以同一个预设的用户登陆。也是因此，我并没有提供“默认用户”、“默认密码”一类的信息供用户使用，你可以随意输入即可登陆。

不过，那些只维护在前端的状态依然可以变动的，典型的如对购物车、收藏夹的增删改。让后端服务保持无状态，而把状态维持在前端中的设计，对服务的伸缩性和系统的鲁棒性都有着极大的益处，多数情况下都是值得倡导的良好设计。而其伴随而来的状态数据导致请求头变大、链路安全性等问题，都会在服务端部分专门讨论和解决。

构建产品

当你将程序用于正式部署时，一般不应部署开发阶段的程序，而是要进行产品化（Product ion）与精简化（Minification），你可以通过以下命令，由node.js驱动webpack来自动完成：

```
# 编译前端代码  
$ npm run build
```

sh

或者使用--report参数，同时输出依赖分析报告：

```
# 编译前端代码并生成报告  
$ npm run build --report
```

sh

编译结果存放在/dist目录中，应将其拷贝至Web服务器的根目录使用。对于Fenix's Bookstore的各个服务端而言，则通常是拷贝到网关工程中静态资源目录下。

与后端联调

同样出于前后端分离的目的，理论上后端通常只应当依据约定的服务协议（接口定位、访问传输方式、参数及模型结构、服务水平协议等）提供服务，并以此为依据进行不依赖前端的独立测试，最终集成时使用的是编译后的前端产品。

不过，在开发期就进行的前后端联合在现今许多企业之中仍是主流形式，由一个人“全栈式”地开发某个功能时更是如此，因此，当要在开发模式中进行联调时，需要修改项目根目录下的main.js文件，使其不导入Mock.js，即如下代码所示的条件语句判断为假

```
/**  
 * 默认在开发模式中启用mock.js代替服务端请求  
 * 如需要同时调试服务端，请修改此处判断条件  
 */  
// eslint-disable-next-line no-constant-condition
```

js

```
if (process.env.MOCK) {  
    require('./api/mock')  
}
```

也有其他一些相反的情况，需要在生产包中仍然继续使用Mock.js提供服务时（譬如Docker镜像icyfenix/bookstore:frontend就是如此），同样应修改该条件，使其结果为真，在开发模式依然导入了Mock.js即可。

工程结构

Fenix's Bookstore的工程结构完全符合vue.js工程的典型习惯，事实上它在建立时就是通过vue-cli初始化的。此工程的结构与其中各个目录的作用主要如下所示：

----build	webpack编译配置，该目录的内容一般不做改动
----config	webpack编译配置，用户需改动的内容提取至此
----dist	编译输出结果存放的位置
----markdown 片)	与项目无关，用于支持markdown的资源（如图片）
----src	
+---api	本地与远程的API接口
+---local	本地服务，如localStorage、加密等
+---mock	远程API接口的Mock
\---json	Mock返回的数据
\---remote	远程服务
+---assets	资源文件，会被webpack哈希和压缩
+---components	vue.js的组件目录，按照使用页面的结构放置
+---home	
+---cart	
+---detail	
\---main	
\---login	
+---pages	vue.js的视图目录，存放页面级组件
\---home	
+---plugins	vue.js的插件，如全局异常处理器
+---router	vue-router路由配置
\---store	vuex状态配置
\---modules	vuex状态按名空间分隔存放

\---static
 缩

静态资源，编译时原样打包，不会做哈希和压

组件

Fenix's Bookstore前端部分基于以下开源组件和免费资源构建：

- [Vue.js](#)：渐进式JavaScript框架
- [Element](#)：一套为开发者、设计师和产品经理准备的基于Vue 2.0的桌面端组件库
- [Axios](#)：Promise based HTTP client for the browser and node.js
- [Mock.js](#)：生成随机数据，拦截 Ajax 请求
- [DesignEvo](#)：一款由PearlMountain有限公司设计研发的logo设计软件

协议

- 本作品代码部分采用[Apache 2.0协议](#)进行许可。遵循许可的前提下，你可以自由地对代码进行修改，再发布，可以将代码用作商业用途。但要求你：
 - 署名：在原有代码和衍生代码中，保留原作者署名及代码来源信息。
 - 保留许可证：在原有代码和衍生代码中，保留Apache 2.0协议文件。
- 本作品文档部分采用[知识共享署名 4.0 国际许可协议](#)进行许可。遵循许可的前提下，你可以自由地共享，包括在任何媒介上以任何形式复制、发行本作品，亦可以自由地演绎、修改、转换或以本作品为基础进行二次创作。但要求你：
 - 署名：应在使用本文档的全部或部分内容时候，注明原作者及来源信息。
 - 非商业性使用：不得用于商业出版或其他任何带有商业性质的行为。如需商业使用，请联系作者。
 - 相同方式共享的条件：在本文档基础上演绎、修改的作品，应当继续以知识共享署名 4.0国际许可协议进行许可。

单体架构：Spring Boot



Release v1.0 build passing coverage 90% License Apache 2.0 Doc License CC 4.0
Author IcyFenix

如果你此时并不曾了解过什么是“The Fenix Project”，建议先阅读[这部分内容](#)。

单体架构是Fenix's Bookstore'第一个版本的服务端实现，它与此后基于微服务（Spring Cloud、Kubernetes）、服务网格（Istio）、无服务（Serverless）架构风格实现的其他版本，在业务功能上的表现是完全一致的。如果你不是针对性地带着解决某个具体问题、了解某项具体工具、技术的目的而来，而是时间充裕，希望了解软件架构的全貌与发展的话，笔者推荐以此工程入手来了解现代软件架构，因为单体架构的结构是相对直观的，易于理解的架构，对后面接触的其他架构风格也起良好的铺垫作用。此外，笔者在对应的文档中详细分析了作为一个架构设计者，会考虑哪些的通用问题，希望把抽象的“架构”一词具象化出来。

运行程序

以下几种途径，可以运行程序，浏览最终的效果：

- 通过Docker容器方式运行：

```
$ docker run -d -p 8080:8080 --name bookstore
icyfenix/bookstore:monolithic
```

sh

然后在浏览器访问：<http://localhost:8080>，系统预置了一个用户（user:icyfenix，pw:123456），也可以注册新用户来测试。

默认会使用HSQLDB的内存模式作为数据库，并在系统启动时自动初始化好了Schema，完全开箱即用。但这同时也意味着当程序运行结束时，所有的数据都将不会被保留。

如果希望使用HSQLDB的文件模式，或者其他非嵌入式的独立的数据库支持的话，也是很简单的。以常用的MySQL/MariaDB为例，程序中也已内置了MySQL的表结构初始化脚本，你可以使用环境变量“PROFILES”来激活Spring Boot中针对MySQL所提供的配置，命令如下所示：

```
$ docker run -d -p 8080:8080 --name bookstore  
icyfenix/bookstore:monolithic -e PROFILES=mysql
```

sh

此时你需要通过Docker link、Docker Compose或者直接在主机的Host文件中提供一个名为“mysql_lan”的DNS映射，使程序能顺利链接到数据库，关于数据库的更多配置，可参考源码中的[application-mysql.yml](#)。

- 通过Git上的源码，以Maven运行：

```
# 克隆获取源码  
$ git clone https://github.com/fenixsoft/monolithic_arch_springboot.git  
  
# 进入工程根目录  
$ cd monolithic_arch_springboot  
  
# 编译打包  
# 采用Maven Wrapper，此方式只需要机器安装有JDK 8或以上版本即可，无需包括Maven在内的其他任何依赖  
# 如在Windows下应使用mvnw.cmd package代替以下命令  
$ ./mvnw package  
  
# 运行程序，地址为localhost:8080  
$ java -jar target/bookstore-1.0.0-Monolithic-SNAPSHOT.jar
```

sh

然后在浏览器访问：<http://localhost:8080>，系统预置了一个用户（user:icyfenix，pw:123456），也可以注册新用户来测试。

- 通过Git上的源码，在IDE环境中运行：

- 以IntelliJ IDEA为例，Git克隆本项目后，在File -> Open菜单选择本项目所在的目录，或者pom.xml文件，以Maven方式导入工程。
- IDEA将自动识别出这是一个SpringBoot工程，并定位启动入口为BookstoreApplication，待IDEA内置的Maven自动下载完所有的依赖包后，运行该类即可启动。
- 如你使用其他的IDE，没有对SpringBoot的直接支持，亦可自行定位到BookstoreApplication，这是一个带有main()方法的Java类，运行即可。
- 可通过IDEA的Maven面板中Lifecycle里面的package来对项目进行打包、发布。
- 在IDE环境中修改配置（如数据库等）会更加简单，具体可以参考工程中application.yml和application-mysql.yml中的内容。

技术组件

Fenix's Bookstore单体架构后端尽可能采用标准的技术组件进行构建，不依赖与具体的实现，包括：

- [JSR 370 : Java API for RESTful Web Services 2.1](#) (JAX-RS 2.1)
RESTful服务方面，采用的实现为Jersey 2，亦可替换为Apache CXF、RESTEasy、WebSphere、WebLogic等
- [JSR 330 : Dependency Injection for Java 1.0](#)
依赖注入方面，采用的实现为SpringBoot 2中内置的Spring Framework 5。虽然在多数场合中尽可能地使用了JSR 330的标准注解，但仍有少量地方由于Spring对@Named、@Inject等注解的支持表现上与本身提供的注解差异，使用了Spring的私有注解。如替换成其他的CDI实现，如HK2，需要较大的改动
- [JSR 338 : Java Persistence 2.2](#)
持久化方面，采用的实现为Spring Data JPA。可替换为Batoo JPA、EclipseLink、OpenJPA等实现，只需将使用CrudRepository所省略的代码手动补全回来即可，无需其他改动。
- [JSR 380 : Bean Validation 2.0](#)
数据验证方面，采用的实现为Hibernate Validator 6，可替换为Apache BVal等其他验证

框架

- [JSR 315 : Java Servlet 3.0](#)

Web访问方面，采用的实现为SpringBoot 2中默认的Tomcat 9 Embed，可替换为Jetty、Undertow等其他Web服务器

有以下组件仍然依赖了非标准化的技术实现，包括：

- [JSR 375 : Java EE Security API specification 1.0](#)

认证/授权方面，在2017年才发布的JSR 375中仍然没有直接包含OAuth2和JWT的直接支持，因后续实现微服务架构时对比的需要，单体架构中选择了Spring Security 5作为认证服务，Spring Security OAuth 2.3作为授权服务，Spring Security JWT作为JWT令牌支持，并未采用标准的JSR 375实现，如Soteria。

- [JSR 353/367 : Java API for JSON Processing/Binding](#)

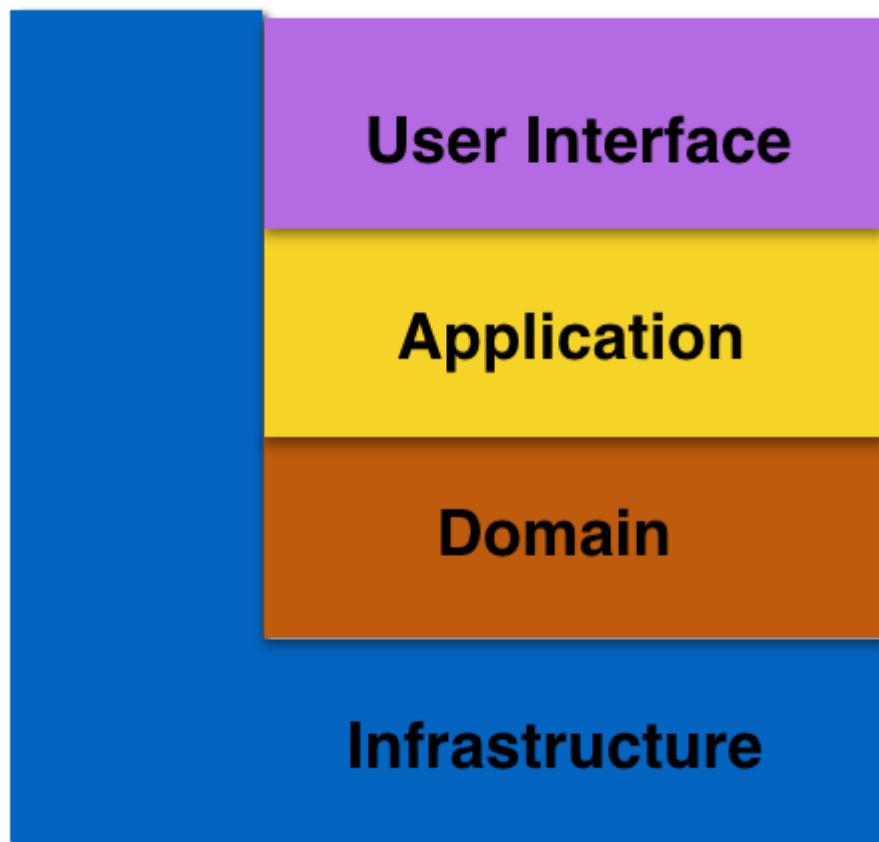
在JSON序列化/反序列化方面，由于Spring Security OAuth的限制（使用JSON-B作为反序列化器时的结果与Jackson等有差异），采用了Spring Security OAuth默认的Jackson，并未采用标准的JSR 353/367实现，如Apache Johnzon、Eclipse Yasson等。

工程结构

Fenix's Bookstore单体架构后端参考（并未完全遵循）了DDD的分层模式和设计原则，整体分为以下四层：

1. Resource：对应DDD中的User Interface层，负责向用户显示信息或者解释用户发出的命令。请注意，这里指的“用户”不一定是使用用户界面的人，可以是位于另一个进程或计算机的服务。由于本工程采用了MVVM前后端分离模式，这里所指的用户实际上是前端的服务消费者，所以这里以RESTFul中的核心概念“资源”（Resource）来命名。
2. Application：对应DDD中的Application层，负责定义软件本身对外暴露的能力，即软件本身可以完成哪些任务，并负责对内协调领域对象来解决问题。根据DDD的原则，应用层要尽量简单，不包含任何业务规则或者知识，而只为下一层中的领域对象协调任务，分配工作，使它们互相协作，这一点在代码上表现为Application层中一般不会存在任何的条件判断语句。在许多项目中，Application层都会被选为包裹事务（代码进入此层事务开始，退出此层事务提交或者回滚）的载体。

3. Domain：对应DDD中的Domain层，负责实现业务逻辑，即表达业务概念，处理业务状态信息以及业务规则这些行为，此层是整个项目的特点。
4. Infrastructure：对应DDD中的Infrastructure层，向其他层提供通用的技术能力，譬如持久化能力、远程服务通讯、工具集，等等。



协议

- 本文档代码部分采用[Apache 2.0 协议](#)进行许可。遵循许可的前提下，你可以自由地对代码进行修改，再发布，可以将代码用作商业用途。但要求你：
 - 署名：在原有代码和衍生代码中，保留原作者署名及代码来源信息。
 - 保留许可证：在原有代码和衍生代码中，保留Apache 2.0协议文件。
- 本作品文档部分采用[知识共享署名 4.0 国际许可协议](#)进行许可。遵循许可的前提下，你可以自由地共享，包括在任何媒介上以任何形式复制、发行本作品，亦可以自由地演绎、修改、转换或以本作品为基础进行二次创作。但要求你：
 - 署名：应在使用本文档的全部或部分内容时候，注明原作者及来源信息。

- **非商业性使用**：不得用于商业出版或其他任何带有商业性质的行为。如需商业使用，请联系作者。
- **相同方式共享的条件**：在本文档基础上演绎、修改的作品，应当继续以知识共享署名4.0国际许可协议进行许可。

微服务 : Spring Cloud



Release v1.0 build passing coverage 81% License Apache 2.0 Doc License CC 4.0

Author IcyFenix

如果你此时并不曾了解过什么是“The Fenix Project”，建议先阅读[这部分内容](#)。

至少到目前，基于Spring Cloud的微服务解决方案仍是以Java为运行平台的微服务中，使用者数量最多的一个分支。这个结果即是Java在服务端应用中长久积累的深厚基础的体现，也是Spring在Java应用中统治性的地位的体现。Spring Cloud令现存数量极为庞大的、基于Spring和Spring Boot的单体系统，得以平滑地迁移到微服务架构中，令这些系统的大部分代码都能够无需或少量修改即可保留重用。微服务兴起的早期，Spring Cloud就集成了[Netflix OSS](#)（以及Spring Cloud Netflix进入维护期后对应的替代组件）所开发的体系化的微服务套件，基本上算“半透明地”解决了在微服务环境中必然会面临的服务发现、远程调用、负载均衡、集中配置等基础问题。

不过，笔者自己并不太认同Spring Cloud Netflix这种以应用代码去解决基础设施功能问题的“解题思路”，以笔者的观点看来，这既是容器化、原生化的微服务基础设施完全成熟之前必然会出现的应用形态，同时也决定了这是微服务进化过程中必然会被替代的过渡形态。无论笔者的看法如何，基于Spring Cloud Netflix的微服务在当前是主流，直至未来不算短的一段时间内仍会是主流，并且以应用的视角，自顶向下观察基础设施在微服务中面临的需求和挑战，用我们熟悉的Java代码来解释分析问题，也有利于对微服务的整体思想的深入理解，所以将它作为我们了解的第一种微服务架构的实现是十分适合的。

需求场景

小书店Fenix's Bookstore生意日益兴隆，客人、货物、营收都在持续增长，业务越发复杂，对信息系统并发与可用方面的要求也越来越高。由于业务属性和质量属性要求的提升，信息系统需要更多的硬件资源去支撑，这是合乎情理的，但是，如果我们把需求场景列的更具体些，便会发现“合理”下面的许多无可奈何之处：

- 譬如，恰逢双十一购物节，短时间内会有大批的交易事件发生，这时候运维的同学对系统进行扩容以应对流量洪峰。但此时增长的业务量并不是均衡的，只有商品交易的活动剧增，其他的活动，如新商品入库、新用户注册这类并未增加多少，此时，面对“铁板一块”的单体系统，运维在做扩容时，只能把“用不上的”商品管理代码、用户管理代码也一并扩容部署。
- 譬如，高性能硬件对性能的提升是有帮助，但对稳定性的提升通常无能为力。业务复杂度的增加促使系统的技术复杂度也在持续增长，当系统不可避免地滑向庞大臃肿时，总伴随有各种难以预料的问题出现；要维持一个庞然大物的健康生存，也对设计、开发、运维各方面的人员都提出越来越高的要求。人力终有穷时，迟早会面临“没有一个人能了解系统的所有细节”的情形；系统的复杂程度也总有极限，持续膨胀的代码终会有崩溃的一刻。
- 譬如，……

微服务的需求场景还可以列举很多，这里就不多列举了，总之，系统发展到一定程度，我们总能找到充分的理由去重构拆分它。在笔者设定的场景中，准备把单体的Fenix's Bookstore拆分为“用户”、“商品”、“交易”三个能够独立运行的子系统，它们将在一系列非功能性模块（认证授权等）和基础设施（配置中心、服务发现等）的支撑下互相协作，以统一的API网关对外提供与原来单体系统在功能上一致的服务，应用视图如下图所示：



运行程序

以下几种途径，可以运行程序，浏览最终的效果：

- 通过Docker容器方式运行：

微服务涉及到多个容器的协作，通过link单独运行容器已经被Docker官方声明为不提倡的方式，所以在工程中提供了专门的配置，以便使用docker-compose来运行：

```
# 下载docker-compose配置文件
$ curl -O
https://raw.githubusercontent.com/fenixsoft/microservice_arch_springcloud/master/docker-compose.yml
```

```
# 启动服务  
$ docker-compose up
```

然后在浏览器访问：<http://localhost:8080>，系统预置了一个用户（user:icyfenix，pw:123456），也可以注册新用户来测试。

- 通过Git上的源码，以Maven编译、运行：

由于笔者已经在配置文件中设置好了各个微服务的默认的地址和端口号，以便于本地调试。如果要在同一台机运行这些服务，并且每个微服务都只启动一个实例的话，那不加任何配置、参数即可正常以Maven编译、以Jar包形式运行。由于各个微服务需要从配置中心里获取具体的参数信息，因此唯一的要求只是“配置中心”的微服务必须作为第一个启动的服务进程，对其他的启动顺序则没有更多要求了。具体的操作过程如下所示：

```
# 克隆获取源码  
$ git clone  
https://github.com/fenixsoft/microservice_arch_springcloud.git  
  
# 进入工程根目录  
$ cd microservice_arch_springcloud  
  
# 编译打包  
# 采用Maven Wrapper，此方式只需要机器安装有JDK 8或以上版本即可，无需包括  
Maven在内的其他任何依赖  
# 克隆后你可能需要使用chmod给mvnw赋予执行权限，如在Windows下应使用mvnw.cmd  
package代替以下命令  
$ ./mvnw package  
  
# 工程将编译出七个SpringBoot Jar  
# 启动服务需要运行以下七个微服务组件  
# 配置中心微服务：localhost:8888  
$ java -jar ./bookstore-microservices-platform-  
configuration/target/bookstore-microservice-platform-configuration-  
1.0.0-SNAPSHOT.jar  
# 服务发现微服务：localhost:8761  
$ java -jar ./bookstore-microservices-platform-  
registry/target/bookstore-microservices-platform-registry-1.0.0-  
SNAPSHOT.jar  
# 服务网关微服务：localhost:8080  
$ java -jar ./bookstore-microservices-platform-  
gateway/target/bookstore-microservices-platform-gateway-1.0.0-  
SNAPSHOT.jar
```

```
# 安全认证微服务：localhost:8301  
$ java -jar ./bookstore-microservices-domain-  
security/target/bookstore-microservices-domain-security-1.0.0-  
SNAPSHOT.jar  
# 用户信息微服务：localhost:8401  
$ java -jar ./bookstore-microservices-domain-  
account/target/bookstore-microservices-domain-account-1.0.0-  
SNAPSHOT.jar  
# 商品仓库微服务：localhost:8501  
$ java -jar ./bookstore-microservices-domain-  
warehouse/target/bookstore-microservices-domain-warehouse-1.0.0-  
SNAPSHOT.jar  
# 商品交易微服务：localhost:8601  
$ java -jar ./bookstore-microservices-domain-  
payment/target/bookstore-microservices-domain-payment-1.0.0-  
SNAPSHOT.jar
```

由于在命令行启动多个服务、通过容器实现各服务隔离、扩展等都较繁琐，笔者提供了一个docker-compose.dev.yml文件，便于开发期调试使用：

```
# 使用Maven编译出JAR包后，可使用以下命令直接在本地构建镜像运行  
$ docker-compose -f docker-compose.dev.yml up
```

sh

以上两种本地运行的方式可任选其一，服务全部启动后，在浏览器访问：<http://localhost:8080>，系统预置了一个用户（user:icyfenix，pw:123456），也可以注册新用户来测试

- 通过Git上的源码，在IDE环境中运行：

- 以IntelliJ IDEA为例，Git克隆本项目后，在File -> Open菜单选择本项目所在的目录，或者pom.xml文件，以Maven方式导入工程。
- 待Maven自动安装依赖后，即可在IDE或者Maven面板中编译全部子模块的程序。
- 本工程下面八个模块，其中除bookstore-microservices-library-infrastructure外，其余均是SpringBoot工程，将这七个工程的Application类加入到IDEA的Run Dashboard面板中。
- 在Run Dashboard中先启动“bookstore-microservices-platform-configuration”微服务，然后可一次性启动其余六个子模块的微服务。

- 配置与横向扩展

工程中预留了一些的环境变量，便于配置和扩展，譬如，对于热点模块，往往需要启动多个微服务扩容，此时需要调整每个服务的端口号。预留的这类环境变量包括：

```
# 修改配置中心的主机和端口，默认为localhost:8888  
CONFIG_HOST  
CONFIG_PORT  
  
# 修改服务发现的主机和端口，默认为localhost:8761  
REGISTRY_HOST  
REGISTRY_PORT  
  
# 修改认证中心的主机和端口，默认为localhost:8301  
AUTH_HOST  
AUTH_PORT  
  
# 修改当前微服务的端口号  
# 譬如，你打算在一台机器上扩容四个支付微服务以应对促销活动的流量高峰  
# 可将它们的端口设置为8601（默认）、8602、8603、8604等  
# 真实环境中，它们可能是在不同的物理机、容器环境下，这时扩容可无需调整端口  
PORT  
  
# SpringBoot所采用Profile配置文件，默认为default  
# 譬如，服务默认使用HSQLDB的内存模式作为数据库，如需调整为MySQL，可将此环境变量调整为mysql  
# 因为笔者默认预置了名为application-mysql.yml的配置，以及HSQLDB和MySQL的数据库脚本  
# 如果你需要支持其他数据库、修改程序中其他的配置信息，可以在代码中自行加入另外的初始化脚本  
PROFILES  
  
# Java虚拟机运行参数，默认为空  
JAVA_OPTS
```

技术组件

Fenix's Bookstore采用基于Spring Cloud微服务架构，微服务部分主要采用了Netflix OSS组件进行支持，它们包括：

- **配置中心**：默认采用[Spring Cloud Config](#)，亦可使用[Spring Cloud Consul](#)、[Spring Cloud Alibaba Nacos](#)代替。
- **服务发现**：默认采用[Netflix Eureka](#)，亦可使用[Spring Cloud Consul](#)、[Spring Cloud ZooKeeper](#)、[etcd](#)等代替。
- **服务网关**：默认采用[Netflix Zuul](#)，亦可使用[Spring Cloud Gateway](#)代替。
- **服务治理**：默认采用[Netflix Hystrix](#)，亦可使用[Sentinel](#)、[Resilience4j](#)代替。
- **进程内负载均衡**：默认采用[Netflix Ribbon](#)，亦可使用[Spring Cloud Loadbalancer](#)代替。
- **声明式HTTP客户端**：默认采用[Spring Cloud OpenFeign](#)。声明式的HTTP客户端其实没有找替代品的必要性，如果需要，可考虑[Retrofit](#)，或者使用[RestTemplate](#)乃至更底层的[OkHTTP](#)、[HttpClient](#)以命令式编程来访问，多写一些代码而已了。

尽管Netflix套件的使用人数很多，但由于Spring Cloud Netflix已进入维护模式，所以笔者均列出了上述组件的代替品。这些组件几乎都是声明式的，这保证了替代它们的成本相当低，只需要更换注解，修改配置，无需改动代码。你在阅读源码时也会发现，三个“platform”开头的服务，基本上没有任何实际代码的存在。

其他与微服务无关的技术组件（REST服务、安全、数据访问，等等），笔者已在[Fenix's Bookstore单体架构](#)中介绍过，在此不再重复。

协议

- 本作品代码部分采用[Apache 2.0协议](#)进行许可。遵循许可的前提下，你可以自由地对代码进行修改，再发布，可以将代码用作商业用途。但要求你：
 - **署名**：在原有代码和衍生代码中，保留原作者署名及代码来源信息。
 - **保留许可证**：在原有代码和衍生代码中，保留Apache 2.0协议文件。
- 本作品文档部分采用[知识共享署名 4.0 国际许可协议](#)进行许可。遵循许可的前提下，你可以自由地共享，包括在任何媒介上以任何形式复制、发行本作品，亦可以自由地演绎、修改、转换或以本作品为基础进行二次创作。但要求你：
 - **署名**：应在使用本文档的全部或部分内容时候，注明原作者及来源信息。
 - **非商业性使用**：不得用于商业出版或其他任何带有商业性质的行为。如需商业使用，请联系作者。

- **相同方式共享的条件** : 在本文档基础上演绎、修改的作品，应当继续以知识共享署名4.0国际许可协议进行许可。

微服务：Kubernetes



Release v1.0 build passing License Apache 2.0 Doc License CC 4.0 Author IcyFenix

如果你此时并不曾了解过什么是“The Fenix Project”，建议先阅读[这部分内容](#)。

2017年，笔者曾在文章中描述其为“[后微服务时代](#)”的开端，这年是容器生态发展历史中具有里程碑意义的一年。在这一年，长期作为Docker竞争对手的[RKT容器](#)一派的领导者CoreOS宣布放弃自己的容器管理系统Fleet，未来将会把所有容器管理的功能移至Kubernetes之上实现。在这一年，容器管理领域的独角兽Rancher Labs宣布放弃其内置了数年的容器管理系统Cattle，提出了“All-in-Kubernetes”战略，从2.0版本开始把1.x版本能够支持多种容器管理工具的Rancher，“升级”为只支持Kubernetes一种容器管理系统。在这一年，Kubernetes的主要竞争者Apache Mesos在9月正式宣布了“[Kubernetes on Mesos](#)”集成计划，由竞争关系转为对Kubernetes提供支持，使其能够与Mesos的其他一级框架（如[HDFS](#)、[Spark](#) 和[Chronos](#)，等等）进行集群资源动态共享、分配与隔离。在这一年，Kubernetes的最大竞争者Docker Swarm的母公司Docker，终于在10月被迫宣布Docker要同时支持Swarm与Kubernetes两套容器管理系统，事实上承认了Kubernetes的统治地位。这场已经持续了三、四年时间，以Docker Swarm、Apache Mesos与Kubernetes为主要竞争者的“容器战争”终于有了明确的结果，Kubernetes登基加冕是容器发展中一个时代的终章，也将是软件架构发展下一个纪元的开端。

需求场景

当引入了基于Spring Cloud的微服务架构后，小书店Fenix's Bookstore初步解决了扩容缩容、独立部署、运维和管理等问题，满足了产品经理不断提出的日益复杂的业务需求。可是，对于团队的开发人员、设计人员、架构人员来说，并没有感觉到工作变得轻松，微服务中的各种新技术名词，如配置中心、服务发现、网关、熔断、负载均衡等等，就够一名新手学习好长一段时间；从产品角度来看，各种Spring Cloud的技术套件，如Config、Eureka、Zuul、Hystrix、Ribbon、Feign等，也占据了产品的大部分编译后的代码容量。之所以微服务架构里，我们选择在应用层面而不是基础设施层面去解决这些分布式问题，完全是因为由硬件构成的基础设施，跟不上由软件构成的应用服务的灵活性的无奈之举。当Kubernetes统一了容器编排管理系统之后，这些纯技术性的底层问题，便开始有了被广泛认可和采纳的基础设施层面的解决方案。为此，Fenix's Bookstore也迎来了它在“后微服务时代”中的下一次架构演进，这次升级的目标主要有如下两点：

- **目标一**：尽可能缩减非业务功能代码的比例。

在Fenix's Bookstore中，用户服务（Account）、商品服务（Warehouse）、交易服务（Payment）三个工程是真正承载业务逻辑的，认证授权服务（Security）可以认为是同时涉及到了技术与业务，而配置中心（Configuration）、网关（Gateway）和服务注册中心（Registry）则是纯技术性。我们希望尽量消除这些纯技术的工程，以及那些依附在其他业务工程上的纯技术组件。

- **目标二**：尽可能在不影响原有的代码的前提下完成迁移。

得益于Spring Framework 4中的Conditional Bean等声明式特性的出现，近年来新发布的技术组件，**声明式编程**（Declarative Programming）已经逐步取代**命令式编程**（Imperative Programming）成为主流。这使得我们可以从目的而不是过程的角度去描述编码意图，使得代码几乎不会与具体技术实现产生耦合，若要更换一种技术实现，只需要调整配置中的声明便可做到。

从升级结果来看，如果仅以Java代码的角度来衡量，本工程与此前基于Spring Cloud的实现没有丝毫差异，两者的每一行Java代码都是一模一样的；其区别是Kubernetes的实现版本中直接删除了配置中心、服务注册中心的工程，在其他工程的pom.xml中也删除了如Eureka、Ribbon、Config等组件的依赖。取而代之的是新增了若干以YAML配置文件为载体的**Skaffold**和Kubernetes的资源描述，这些资源描述文件，将会动态构建出DNS服务器、服务负载均衡器等一系列虚拟化的基础设施，去代替原有的应用层面的技术组件。升级改造之后的应用架构如下图所示：



运行程序

在已经部署Kubernetes集群的前提下，通过以下几种途径，可以运行程序，浏览最终的效果：

- 直接在Kubernetes集群环境上运行：

工程在编译时已通过Kustomize产生出集成式的资源描述文件，可通过该文件直接在Kubernetes集群中运行程序：

```
# 资源描述文件
$ kubectl apply -f
https://raw.githubusercontent.com/fenixsoft/microservice_arch_kubernetes/master/bookstore.yaml
```

当所有的Pod都处于正常工作状态后（这个过程一共需要下载几百MB的镜像，尤其是Docker中没有各层基础镜像缓存时，请根据自己的网速保持一定的耐心。未来GraalVM对Spring Cloud的支持更成熟一些后，可以考虑采用GraalVM来改善这一点），在浏览器访问：<http://localhost:30080>，系统预置了一个用户（user:icyfenix，pw:123456），也可以注册新用户来测试。

- 通过Skaffold在命令行或IDE中以调试方式运行：

一般开发基于Kubernetes的微服务应用，是在本地针对单个服务编码、调试完成后，通过CI/CD流水线部署到Kubernetes中进行集成的。如果只是针对集成测试，这并没有什么问题，但同样的做法应用在开发阶段就不十分不便了，我们不希望每做一处修改都要经过一次CI/CD流程，这将非常耗时且难以调试。

Skaffold是Google在2018年开源的一款加速应用在本地或远程Kubernetes集群中构建、推送、部署和调试的自动化命令行工具，对于Java应用来说，它可以帮助我们做到监视代码变动，自动打包出镜像，将镜像打上动态标签并更新部署到Kubernetes集群，为Java程序注入开放JDWP调试的参数，并根据Kubernetes的服务端口自动在本地生成端口转发。以上都是根据 `skaffold.yaml` 中的配置来进行的，开发时skaffold通过 `dev` 指令来执行这些配置，具体的操作过程如下所示：

```
# 克隆获取源码  
$ git clone  
https://github.com/fenixsoft/microservice_arch_kubernetes.git && cd  
microservice_arch_kubernetes  
  
# 编译打包  
$ ./mvnw package  
  
# 启动Skaffold  
# 此时将会自动打包Docker镜像，并部署到Kubernetes中  
$ skaffold dev
```

服务全部启动后，在浏览器访问：<http://localhost:30080>，系统预置了一个用户（user:icyfenix，pw:123456），也可以注册新用户来测试

由于面向的是开发环境，基于效率原因，笔者并没有像传统CI工程那样直接使用Maven的Docker镜像来打包Java源码，这决定了构建Dockerfile时，要监视的变动目标将是Jar

文件而不是Java源码，即Skaffold监视的是Jar包的变动，只当进行Maven编译、输出了新的Jar包后才会更新镜像。这样做一方面是考虑到在Maven镜像中打包不便于利用本地的仓库缓存，尤其在国内网络中，速度实在难以忍受；另外一方面，是笔者其实并不希望每保存一次源码时，都自动构建和更新一次镜像，毕竟比起传统的HotSwap或者Spring Devtool Reload来说，更新镜像重启Pod是一个更加重负载的操作。未来CNCF的Buildpack¹²成熟之后，应该可以绕过笨重的Dockerfile，对打包和容器热更新做更加精细化的控制。

另外，对于有IDE调试需求的同学，推荐采用Google Cloud Code¹³（Cloud Code同时提供了VS Code和IntelliJ Idea的插件）来配合Skaffold使用，毕竟是一个公司出品的产品，搭配起来能获得几乎与本地开发单体应用一致的体验。

技术组件

Fenix's Bookstore采用基于Kubernetes的微服务架构，并采用Spring Cloud Kubernetes做了适配，其中主要的技术组件包括：

- **容器环境感知**：Spring Cloud Kubernetes本身引入了Fabric8的Kubernetes Client¹⁴作为容器环境感知，不过引用的版本相当陈旧，如Spring Cloud Kubernetes 1.1.2中采用的是Fabric8 Kubernetes Client 4.4.1，Fabric8提供的兼容性列表中该版本只支持到Kubernetes 1.14，实测在1.16上也能用，但是在1.18上无法识别到最新的Api-Server，因此Maven引入依赖时需要手工处理，排除旧版本，引入新版本（本工程采用的是4.10.1）。
- **配置中心**：采用Kubernetes的ConfigMap来管理，通过Spring Cloud Kubernetes Config¹⁵自动将ConfigMap的内容注入到Spring配置文件中，并实现动态更新。
- **服务发现**：采用Kubernetes的Service来管理，通过Spring Cloud Kubernetes Discovery¹⁶自动将HTTP访问中的服务转换为FQDN¹⁷。
- **负载均衡**：采用Kubernetes Service本身的负载均衡能力实现（就是DNS负载均衡），可以不再需要Ribbon这样的客户端负载均衡了。Spring Cloud Kubernetes从1.1.2开始也已经移除了对Ribbon的适配支持，也（暂时）没有对其代替品Spring Cloud LoadBalancer提供适配。
- **服务网关**：网关部分仍然保留了Zuul，未采用Ingress代替。这里有两点考虑，一是Ingress Controller不算是Kubernetes的自带组件，它可以有不同的选择（KONG、Nginx、Haproxy，等等），同时也需要独立安装，作为演示工程，出于环境复杂度最小化考虑

未使用Ingress；二是Fenix's Bookstore的前端工程是存放在网关中的，移除了Zuul之后也仍然要维持一个前端工程的存在，不能进一步缩减工程数量，也就削弱了移除Zuul的动力。

- **服务熔断**：仍然采用Hystrix，Kubernetes本身无法做到精细化的服务治理，包括熔断、流控、监视，等等，我们将在基于Istio的服务网格架构中解决这个问题。
- **认证授权**：仍然采用Spring Security OAuth 2，Kubernetes的RBAC授权可以解决服务间的安全访问问题，但Security是跨越了业务和技术的边界的，认证授权模块本身仍承担着对前端用户的认证、授权职责，这部分是与业务相关的。

协议

- 本文档代码部分采用[Apache 2.0协议](#)进行许可。遵循许可的前提下，你可以自由地对代码进行修改，再发布，可以将代码用作商业用途。但要求你：
 - **署名**：在原有代码和衍生代码中，保留原作者署名及代码来源信息。
 - **保留许可证**：在原有代码和衍生代码中，保留Apache 2.0协议文件。
- 本作品文档部分采用[知识共享署名 4.0 国际许可协议](#)进行许可。遵循许可的前提下，你可以自由地共享，包括在任何媒介上以任何形式复制、发行本作品，亦可以自由地演绎、修改、转换或以本作品为基础进行二次创作。但要求你：
 - **署名**：应在使用本文档的全部或部分内容时候，注明原作者及来源信息。
 - **非商业性使用**：不得用于商业出版或其他任何带有商业性质的行为。如需商业使用，请联系作者。
 - **相同方式共享的条件**：在本文档基础上演绎、修改的作品，应当继续以知识共享署名 4.0国际许可协议进行许可。

服务网格：Istio



Release v1.0 build passing License Apache 2.0 Doc License CC 4.0 Author IcyFenix

如果你此时并不曾了解过什么是“The Fenix Project”，建议先阅读[这部分内容](#)。

当软件架构演进至基于Kubernetes实现的微服务时，已经能够相当充分地享受到虚拟化技术发展的红利，如应用能够灵活地扩容缩容、不再畏惧单个服务的崩溃消亡、立足应用系统更高层来管理和编排各服务之间的版本、交互。可是，单纯的Kubernetes仍然不能解决我们面临的所有分布式技术问题，在此前对基于Kubernetes的架构中“[技术组件](#)”的介绍里，笔者已经说明光靠着Kubernetes本身的虚拟化基础设施，难以做到精细化的服务治理，譬如熔断、流控、监视，等等；而即使是那些它可以提供支持的分布式能力，譬如通过DNS与服务来实现的服务发现与负载均衡，也只能说是初步解决了的分布式中如何调用服务的问题而已，只靠DNS难以满足根据不同的配置规则、协议层次、均衡算法等去调节负载均衡的执行过程这类高级的配置需求。Kubernetes提供的虚拟化基础设施是我们尝试从应用中剥离分布式技术代码踏出的第一步，但只从微服务的灵活与可控这一点而言，基于Kubernetes实现的版本其实比上一个Spring Cloud版本里用代码实现的效果有所倒退的，这也是当时我们未放弃Hystrix、Spring Security OAuth 2等组件的原因。

Kubernetes给予了我们强大的虚拟化基础设施，这是一把好用的锤子，但我们却不必把所有问题都看作钉子，不必只局限于纯粹基础设施的解决方案。现在，基于Kubernetes之上构筑的服务网格（Service Mesh）是目前最先进的架构风格，即通过中间人流量劫持的方式，介乎于应用和基础设施之间的边车代理（Sidecar）来做到既让用户代码可以专注业务需求，不必关注分布式的实现，又能实现几乎不亚于此前Spring Cloud时代的那种通过

代码来解决分布式问题的可配置、安全和可观测性。这一个目标，现在已成为了最热门的服务网格框架Istio的Slogan：“Connect, secure, control, and observe services”。

需求场景

得益于Kubernetes的强力支持，小书店Fenix's Bookstore已经能够依赖虚拟化基础设施进行扩容缩容，将用户请求分散到数量不定的Pod中处理，可以应对相当规模的用户量了。不过，随着Kubernetes集群中的Pod数量规模越来越庞大，到一定程度之后，运维的同学无奈地表示已经不可能够依靠人力来跟进微服务中出现的各种问题了：一个请求在哪个服务上调用失败啦？是A有调用B吗？还是C调用D时出错了？为什么这个请求、页面忽然卡住了？怎么调度到这个Node上的服务比其他Node慢那么多？这个Pod有Bug，消耗了大量的TCP链接数……

而另外一方面，随着Fenix's Bookstore程序规模与用户规模的壮大，开发团队人员数量也变得越来越多。尽管根据不同微服务进行拆分，可以将每个服务的团队成员都控制于“2 Pizza Teams”的范围以内，但一个很现实的问题是高端技术人员的数量总是有限的，人多了就不可能保证每个人都是精英，如何让普通的、初级的程序员依然能够做出靠谱的代码，成为这一阶段技术管理者的要重点思考的难题。这时候，团队内部出现了一种声音：微服务太复杂了，已经学不过来了，让我们回归单体吧……

在上述故事背景下，Fenix's Bookstore迎来了它的下一次技术架构的演进，这次的进化的目标主要有以下两点：

- **目标一**：实现在大规模虚拟服务下可管理、可观测的系统。

必须找到某种方法，针对应用系统整体层面，而不是针对单一微服务来连接、调度、配置和观测服务的执行情况。此时，可视化整个系统的服务调用关系，动态配置调节服务节点的断路、重试和均衡参数，针对请求统一收集服务间的处理日志等功能就不再是系统锦上添花的外围功能了，而是关乎系统是否能够正常运行、运维的必要支撑点。

- **目标二**：在代码层面，裁剪技术栈深度，回归单体架构中基于Spring Boot的开发模式，而不是Spring Cloud或者Spring Cloud Kubernetes的技术架构。

我们并不是要去开历史的倒车，相反，我们是很贪心地希望开发重新变得简单的同时，又不能放弃现在微服务带来的一切好处。在这个版本的Fenix's Bookstore里，所有与Spring Cloud相关的技术组件，如上个版本遗留的Zuul网关、Hystrix断路器，还有上个版本新引入用于感知适配Kubernetes环境的Spring Cloud Kubernetes都将会被拆除掉。如

果只观察单个微服务的技术堆栈，它与最初的单体架构几乎没有任何不同——甚至还更加简单了，连从单体架构开始一直保护着服务调用安全的Spring Security都移除掉（由于Fenix's Bookstore借用了Spring Security OAuth2的密码模式做为登陆服务的端点，所以在JAR包层面Spring Security还是存在的，但其用于安全保护的Servlet和Filter已经被关闭掉）

从升级目标可以明确地得到一种导向，我们必须控制住服务数量膨胀后传递到运维团队的压力，让“每运维人员能支持服务的数量”这个比例指标有指数级地提高才能确保微服务下运维团队的健康运作。对于开发团队，我们可以只要求一小部分核心的成员对微服务、Kubernetes、Istio等技术有深刻的理解即可，其余大部分开发人员，仍然可以基于最传统、普通的Spring Boot技术栈来开发功能。升级改造之后的应用架构如下图所示：



运行程序

在已经部署Kubernetes与Istio的前提下，通过以下几种途径，可以运行程序，浏览最终的效果：

- 在Kubernetes无Sidecar状态下运行：

在业务逻辑的开发过程中，或者其他不需要双向TLS、不需要认证授权支持、不需要可观测性支持等非功能性能力增强的环境里，可以不启动Envoy（但还是要安装Istio的，因为用到了Istio Ingress Gateway），工程在编译时已通过Kustomize产生出集成式的资源描述文件：

```
# Kubernetes without Envoy资源描述文件  
$ kubectl apply -f  
https://raw.githubusercontent.com/fenixsoft/servicemesh_arch_istio/master/bookstore-dev.yaml
```

请注意资源文件中对Istio Ingress Gateway的设置是针对Istio默认安装编写的，即以“istio-ingressgateway”作为标签，以LoadBalancer形式对外开放80端口，对内监听8080端口。在部署时可能需要根据实际情况进行调整，你可观察以下命令的输出结果来确认这一点：

```
$ kubectl get svc istio-ingressgateway -n istio-system -o yaml
```

在浏览器访问：<http://localhost>，系统预置了一个用户（user:icyfenix，pw:123456），也可以注册新用户来测试。

- 在Istio服务网格环境下运行：

工程在编译时已通过Kustomize产生出集成式的资源描述文件，可通过该文件直接在Kubernetes with Envoy集群中运行程序：

```
# Kubernetes with Envoy 资源描述文件  
$ kubectl apply -f  
https://raw.githubusercontent.com/fenixsoft/servicemesh_arch_istio/master/bookstore.yaml
```

当所有的Pod都处于正常工作状态后（这个过程一共需要下载几百MB的镜像，尤其是Docker中没有各层基础镜像缓存时，请根据自己的网速保持一定的耐心。未来GraalVM对

Spring Cloud的支持更成熟一些后，可以考虑采用GraalVM来改善这一点），在浏览器访问：<http://localhost>，系统预置了一个用户（user:icyfenix，pw:123456），也可以注册新用户来测试。

- 通过Skaffold在命令行或IDE中以调试方式运行：

这个运行方式与此前调试Kubernetes服务是完全一致的。在本地针对单个服务编码、调试完成后，通过CI/CD流水线部署到Kubernetes中进行集成的。如果只是针对集成测试，这并没有什么问题，但同样的做法应用在开发阶段就不十分不便了，我们不希望每做一处修改都要经过一次CI/CD流程，这将非常耗时且难以调试。

Skaffold是Google在2018年开源的一款加速应用在本地或远程Kubernetes集群中构建、推送、部署和调试的自动化命令行工具，对于Java应用来说，它可以帮助我们做到监视代码变动，自动打包出镜像，将镜像打上动态标签并更新部署到Kubernetes集群，为Java程序注入开放JDWP调试的参数，并根据Kubernetes的服务端口自动在本地生成端口转发。以上都是根据 `skaffold.yaml` 中的配置来进行的，开发时skaffold通过 `dev` 指令来执行这些配置，具体的操作过程如下所示：

```
# 克隆获取源码  
$ git clone https://github.com/fenixsoft/servicemesh_arch_istio.git  
&& cd servicemesh_arch_istio  
  
# 编译打包  
$ ./mvnw package  
  
# 启动Skaffold  
# 此时将会自动打包Docker镜像，并部署到Kubernetes中  
$ skaffold dev
```

服务全部启动后，在浏览器访问：<http://localhost>，系统预置了一个用户（user:icyfenix，pw:123456），也可以注册新用户来测试，注意这里开放和监听的端口同样取决于Istio Ingress Gateway，可能需要根据系统环境进行调整

- 调整代理自动注入

项目提供的资源文件中，默认是允许边车代理自动注入到Pod中的，这会导致服务需要有额外的容器初始化过程。开发期间，我们可能需要关闭自动注入以提升容器频繁改动、重新部署时的效率。如需关闭代理自动注入，请自行调整bookstore-kubernetes-ma

nifests目录下的bookstore-namespaces.yaml资源文件，根据需要将istio-injection修改为enable或者disable。

如果关闭了边车代理，意味着你的服务丧失了访问控制（以前是基于Spring Security实现的，在Istio版本中这些代码已经被移除）、断路器、服务网格可视化等一系列依靠Envoy代理所提供能力。但这些能力是纯技术的，与业务无关，并不影响业务功能正常使用，所以在本地开发、调试期间关闭代理是可以考虑的。

技术组件

Fenix's Bookstore采用基于Istio的服务网格架构，其中主要的技术组件包括：

- **配置中心**：通过Kubernetes的ConfigMap来管理。
- **服务发现**：通过Kubernetes的Service来管理，由于已经不再引入Spring Cloud Feign了，所以在OpenFeign中，直接使用短服务名进行访问。
- **负载均衡**：未注入边车代理时，依赖KubeDNS实现基础的负载均衡，一旦有了Envoy的支持，就可以配置丰富的代理规则和策略。
- **服务网关**：依靠Istio Ingress Gateway来实现，已经移除了Kubernetes版本中保留的Zuul网关。
- **服务容错**：依靠Envoy来实现，已经移除了Kubernetes版本中保留的Hystrix。
- **认证授权**：依靠Istio的安全机制来实现，实质上已经不再依赖Spring Security进行ACL控制，但Spring Security OAuth 2仍然以第三方JWT授权中心的角色存在，为系统提供终端用户认证，为服务网格提供令牌生成、公钥JWKS等支持。

协议

- 本作品代码部分采用Apache 2.0协议进行许可。遵循许可的前提下，你可以自由地对代码进行修改，再发布，可以将代码用作商业用途。但要求你：
 - **署名**：在原有代码和衍生代码中，保留原作者署名及代码来源信息。
 - **保留许可证**：在原有代码和衍生代码中，保留Apache 2.0协议文件。
- 本作品文档部分采用知识共享署名 4.0 国际许可协议进行许可。遵循许可的前提下，你可以自由地共享，包括在任何媒介上以任何形式复制、发行本作品，亦可以自由地演

绎、修改、转换或以本作品为基础进行二次创作。但要求你：

- **署名**：应在使用本文档的全部或部分内容时候，注明原作者及来源信息。
- **非商业性使用**：不得用于商业出版或其他任何带有商业性质的行为。如需商业使用，请联系作者。
- **相同方式共享的条件**：在本文档基础上演绎、修改的作品，应当继续以知识共享署名4.0国际许可协议进行许可。

无服务 : AWS Lambda



Release v1.0

License Apache 2.0

Doc License CC 4.0

Author IcyFenix

如果你此时并不曾了解过什么是“[The Fenix Project](#)”，建议先阅读[这部分内容](#)。

无服务架构 (Serverless) 与微服务架构本身没有继承替代关系，它们并不是同一种层次的架构，无服务的云函数可以作为微服务的一种实现方式，甚至可能是未来很主流的实现方式。在这部文档中我们的话题主要还是聚焦在如何解决分布式架构下的种种问题，相对而言无服务架构并非重点，不过为保证架构演进的完整性，笔者仍然建立了无服务架构的简单演示工程。

运行程序

Serverless架构的Fenix's Bookstore基于[亚马逊AWS Lambda](#)平台运行，这是最早商用，也是目前全球规模最大的Serverless运行平台。从2018年开始，中国的主流云服务厂商，如阿里云、腾讯云都推出了各自的Serverless云计算环境，如需在这些平台上运行Fenix's Bookstore，应根据平台提供的Java SDK对StreamLambdaHandler的代码进行少许调整。

假设你已经完成[AWS注册](#)、配置[AWS CLI环境](#)以及IAM账号的前提下，可通过以下几种途径，可以运行程序，浏览最终的效果：

- 通过AWS SAM (Serverless Application Model) Local在本地运行：
AWS CLI中附有SAM CLI，但是版本较旧，可通过[如下地址](#)安装最新版本的SAM CL

I。 另外 , SAM需要Docker运行环境支持 , 可参考[此处部署](#)。

首先编译应用出二进制包 , 执行以下标准Maven打包命令即可 :

```
$ mvn clean package
```

根据pom.xml中assembly-zip的设置 , 打包将不会生成SpringBoot Fat JAR , 而是产生适用于AWS Lambda的ZIP包。 打包后 , 确认已在target目录生成ZIP文件 , 且文件名称与代码中提供了sam.yaml中配置的一致 , 在工程根目录下运行如下命令启动本地SAM测试 :

```
$ sam local start-api --template sam.yaml
```

sh

在浏览器访问 : <http://localhost:3000> , 系统预置了一个用户 (user:icyfenix , pw:123456) , 也可以注册新用户来测试。

- 通过AWS Serverless CLI将本地ZIP包上传至云端运行 :

确认已配置AWS凭证后 , 工程中已经提供了serverless.yml配置文件 , 确认文件中ZIP的路径与实际Maven生成的一致 , 然后在命令行执行 :

```
$ sls deploy
```

sh

此时Serverless CLI会自动将ZIP文件上传至AWS S3 , 然后生成对应的Layers和API Gateway , 运行结果如下所示 :

```
$ sls deploy
Serverless: Packaging service...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Uploading service bookstore-serverless-awslambda-1.0-SNAPSHOT-lambda-package.zip file to S3 (53.58 MB)...
Serverless: Validating template...
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
.
.
.
Serverless: Stack update finished...
Service Information
```

```
service: spring-boot-serverless
stage: dev
region: us-east-1
stack: spring-boot-serverless-dev
resources: 10
api keys:
  None
endpoints:
  GET - https://cc1oj8hirl.execute-api.us-east-1.amazonaws.com/dev/
functions:
  springBootServerless: spring-boot-serverless-dev-
springBootServerless
layers:
  None
Serverless: Removing old service artifacts from S3...
```

访问输出结果中的地址（譬如上面显示的<https://cc1oj8hirl.execute-api.us-east-1.amazonaws.com/dev/>）即可浏览结果。

需要注意，由于Serverless对响应速度的要求本来就较高，与Java本身就运行方式就多少存在矛盾（笔者在GraalVM的“[向原生迈进](#)”一文有详细解释），所以不建议再采用HSQLDB数据库来运行程序了，每次冷启动都重置一次数据库本身也并不合理。代码中有提供MySQL的Schema，建议采用AWS RDB MySQL/MariaDB作为数据库来运行。

协议

- 本作品代码部分采用[Apache 2.0协议](#)进行许可。遵循许可的前提下，你可以自由地对代码进行修改，再发布，可以将代码用作商业用途。但要求你：
 - **署名**：在原有代码和衍生代码中，保留原作者署名及代码来源信息。
 - **保留许可证**：在原有代码和衍生代码中，保留Apache 2.0协议文件。
- 本作品文档部分采用[知识共享署名 4.0 国际许可协议](#)进行许可。遵循许可的前提下，你可以自由地共享，包括在任何媒介上以任何形式复制、发行本作品，亦可以自由地演绎、修改、转换或以本作品为基础进行二次创作。但要求你：
 - **署名**：应在使用本文档的全部或部分内容时候，注明原作者及来源信息。
 - **非商业性使用**：不得用于商业出版或其他任何带有商业性质的行为。如需商业使用，请联系作者。

- **相同方式共享的条件**：在本文档基础上演绎、修改的作品，应当继续以知识共享署名4.0国际许可协议进行许可。

服务架构演进史

服务架构的演进历史这一章，我们借讨论历史之名，来梳理微服务发展里程中出现的大量名词、概念，借着微服务的演变过程，我们将从这些概念起源的最初，去分析它们是什么、它们取代了什么、以及它们为什么能够在斗争中取得成功，为什么变得不可或缺的支撑，又或者它们为什么会失败，在竞争中被淘汰，或逐渐湮灭于历史的烟尘当中。

- **原始分布式时代**：使用多个独立的分布式服务共同构建一个更大型系统，尽可能促使服务交互透明与简单，令开发人员不必过份关注他们访问的方法或其他资源是位于本地还是远程。
- **单体系统时代**：“单体”只是表明系统中主要的过程调用都是进程内调用，不会发生进程间通讯，仅此而已。
- **SOA时代**：面向服务的架构是第一次系统性地成功解决分布式服务主要问题的架构模式。
- **微服务时代**：微服务是一种通过多个小型服务组合来构建单个应用的架构风格，这些服务围绕业务能力而非特定的技术标准来构建。各个服务可以采用不同的编程语言，不同的数据存储技术，运行在不同的进程之中。服务采取轻量级的通讯机制和自动化的部署机制实现通讯与运维。
- **后微服务时代**：从软件层面独力应对微服务架构问题，发展到软硬一体，合力应对架构问题的时代，此即为“后微服务时代”。
- **无服务时代**：如果说微服务架构是分布式系统这条路的极致，那无服务架构，也许就是“不分布式”的云端系统这条路的起点。

原始分布式时代

Unix的分布式设计哲学

Simplicity of both the interface and the implementation are more important than any other attributes of the system — including correctness, consistency, and completeness

保持接口与实现的简单性，比系统的任何其他属性，包括准确性、一致性和完整性，都来得更加重要。

—— Richard P. Gabriel [\[1\]](#) , The Rise of 'Worse is Better' [\[2\]](#) , 1991

可能与绝大多数人心中的认知会有差异，“使用多个独立的分布式服务共同构建一个更大型系统”的设想与实际尝试，反而要比今天大家所了解的大型单体系统出现的时间更早。

在20世纪的70年代末期到80年代初，计算机科学刚经历了从以大型机为主向以微型机为主的蜕变，计算机逐渐从一种存在于研究机构、实验室当中的科研设备，转变为存在于商业企业中的生产设备，或者是面向家庭、个人用户的娱乐设备。此时的微型计算机系统通常具有16位寻址能力、不足5MHz时钟频率的处理器和128KB左右的内存地址空间。譬如著名英特尔处理器的鼻祖，[Intel 8086处理器](#) [\[3\]](#)就是在1978年研制成功，流行于80年代中期，甚至一直持续到90年代初期仍有生产销售。

囿于当时计算机硬件局促的运算处理能力，已直接妨碍到了单台计算机上信息系统软件能够达到的最大规模。为突破硬件算力的限制，各个高校、研究机构、软硬件厂商开始分头探索使用多台计算机共同协作来支撑同一套软件系统运行的可行性。这阶段是对分布式架构最原始的探索与研究，但仅从技术角度来看，这个阶段的探索称得上的硕果累累，成绩斐然。提出的很多技术、概念对*nix系统后续的发展，乃至对今天计算机科学的诸多领域都产生了巨大而深远的影响，直接牵引了后续软件架构演化进程。譬如，惠普（及后来被惠普收购的Apollo）提出的[网络运算架构](#) [\[4\]](#)（Network Computing Architecture，NCA）是未来远程服务调用的雏形；卡内基·梅隆大学提出的[AFS文件系统](#) [\[5\]](#)是日后分布式文件系统

的最早实现（顺便一提，AFS中的A是Andrew的意思，意为纪念Andrew Carnegie和Andrew Mellon）；麻省理工学院提出的[Kerberos协议](#)是服务认证和访问控制（ACL）的基础性协议，是分布式服务安全性的重要支撑，目前仍被用于实现包括Windows和MacOS在内众多操作系统的登陆、认证功能，等等。

为了避免[Unix系统的版本战争](#)在分布式领域中重演，Unix系统标准化组织[开放软件基金会](#)（Open Software Foundation，OSF，也即后来的“国际开放标准组织”）邀请了各主要的研究厂商参与，共同制订了名为“[分布式运算环境](#)”（Distributed Computing Environment，DCE）的软件架构公约，DCE包括了一整套完整的分布式服务组件的规范与实现，譬如源自NCA的远程服务调用规范（Remote Procedure Call，RPC），当时被称为[DCE/RPC](#)，与后来Sun公司向互联网工程任务组（Internet Engineering Task Force，IETF）提交的基于通用TCP/IP协议的远程服务标准[ONC RPC](#)被认为是现代RPC的共同鼻祖；源自AFS的分布式文件系统（Distributed File System，DFS）规范，当时被称为[DCE/DFS](#)；源自Kerberos的服务认证规范；还有时间服务、命名与目录服务，以及当今程序中很常用的UUID也是在DCE中定义的。

由于OSF本身的Unix背景，当时研究这些分布式技术，通常有一个预设的重要原则是实现分布式环境中的服务调用、资源访问、数据存储等操作尽可能的透明化、简单化，使开发人员不必过于关注他们访问的方法或其他资源是位于本地还是远程。这样的主旨非常符合一贯的[Unix设计哲学](#)（有过几个版本的不同说法，这里指的是Common Lisp作者Richard P. Gabriel提出的简单优先“Worse is Better”原则），但这个过于理想化的目标背后其实蕴含着彼时根本不可能完美解决的技术困难。“调用远程方法”与“调用本地方法”尽管只是两字之差，但若要同时兼顾到简单、透明、性能、正确、鲁棒、一致的话，两者的复杂度就完全不可同日而语。且不说远程方法无法去做本地方法那些以内联为代表的传统编译优化来提升速度，光是“远程”二字带来的网络环境下的新问题，如远程的服务在哪里（服务发现）、有多少个（负载均衡）、网络出现分区、超时或者服务出错了怎么办（熔断、隔离、降级）、方法的参数与返回结果如何表示（序列化协议）、如何传输（传输协议）、服务权限如何管理（认证、授权）、如何保证通信安全（网络安全层）、如何令调用不同机器的服务能返回相同的结果（分布式数据一致性）等一系列问题就需要设计者耗费大量心思。

面对重重困难与压力，DCE不仅从零开始、从无到有全部解决了这些问题，构建出大量的分布式基础组件与协议，而且还真的尽力去做到了相对意义的“透明”，譬如你在DFS上访问文件，如果不考虑性能上的差异的话，就很难感受到它与本地磁盘文件系统有什么不

同。可是，一旦考虑性能上的差异，分布式和本地的鸿沟是无比深刻的，这是数量级上的差距，是不可调和的差距。尤其是在那个年代的机器硬件限制下，开发者为了让程序在运行效率上可以接受，不得不局限仅在方法本身运行时间很长，可以相对忽略远程调用成本时的情况下才考虑分布式，如果方法本身运行时长不够，就要人为用各种奇技淫巧地刻意构造出这样的场景，譬如将几个原本毫无关系的方法打包到一个方法内，一块进行远程调用。这一方面本身就与使用分布式来突破硬件算力，提升性能的初衷相互矛盾，需要小心平衡；另一方面，此时的开发人员实际上仍然必须每时每刻都意识到自己是在编写分布式的程序，不可轻易踏过本地与远程的界限，设计向性能做出的妥协，令DCE“尽量简单透明”的努力几乎全部付诸东流，本地与远程无论是编码、运行还是效率角度上看，都有着天壤之别，设计一个能运作良好的分布式应用，变得需要极高的编程技巧和各方面的知识去支撑，这时候反而是人员本身对软件规模的约束，超过机器算力上的约束了。

对DCE的研究是计算机科学中第一次有组织领导、有标准可循、有巨大投入的分布式计算的尝试，但无论是DCE还是稍后出现的CORBA，从结果来看，都不能称取得了成功，将一个系统直接拆分到不同的机器之中，这样做带来的服务的发现、跟踪、通讯、容错、隔离、配置、传输、数据一致性和编码复杂度等方面的问题，所付出的代价远远超过了分布式所取得的收益。亲身经历过那个年代的计算机科学家、IBM院士Kyle Brown事后曾经评价道：“这次尝试最大的收获就是对RPC、DFS等概念的开创，以及得到了一个价值千金的教训：**某个功能能够进行分布式，并不意味着它就应该进行分布式，强行追求透明的分布式操作，只会自寻苦果**”。

原始分布式时代的教训

Just because something **can** be distributed doesn't mean it **should** be distributed. Trying to make a distributed call act like a local call always ends in tears

某个功能能够进行分布式，并不意味着它就应该进行分布式，强行追求透明的分布式操作，只会自寻苦果

—— [Kyle Brown](#)，IBM Fellow，[Beyond Buzzwords: A Brief History of Microservices Patterns](#)，2016

以上结论是有违Unix设计哲学的，但也是当时现实情况下不得不做出的让步。摆在计算机科学面前有两条通往更大规模软件系统的道路，一条是尽快提升单机的处理能力，以避免分布式带来的种种问题；另一条路是找到更完美的解决如何构筑分布式系统的方案。

上世纪80年代正是摩尔定律开始稳定发挥作用的黄金时期，微型计算机的性能以每两年即增长一倍的惊人速度提升，硬件算力束缚软件规模的链条很快变得松动，信息系统进入了以单台或少数几台计算机即可作为服务器来支撑大型信息系统运作的单体时代，且在很长的一段时间内，单体系统都将是软件架构的主流。尽管如此，对于另外一条路径，即对分布式计算、远程服务调用的探索却也从未有过中断。关于远程服务调用这个关键问题的历史、发展与现状，笔者还会在服务设计风格的“远程服务调用”部分，以现代RPC和RESTful为主角来进行更详细的讲述。而对于在原始分布式时代中遭遇到的其他问题，也还将会在软件架构演进后面几个时代里被反复提起。

原始分布式时代提出的构建“符合Unix的设计哲学的”、“如同本地调用一般简单透明的”分布式系统这个目标，是软件开发者对分布式系统最初的美好愿景，迫于现实，它会在一定时期内被妥协、被舍弃，分布式将会经过一段越来越复杂的发展进程。但是，到了三十多年以后的未来，随着微服务的逐渐成熟完善，成为大型软件的主流架构风格以后，这个美好的愿景终将还是会重新被开发者拾起。

单体系统时代

单体架构 (Monolithic)

“单体”只是表明系统中主要的过程调用都是进程内调用，不会发生进程间通讯，仅此而已。

单体架构是今天绝大部分软件开发者都学习、实践过的一种软件架构，许多介绍微服务的书籍和技术资料中也常把这种架构形式的应用称作“[巨石系统](#)” (Monolithic Application)。“单体架构”在整个软件架构演进的历史进程里，是出现的时间最早、应用的范围最广、使用人数最多、统治的历史最长的一种架构风格，但“单体”这个名称，却是要到微服务开始流行之后才“事后追认”所形成的概念，此前，并没有多少人将“单体”视作为一种架构来看待，如果你去查找软件架构的开发资料，可以轻而易举地找出大量以微服务为主题的书籍和文章，却很难找出专门教你如何开发单体系统的任何形式的材料，这一方面体现了单体架构本身的简单性，另一方面，也体现出相当长的时间尺度里，大家都已经习惯了软件架构就应该是单体这种样子。

剖析单体架构之前，我们有必要先厘清一个概念误区，许多微服务的资料里，单体系统往往是以“反派角色”的身份登场的，譬如著名的微服务入门书《[微服务架构设计模式](#)》，第一章的名字就是“逃离单体的地狱”。这些材料所讲的单体系统，其实都是有一个没有明说的隐含定语：“**大型的单体系统**”。对于小型系统——即由单台机器就足以支撑其良好运行的系统，单体不仅易于开发、易于测试、易于部署，且由于系统中各个功能、模块、方法的调用过程都是进程内调用，不会发生[进程间通讯](#) (Inter-Process Communication , IPC)。RPC属于IPC的一种特例，但请注意这里两个“PC”不是同个单词的缩写)，因此连运行效率也是最高的一种架构风格，完全不应该被贴上“反派角色”的标签，反倒是那些爱赶技术潮流却不顾需求现状的微服务吹捧者更像是个反派。单体系统的缺陷，必须基于软件的性能需求超过了单机，软件的开发人员规模明显超过了“[2 Pizza Teams](#)”范畴的前提下才有讨论的价值，因此，本文后续讨论中所说的单体，均应该是特指“大型的单体系统”，也正因如此，本节中说到“单体是出现最早的架构风格”，与上一节介绍原始分布式时代开篇

提到的“使用多个独立的分布式服务共同构建一个更大型系统的设想与实际尝试，反而要比今天大家所了解的大型单体系统出现的时间更早”实际并无矛盾。

单体系统

Monolith means composed all in one piece. The Monolithic application describes a single-tiered software application in which different components combined into a single program from a single platform.

—— [Monolithic Application](#) , Wikipedia

尽管“Monolithic”这个词语本身的意思“巨石”确实是带有一些“不可拆分”的隐含意味，但我们也不能简单粗暴地把单体系统在维基百科上的定义“All in One Piece”翻译成“铁板一块”，它其实更接近于自给自足（Self-Contained）的含义。这种“铁板一块”的译法不能全算作是段子，笔者相信肯定有一部分人说起单体架构、巨石系统的缺点，第一个在脑海中闪过的印象就是它的“不可拆分”，难以扩展，因此才不能支撑越来越大的软件规模。这种想法是有失偏颇的，至少不完整。从纵向角度来看，现代信息系统中，笔者从未见过实际生产环境里的哪个大型的系统是完全不分层的。分层架构（Layered Architecture）已是现在几乎所有信息系统建设中都普遍认可、普遍采用的软件设计方法，无论是单体还是微服务，抑或是其他架构风格，都会对代码进行纵向拆分，收到的外部的请求在各层之间以不同形式的数据结构进行流转传递，触及最末端的数据库后依次返回响应。在这个意义上的“可拆分”，单体架构完全不会展露出丝毫的弱势，反而还可能会因更容易开发、部署、测试而获得一些便捷性上的好处。



分层架构示意

图片来自O'Reilly的开放文档《Software Architecture Patterns》

而在横向角度的“可拆分”上，单体架构也可以支持按照技术、功能、职责等角度，将软件拆分为各种模块，以便重用和团队管理。单体系统并不意味着就只能有一个整体的程序封装形式，如果需要，它完全可以由多个JAR、WAR、DLL、Assembly或者其他模块格式来构成。即使是以横向扩展（Scale Horizontally）的角度来衡量，在负载均衡器之后同时部署若干个单体系统的副本，以达到分摊流量压力的效果也是轻而易举可以实现的需求。

在“拆分”这方面，单体系统的真正缺陷不在如何拆分，而在是拆分之后的隔离与自治能力上的欠缺。由于所有代码都运行在同一个进程空间之内，所有模块、方法的调用都无需考虑网络分区、对象复制这些麻烦事和性能的损失。获得了进程内调用的简单、高效这些好处的同时，也意味着如果任何一部分代码出现了缺陷，过度消耗能进程空间内的公共资源，所造成的影响也是全局性的，难以隔离的。譬如出现了内存泄漏、线程爆炸、阻塞、死循环、端口占用过多等问题，都将会影响整个程序而不仅仅是某一个功能、模块本身的正常运作，如果消耗的是某些更高层次的公共资源，譬如数据库连接池泄漏，影响还将会波及到集群中其他横向扩展的单体副本的正常工作。

同样是由于所有代码都共享着同一个进程空间，代码无法隔离，也就无法（其实还是有办法，譬如使用OSGi这种运行时模块化框架，就是很别扭很复杂）做到单独停止、更新、升级某一部分代码，因为不可能有“停掉半个进程，重启1/4个进程”这样不合逻辑的操作，所以从动态可维护性来说，单体系统也是有所不足的，程序升级、修改缺陷往往需要制定专门的停机更新计划，做灰度发布也相对更加复杂。

如果说共享同一进程获得简单、高效的代价是同时损失了各个功能模块的自治、隔离能力，那这两者孰轻孰重呢？这个问题的潜台词其实是比较微服务、单体架构哪种更先进、优秀？笔者认为“先进和优秀”不可能是绝对的，这点可以举一个非常浅显的例子加以说明。譬如，沃尔玛将超市分为仓储部、采购部、安保部、库存管理部、巡检部、质量管理部门、市场营销部，等等，可以划清职责，明确边界，让管理能力能支持企业的成长规模；但如果你家楼下开的小卖部，爸、妈加儿子，再算上看家的中华田园犬小黄一共也就只有四名员工，也去追求“先进管理”，划分仓储部、采购部、库存管理部……那纯粹是给自己找麻烦。单体架构下，哪怕是信息系统中两个相互毫无关联的子系统，也必须部署到一起。当系统规模小时这是优势，但系统规模大的时候、程序需要修改时候的部署成本、技术升级时的迁移成本都会变得高昂。按前面的例子来说，就是当公司小时，让安保部和质检部两个不相干的部门在同一栋大楼中办公是节约资源，但当公司人数增加，办公室已经拥挤不堪，也最多只能在楼顶加盖新楼层（相当于增强硬件性能），而不能让安保、质检分开地方办公，这才是缺陷所在。

由于隔离能力的缺失，除了难以阻断错误传播、不便于动态更新程序以外，还会带来难以技术异构（每个模块的代码都通常需要使用一样的程序语言，一样的编程框架。单体系统的技术栈异构不是一定做不到，譬如JNI就可以让Java混用C/CPP，但是这也是很麻烦的事，是迫不得已下的选择）等困难。不过，笔者看来，以上列举的这些问题都还不足以构成今天以微服务去代替单体系统成为潮流趋势的根本原因，笔者认为最根本的一点是：单体系统并不兼容“Phoenix”的特性。这种架构风格潜在的观念是希望系统的每一个部件，甚至每一处代码都尽量可靠，不出、少出缺陷，致力于构筑一个7×24小时不间断的可靠系统。这种观念在小规模软件上能运作良好，但系统越大，交付一个可靠的单体系统就变得越来越具有挑战性。如本文档的开篇语《什么是The Fenix Project》所说，正是随着软件架构演进，构筑可靠系统从“追求尽量不出错”，到正视“出错是必然”的观念转变，才是微服务架构得以挑战并逐步开始代替运作了数十年单体架构的根本驱动力所在。

不过，即使是为了允许程序出错，为了获得隔离、自治的能力，为了可以技术异构等目标，也并不意味着一定要依靠今天的微服务架构才能解决。在新旧世纪之交，人们曾经探

索过几种服务的拆分方法，将一个大的单体系统拆分为若干个更小的、不运行在同一个进程的独立服务，这些服务拆分的方法后来导致了面向服务架构（Service-Oriented Architecture）的一段兴盛期，我们称其为“[SOA时代](#)”。

SOA时代

SOA架构（Service-Oriented Architecture）

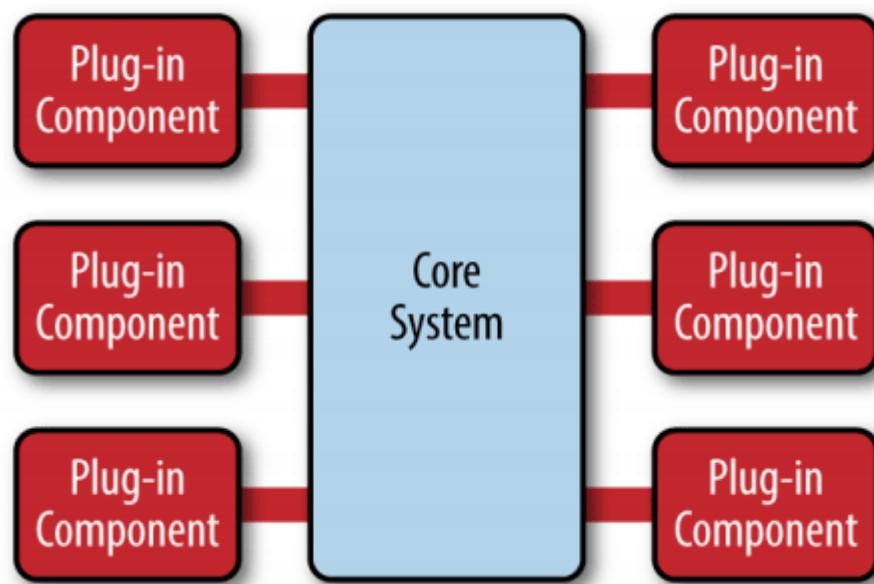
面向服务的架构是一次具体地、系统性地成功解决分布式服务主要问题的架构模式。

为了对大型的单体系统进行拆分，让每一个子系统都能独立地部署、运行、更新，开发者们曾经尝试过多种方案，笔者列举以下三种较有代表性的架构模式，分别为：

- **烟囱式架构**（Information Silo Architecture）：信息烟囱又名信息孤岛（Information Island），使用这种架构的系统也被称为孤岛式信息系统或者烟囱式信息系统。它指的是一种完全不与其他相关信息系统之间进行互操作或者说协调工作的信息系统。这样的系统其实并没有什么“架构设计”可言，承接着上一节中企业与部门的例子来说，如果两个部门真的完全不会发生任何交互，就并没有什么理由强迫把它们必须在一栋楼里办公；两个不发生交互的信息系统，让他它们使用独立的数据库、服务器即可完成拆分，而唯一的问题，也是致命的问题是，企业中真的存在完全不发生交互的部门？对于两个信息系统来说，哪怕真的毫无业务往来关系，但系统的人员、组织、权限等等主数据，会是完全独立、没有任何重叠的吗？这样“独立拆分”、“老死不相往来”的系统，显然不可能是企业所希望见到的。
- **微内核架构**（Microkernel Architecture）：微内核架构也被称为插件式架构（Plug-in Architecture）。既然烟囱式架构中，两个没有业务往来关系的系统也可能需要共享人员、组织、权限等一些的公共的主数据，那不妨就将这些主数据，连同其他可能被各子系统使用到的公共服务、数据、资源集中到一块，成为一个被所有业务系统共同依赖的核心系统（Kernel，也称为Core System），具体的业务系统就以插件模块（Plug-in Modules）的形式存在，这样便可提供可扩展的，灵活的，天然隔离的功能特性。
这种模式很适合桌面应用程序，也经常在Web应用程序中使用。以更高层抽象来看，任何计算机系统都是由各种架构的软件互相配合工作来实现各种功能的，本文列举的各种不同的架构模式一般都可视作整个系统的一种插件。对于产品型应用程序来说，如果我

们想将新特性或者功能及时加入系统，微内核架构会是一种不错的选择。微内核的架构也可以嵌入到其它的架构模式之中，通过插件的方式来提供逐步演化的功能和增量开发。所以如果你准备实现一个能够支持二次开发的软件系统，微内核是一种良好的架构模式。

不过，微内核架构也有它的局限和使用前提，它假设系统中各个插件模块之间是互不认识（不可预知系统会安装哪些模块），这些插件会访问内核中一些公共的资源，但不会发生直接交互。可是，无论是在企业信息系统还是互联网，这一前提假设在许多场景中都并不成立，我们必须找到办法，既能拆分出独立的系统，也能让拆分后的子系统之间可以顺畅地互相调用通讯。



微内核架构示意

图片来自O'Reilly的开放文档《Software Architecture Patterns》

- **事件驱动架构** (Event-Driven Architecture)：为了能让子系统互相通讯，一种可行的方案是在子系统之间建立一套事件队列管道 (Event Queues)，来自系统外部的消息将以事件的形式发送至管道中，各个子系统可以从管道里获取自己感兴趣、可以处理的事件消息，可以为事件新增或者修改其中的附加信息，甚至可以自己发布一些新的事件到管道队列中去，如此，每一个消息的处理器都是独立的，高度解耦的，但又能与其他处理器（如果存在该消息处理器的话）通过事件管道进行互动。



事件驱动架构示意

图片来自O'Reilly的开放文档《Software Architecture Patterns》

当系统演化至事件驱动架构时，原始分布式时代结尾中提到的第二条通往大规模软件的路径，即仍在并行发展的远程服务调用也迎来了SOAP协议的诞生（详见[远程服务调用](#)一文），此时“面向服务的架构”（Service Oriented Architecture，SOA）已经有了它登上软件架构舞台所需要的全部前置条件。SOA的概念最早由Gartner公司在1994年提出，2006年，由IBM、Oracle、SAP等公司共同成立了OSOA联盟（Open Service Oriented Architecture），用于联合制定和推进SOA相关行业标准。2007年，在[结构化资讯标准促进组织](#)（Organization for the Advancement of Structured Information Standards，OASIS）倡议与支持下，OSOA由一个软件厂商组成的松散联盟，转变为一个制定行业标准的国际组织，联合OASIS共同新成立了的[Open CSA](#)组织（Open Composite Services Architecture），这便是SOA的“官方管理机构”。

当软件架构发展至SOA时代，其中的许多概念、思想都已经能在今天微服务中找到对应的身影了。服务之间的松散耦合、注册、发现、治理，隔离、编排，等等。这些今天微服务中耳熟能详的名词概念，大多数也是在分布式服务刚被提出时就已经可以预见到的困难。SOA针对这些问题，乃至于针对“软件开发”这件事情本身，进行了更加系统性、更加具体的探索。

- “更具体”体现在尽管SOA本身还是属抽象概念，而不是特指某一种具体的技术，但它比单体架构和前面所说的三种架构模式都要更具可操作性、细节充实了很多，已经不能简单视其为一种架构风格，可以称为是一套软件架构的基础平台了。它拥有领导制定技术标准的组织Open CSA；有清晰软件设计的指导原则，譬如服务的封装性、自治、松耦合、可重用、可组合、无状态，等等；明确了采用SOAP作为远程调用的协议，依靠SOAP协议族（WSDL、UDDI和一大票WS-*协议）来完成服务的发布、发现和治理；利用一个被称为企业服务总线（Enterprise Service Bus，ESB）的消息管道来实现各个子系统之间的通讯交互，令各服务间在ESB调度下无需相互依赖却能相互通讯，既带来了服务松耦合的好处，也为以后可以进一步实现业务流程编排（Business Process Management，BPM）提供了基础；使用服务数据对象（Service Data Object，SDO）来访问和表示数据，使用服务组件架构（Service Component Architecture，SCA）来定义服务封装的形式和服务运行的容器，等等。在这一整套成体系可以互相精密协作的技术组件支持下，从技术可行性这一个角度来评判的话，SOA可以算是成功地解决了分布式环境下出现的主要技术问题。
- “更系统”所指的是SOA的宏大理想，它最根本的目标是希望总结出一套自上而下的软件研发方法论，希望做到企业只需要跟着SOA的思路，就能够一揽子解决掉诸如如何挖掘需求、如何将需求分解为服务能力、如何编排已有服务、如何开发测试部署新的功能等软件开发过程中的全套问题，这里面技术确实是重点难点，但也仅占其中的一环。如果这个目标真的能够达成，软件开发就有可能从此迈进工业化大生产的阶段，试想如果有一天写出符合客户需求的软件会像写八股文一样有迹可循、有法可依，那对软件开发者来说也许是无趣的，但整个社会实施信息化的效率肯定会有大幅的提升。

SOA在21世纪最初的十年里曾经盛行一时，有IBM等一众行业巨头厂商为其呐喊冲锋，吸引了不少软件开发商、尤其是企业级软件的开发商的跟随，最终却还是偃旗息鼓，沉寂了下去。笔者曾在[远程服务调用](#)一文中提到SOAP协议被逐渐边缘化的本质原因：过于严格的规范定义带来过度的复杂性。而构建在SOAP基础之上的ESB、BPM、SCA、SDO等諸多上层建筑，进一步加剧了这种复杂性。开发信息系统毕竟不是作八股文章，过于精密的流程和理论也需要懂得复杂概念的专业人员才能够驾驭，SOA诞生的那一天起，就已经注定了它只能是少数系统阳春白雪式的精致奢侈品，它可以实现多个异构大型系统之间的复杂集成交互，却很难作为一种具有广泛普适性的软件架构风格来推广。SOA最终没有获得成功的致命伤与当年的EJB如出一辙，尽管有Sun Microsystems和IBM等一众巨头在背

后力挺，EJB仍然败于以Spring、Hibernate为代表的“草根框架”，可见一旦脱离人民群众，终究会淹没在群众的海洋之中，连信息技术也不曾例外过。

当你读到这一段的时候，不妨重新翻到开头，回头想一想“如何使用多个独立的分布式服务共同构建一个更大型系统”这个问题，再回顾下“原始分布式时代”一节中Unix DCE提出的分布式服务的主旨：“让开发人员不必关心服务是远程还是本地，都能够透明地调用服务或者访问资源”。经过了三十年的技术进步，信息系统经历了巨石、烟囱、微内核、事件驱动、SOA等等的架构模式，应用受架构复杂度的牵绊却是越来越大，已经距离“透明”二字越来越远了，这是否算不自觉间忘记掉了当年的初心？接下来我们所谈论的微服务时代，似乎正是带着这样的自省式的问句而开启的。

微服务时代

微服务架构（Microservices）

微服务是一种通过多个小型服务组合来构建单个应用的架构风格，这些服务围绕业务能力而非特定的技术标准来构建。各个服务可以采用不同的编程语言，不同的数据存储技术，运行在不同的进程之中。服务采取轻量级的通讯机制和自动化的部署机制实现通讯与运维。

“微服务”这个技术名词最早在2005年就已经被提出，它是由Peter Rodgers博士在2005年度的云计算博览会（Web Services Edge 2005）上首次使用，当时的说法是“Micro-Web-Service”，指的是一种专注于单一职责的、语言无关的、细粒度Web服务（Granular Web Services）。“微服务”一词并不是Peter Rodgers直接凭空创造出来的概念，初生的微服务可以说是SOA发展时催生的产物，就如同EJB推广过程中催生了Spring和Hibernate那样。这一阶段的微服务是作为一种SOA的轻量化的补救方案而被提出的。时至今日，在英文版的维基百科上，仍然将微服务定义为一种SOA的变种形式，所以微服务在最初阶段与SOA、Web Service这些概念有所牵扯也完全可以理解，但现在来看，维基百科对微服务的定义已经颇有些过时了。

What is microservices

Microservices is a software development technique — a variant of the service-oriented architecture (SOA) structural style.

—— Wikipedia , [Microservices](#) ↗

微服务的概念提出后，将近10年的时间里面，都并没有受到太多的追捧，如果只是对现有SOA架构的修修补补，确实是难以唤起广大技术人员的更多激情了。不过，在这10年时间里，微服务本身也在思考、蜕变。2012年，在波兰克拉科夫举行的“33rd Degree Conference”大会上，Thoughtworks首席咨询师James Lewis做了题为《Microservices - Java, the U

nix Way》的主题演讲，其中提到了单一服务职责、康威定律、自动扩展、领域驱动设计等原则，却只字未提SOA，反而提倡应该重拾Unix的设计哲学（As Well Behaved Unix Services），这点仿佛与笔者在前一节所说的“初心与自省”在遥相呼应。微服务已经迫不及待地要脱离SOA的附庸，成为一种独立的架构风格，也许，还将会是SOA的革命者。

微服务真正的崛起是在2014年，相信阅读此文的大多数读者，也是从Martin Fowler与James Lewis合写的文章《Microservices: a definition of this new architectural term》中首次了解到微服务的，并不是指各位一定读过这篇文章，或者准确地说，今天各位所了解的“微服务”是这篇文章中提出的“微服务”。在此文中，定义了现代微服务的概念：“**微服务是一种通过多个小型服务组合来构建单个应用的架构风格，这些服务围绕业务能力而非特定的技术标准来构建。各个服务可以采用不同的编程语言，不同的数据存储技术，运行在不同的进程之中。服务采取轻量级的通讯机制和自动化的部署机制实现通讯与运维**”。此外，文中列举出了微服务的九个核心的业务与技术特征，笔者将其一一列出并解读如下：

- **围绕业务能力构建**（Organized around Business Capabilities），这里再次强调了康威定律的重要性，有怎样结构、规模、能力的团队，就会产生出对应结构、规模、能力的产品。这个结论不是某个团队、某个公司遇到的巧合，而是必然的演化结果。如果本应归属同一个产品内的功能被划分在不同团队中，必然就会产生大量的跨团队沟通协作，跨越团队边界无论在管理、沟通、工作安排上都有更高昂的成本，高效的团队自然会针对其进行改进，当团队、产品磨合调节稳定之后，团队与产品就会拥有一致的结构。
- **分散治理**（Decentralized Governance），这是要表达“谁家孩子谁来管”的意思，服务对应的开发团队有直接对服务运行质量负责的责任，也应该有着不受外界干预地掌控服务各个方面权力，譬如选择与其他服务异构的技术来实现自己的服务。这一点在真正实践时多少存有宽松的处理余地，大多数的公司都不会在某一个服务用Java，另一个用Python，下一个用Golang，通常都会统一主流语言，乃至有统一的技术栈或专有的技术平台。微服务不提倡也并不反对这种“统一”，只要负责提供和维护基础技术栈的团队，有被各方依赖的觉悟，要有“经常被凌晨3点的闹钟吵醒”的心理准备就好。微服务更加强调的是确实有必要技术异构时，应能够有选择“不统一”的权利，譬如不应该强迫Node.js去开发报表页面，要做人工智能计算时，应该可以选择Python，等等。
- **通过服务来实现独立自治的组件**（Componentization via Services），之所以强调通过“服务”（Service）而不是“类库”（Library）来构建组件，是指类库在编译期静态链接到程序中的，通过本地调用来提供功能，而服务是进程外组件，通过远程调用来提供功

能。前面的文章里我们已经分析过，尽管远程服务有更高昂的调用成本，但这是为组件带来隔离与自治能力的必要代价。

- **产品化思维** (Products not Projects)，避免把软件研发视作要去完成某种功能，而是视作一种持续改进、提升的过程。譬如，不应该把运维看作就是运维团队的事，把开发看作就是开发团队的事，团队应该为软件产品的整个生命周期负责，开发者不仅应该知道软件如何开发，还应该知道它如何运作，用户如何反馈，乃至售后支持工作是怎样进行的，这里服务的用户不一定是最终用户，也可能是消费这个服务的另外一个服务。以前单体架构下，程序的规模决定了无法让全部人员都关注完整的产品，组织中会有开发、运维、支持等细致的分工的成员只关注于自己的一块工作，但在微服务下，希望团队中每个人都具有产品化思维是可取的。
- **数据去中心化** (Decentralized Data Management)，微服务明确地提倡数据应该按领域分散管理、更新、维护、存储，单体服务中通常一个系统的各个功能模块会使用同一个数据库，诚然中心化的存储天生就更容易避免一致性问题，但是，同一个数据实体在不同服务的视角里，它的抽象形态往往也是不同的。譬如，Bookstore应用中的书本，在销售领域的中关注的是价格，在仓储领域中关注的库存数量，在商品展示领域中关注的是书籍的介绍信息，如果作为中心化的存储，所有这里领域都必须修改和映射到同一个实体之中，这便使得不同的服务可能会互相产生影响而丧失掉独立性。尽管在分布式中要处理好一致性的问题也很困难，很多时候都没法使用传统的事务处理来保证，但是两害相权取其轻，有一些必要的代价是值得付出的。
- **强终端弱管道** (Smart Endpoints and Dumb Pipes)，弱管道 (Dumb Pipes) 几乎算是直接指名道姓地反对SOAP和ESB的那一堆复杂的通讯机制，ESB可以处理消息的编码加工、业务规则转换等；BPM可以集中编排企业业务服务；SOAP有几十个WS-*协议族在处理了事务、一致性、认证授权等一系列工作，这些构筑在通讯管道上的功能也许有某个系统中有一部分服务是需要的，但对于另外更多的服务则是强加进来的负担。如果服务需要上面的某一种功能或能力，应该在服务自己的Endpoint上解决，而不是在通讯管道上一揽子处理。微服务提倡类似于经典Unix过滤器那样简单直接的通讯方式，RESTful风格的通讯在微服务中是比较适合的选择。
- **容错性设计** (Design for Failure)，不再虚幻地追求服务永远稳定，而是接受服务总会出错的现实，要求在微服务的设计中，有自动的机制对其依赖的服务能够进行快速故障检测，在持续出错的时候进行隔离，在服务恢复的时候重新联通。所以“断路器”这类设施，对实际生产环境的微服务来说并不是可选的外围组件，而是一个必须的支撑点，如果没有容错性的设计，系统很容易就会被因为一两个服务的崩溃所带来的雪崩效应淹

没。可靠系统完全可由会出错的服务组成，这是微服务最大的价值所在，也是这部开源文档标题“The Fenix Project”的含义。

- **演进式设计**（Evolutionary Design），容错性设计承认服务会出错，演进式设计则是承认服务会被报废淘汰。一个良好设计的服务，应该是能够报废的，而不是期望得到长久的发展，如果一个系统中出现不可更改、无可替代的服务，这并不能说明这个服务是多么的重要，反而是一种系统设计上脆弱的表现，微服务带来的独立、自治，也是在反对这种脆弱性的表现。
- **基础设施自动化**（Infrastructure Automation）：基础设施自动化，如CI/CD的长足发展，显著减少了构建、发布、运维工作的复杂性。由于运维的服务数量比起单体架构要有数量级的增长，使用微服务的团队更加依赖于基础设施的自动化，人工是无法运维成百上千乃至成千上万级别的服务的。

《Microservices》一文中对微服务特征的描写已经相当具体了，此文中除了定义的微服务是什么，还专门申明了微服务不是什么——微服务不是SOA的变体或衍生品，应该明确地与SOA划清了界线，不再贴上任何SOA的标签。如此，微服务的概念才算是一种真正丰满、独立、具体的架构风格，为它在未来的几年时间里如明星一般闪耀崛起于技术舞台铺下了理论基础。

Microservices and SOA

This common manifestation of SOA has led some microservice advocates to reject the SOA label entirely, although others consider microservices to be one form of SOA, perhaps service orientation done right. Either way, the fact that SOA means such different things means it's valuable to have a term that more crisply defines this architectural style

由于与SOA具有一致的表现形式，这让微服务的支持者更加迫切地拒绝再被打上SOA的标签，尽管有一些人坚持认为微服务就是SOA的一种变体形式，也许从面向服务方面这个方面来说是对的，但无论如何，SOA与微服务都是两种不同的东西，正因如此，使用一个别的名称来简明地定义这种架构风格就显得更有必要。

—— Martin Fowler / James Lewis , [Microservices](#) ↗

从以上微服务的定义和特征中还可以明显地感觉到，微服务追求的是更加自由的架构风格，摒弃了几乎所有SOA中可以抛弃的约束和规定，提倡以“实践标准”代替“规范标准”。可是，如果没有了统一的规范和约束，以前SOA所解决的那些分布式服务的问题，不也就一

下子都重新都出现了吗？的确如此，服务的注册发现、跟踪治理、负载均衡、故障隔离、认证授权、伸缩扩展、传输通讯、事务处理，等等，这些问题，微服务中不再会有统一的解决方案，即使只讨论Java范围内会使用到的微服务，光一个服务间通讯问题，可以列入解决方案的候选清单的就有：RMI（Sun/Oracle）、Thrift（Facebook）、Dubbo（阿里巴巴）、gRPC（Google）、Motan2（新浪）、Finagle（Twitter）、brpc（百度）、Arvo（Hadoop）、JSON-RPC、REST，等等；光一个服务发现问题，可以选择的就有：Eureka（Netflix）、Consul（HashiCorp）、Nacos（阿里巴巴）、ZooKeeper（Apache）、Etcd（CoreOS）、CoreDNS（CNCF），等等。其他领域的情况也是与此类似，总之，完全是八仙过海，各显神通的局面。

微服务所带来的自由是一把双刃开锋的宝剑，当软件架构者拿起这把宝剑，一刀指向SOA定下的复杂技术标准，将选择的权力夺回的同一时刻，另外一刀也正朝向着自己映出冷冷的寒光。微服务时代中，软件研发本身的复杂度应该说是有所降低，一个简单服务，并不见得就会同时面临分布式中所有的问题，也就没有必要背上SOA那百宝袋般沉重的技术包袱。需要解决什么问题，就引入什么工具；团队熟悉什么技术，就使用什么框架。此外，像Spring Cloud这样的胶水式的全家桶工具集，通过一致的接口、声明和配置，进一步屏蔽了源自于具体工具、框架的复杂性，降低了在不同工具、框架之间切换的成本，所以，作为一个普通的服务开发者，作为一个“螺丝钉”式的程序员，微服务架构是友善的。可是，微服务对架构者是满满的恶意，对架构能力要求已提升到史无前例的程度，笔者在这部文档的多处反复强调过，技术架构者的第一职责就是做决策权衡，有利有弊才需要决策，有取有舍才需要权衡，如果架构者本身的知识面不足以覆盖所需要决策的内容，不清楚其中利弊，恐怕也就无可避免地陷入选择困难症的困境之中。

微服务时代充满着自由的气息，微服务时代充斥着迷茫的选择。软件架构不会止步于自由，微服务仍不是架构探索终点，如果有下一个时代，我希望是信息系统能同时拥有微服务的自由权利，围绕业务能力构建自己的服务而不受技术规范管束，但同时又不必以承担自行解决分布式的问题的责任为代价。管他什么利弊权衡！小孩子才做选择题，成年人全部都要！

后微服务时代

后微服务时代（Cloud Native）

从软件层面独力应对微服务架构问题，发展到软、硬一体，合力应对架构问题的时代，此即为“后微服务时代”。

在微服务架构中，有一些必须解决的问题，比如注册发现、跟踪治理、负载均衡、传输通讯等。这些问题其实在SOA时代甚至可以说自从原始分布式时代起就一直存在了。既然只要是分布式架构的系统，就无法完全避免这些问题，那我们不妨换个思路来想一下：这些问题一定要由分布式系统自己来解决吗？

我们先不纠结于微服务或者什么别的架构，直接来看待这些问题与它们最常见的解决方法。如果某个系统需要伸缩扩容，通常会购买新的服务器，多部署几套副本实例；如果某个系统需要解决负载均衡问题，通常会布置负载均衡器，选择恰当的均衡算法来分流；如果需要解决传输安全问题，通常会布置TLS传输链路，配置好CA证书以保证通讯不被窃听篡改；如果需要解决服务发现问题，通常会设置DNS服务器，让服务访问依赖稳定的记录名而不是易变的A地址、SRV地址等记录值，等等。计算机科学经过多年的发展，这些问题大多都有了专职化的基础设施去解决，而之所以微服务时代，我们不得不在应用服务层面而不是基础设施层面去解决这些分布式问题，完全是因为由硬件构成的基础设施，跟不上由软件构成的应用服务的灵活性的无奈之举。软件可以只使用键盘就能拆分出不同的服务，只通过拷贝、启动就能够伸缩扩容服务，硬件难道也可以通过敲键盘就变出相应的应用服务器、负载均衡器、DNS服务器、网络链路这些设施吗！嗯？好像也可以啊？

行文至此，估计大家已经听出下面要说的是[虚拟化](#)技术和[容器化](#)技术了。微服务时代所取得的成就，本身就离不开以Docker为代表的早期容器化技术的巨大贡献。在此之前，笔者从来没有提过“容器”二字，这并不是刻意冷落，而是早期的容器只被简单地视为一种可快速启动的服务运行环境，目的是方便于程序的分发部署，这个阶段针对单个服务的容器并未真正参与到分布式问题的解决之中。尽管2014年微服务真正崛起的时候，Docker Swarm（2013年）和Apache Mesos（2012年）已经存在，更早之前也出现过[软件定义网](#)

络（Software-Defined Networking，SDN）、软件定义存储（Software-Defined Storage，SDS）等技术，但是，被业界广泛认可、普遍采用的通过虚拟化的基础设施去解决分布式架构问题的方案，应该要从2017年Kubernetes赢得容器战争的胜利开始算起。

2017年是容器生态发展历史中具有里程碑意义的一年。在这一年，长期作为Docker竞争对手的RKT容器一派的领导者CoreOS宣布放弃自己的容器管理系统Fleet，未来将会把所有容器管理的功能移至Kubernetes之上去实现。在这一年，容器管理领域的独角兽Rancher Labs宣布放弃其内置了数年的容器管理系统Cattle，提出了“All-in-Kubernetes”战略，从2.0版本开始把1.x版本能够支持多种容器管理工具的Rancher，“升级”为只支持Kubernetes一种容器管理系统。在这一年，Kubernetes的主要竞争者Apache Mesos在9月正式宣布了“Kubernetes on Mesos”集成计划，由竞争关系转为对Kubernetes提供支持，使其能够与Mesos的其他一级框架（如HDFS、Spark 和Chronos，等等）进行集群资源动态共享、分配与隔离。在这一年，Kubernetes的最大竞争者Docker Swarm的母公司Docker，终于在10月被迫宣布Docker要同时支持Swarm与Kubernetes两套容器管理系统，事实上承认了Kubernetes的统治地位。这场已经持续了三、四年时间，以Docker Swarm、Apache Mesos与Kubernetes为主要竞争者的“容器编排战争”终于有了明确的结果，Kubernetes登基加冕是容器发展中一个时代的终章，也将是软件架构发展下一个纪元的开端。笔者在下表列出了在同一个分布式服务的问题在Spring Cloud中提供的应用层面的解决方案与在Kubernetes中提供的基础设施层面的解决方案，尽管因为各自出发点不同，解决问题的方法和效果都有所差异，但这无疑是提供了一条全新的、前途更加广阔的解题思路。

	Kubernetes	Spring Cloud
弹性伸缩	Autoscaling	N/A
服务发现	KubeDNS / CoreDNS	Spring Cloud Eureka
配置中心	ConfigMap / Secret	Spring Cloud Config
服务网关	Ingress Controller	Spring Cloud Zuul
负载均衡	Load Balancer	Spring Cloud Ribbon
服务安全	RBAC API	Spring Cloud Security
跟踪监控	Metrics API / Dashboard	Spring Cloud Turbine
降级熔断	N/A	Spring Cloud Hystrix

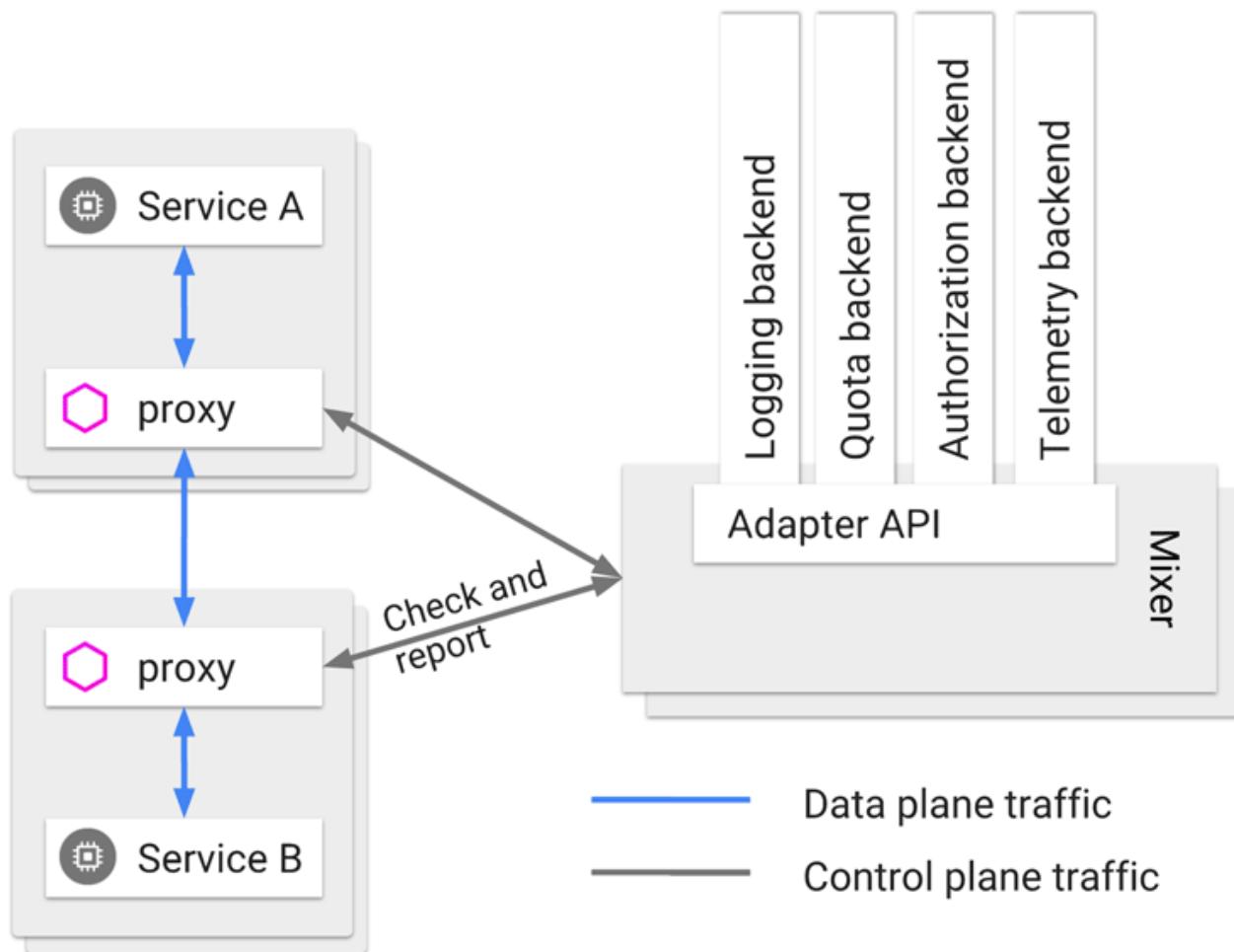
“前途广阔”不仅仅是一句恭维赞赏的客气话，当虚拟化的基础设施从单个服务的容器发展至由多个容器构成的服务集群，以及集群所需的所有通讯、存储设施时，软件与硬件的界限便开始模糊。一旦硬件能够跟上软件的灵活性，那些与业务无关的技术性问题便有可能能从软件层面剥离，悄无声息地解决于硬件基础设施之内，让软件得以只专注业务，真正“围绕业务能力构建”团队与产品。如此，DCE中未能实现的“透明的分布式应用”成为可能，Martin Flower设想的“[凤凰服务器](#)”成为可能，Chad Fowler提出的“[不可变基础设施](#)”成为可能，从软件层面独力应对分布式架构所带来的各种问题，发展到应用代码与基础设施软、硬一体，合力应对架构问题的时代，现在常被媒体冠以“云原生”这个颇为抽象的名字加以宣传。云原生时代与此前微服务时代中追求的目标并没有本质改变，在服务架构演进的历史进程中，笔者更愿意称其为“后微服务时代”。

Kubernetes成为容器战争胜利者标志着后微服务时代的开端，但Kubernetes并没有能够完美解决全部的分布式问题——“不完美”的意思是，仅从功能灵活强大这点而言，Kubernetes一般还不如之前的Spring Cloud方案。这是因为有一些问题处于应用系统与基础设施的边缘，使得完全在基础设施层面中确实很难精细化地解决。举个例子，譬如微服务A调用了微服务B中发布的两个服务，称为B1和B2，假设B1表现正常但B2出现了持续的500错，那在达到一定阈值之后就应该对B2进行熔断，以避免产生[雪崩效应](#)。如果仅在基础设施层面来处理，这会遇到一个两难问题，切断A到B的网络通路则会影响到B1的正常调用，不切断的话则持续受B2的错误影响。



这种问题在以前通过Spring Cloud这类应用代码实现的微服务中并不难处理，反正是使用代码（或者配置）来解决问题，只要合乎逻辑，想要去做什么功能，只受限于开发人员的想象力与技术能力，但基础设施是针对整个容器的整体管理的，粒度就相对粗旷。类似的情况不仅仅在断路器上出现，服务的监控、认证、授权、安全、负载均衡等等都有可能面临进行细化管理的需求，譬如服务调用时的负载均衡，往往需要根据流量特征，调整负载均衡的层次、算法，等等，而DNS尽管能实现一定程度的负载均衡，但通常并不能满足这些额外的需求。

为了解决这一类问题，微服务基础设施很快进行了第二次进化，引入了今天被称为“[服务网格](#)”（Service Mesh）的“边车代理模式”（Sidecar Proxy）。所谓的“边车”是一种带挎斗的三轮摩托，我小时候还算常见，现在基本就只在抗日神剧中才会看到了。这里指的意思是会由系统自动在服务的资源容器（指Kubernetes的Pod）中注入一个通讯代理服务器（相当于那个挎斗），以类似网络安全里中间人攻击的方式进行流量劫持，通过iptable或者ipvs，在应用毫无感知的情况下，悄然接管掉应用所有对外通讯。这个代理除了实现正常的服务调用间通讯外（称为数据平面通讯），同时还接受来自控制器的指令（称为控制平面通讯），根据控制平面中的配置，对数据平面通讯的内容进行分析，以实现熔断、认证、度量、监控、负载均衡等各种附加功能。这样便实现了即不需要在应用层面附带额外的代码，也提供了几乎不亚于应用代码的精细管理能力。



边车代理流量示意

图来自Istio的[配置文档](#)，图中的Mixer在Istio 1.5之后已经取消，这里仅作示意

很难从概念上判定清楚一个与应用系统运行于同一资源容器之内的代理服务到底应该算软件还是算基础设施，但它对应用是透明的，不需要改动任何软件代码就可以实现的服务治理，这便足够了。服务网格在2018年才火起来，今天它仍然是个新潮的概念，Istio和Envoy

y的发展时间尚短，仍然未完全成熟，甚至连Kubernetes也还算是个新生事物（以它开源的日期来计算）。但笔者相信，未来几年Kubernetes将会成为服务器端标准的运行环境，如同在此之前Linux；服务网格将会成为微服务之间通讯交互的主流模式，把“选择什么通讯协议”、“如何做认证授权”之类的技术问题隔离于应用软件之外，取代今天Spring Cloud全家桶中大部分组件的功能，这是最理想的[Smart Endpoints](#)解决方案，微服务只需要考虑业务本身的逻辑。

上帝的归上帝，凯撒的归凯撒，业务与技术完全分离，远程与本地完全透明，也许这就是最好的时代了吧？

无服务时代

无服务架构 (Serverless)

如果说微服务架构是分布式系统这条路的极致，那无服务架构，也许就是“不分布式”的云端系统这条路的起点。

进行分布式的目的，最初是由于单台机器的性能无法满足系统的运行需要，尽管后来架构演进过程中，容错能力、技术异构、职责划分等各方面因素都成为架构需要考虑的问题，但其中获得更好性能的需求在架构设计中比重依然很大。对软件研发而言，不去做分布式无疑是最简单的，如果单台服务器的性能可以是无限的，那架构演进的结果肯定会与今天有很大的差别，分布式也好，容器化也好，微服务也好，恐怕都未必会出现，最起码不必是如今天这个样子。

绝对意义上的无限性能必然是不存在的，但在云计算落地已有十年时间的今日，相对意义的无限性能已经成为了现实。在工业界，2012年，iron.io公司¹率先提出了“无服务”（Serverless，应该翻译为“无服务器”才合适，但现在称“无服务”已形成习惯了）的概念，2014年开始，AWS发布了命名为Lambda的商业化无服务应用，并在后续的几年里逐步得到开发者认可，发展成目前世界上最大的无服务的运行平台；到了2018年，中国的阿里云、腾讯云等厂商也开始跟进，发布了旗下的无服务的产品，“无服务”已成了近期技术圈里的“网红”之一。

在学术界，2009年，云计算概念刚提出的早期，UC Berkeley大学曾发表的论文《Above the Clouds: A Berkeley View of Cloud Computing²》，文中预言的云计算的价值、演进和普及在过去的十年里一一得到验证。十年之后的2019年，UC Berkeley的第二篇有着相同命名风格的论文《Cloud Programming Simplified: A Berkeley View on Serverless Computing³》发表，再次预言未来“无服务将会发展成为未来云计算的主要形式”，由此可见，“无服务”也同样是被主流学术界所认可发展方向（之一）。

We predict that serverless computing will grow to dominate the future of cloud computing
我们预测无服务将会发展成为未来云计算的主要形式

—— Cloud Programming Simplified: A Berkeley View on Serverless Computing ↗, 2019

无服务今天还没有一个权威的“官方”定义，但它的概念并没有前面各种架构那么复杂，本来无服务也是以“简单”为主要卖点的，它只涉及两块内容：后端设施（Backend）和函数（Function）。

- **后端设施**是指数据库、消息队列、日志、存储，等等这一类用于支撑业务逻辑运行，但本身无业务含义的技术组件，这些后端设施都运行在云中，无服务中称其为“后端即服务”（Backend as a Service，BaaS）。
- **函数**就是指的业务逻辑代码，这里函数的概念与粒度，都已经很接近于程序编码角度的函数了，其区别是无服务中的函数运行在云端，不必考虑算力问题，不必考虑容量规划（从技术角度可以不考虑，从计费的角度你的钱包够不够用还是要掂量一下的），无服务中称其为“函数即服务”（Function as a Service，FaaS）。

无服务的愿景是让开发者只需要纯粹地关注业务，不需要考虑技术组件，后端的技术组件是现成的，可以直接取用，没有采购、版权和选型的烦恼；不需要考虑如何部署，部署过程完全是托管到云端的，工作由云端自动完成；不需要考虑算力，有整个数据中心支撑，算力可以认为是无限的；也不需要操心运维，维护系统持续平稳运行是云服务商的责任而不再是开发者的责任。在UC Berkeley的论文中，把无服务架构下开发者不再关心这些技术层面的细节，类比成当年软件开发从汇编语言踏进高级语言的发展过程，开发者可以不去关注寄存器、信号、中断等与机器底层相关的细节，从而令生产力得到极大地解放。

无服务架构的远期前景也许是很美好的，但笔者自己对无服务中短期内的发展并没有那么乐观，与单体架构、微服务架构不同，无服务架构一些天生的特点决定了它现在不是，以后如果没有重大变革的话，估计也很难成为一种普适性的架构模式，它对一些适合的应用确实能够降低开发和运维环节的成本，譬如不需要交互的离线大规模计算，又譬如多数Web资讯类网站、小程序、公共API服务、移动应用服务端等都契合于无服务架构所擅长的短链接、无状态、适合事件驱动的交互形式；但另一方面，对于那些信息管理系统、网络游戏等应用，又或者说所有具有业务逻辑复杂，依赖服务端状态，响应速度要求较高，需要长链接，等等这些特征的应用，无服务架构至少目前是相对并不合适的。这是因为无服务天生“无限算力”的假设就决定了它必须要按使用量（函数运算的时间和内存）计费以控制消耗算力的规模，因而函数不会一直以活动状态常驻服务器，请求到了才会开始运行，这导致了函数不便依赖服务端状态，也导致了函数会有冷启动时间，响应的性能不可能太好

(目前无服务的冷启动过程大概是在数十到百毫秒级别 , 对于 Java 这类启动性能差的应用 , 甚至能到接近秒的级别) 。

无论如何 , 云计算毕竟是大势所趋 , 今天信息系统建设的概念和观念 , 在 (较长尺度的) 明天都是会转变成适应云端的 , 笔者并不怀疑 Serverless+API 的设计方式会成为以后其中一种主流的软件形式 , 届时无服务还会有更广阔的应用空间。

如果说微服务架构是分布式系统这条路当前所能做到的极致 , 那无服务架构 , 也许就是 “ 不分布式 ” 的云端系统这条路的起点。虽然在顺序上笔者将 “ 无服务 ” 安排到了 “ 微服务 ” 和 “ 云原生 ” 时代之后 , 但它们两者并没有继承替代关系 , 强调这点是为了避免有读者从两者的名称与安排的顺序中产生 “ 无服务就会比微服务更加先进 ” 的错误想法。笔者相信软件开发的未来不会只存在某一种 “ 最先进的 ” 架构风格 , 多种具针对性的架构风格同时并存 , 是软件产业更有生命力的形态。笔者同样相信软件开发的未来 , 多种架构风格将会融合互补 , “ 分布式 ” 与 “ 不分布式 ” 的边界将逐渐模糊 , 两条路线在云端的数据中心中交汇。今天已经能初步看见一些使用无服务的云函数去实现微服务架构的苗头了 , 将无服务作为技术层面的架构 , 将微服务视为应用层面的架构 , 把它们组合起来使用是完全合理可行的。以后 , 无论是通过物理机、虚拟机、容器 , 抑或是无服务云函数 , 都会是微服务实现方案的候选选项之一。

本节是架构演进历史的最后一篇 , 如引言所说 , 我们谈历史 , 重点不在考古 , 而是借历史之名 , 理解好每种架构出现的意义与淘汰的原因 , 为的是更好地解决今天的现实问题 , 寻找出未来架构演进的发展道路。对于架构演进的未来发展 , 2014 年 , Martin Fowler 与 James Lewis 在《 Microservices 》的结束语中曾写到 , 他们对于微服务日后能否被大范围地推广 , 最多只能持有谨慎的乐观 , 无服务方兴未艾的今天 , 与那时微服务的情况十分相近 , 笔者对无服务日后的推广同样持乐观谨慎的乐观态度。软件开发的最大挑战就在于只能在不完备的信息下决定当前要处理的问题。时至今日 , 依然很难预想在架构演进之路的前方 , 微服务和无服务之后还会出现何种形式的架构风格 , 但这也契合了图灵的那句名言 : 尽管目光所及之处 , 只是不远的前方 , 即使如此 , 依然可以看到那里有许多值得去完成的工作在等待我们。

We can only see a short distance ahead, but we can see plenty there that needs to be done.
尽管目光所及之处 , 只是不远的前方 , 即使如此 , 依然可以看到那里有许多值得去完成的工作在等待我们。

—— Alan Turing , Computing Machinery and Intelligence , 1950

远程访问

在软件业发展的初期，程序编写都是以算法为核心的，程序员会把数据和过程分别作为独立的部分来考虑，数据代表问题空间中的客体，程序代码则用于处理这些数据，这种思维方式直接站在计算机的角度去抽象问题和解决问题，被称为面向过程的编程思想。与此类似，后来出现的面向对象的编程思想则站在现实世界的角度去抽象和解决问题，它把数据和行为都看作是对象的一部分，这样可以让程序员能以符合现实世界的思维方式来编写和组织程序。

这两种思想出现的时间有先后，但在人类使用计算机语言来处理数据的工作中，无论提倡以计算机的思维还是提倡以人类的思维来抽象问题，都是合乎逻辑的。经过了上世纪90年代末到21世纪初期面向对象编程的火热之后，如今又出现了另一种考虑如何对内封装逻辑、对外重用服务的新思想：面向资源的编程思想。这种思想是把问题空间中的数据对象作为抽象的主体，把解决问题时从输入数据到输出结果的处理过程，看作是一个（组）数据资源的状态不断发生变换而导致的结果。这种思想有其生根的土壤基础：在跨越进程、跨越网络主机、跨越编程语言的分布式系统中，人们尝试过将之前在单进程应用里行之有效的面向过程、面向对象的服务设计方法改造迁移，使之适应分布式环境，这项工作总体上获得了成功，但在分布式环境里多少还是出现了一些不适与瑕疵，所以为另一种服务设计风格，即面向资源的编程思想留出了成长的空间。

尽管在2020年还谈论什么RESTful、RPC，大概是确实有点落伍了，可这个问题是一个架构设计者必须有明确取舍权衡的重要技术决策，今天笔者仍准备来谈一下这个话题。

远程服务调用

远程服务调用（ Remote Procedure Call , RPC ）在计算机科学中已经存在有超过40年时间了，但在今天仍然可以在Quora、知乎等网站上随处可见“什么是RPC？”、“如何评价某某RPC技术？”、“RPC好还是REST好？”之类的问题，仍然“每天”都有新的不同形状的RPC轮子被发明制造出来，仍然有层出不穷的文章去比对谷歌gRPC、阿里Dubbo等等各个厂家的RPC技术优劣。

在计算机这个技术快速更迭的领域，以上情景并不是常见的现象，这一方面是由于微服务风潮带来的热度，另外一方面，也不得不承认，部分开发者对RPC本身解决什么问题、如何解决这些问题、为什么要这样解决都或多或少存在认知模糊。本篇中，笔者将尽可能从根源到现状，从表现到本质去解析清楚RPC的来龙去脉。

进程间通讯

尽管今天的大多数RPC技术已经不再追求这个目标了，但无可否认，RPC出现的最初目的，就是为了让计算机能够跟调用本地方法一样去调用远程方法。所以，我们先来看一下本地方法调用时，会发生什么。笔者通过以下这段Java风格的伪代码，来定义几个概念：

```
// 调用者 (Caller)      : main()
// 被调用者 (Callee)    : println()
// 调用点 (Call Site)   : 发生方法调用的指令流位置
// 调用参数 (Parameter) : 由Caller传递给Callee的数据，即“hello world”
// 返回值 (Retval)     : 由Callee传递给Caller的数据，如果方法正常完成，是
void，如果方法异常完成，是对应的异常
public static void main(String[] args) {
    System.out.println("hello world");
}
```

java

在完全不考虑编译器优化的前提下，程序运行至调用 `println()` 的这一行时，计算机（物理机或者虚拟机）会做以下这些事情：

1. **传递方法参数**：将字符串 `helloworld` 的引用压栈。
2. **确定方法版本**：确定 `println()` 方法的版本其实并不是一个简单的过程，不论是编译时静态解析也好，是运行时动态分派也好，总之必须根据某些语言规范中明确定义原则，找到明确的 `Callee`，“明确”是指唯一的一个 `Callee`，或者有严格优先级的多个 `Callee`，譬如不同的重载版本。笔者曾在《深入理解Java虚拟机》中用一整章篇幅介绍该过程，这里就不赘述了。
3. **执行方法**：从栈中获得 `Parameter`，以此为输入，执行 `Callee` 内部的逻辑。
4. **传回执行结果**：将 `Callee` 的执行结果压栈，并将指令流恢复到 `Call Site` 处继续向下执行。

接下来，我们考虑当 `println()` 方法不在当前进程的内存地址空间中的情况。很显然，此时第一步如何将参数传递给方法就无法做下去，把参数在调用进程的内存中压栈，对于另外一个进程执行的方法毫无意义。我们面临的第一个问题是两个进程之间，要有交换数据的手段，这件事情被称为“[进程间通讯](#)”（Inter-Process Communication，IPC）。可以考虑的办法有以下几种：

- **管道（Pipe）或者具名管道（Named Pipe）**：管道类似于两个进程间的桥梁，用于进程间传递少量的字符流或字节流。普通管道可用于有亲缘关系进程（由一个进程启动的另外一个进程）间的通信，具名管道摆脱了普通管道没有名字的限制，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。管道典型的应用就是命令行中的 `|` 操作符，譬如：

```
ps -ef | grep java
```

sh

以上命令就通过管道操作符 `|` 将 `ps` 命令的标准输出通过管道连接到 `grep` 命令的标准输入上。

- **信号（Signal）**：信号用于通知目标进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程自身。信号的典型应用是 `kill` 命令，譬如：

```
kill -9 pid
```

sh

以上就是由 Shell 进程向指定 PID 的进程发送 SIGKILL 信号。

- **信号量** (Semaphore) : 信号量用于两个进程之间同步协作手段，它相当于操作系统提供的一个特殊变量，你可以在上面进行wait()和notify()操作。
- **消息队列** (Message Queue) : 以上三种方式只适合传递少量信息，POSIX标准中有定义消息队列用于进程间通讯。进程可以向队列中添加消息，被赋予读权限的进程则可以从队列消费消息。消息队列克服了信号承载信息量少，管道只能用于无格式字节流以及缓冲区大小受限等缺点。
- **共享内存** (Shared Memory) : 允许多个进程可以访问同一块内存空间，这是效率最高的进程间通讯形式。进程的内存地址空间是独立隔离的，但操作系统提供了让进程主动创建、映射、分离、控制某一块内存的接口。由于内存是多进程共享的，所以往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
- **套接字接口** (Socket) : 以上两种方式只适合单机多进程间的通讯，套接字接口是更为普适的进程间通信机制，可用于不同机器之间的进程通信。起初是由Unix系统的BSD分支开发出来的，但现在已经移植到所有*nix系统上。基于效率考虑，当仅限于本机进程间通讯时，套接字接口是被优化过的，不会经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等操作，只是简单地将应用层数据从一个进程拷贝到另一个进程，此时可以称之为Unix Domain Socket。

通信的成本

之所以花费那么多篇幅来介绍IPC的手段，是因为最初计算机科学家们的想法，就是将RPC作为IPC的一种特例来看待的（其实现在分类上这么说也仍然合适，只是到具体操作手段上不会这么做了）。请读者特别关注最后一种基于套接字接口的通讯方式（IPC Socket），它不仅普适，能够支持基于的网络多机进程间通讯，而且被许多实践验证过是有效的，譬如X Window服务器和GUI程序之间的交互就是由这套机制来实现。此外，这样做有一个看起来无比诱人的好处，由于IPC Socket是操作系统提供的标准接口，完全有可能把远程方法调用的通讯细节隐藏在操作系统底层，从应用层面上看来可以做到远程调用与本地方法调用几乎完全一致。事实上，在原始分布式时代的初期确实是奔着这个目标去做的，但这种透明的调用形式却反而造成了程序员误以为通信是无成本的假象，从而被滥用以至于显著降低了分布式系统的性能。1987年，当“透明的RPC调用”一度成为主流范式的时候，Andrew Tanenbaum教授曾发表了论文《[A Critique of the Remote Procedure Call Paradigm](#)》，对这种透明的RPC范式提出了一系列质问：

- 两个进程通讯，谁作为服务端，谁作为客户端？
- 怎样进行异常处理？异常该如何让调用者获知？
- 服务端出现多线程竞争之后怎么办？
- 如何提高网络利用的效率，譬如连接是否可被多个请求复用以减少开销？是否支持多播？
- 参数、返回值如何表示？应该有怎样的字节序？
- 如何保证网络的可靠性？譬如调用期间某个链接忽然断开了怎么办？
- 发送的请求服务端收不到回复该怎么办？
-

论文的中心观点是：本地调用与远程调用当做一样处理这是犯了方向性的错误，把系统间的调用做成透明，反而会增加程序员工作的复杂度。此后几年，关于RPC应该如何发展、如何实现的论文层出不穷，支持者有之，反对者有之，冷静分析者有之，狂热唾骂者有之，但历史逐渐证明Andrew Tanenbaum的预言是正确的。最终，到1994年至1997年间，由ACM和Sun院士Peter Deutsch[☞](#)、套接字接口发明者Bill Joy[☞](#)、Java之父James Gosling[☞](#)等一众在Sun Microsystems工作的大佬们共同总结了通过网络进行分布式运算的八宗罪[☞](#)（8 Fallacies of Distributed Computing）：

1. The network is reliable —— 网络是可靠的
2. Latency is zero —— 延迟是不存在的
3. Bandwidth is infinite —— 带宽是无限的
4. The network is secure —— 网络是安全的
5. Topology doesn't change —— 拓扑结构是一成不变的
6. There is one administrator —— 总会有一个管理员
7. Transport cost is zero —— 不考虑传输成本
8. The network is homogeneous —— 网络是同质化的

以上这八条被认程序员在网络编程中经常忽略的八大问题，潜台词就是如果远程服务调用要弄透明化的话，就必须为这些罪过埋单，这算是给RPC是否能等同于IPC来实现暂时定下了一个具有公信力的结论。至此，RPC应该是一种高层次的或者说语言层次的特征，而不是像IPC那样，是低层次的或者说系统层次的特征成为工业界、学术界的主流观点。

在1980年代初期，传奇的施乐Palo Alto研究中心[☞](#)发布了基于Cedar语言的RPC框架Lupine，并实现了世界上第一个基于RPC的商业应用Courier，这里施乐PARC所定义的“远程服

务调用”的概念就是符合以上对RPC的结论的，所以，尽管此前已经有用其他名词指代RPC这种操作，一般仍认为RPC的概念最早由施乐公司所提出。

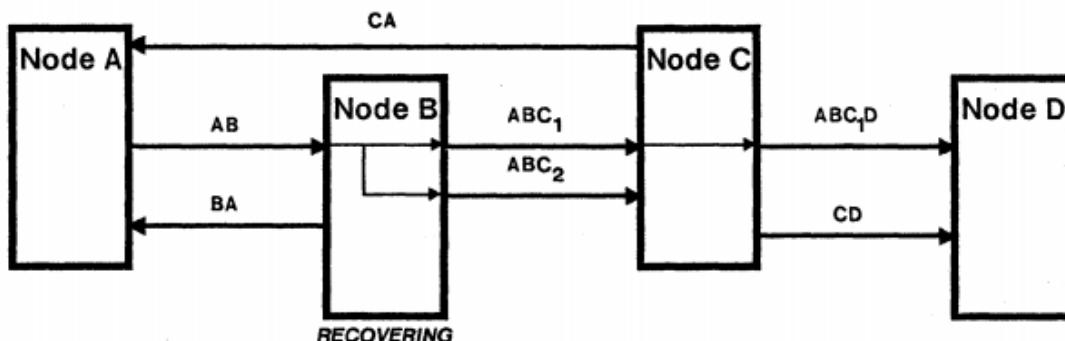
额外知识：首次提出远程服务调用的定义

Remote procedure call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel.

Remote Procedure Call

by Bruce Jay Nelson

CSL-81-9 May 1981



—— Bruce Jay Nelson , Remote Procedure Call , Xerox PARC , 1981

三个基本问题

80年代中后期，惠普和Apollo提出了网络运算架构（Network Computing Architecture, NCA）的设想，并随后在DCE项目中发展成在Unix系统下的远程服务调用框架DCE/RPC，笔者曾经在“原始分布式时代”中介绍过DEC，这是历史上第一次对分布式有组织的探索尝试，由于DEC本身是基于Unix操作系统的，所以DEC/RPC也仅面向于在Unix系统程序之间通用（不全对，微软COM/DCOM的前身MS RPC是DCE的一种变体版本）。在198

8年，Sun Microsystems起草并向[互联网工程任务组](#)（Internet Engineering Task Force，IETF）提交了[RFC 1050](#)规范，此规范中设计了一套面向于广域网或混合网络环境的、基于TCP/IP网络的、支持C语言的RPC协议，后被称为[ONC RPC](#)（Open Network Computing RPC，也被称为Sun RPC），这两个RPC协议就可以算是如今各种RPC协议的鼻祖了，从它们开始，直至接下来这几十年来所有流行过的RPC协议，都不外乎通过各种手段来解决以下三个基本问题：

- **如何表示数据**：这里数据包括了传递给方法的参数，以及方法的返回值。无论是将参数传递给另外一个进程，还是从另外一个进程中取回执行结果，都涉及到它们应该如何表示的问题。进程内的方法调用，使用程序语言内置的和程序员自定义的数据类型就很容易解决数据表示问题，远程方法调用则完全可能面临交互双方分属不同程序语言的情况；即使只支持同一种语言RPC协议，在不同硬件指令集、不同操作系统下，也完全可能有不一样表现细节，譬如数据宽度、字节序的差异等等。行之有效的方法是将交互双方所涉及的数据转换为某种事先约定好的中立数据流格式来进行传输，将数据流转换回不同语言中对应的数据类型来进行使用，这个操作相信大家都很熟悉，就是序列化与反序列化。每种RPC协议都应该要有对应的序列化协议，如：
 - ONC RPC的[External Data Representation](#)（XDR）
 - CORBA的[Common Data Representation](#)（CDR）
 - Java RMI的[Java Object Serialization Stream Protocol](#)
 - gRPC的[Protocol Buffers](#)
 - Web Service的[XML Serialization](#)
 - 众多轻量级RPC支持的[JSON Serialization](#)
 -
- **如何传递数据**：准确地说，是指如何通过网络，在两个服务Endpoint之间相互操作、交换数据。这通常指的是应用层协议，实际传输一般是基于标准的TCP、UDP等传输层协议来完成的。两个服务交互不是只扔个序列化数据流来表示参数和结果就行的，许多在此之外信息，譬如异常、超时、安全、认证、授权、事务，等等，都可能存在双方交换信息的需求。在计算机科学中，专门有一个名称“[Wire Protocol](#)”来用于表示这种两个Endpoint之间交换这类数据的行为，常见的Wire Protocol有：
 - Java RMI的[Java Remote Message Protocol](#)（JRMP，也支持[RMI-IIOP](#)）
 - CORBA的[Internet Inter ORB Protocol](#)（IIOP，是GIOP协议在IP协议上的实现版本）
 - DDS的[Real Time Publish Subscribe Protocol](#)（RTPS）

- Web Service的Simple Object Access Protocol (SOAP)
 - 如果要求足够简单，双方都是HTTP Endpoint，直接使用HTTP也是可以的（如JSON-RPC）
 -
- **如何表示方法**：这在本地方法调用中也不成问题，编译器或者解释器会根据语言规范，将调用的方法转换为进程地址空间中方法入口位置的指针。不过一旦考虑不同语言，事情又麻烦起来，每门语言的方法签名都可能有所差别，所以如“何表示一个方法”，“如何找到这些方法”还是得弄个统一的标准。这个标准做起来可以很简单，只要给程序中每个方法都规定一个唯一的绝不重复的编号，调用时不用管它什么方法签名定义，直接传这个编号就可以找到对应的方法。这种听起无比寒碜的办法，还真的就是DCE/RPC的最初准备的解决方案。虽然最终DCE还是弄出了一套语言无关的[接口描述语言](#) (Interface Description Language , IDL)，成为此后许多RPC参考或依赖的基础（如CORBA的OMG IDL），但那个唯一的“绝不重复”的编码方案[UUID](#)却广为流传开来，今天已广泛应用于程序开发的方方面面。这类用于表示方法的协议还有：
- Android的[Android Interface Definition Language](#) (AIDL)
 - CORBA的[OMG Interface Definition Language](#) (OMG IDL)
 - Web Service的[Web Service Description Language](#) (WSDL)
 - JSON-RPC的[JSON Web Service Protocol](#) (JSON-WSP)
 -

以上三个RPC中的基本问题，我们都可以在本地方法调用中找到对应的操作。RPC的思想始于本地方法调用，尽管早已不再追求实现成与本地方法调用完全一致，但其发展仍然带有本地方法调用的深刻烙印，抓住两者间的联系来类比，对我们理解RPC的本质很有好处。

统一的RPC

DEC/RPC与ONC RPC都有很浓厚的Unix痕迹，其实并没有真正在Unix系统以外大规模流行过，而且它们还有一个“大问题”：只支持在传递值而不支持传递对象（ONC RPC的XDR的序列化器能用于序列化结构体，但结构体毕竟不是对象），这两门RPC协议都是面向C语言设计的，根本就没有对象的概念。而90年代正好又是[面向对象编程](#) (Object-Oriented Programming , OOP) 风头正盛的年代，所以在1991年，[对象管理组织](#) (Object Management Group , OMG) 便发布了跨进程、面向异构语言的、支持面向对象的服务调用协

议：CORBA 1.0 (Common Object Request Broker Architecture)，1.0和1.1版本只提供了C、C++语言的支持，到了末代的CORBA 3.0版本，不仅支持了C、C++、Java、Lisp、Object Pascal、Python、Ruby等多种主流编程语言，还支持了Smalltalk、Ada、COBOL等已经半截入土的非主流语言，阵营不可谓不强大。这是一套由国际标准组织牵头，由多家软件提供商共同参与的分布式规范，当时影响力只有微软私有的[DCOM](#)可以与之稍微抗衡，但微软的DCOM与DCE一样，是受限于操作系统的（不过比DCE厉害的是DCOM能跨语言哟），所以同时支持跨系统、跨语言的CORBA原本是最有机会统一RPC这个细分领域的有力竞争者。

但无奈CORBA本身设计得实在是太过于啰嗦繁琐，甚至有些规定简直到了荒谬的程度——写一个对象请求代理（ORB，这是CORBA中的关键概念）大概要200行代码，其中大概有170行都是纯粹无用的废话（这句带有鞭尸性质的得罪人的评价不是笔者写的，是CORBA的首席科学家Michi Henning在文章《[The Rise and Fall of CORBA](#)》中自己说的）。另一方面，为CORBA制定规范的专家逐渐脱离实际，做出CORBA规范晦涩难懂，各家语言的厂商都有自己的解读，结果弄出来的CORBA实现互不兼容，实在是对CORBA号称支持众多异构语言的莫大讽刺。这也间接造就了稍后W3C Web Service出现，CORBA与Web Service竞争时犹如十八路诸侯讨董卓，互乱阵脚一触即溃的惨败局面。最终下场是CORBA与DCOM一同被扫进计算机历史的博物馆中。

CORBA没有把握住统一RPC的大好机遇，很快另外一个更有希望的机会再次降临。1998年，XML 1.0发布，并成为[万维网联盟](#)（World Wide Web Consortium，W3C）的推荐标准。1999年末，SOAP 1.0 (Simple Object Access Protocol) 规范的发布，它代表着一种被称为“Web Service”的全新的RPC协议的诞生。Web Service是由微软和DevelopMentor公司共同起草的远程服务协议，随后提交给W3C投票成为国际标准，所以Web Service也被称为W3C Web Service。Web Service采用了XML作为远程过程调用的序列化、接口描述、服务发现等所有编码的载体，当时XML是计算机工业最新的银弹，只要是定义为XML的东西几乎就都被认为是好的，风头一时无两，连微软自己都主动宣布放弃DCOM，迅速转投Web Service的怀抱。

Web Service没有天生属于哪家公司的烙印，商业运作非常成功，很受市场欢迎，大量的厂商都想分一杯羹。但从技术角度来看，它设计得也并不优秀，甚至同样可以说是有显著缺陷的。对于开发者而言，Web Service的一大缺点是它那过于严格的数据和接口定义所带来的性能问题，尽管Web Service吸取了CORBA失败的教训，不需要程序员手工去编写对象的描述和服务代理，可是，XML作为一门描述性语言本身信息密度就较为低下（都不用与

二进制协议比，与今天的JSON或YAML比一下就知道了），Web Service又是跨语言的RPC协议，这使得一个简单的字段，为了在不同语言中不会产生歧义，要以XML描述去清楚的话，往往比原本存储这个字段值的空间多出十几倍、几十倍乃至上百倍。这个特点一方面导致了使用Web Service必须要专门的客户端去调用和解析SOAP内容，也需要专门的服务去部署（如Java中的Apache Axis/CXF），更关键的是导致了每一次数据交互都包含大量的冗余信息，性能奇差无比。

如果只是需要客户端、传输性能差也就算了，又不是不能用。既然选择了XML，获得自描述能力，本来也就没有打算把性能放到第一位，但Web Service还有另外一点原罪：贪婪。“贪婪”是指它希望在一套协议上一揽子解决分布式计算中可能遇到的所有问题，这导致Web Service生出了一整个家族的协议出来（这句话居然不是拟人修辞）。Web Service协议家族中，除它本身外包括了的SOAP、WSDL、UDDI协议之外，还有一堆几乎说不清有多少个、以WS-*命名的、用于解决事务、一致性、事件、通知、业务描述、安全、防重放……等等的子功能协议，子子孙孙无穷无尽，对开发者造成了非常沉重的学习负担，这次算是真得了罪惨了开发者，谁爱用谁用去。

当程序员们对Web Service的热情迅速兴起，又逐渐冷却之后，自己也不仅开始反思：那些面向透明的、简单的RPC协议，如DCE/RPC、DCOM、Java RMI，要么依赖于操作系统，要么依赖于特定语言，总有一些先天约束；那些面向通用的、普适的RPC协议；如CORBA，就无法逃过使用复杂性的困扰，CORBA的OMG IDL、ORB都是很好的佐证；而那些意图通过技术手段来屏蔽复杂性的RPC协议，如Web Service，又不免受到性能问题的束缚。简单、普适、高性能这三点，似乎真的难以同时满足。

分裂的RPC

由于一直没有一个同时满足以上三点的“完美RPC协议”出现，所以远程服务器调用这个小小领域里，逐渐进入了群雄混战、百家争鸣的战国时代，距离“统一”是越来越远，并一直延续至今。现在，已经相继出现过RMI（Sun/Oracle）、Thrift（Facebook/Apache）、Dubbo（阿里巴巴/Apache）、gRPC（Google）、Motan2（新浪）、Finagle（Twitter）、brpc（百度）、.NET Remoting（微软）、Arvo（Hadoop）、JSON-RPC 2.0（公开规范，JSON-RPC工作组）……等一系列的协议/框架。这些RPC功能、特点不尽相同，有的是某种语言私有，有的能支持跨越多门语言，有的运行在HTTP协议之上，有的能直接运行于TCP/UDP之上的，但肯定不存在哪一款是“最完美的RPC”。今时今日，任何一款具有生命

力的RPC框架，都不再去追求大而全的“完美”，而是有自己的针对性特点作为主要的发展方向，譬如：

- 朝着**面向对象发展**，不满足于RPC将面向过程的编码方式带到分布式，希望在分布式系统中也能够进行跨进程的面向对象编程，代表为RMI、.NET Remoting，之前的CORBA和DCOM也可以归入这类，这条线有一个别名叫做**分布式对象**（Distributed Object）。
- 朝着**性能发展**，代表为gRPC和Thrift，SOAP的墓志铭上还刻着“信息密度（Payload所占传输数据的比例大小，使用的传输协议和协议的设计都会影响到这点）和序列化效率是对性能影响最大的因素”，gRPC和Thrift都有自己优秀的私有序列化器，而传输协议，gRPC是基于HTTP2的，支持多路复用和Header压缩，Thrift则直接基于TCP协议自己来处理编码。
- 朝着**简化发展**，代表为JSON-RPC，说要选功能最强、速度最快的RPC可能会有争议，但选功能弱的、速度慢的，JSON-RPC肯定会候选人中之一。牺牲了功能和效率，换来的是协议的简单，接口与格式都更为通用，尤其适合支持Web浏览器这类一般不会有额外协议、客户端支持的应用场合。
-

经历了RPC框架的“战国时代”，开发者们终于认可了不同的RPC框架所提供的不同特性或多或少是有矛盾的，很难有某一种框架说“我全部都要”。要把面向对象那套全搬过来，就注定不会太简单（如建Stub、Skeleton就很烦了，即使由IDL生成也很麻烦）；功能多起来，协议就要弄得复杂，效率一般就会受影响；要简单易用，那很多事情就必须遵循约定而不是配置才行；要重视效率，那就需要采用二进制的序列化器和较底层的传输协议，支持的语言范围容易受限。也正是每一种RPC框架都有不完美的地方，所以才导致不断有新的RPC轮子出现，决定了选择框架时在获得一些利益的同时，要付出另外一些代价。到了最近几年，RPC框架有明显的朝着插件化方向发展的趋势，不再选择自己去解决RPC的全部三个问题（表示数据、传递数据、表示方法），而是将全部或者一部分问题设计为扩展点，实现核心能力的可配置，再辅以外围功能，如负载均衡、服务注册、可观测性等方面的支持。这一类框架的代表有Facebook的Thrift与阿里的Dubbo（现在两者都是Apache的）。以Dubbo的序列化器为例，它默认采用Hessian 2作为序列化器，如果你有JSON的需求，可以替换为Fastjson，如果你对性能有更高的需求，可以替换为Kryo、FST、Protocol Buffers等，如果不想依赖其他包，直接使用JDK自带的序列化器也可以，这种设计在一定程度上缓解了RPC框架必须取舍，难以完美的缺憾。

最后，笔者提个问题，大家不妨来反思一下：开发一个分布式系统，是不是就一定要用RPC呢？RPC的三大问题源自于对本地方法调用的类比模拟，如果我们把思维从“方法调用”的约束中挣脱，那参数与结果如何表示、方法如何表示、数据如何传递这些问题都会海阔天空，拥有焕然一新的视角。但是我们写程序，真的可能不面向方法来编程吗？这就是笔者下一篇准备谈的话题了。

后记：前文提及的DCOM、CORBA、Web Service的失败时，可能笔者的口吻多少有一些戏虐，这只是落笔行文的方式，这些框架即使没有成功，但作为早期的探索先驱，并没有什么该去讽刺的地方。而且它们的后续发展，都称得上是知耻后勇的表现，反而值得我们去赞赏。譬如说到CORBA的消亡，OMG痛定思痛之后，提出了基于RTPS协议栈的“[数据分发服务](#)”商业标准（Data Distribution Service，DDS，“商业”就是要付费使用的意思），如今主要流行于物联网领域，能够做到微秒级延时，还能支持大规模并发通讯。譬如说到DCOM的失败和Web Service的式微，微软在它们的基础上推出的[.NET WCF](#)（Windows Communication Foundation，Windows通信基础），不仅同时将REST、TCP、SOAP等不同形式的调用自动封装为完全一致的如同本地方法调用一般的程序接口，还依靠自家的“地表最强IDE”Visual Studio将工作量减少到只需要指定一个远程服务地址，就可以获取服务描述、绑定各种特性（譬如安全传输）、自动生成客户端调用代码、甚至还能选择同步还是异步之类细节的程度。尽管这东西只支持.NET平台，而且与传统Web Service一样采用XML描述，但使用起来体验真的是异常地畅快，能挽回Web Service中得罪开发者丢掉的全部印象分。

REST服务设计风格

REST无论是思想上、概念上、还是应用目标上，它与各种RPC协议只能算是有所关联，有一些重合，但本质上并不是同一类型的东西。所谓思想上的不同，是指面向过程的编程思想与面向资源的编程思想之间的差异，至于什么是面向资源编程，后文中我们再详谈。

而概念上的不同主要是指REST并不是一种远程服务调用协议，甚至可以把定语也去掉，它就不是一种协议。协议都带有一定的规范性和强制性，最起码也该有个规约文档吧，譬如JSON-RPC，它再简单，也要有个《[JSON-RPC Specification](#)》来规定它的格式细节、异常、响应码等信息，但REST并没有这些东西，尽管有一些指导原则，实际上并不受任何强制约束。常有人批评某个系统接口“设计得不够RESTful”，其实这句话本身就有争议，RESTful只是风格而不是规范、协议，而且能完全达到REST所有指导原则的系统也是很少见的，这一点我们同样将在稍后详细讨论。

至于应用目标，REST与RPC在范围上是确有一些重合的，不过重合的区域有多大却是见仁见智。上一节提到了当前的RPC协议框架都各有侧重点，并且列举了RPC一些发展方向，这里面分布式对象这一条线的应用与REST可以说是毫无关系；而能够重视“效率”这个方向的应用，基本上就限制了只能是后端服务（前端应用对于网络协议、序列化器这两点都没有选择的余地，想要高效率也有心无力），在分布式服务各个后端节点之间通讯这一块，REST虽然照样可以用于任何语言（只要有个HTTP Client就可以用）之间的调用，而且是微服务中推荐的通讯方式，但在需要追求效率的后端应用场景里，REST提升传输效率的潜力非常有限，为性能而选择REST真算不得是个好决定。我们开发的REST服务，大多数的是提供给前端或效率不处于主要关注点的部分后端场景去消费的。在前端这一块，一众RPC里最多也就是JSON-RPC有机会与REST产生竞争，其他所有RPC协议、框架，哪怕是能够支持HTTP协议，哪怕提供了JavaScript版本的客户端（如gRPC-Web），也只是存在前端使用的[理论可行性](#)，很少见有实际项目把它们真应用到浏览器上的。

但尽管有如此多的不同，这两者还是产生了很多的比较与争论，就如同当年面向对象与面向过程一样，非得争出个高低不可。网上许多REST vs RPC的口水仗中说REST不好的，通常也并不是支持哪个RPC框架/协议比它好用，大多都只是不赞成REST的设计风格，心中说的本意其实是“面向资源编程”的思想不好，不如“面向过程编程”来得好用好理解。

理解REST

个人会有好恶偏爱，但计算机科学是务实的，有了RPC，还会提出REST，有了面向过程编程之后，还能产生面向资源编程，并引起广泛的关注、使用和讨论，后者一定是一些前者没有的闪光点，或者解决、避免了一些面向过程中的缺陷。我们不妨先去理解REST为什么出现、解决什么问题、方法是什么，然后再来讨论评价它。

许多人都知道REST源于Roy Thomas Fielding在2000年发表的博士论文：《Architectural Styles and the Design of Network-based Software Architectures》[»](#)，此文的确是REST的源头，但我们不能忽略Fielding的身份和之前工作的背景，这对理解REST的设计思想至关重要。

首先，Fielding是一名很优秀的软件工程师，他是Apache服务器的核心开发者，后来成为了著名的Apache软件基金会[»](#)的联合创始人；同时，Fielding也是HTTP 1.0协议（1996年发布）的专家组成员，后来还成为了HTTP 1.1协议（1999年发布）的负责人。HTTP 1.1协议设计的极为成功，以至于发布之后长达十年的时间里，都没有多少人认为有修订的必要。用来指导HTTP 1.1协议设计的理论和思想，最初是以备忘录的形式在专家组成员之间交流，除了IETF、W3C的专家外，并没有在外界广泛流传。



Roy Thomas Fielding

从时间上看，对HTTP 1.1协议的设计工作贯穿了Fielding的整个博士研究生生涯，当起草HTTP 1.1协议的工作完成后，Fielding回到了加州大学欧文分校继续攻读自己的博士学位。第二年，他更为系统、严谨地阐述了这套理论框架，并且以这套理论框架导出了一种新的编程风格，他为这种风格取了一个很多人难以理解，但是今天已经广为人知的名字REST（**R**epresentational State Transfer），即“表征状态转移”的缩写。

哪怕对编程和网络都很熟悉的同学，只从标题中也不太可能直接弄明白什么叫“表征”、啥东西的“状态”、从哪“转移”到哪。尽管在论文原文中确有论述这些概念，但写得确实相当晦涩（不想读英文的同学从此处获得中文翻译版本[↗](#)），笔者推荐一种比较容易理解REST思想方式是先理解什么是HTTP，再配合一些实际例子来进行类比，你会发现“REST”实际上是“HTT”（**H**yper **T**ext **T**ransfer）的进一步抽象，两者就如同接口与实现类之间的关系一般。

HTTP中使用的“超文本”一词是美国社会学家Theodor Holm Nelson在1967年于《[Brief Words on the Hypertext](#)》一文里提出的，下面引用的是他本人在1992年修正后的定义：

Hypertext

By now the word "hypertext" has become generally accepted for branching and responding text, but the corresponding word "hypermedia", meaning complexes of branching and responding graphics, movies and sound – as well as text – is much less used. Instead they use the strange term "interactive multimedia": this is four syllables longer, and does not express the idea of extending hypertext.

—— Theodor Holm Nelson [Literary Machines](#)↗, 1992

以上定义描述的“超文本（或超媒体）”是一种“能够对操作进行判断和响应的文本（或声音、图像等）”，这个概念在上世纪60年代提出时应该还属于科幻的范畴，但是今天大众已经完全接受了它，互联网中一段文字可以点击、可以触发脚本执行、可以调用服务端，这一切已稀松平常，毫不稀奇。那我们继续尝试从“超文本”或者“超媒体”的含义来理解什么是“表征”以及REST中其他关键概念，笔者使用一个具体事例来将其描述如下：

- **资源**（Resource）：譬如你现在正在阅读一篇名为《REST服务设计风格》的文章，这篇文章中的内容本身（你将其视作是某种信息、数据）我们称之为“资源”。无论你是在

网上看的网页、是打印出来看的文字稿、是在电脑屏幕上阅读抑或是手机上浏览，尽管呈现的样子各不相同，但其中的信息是不变的，你所阅读的仍是同一个“资源”。

- **表征 (Representation)**：当你通过电脑浏览器阅读此文章时，浏览器向服务端发出请求“我需要这个资源的HTML格式”，服务端向浏览器返回的这个HTML就被称为“表征”，你可能通过其他方式拿到本文的PDF、Markdown、RSS等其他形式的版本，它们也同样是一个资源的多种表征。可见“表征”这个概念是指信息与用户交互时的表示形式，这与我们应用分层中常说的“表示层”（Presentation Layer）的语义其实是一致的。
- **状态 (State)**：当你把这篇文章阅读完毕，想看下一篇是什么内容的时候，你向服务器请求“给我下一篇”，但是“下一篇”是个相对概念，必须依赖“当前你正在阅读的文章是哪一篇”才能正确回应，这类在特定语境中才能产生的上下文信息即被称为“状态”。我们所说的有状态（Stateful）还是无状态（Stateless），都是只相对于服务端来说的，服务器要完成“取下一篇”的请求，要么自己记住用户的状态（这个用户现在阅读的是哪一篇文章，这是有状态），要么客户端来记住状态，在请求的时候明确告诉服务器（我正在阅读某某文章，现在要读下一篇，这是无状态）。
- **转移 (Transfer)**：无论状态是由服务端还是客户端来提供的，“取下一篇”这个行为逻辑必然只能由服务端来提供。服务器通过某种方式，把“用户当前阅读的文章”转变成“下一篇”，这就被称为“**表征状态转移**”

借着这个故事的上下文，笔者顺便再介绍几个现在不涉及但稍后要用到的概念名词：

- **统一接口 (Uniform Interface)**：上面说的“服务器通过某种方式”具体是什么方式？请把本文拉到结尾处，右下角有下一篇的URI超链接地址，这是服务端渲染这篇文章时就预置好的，点击它让页面跳转到下一篇，就是一种所谓的“某种方式”。但URI的含义是统一资源标识符，如何能表达出“转移”的含义呢？HTTP协议中提前约定好了一套“统一接口”，包括：GET、HEAD、POST、PUT、DELETE、TRACE、OPTIONS七种操作，任何一个支持HTTP协议的服务器都会遵守这套规定，对特定的URI采取这些操作，服务器自然就会触发相应的表征状态转移。
- **超文本驱动 (Hypertext Driven)**：尽管表征状态转移是由浏览器主动向服务器发出请求，该请求导致了“在我们浏览器的屏幕上显示出了下一篇的内容”这个结果的出现，但浏览器其实根本不知道系统中这套转移逻辑。它根据是用户输入的URI地址请求网站首页，服务器给予的首页超文本内容，我们是通过内部的超链接导航到了这篇文章，阅读结束时再导航到下一篇。浏览器作为所有网站的通用的客户端，任何网站的导

航（状态转移）行为都是不可能预置于浏览器之中，而是由服务器每一个请求中的返回信息（超文本）来驱动的。这点大家习以为常，但其实与其他带有客户端的软件有很本质的区别，在那些软件中，业务逻辑往往是预置于客户端之中的，有专门的页面控制器（无论在服务端还是在客户端中）来驱动页面的状态转移。

- **自描述消息**（Self-Descriptive Messages）：由于资源的表征可能存在多种不同形态，在消息中应当有明确的信息来告知客户端该消息的类型以及该如何处理这条消息。一种被广泛采用的自描述方法是在名为“Content-Type”的HTTP Header中标识出[互联网媒体类型](#)（MIME type），譬如“Content-Type : application/json; charset=utf-8”，则说明该资源会以JSON的格式来返回，请使用UTF-8字符集进行处理。

RESTful的系统

当你理解了上面这些概念之后，我们就可以开始讨论面向资源的编程思想与REST所提出的几个具体的软件架构设计原则了。请注意，Fielding提出REST时所谈论的范围是“架构风格与网络的软件架构设计”（Architectural Styles and Design of Network-based Software Architectures），而不是现在被人们所狭义理解的一种“服务（API）设计风格”，这两者的范围差别就好比本站全站所谈论的话题“现代软件架构探索”与本篇文章谈论的“服务设计风格”一般，前者是后者的一个很大的超集，尽管基于本文的主题和多数人的关注点考虑，后面还是会着重以“服务设计风格”的视角来讨论，但事前我们至少应该知道它们范围上的差别。

Fielding认为，一套理想的、完全满足REST的系统应该满足以下六个原则：

1. 服务端与客户端分离（Client-Server）

将用户界面所关注的逻辑和数据存储所关注的逻辑分离开来有助于提高用户界面的跨平台的可移植性，这一点正越来越受到广大开发者所认可，以前完全基于服务端控制和渲染（如JSF这类）框架实际用户已甚少，而在服务端进行界面控制（Controller），通过服务端或者客户端的模版渲染引擎来进行界面渲染的框架（如Struts、SpringMVC这类）也受到了颇大的冲击。这一点主要推动力量与REST可能关系并不大，前端技术（从ES规范，到语言实现，到前端框架等）的近年来的高速发展，使得前端表达能力大幅度加强才是真正的幕后推手。由于前端的日渐强势，现在还流行起由前端代码反过来驱动服务端进行渲染的SSR（Server-Side Rendering）技术，在Serverless、SEO等场景中已经占领了一块领地。

2. 无状态 (Stateless)

这是REST的一条关键原则，部分开发者在做服务接口规划时，觉得RESTful风格的API怎么设计都别扭，很有可能的一种原因是在服务端持有着比较重的状态。REST希望服务器能不负责维护状态，每一次从客户端发送的请求中，应包括所有的必要的上下文信息，会话信息也由客户端保存维护，服务器端依据客户端传递的状态信息来行业务处理，并且驱动整个应用的状态变迁。至于客户端承担状态维护职责后的认证、授权等各方面的可信问题，都会有针对性的解决方案（这部分可参见[安全架构](#)中的介绍）

但必须承认的现状是，目前大多数的系统是达不到这个要求的，越复杂、越大型的系统越是如此。服务端无状态可以在分布式环境中获得非常高价值的好处，但大型系统的上下文状态数量完全可能膨胀到让客户端在每次请求时提供变得不切实际的程度，在服务端的内存、会话、数据库或者集中式缓存等地方持有一定的状态成为一种是事实上仍然被广泛使用的主流的方案。

3. 可缓存 (Cacheability)

无状态服务虽然提升了系统的可见性、可靠性和可伸缩性，但降低了系统的网络性。这句话通俗的解释就是，某个功能使用有状态的架构只需要一次请求就能完成，而无状态的服务则可能会需要多个请求，或者在请求中带有冗余的信息。为了缓解这个矛盾，REST希望软件系统能够如同万维网一样，客户端和中间的通讯传递者（代理）可以将部分服务端的应答缓存起来。当然，应答中必须明确地或者间接地表明本身是否可以进行缓存，以避免客户端在将来进行请求的时候得到过时的数据。运作良好的缓存机制可以减少客户端、服务器之间的交互，甚至有些场景中可以完全避免交互，这就进一步提了高性能。

4. 分层系统 (Layered System)

这里所指的并不是表示层、服务层、持久层这种意义上的应用分层。而是指客户端一般不需要知道是否直接连接到了最终的服务器，抑或是路径上的中间服务器。中间服务器可以通过负载均衡和共享缓存的机制提高系统的可扩展性，这样也可便于缓存、伸缩和安全策略的部署。譬如，一种典型的应用是内容分发网络（CDN），如你现在访问这个站点，你所发出的请求一般（假设你在中国国境内的话）并不是直接访问位于GitHub Pages的源服务器，而是访问了位于腾讯云的CDN，但你并不需要感知到这一点。我们将在“[透明多级分流系统](#)”中讨论如何构建可缓存的分层系统。

5. 统一接口 (Uniform Interface)

这是REST的另一条关键原则，REST希望开发者面向资源编程，希望设计软件系统的核 心放在抽象系统该有哪些资源，而不是抽象系统该有哪些行为（服务）。对资源的操作是可数的、固定的、统一的，由于REST并没有设计新的协议，所以这些操作都借用了H

HTTP协议中固有的操作命令来完成。

这一点也是REST最容易陷入争论的地方，基于网络的软件系统，到底是面向资源更好，还是面向服务更好，这事情哪怕到了今天仍然是没有个定论，也许永远都没有。但是，有一个基本清晰的结论是，面向资源编程的抽象程度通常更高，这意味着坏处是往往距离人类的思维方式更远，而好处是往往通用程度会更好。这样诠释REST大概本身就挺抽象的，笔者还是举个例子来说明：譬如几乎每个系统都有的登录和注销功能，如果你理解成登录对应于login()服务，注销对应于logout()服务这样两个服务，这是“符合人类思维”的；如果你理解成登录是CREATE Session，注销是REMOVE Session，这样你只需要设计一种“Session资源”即可满足需求，甚至以后对Session的其他需求，譬如查询或者修改登陆用户的信息，都可以在这一套设计中囊括在内，这便是“抽象程度更高”带来的好处。

想要在架构设计中合理恰当地利用统一接口，Fielding建议系统应能做到每次请求中都包含资源的ID，所有操作均通过资源ID来进行；建议每个资源都应该是自描述的消息；建议通过超文本来驱动应用状态的转移。

6. 按需代码 (Code-On-Demand)

这被Fielding列为一条可选原则。按需代码指任何按照客户端软件（譬如浏览器）的请求，将可执行的软件程序从服务器计算机发送到客户端的技术。这是可选的原因并非是它特别难以达到，而更多是出于必要性和性价比的考虑。举个例子，譬如你使用Element-UI组件库开发一个Web应用，但其实只用了里面一两个组件，却没有仔细配置好babel-plugin-component来做按需引入，一下子把几十个组件都打包进脚本中，这便是没有贯彻好按需代码的原则。这类事情（引入一个类库可能只使用其中很少量的一部分代码）是相当普遍的，我个人并不赞成不考虑实际场景的唯性能论，在关键场景肯定要抠细节，但所有场景都无限度的“精益求精”并无必要。

REST的基本思想是面向资源来抽象问题，它与此前流行的编程思想——面向过程的编程在抽象主体上有本质的差别。在REST提出以前，人们设计分布式系统服务的唯一方案就只有RPC，RPC是将本地的方法调用思路迁移到远程方法调用上，开发者是围绕着“方法”去设计服务的，譬如CORBA、RMI、DCOM，等等。这样做的坏处不仅是“如何在异构系统间表示一个方法”、“如何获得接口能够提供的方法清单”都成了需要专门协议去解决的问题（RPC的三大基本问题之一），更在于服务的每个方法都是不同的，服务使用者必须逐个学习才能正确地使用它们。Google在《Google API Design Guide》中曾经写下这样一段话：

Traditionally, people design RPC APIs in terms of API interfaces and methods, such as CORBA and Windows COM. As time goes by, more and more interfaces and methods are introduced. The end result can be an overwhelming number of interfaces and methods, each of them different from the others. Developers have to learn each one carefully in order to use it correctly, which can be both time consuming and error prone

以前，人们面向方法去设计RPC API，譬如CORBA和DCOM，随着时间推移，接口与方法越来越多却又各不相同，开发人员必须了解每一个方法才能正确使用它们，这样既耗时又容易出错。

—— Google API Design Guide, 2017

REST提出以资源为主体进行服务设计的风格，为它带来不少好处（自然也有坏处，笔者将在最后谈论REST的不足与争议），譬如：

- 降低的服务接口的学习成本。统一接口（Uniform Interface）是REST的重要标志，将对资源的标准操作都映射到了标准的HTTP方法上去，这些方法对每个资源的语义都是一致的，不需要刻意学习，更不会有什麼Interface Description Language之类的协议存在。
- 资源具有层次结构。以方法为中心抽象的API，由于方法是动词，逻辑上决定了它们都是互相独立的，但以资源为中心抽象的API，由于资源是名词，天然就可以产生集合与层次结构，譬如： 用户 资源会拥有多个 消息 资源、一个 个人资料 资源、一部 购物车 资源，购物车中有多本 书籍 资源，很容易在API中构造出这些资源的集合关系、层次关系，而且是符合人们长期在单机、网络中资源管理的直觉的。相信你不需要专门阅读方法说明书，也可以知道获取用户 icyfenix 的购物车中的第2本书应该表示为：

```
GET /users/icyfenix/cart/2
```

- REST绑定于HTTP协议。面向资源编程不是必须构筑在HTTP之上，但REST是，这是优点，也是缺点。因为HTTP本来就是面向资源而设计的网络协议，纯粹只用HTTP（而不是SOAP over HTTP那样在再构筑协议）带来的好处是RPC中的Wire Protocol问题无需再多考虑了，REST将复用HTTP协议中已经定义的语义和相关基础支持来解决。HTTP协议已经有效运作了30年，其相关的技术基础设施已是千锤百炼，无比成熟。而坏处自然是，当你想去考虑那些HTTP不提供的特性时，将束手无策。

-

以上列举的面向资源的优点，并非要证明它比面向过程、面向对象更优秀。笔者只想说明它们各有所长，只想说明在互连网中，面向资源来进行交互是这30年HTTP培养出来的用户习惯。是否能够选用RESTful的API设计风格，最需要权衡的是你的需求场景、你团队的设计和开发人员是否能够适应面向资源的思想来设计软件，来编写代码。

RMM成熟度

前面我们花费大量篇幅讨论了REST的思想、概念和指导原则等理论方面的内容，在这个小节里，我们把重心放在实践上，同时把目光从整个软件架构设计聚焦到REST服务接口，以切合本节的题目“服务设计风格”，也顺带填了前面埋下的“如何评价服务是否RESTful”的坑。

《[RESTful Web APIs](#)》和《[RESTful Web Services](#)》的作者Leonard Richardson曾提出过一个衡量“服务有多么REST”的Richardson成熟度模型（[Richardson Maturity Model](#)），便于那些原本不使用REST的服务，能够逐步地导入REST。Richardson将服务接口“REST的程度”从低到高，分为0至4级：

0. The Swamp of [Plain Old XML](#)：完全不REST。另外，关于POX这说法，SOAP表示感觉有被冒犯到。
1. Resources：开始引入资源的概念。
2. HTTP Verbs：引入统一接口，映射到HTTP协议的方法上。
3. Hypermedia Controls：在本文里面的说法是“超文本驱动”，在Fielding论文里的说法是“Hypertext As The Engine Of Application State，HATEOAS”，都是指同一件事情。

我们借用Martin Fowler撰写的关于RMM成熟度模型的[文章](#)中的实际例子（原文是XML写的，我简化了一下），来实际看一下四种不同程度的REST反应到实际API是怎样的。假设你是一名软件工程师，接到需求（也被我尽量简化了）的UserStory是这样的：

医生预约系统

作为一名病人，我想要从系统中得知指定日期内我熟悉的医生是否具有空闲时间，以便于我向该医生预约就诊。

第0级

医院开放了一个/appointmentService的Web API，传入日期、医生姓名作为参数，可以得到该时间段该名医生的空闲时间，该API的一次HTTP调用如下所示：

```
POST /appointmentService?action=query HTTP/1.1  
  
{date: "2020-03-04", doctor: "mjones"}
```

然后服务器会传回一个包含了所需信息的回应：

```
HTTP/1.1 200 OK  
  
[  
  {start:"14:00", end: "14:50", doctor: "mjones"},  
  {start:"16:00", end: "16:50", doctor: "mjones"}  
]
```

得到了医生空闲的结果后，我觉得14:00的时间比较合适，于是进行预约确认，并提交了我的基本信息：

```
POST /appointmentService?action=confirm HTTP/1.1  
  
{  
  appointment: {date: "2020-03-04", start:"14:00", doctor: "mjones"},  
  patient: {name: xx, age: 30, .....}  
}
```

如果预约成功，那我能够收到一个预约成功的响应：

```
HTTP/1.1 200 OK  
  
{  
  code: 0,
```

```
        message: "Successful confirmation of appointment"  
    }
```

如果发生了问题，譬如有人在我前面抢先预约了，那么我会在响应中收到某种错误信息：

```
HTTP/1.1 200 OK  
  
{  
    code: 1  
    message: "doctor not available"  
}
```

到此，整个预约服务宣告完成，直接明了，我们采用的是非常直观的基于RPC风格的服务设计似乎很容易就解决了所有问题……吗？

第1级

通往REST的第一步是引入资源的概念，在API中基本的体现是围绕着资源而不是过程来设计服务，说的直白一点，可以理解为服务的Endpoint应该是一个名词而不是动词。此外，每次请求中都应包含资源的ID，所有操作均通过资源ID来进行。

```
POST /doctors/mjones HTTP/1.1  
  
{date: "2020-03-04"}
```

然后服务器传回一个包含了ID信息，注意，ID是资源的唯一编号，有ID即代表“医生的档期”被视为一种资源：

```
HTTP/1.1 200 OK  
  
[  
    {id: 1234, start:"14:00", end: "14:50", doctor: "mjones"},  
    {id: 5678, start:"16:00", end: "16:50", doctor: "mjones"}  
]
```

我还是觉得14:00的时间比较合适，于是又进行预约确认，并提交了我的基本信息：

```
POST /schedules/1234 HTTP/1.1
```

```
{name: xx, age: 30, ....}
```

后面预约成功或者失败的响应消息在这个级别里面与之前一致，就不重复了。比起第0级，第1级的服务抽象程度有所提高，但至少还有三个问题并没有解决，一是只处理了查询和预约，如果我临时想换个时间，要调整预约，或者我的病忽然好了，想删除预约，这都需要提供新的服务接口。二是处理结果响应时，只能靠着结果中的code、message这些字段做分支判断，每一套服务都要设计可能发生错误的code，这很难考虑全面，而且也不利于对某些通用的错误做统一处理；三是并没有考虑认证授权等安全方面的内容，譬如要求只有登陆用户才允许查询医生档期时间，某些医生可能只对VIP开放，需要特定级别的病人才能预约等等。

第2级

第1级遗留三个问题都可以靠引入统一接口来解决。HTTP协议的七个标准方法是经过精心设计的，几乎能涵盖资源可能遇到的所有操作场景（这其实更取决于架构师的抽象能力）。REST的做法是把不同业务需求抽象为对资源的增加、修改、删除等操作来解决第一个问题；使用HTTP协议的Status Code，可以涵盖大多数资源操作可能出现的异常（而且也是可以自定义扩展的），以此解决第二个问题；依靠HTTP Header中携带的额外认证、授权信息来解决第三个问题（这个在实战中并没有体现，请参考安全架构中的“[凭证](#)”相关内容）。

按这个思路，获取医生档期，应采用具有查询语义的GET操作进行：

```
GET /doctors/mjones/schedule?date=2020-03-04&status=open HTTP/1.1
```

然后服务器会传回一个包含了所需信息的回应：

```
HTTP/1.1 200 OK
```

```
[
```

```
 {id: 1234, start:"14:00", end: "14:50", doctor: "mjones"},
```

```
{id: 5678, start:"16:00", end: "16:50", doctor: "mjones"}  
]
```

我仍然觉得14:00的时间比较合适，于是双进行预约确认，并提交了我的基本信息，用以创建预约，这是符合POST的语义的：

```
POST /schedules/1234 HTTP/1.1  
  
{name: xx, age: 30, .....}
```

如果预约成功，那我能够收到一个预约成功的响应：

```
HTTP/1.1 201 Created  
  
Successful confirmation of appointment
```

如果发生了问题，譬如有人在我前面抢先预约了，那么我会在响应中收到某种错误信息：

```
HTTP/1.1 409 Conflict  
  
doctor not available
```

第3级

第2级是目前绝大多数系统所到达的REST级别，但仍不是不够完美的，至少还存在一个问题：你是如何知道预约mjones医生的档期是需要访问“/schedules/1234”这个服务Endpoint的？也许你甚至第一时间无法理解为何我会有这样的疑问，这当然是程序代码写的呀！但REST并不认同这种已烙在程序员脑海中许久的想法。RMM中的Hypermedia Controls、Fielding论文中的HATEOAS和现在提的比较多的“超文本驱动”，所希望的是除了第一个请求是有你在浏览器地址栏输入所驱动之外，其他的请求都应该能够自描述清楚后续可能发生的状态转移，由超文本自身来驱动。所以，当你输入了查询的指令之后：

```
GET /doctors/mjones/schedule?date=2020-03-04&status=open HTTP/1.1
```

服务器传回的响应信息应该包括诸如如何预约档期、如何了解医生信息等可能的后续操作：

```
HTTP/1.1 200 OK

{
  schedules : [
    {
      id: 1234, start:"14:00", end: "14:50", doctor: "mjones",
      links: [
        {rel: "comfirm schedule", href: "/schedules/1234"}
      ]
    },
    {
      id: 5678, start:"16:00", end: "16:50", doctor: "mjones",
      links: [
        {rel: "comfirm schedule", href: "/schedules/5678"}
      ]
    }
  ],
  links: [
    {rel: "doctor info", href: "/doctors/mjones/info"}
  ]
}
```

如果做到了第3级REST，那服务端的API和客户端也是完全解耦的，你要调整服务数量，或者同一个服务做API升级将会变得非常简单。

不足与争议

以下是笔者所见过的怀疑REST能否在实践中真正良好应用的争议问题，笔者将自己的观点总结如下：

- **面向资源的编程思想只适合做CRUD，只有面向过程、面向对象编程才能处理真正复杂的业务逻辑**

这是遇到最多的一个问题。HTTP的四个最基础的命令POST、GET、PUT和DELETE很容易让人直接联想到CRUD操作，以至于在脑海中自然产生了直接的对应。REST所能涵盖的范围当然远不止于此，不过要说POST、GET、PUT和DELETE对应于CRUD其

实也没什么不对，只是这个CRUD必须泛化去理解，它们涵盖了信息在客户端与服务端之间如何流动的几种主要方式，所有基于网络的操作逻辑，都可以对应到信息在服务端与客户端之间如何流动来理解，有的场景里比较直观，而另一些场景中可能比较抽象。针对那些比较抽象的场景，如果真不好把HTTP方法映射为资源的所需操作，REST也并非刻板的教条，用户是可以使用自定义方法的，按Google推荐的REST API风格，[自定义方法](#)应该放在资源路径末尾，嵌入冒号加自定义动词的后缀。譬如，我将删除操作映射到标准DELETE方法上，此外还要提供一个 `恢复删除` 的API，那它可能会被设计为：

```
POST /user/user_id/cart/book_id:undelete
```

如果你设计一个 `回收站` 的资源，在那里保留着还能被恢复的商品，将 `恢复删除` 视为对该资源某个状态值的修改，映射到 `PUT` 或者 `PATCH` 方法上，这也是一种完全可行的设计。

最后，笔者再重复一遍，面向资源的编程思想与另外两种主流编程思想只是抽象问题时所处的立场不同，只有选择问题，没有高下之分：

- 面向过程编程时，为什么要以算法和处理过程为中心，输入数据，输出结果？当然是为了符合计算机世界中主流的交互方式。
 - 面向对象编程时，为什么要将数据和行为统一起来、封装成对象？当然是为了符合现实世界的主流的交互方式。
 - 面向资源编程时，为什么要将数据（资源）作为抽象的主体，把行为看作是统一的接口？当然是为了符合网络世界的主流的交互方式。
- **REST与HTTP完全绑定，不适合应用于要求高性能传输的场景中**

我个人很大程度上赞同此观点，但并不认为这是REST的缺陷，锤子不能当扳手用并不是锤子的质量有问题。面向资源编程与协议无关，但是REST（特指Fielding论文中所定义的REST，而不是泛指面向资源的思想）的确依赖着HTTP协议的标准方法、状态码、协议头等各个方面。HTTP并不是传输层协议，它是应用层协议，如果仅将HTTP当作传输是不恰当的（SOAP：再次感觉有被冒犯到）。对于需要直接控制传输（如二进制细节/编码形式/报文格式/连接方式等）细节的场景中，REST确实不合适，这些场景往往存在于服务集群的内部节点之间，这也是之前我曾提及的，REST和RPC尽管应用确有所重合，但重合的范围有多大就是见仁见智的事情。

- REST不利于事务支持

这个问题首先要看你怎么看待“事务（Transaction）”这个概念。如果“事务”指的是数据库那种的狭义的刚性ACID事务，那分布式系统本身与此就是有矛盾的（CAP不可兼得），这是分布式的问题而不是REST的问题。如果“事务”是指通过服务协议或架构，在分布式服务中，获得对多个数据同时提交的统一协调能力（2PC/3PC），譬如[WS-AtomicTransaction](#)、[WS-Coordination](#)这样的功能性协议，这REST确实不支持，假如你已经理解了这样做的代价，仍决定要这样做的话，Web Service是比较好的选择。如果“事务”是指希望保障数据的最终一致性，说明你已经放弃刚性事务了，这才是分布式系统中的主流，使用REST肯定不会有阻碍，谈不上“不利于”（当然，对此REST也并没有什么帮助，这完全取决于你系统的事务设计，我们在[事务处理](#)中再详细讨论）

- REST没有传输可靠性支持

是的，并没有。在HTTP中你发送出去一个请求，通常会收到一个与之相对的响应，譬如HTTP/1.1 200 OK或者HTTP/1.1 404 Not Found诸如此类的。但如果你没有收到任何响应，那就无法确定消息到底是没有发送出去，抑或是没有从服务端返回回来，这其中的关键差别是服务端到底是否被触发了某些处理？应对传输可靠性最简单粗暴的做法是把消息再重发一遍。这种简单处理能够成立的前提是服务应具有[幂等性](#)（Idempotency），即服务被重复执行多次的效果与执行一次是相等的。HTTP协议要求GET、PUT和DELETE应具有幂等性，我们把REST服务映射到这些方法时，也应当保证幂等性。对于POST方法，曾经有过一些专门的提案（如[POE](#)，POST Once Exactly），但并未得到IETF的通过。对于POST的重复提交，浏览器会出现相应警告，如Chrome中“确认重新提交表单”的提示，对于服务端，就应该做预校验，如果发现可能重复，返回HTTP/1.1 425 Too Early。另，Web Service中有[WS-ReliableMessaging](#)功能协议用于支持消息可靠投递。类似的，由于REST没有采用额外的Wire Protocol，所以不仅是事务、可靠传输这些，一定还可以在WS-*协议中找到很多REST不支持的特性。

- REST缺乏对资源进行“部分”和“批量”的处理能力

这个观点我是认同的，我认为这很可能是未来面向资源的思想和API设计风格的发展方向。REST开创了面向资源的服务风格，却肯定仍并不完美。以HTTP协议为基础给REST带来了极大的便捷（不需要额外协议，不需要重复解决一堆基础网络问题，等等），但也是HTTP本身成了束缚REST的无形牢笼。我仍通过具体例子来解释REST这方面的局限性：譬如你仅仅想获得某个用户的姓名，RPC风格中可以设计一个“getUsernameByUserId”的服务，返回一个字符串，尽管这种服务的通用性实在称不上“设计”二字，但确实可

以工作；而REST风格中你将向服务端请求整个用户对象，然后丢弃掉返回的结果中该用户的其他属性，这便是一种Overfetching。REST的应对手段是通过位于中间节点或客户端缓存来缓解这种问题，但此缺陷的本质是由于HTTP协议完全没有对请求资源的结构化描述能力（但有非结构化的部分内容获取能力，即今天多用于端点续传的[Range Header](#)），所以返回资源的哪些内容、以什么数据类型返回等等，都不可能得到协议层面的支持，要做你就只能自己在GET方法的Endpoint上设计各种参数来实现。而另外一方面于此相对的缺陷是对资源的批量操作的支持，有时候我们不得不为此而专门设计一些抽象的资源才能应对。譬如你准备把某个用户的名字增加一个“VIP”前缀，提交一个PUT请求修改这个用户的名称即可，而你要给1000个用户加VIP时，就不得不先创建一个（如名为“VIP-Modify-Task”）任务资源，把1000个用户的ID交给这个任务，最后驱动任务进入执行状态（真去调用1000次PUT，浏览器会回应你HTTP/1.1 429 Too Many Requests，老板则会揍你一顿）。又譬如你去网店买东西，下单、冻结库存、支付、加积分、扣减库存这一系列步骤会涉及到多个资源的变化，你可能面临不得不创建一种“事务”的抽象资源，或者用某种具体的资源（譬如“结算单”）贯穿这个过程的始终，每次操作其他资源时都带着事务或者结算单的ID。HTTP协议由于本身的无状态性，会相对不适应（并非不能够）处理这类业务场景。

解决这类问题，目前看起来一种理论上较优秀的解决方案是[GraphQL](#)，这是由Facebook提出并开源的一种面向资源API的数据查询语言（如同SQL一样，挂了个“查询语言”的名字，但CRUD都做）。比起依赖HTTP无协议的REST，GraphQL可以说是另一种“有协议”的、更彻底的面向资源的服务方式。然而凡事都有两面，离开了HTTP，它又面临着几乎所有RPC框架所遇到的那个如何推广交互接口的问题。

事务处理

一致性 (Consistency)

一致性在数据科学中是有严肃定义、且有多种细分类型的概念，在“分布式的基石”讨论分布式共识算法时说的一致性，与这里的数据库状态的一致性严格来说并不能直接等同，具体差别我们将在分布式共识算法中继续探讨。

事务处理几乎是每一个信息系统中都会涉及到的问题，它存在的意义就是为了保证系统中数据是正确的，不同数据间不会产生矛盾，即数据状态的一致性 (Consistency)。

理论上，达成这个目标需要三方面共同努力来保障：

- **原子性 (Atomic)**：在同一项业务处理过程中，事务保证了多个对数据的修改，要么同时成功，要么一起被撤销。
- **隔离性 (Isolation)**：在不同的业务处理过程中，事务保证了各自业务正在读、写的数据互相独立，不会彼此影响。
- **持久性 (Durability)**：事务应当保证所有成功被提交的数据修改都能够正确地被持久化，不丢失数据。

以上即事务的“ACID”的概念提法，笔者自己对这种已经形成习惯的“ACID”的提法是不太认同的，上述四种特性并不正交，A、I、D是手段，C是目的，完全是为了拼凑个单词缩写才弄到一块去，误导的弊端已经超过了易于传播的好处。

事务的概念最初是源于数据库，但今天的信息系统中已经不再局限于数据库本身，所有需要保证数据正确性（一致性）的场景中，包括但不限于数据库、缓存、[事务内存](#)、消息、队列、对象文件存储，等等，都有可能会涉及到事务处理。当一个服务只操作一个数据源时，通过A、I、D来获得一致性是相对容易的，但当一个服务涉及到多个不同的数据源，甚至多个不同服务同时涉及到多个不同的数据源时，这件事情就变得很困难，有时需要付出很大乃至是不切实际的代价，因此业界探索过许多其他方案，在确保可操作的前

提下获得尽可能高的一致性保障，事务处理由此才从一个具体操作上的“编程问题”上升成一个需要仔细权衡的“架构问题”。

人们在探索这些事务方案的过程中，产生了许多新的思路和概念，有一些概念看上去并不那么直观，在本章里，笔者会通过同一个具体事例在不同的事务方案中如何处理来贯穿、理顺这些概念。

场景事例

Fenix's Bookstore是一个在线书店。当一份商品成功售出时，需要确保以下三件事情被正确地处理：

- 用户的账号扣减相应的商品款项
- 商品仓库中扣减库存，将商品标识为待配送状态
- 商家的账号增加相应的商品款项

接下来，笔者将逐一介绍在“单个服务使用单个数据源”、“单个服务使用多个数据源”、“多个服务使用单个数据源”以及“多个服务使用多个数据源”的不同场景下，我们可以采用哪些手段来保证以上场景实例的正确性。

本地事务

本地事务（Local Transactions）其实应该翻译成“局部事务”才好与稍后的“全局事务”相对应，不过现在“本地事务”的译法似乎已经成为主流，笔者就不去纠结名称了。本地事务是指仅操作特定单一事务资源的、不需要“全局事务管理器”进行协调的事务，如果这个定义现在不能理解的话，不妨暂且先放下，等读完下一节“全局事务”后且回过头来想一想，对比一下。

本地事务是最基础的一种事务处理方案，通常只适用于单个服务使用单个数据源的场景，它是直接依赖于数据源（典型如数据库系统）本身的事务能力来工作的，在程序代码层面，最多只能对事务接口做了一层标准化的包装（如JDBC接口），并不能深入参与到事务的运作过程当中，事务的开启、终止、提交、回滚、嵌套、设置隔离级别、乃至与应用代码贴近的传播方式，全部都要依赖底层数据库的支持，这一点与后续介绍的XA、TCC、Saga等主要靠应用程序代码来实现的事务有着十分明显的区别。举个具体的例子，假设你的代码调用了JDBC中的Transaction::rollback()方法，方法的成功执行并不代表事务就已经被成功回滚，如果数据表采用引擎的是MyISAM¹，那rollback()方法便是一项没有意义空操作。因此，我们要想深入地讨论本地事务，便不得不越过应用代码的层次，去了解一些数据库本身的事务实现原理，弄明白传统数据库管理系统是如何实现ACID的。

如今研究事务的实现原理，必定会追溯到ARIES²理论（Algorithms for Recovery and Isolation Exploiting Semantics，翻译过来是“基于语义的恢复与隔离算法”，起这拗口的名字应该多少也有些拼凑“ARIES”这单词目的，跟ACID一样地恶趣味）。不能说所有的数据库都实现了ARIES理论，但现代的主流关系型数据库（Oracle、MS SQLServer、MySQL-InnoDB、IBM DB2、PostgreSQL，等等）在事务实现上都深受该理论的影响。

上世纪90年代，IBM Almaden研究院³总结了研发原型数据库系统“IBM System R”的经验，发表了ARIES理论中最主要的三篇论文，其中《ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging⁴》着重解决了事务的ACID的两个属性：原子性（A）和持久性（D）在算法层面上应当如何实现。而另一篇《ARIES/KVL: A Key-Value Locking Method for Concurrency Control

of Multiaction Transactions Operating on B-Tree Indexes» 则是现代数据库隔离性（I）奠基式的文章，我们先从原子性和持久性说起。

实现原子性和持久性

原子性和持久性在事务里是密切相关的两个属性，原子性保证了事务的多个操作要么都生效要么都不生效，不会存在中间状态；持久性保证了一旦事务生效，就不会再因为任何原因而导致其修改的内容被撤销或丢失。显而易见，数据必须要成功写入磁盘、磁带等持久化存储器后才能拥有持久性，只存储在内存中的数据，一旦遇到程序忽然崩溃、数据库崩溃、操作系统崩溃，机器突然断电宕机（后文我们都统称为 崩溃，Crash）等情况就会丢失。实现原子性和持久性所面临的困难是“写入磁盘”这个操作不会是原子的，不仅有“写入”与“未写入”，还客观地存在着“正在写”的中间状态。

按照“[事务处理](#)”里列出示例场景，从Fenix's Bookstore购买一本书需要修改三个数据：在用户账户中减去货款、在商家账户中增加货款、在商品仓库中标记一本书为配送状态，由于写入存在中间状态，可能发生以下情形：

- **未提交事务，写入后崩溃**：程序还没修改完三个数据，数据库已经将其中一个或两个数据的变动写入了磁盘，此时出现崩溃，一旦重启之后，数据库必须要有办法得知崩溃前发生过一次不完整的购物操作，将已经修改过的数据从磁盘中恢复成没有改过的样子，以保证原子性。
- **已提交事务，写入前崩溃**：程序已经修改完三个数据，数据库还未将全部三个数据的变动都写入到磁盘，此时出现崩溃，一旦重启之后，数据库必须要有办法得知崩溃前发生过一次完整的购物操作，将还没来得及写入磁盘的那部分数据重新写入，以保证持久性。

这种数据恢复操作被称为 崩溃恢复（Crash Recovery，也有称作Failure Recovery或Transaction Recovery），为了能够顺利地完成崩溃恢复，在磁盘中写数据就不能像程序修改内存中变量值那样，直接改变某表某行某列的某个值，必须将修改数据这个操作所需的所有信息（譬如修改什么数据、数据物理上位于哪个内存页和磁盘块中、从什么值改成什么值，等等），以日志的形式（日志特指仅进行顺序追加的文件写入方式，这是最高效的写入方式）先记录到磁盘中。只有在日志记录全部都安全落盘，见到代表事务成功提交的“Commit Record”后，数据库才会根据日志上的信息对真正的数据进行修改，修改完成后，在

日志中加入一条“End Record”表示事务已完成持久化，这种事务实现方法被称为“Commit Logging”。

额外知识：Shadow Paging

通过日志实现事务的原子性和持久性是当今的主流方案，但并非唯一的选择。除日志外，还有另外一种称为“Shadow Paging”（有中文资料翻译为“影子分页”）的事务实现机制，常用的轻量级数据库SQLite Version 3采用的就是Shadow Paging。

Shadow Paging的大体思路是对数据的变动会写到硬盘的数据中，但并不是直接就地修改原先的数据，而是先将数据复制一份副本，保留原数据，修改副本数据。在事务过程中，被修改的数据会同时存在两份，一份修改前的数据，一份是修改后的数据，这也是“影子”（Shadow）这个名字的由来。当事务成功提交，所有数据的修改都成功持久化之后，最后一步要修改数据的引用指针，将引用从原数据改为新复制出来修改后的副本，最后的“修改指针”这个操作将被认为是原子操作，所以Shadow Paging也可以保证原子性和持久性。Shadow Paging相对简单，但涉及到隔离性与锁时，Shadow Paging实现的事务并发能力相对有限，因此在高性能的数据库中应用不多。

Commit Logging保障数据持久性、原子性的原理并不难想明白：首先，日志一旦成功写入 Commit Record，那整个事务就是成功的，即使修改数据时崩溃了，重启后根据已经写入磁盘的日志信息恢复现场、继续修改数据即可，这保证了持久性。其次，如果日志没有写入成功就发生崩溃，系统重启后会看到一部分没有Commit Record的日志，那将这部分日志标记为回滚状态即可，整个事务就像完全没好有发生过一样，这保证了原子性。

Commit Logging实现事务简单清晰，也有一些数据库就是采用Commit Logging机制来实现事务的（较具代表性的是阿里的OceanBase）。但是，Commit Logging存在一个巨大的缺陷：所有对数据的真实修改都必须发生在事务提交、日志写入了Commit Record之后，即使事务提交前磁盘I/O有足够的空闲、即使某个事务修改的数据量非常庞大，占用大量的内存缓冲，无论何种理由，都决不允许在事务提交之前就开始修改磁盘上的数据，这一点对提升数据库的性能是很不利的。为了解决这个缺陷，前面提到的ARIES理论终于可以登场，ARIES提出了“Write-Ahead Logging”的日志改进方案，名字里所谓的“提前写入”（Write-Ahead），就是允许在事务提交之前，提前写入变动数据的意思。

Write-Ahead Logging先将何时写入变动数据，按照事务提交时点为界，分为了FORCE和STEAL两类：

- **FORCE**：当事务提交后，要求变动数据必须同时完成写入则称为FORCE，如果不强制变动数据必须同时完成写入则称为NO-FORCE。现实中绝大多数数据库采用的都是NO-FORCE策略，只要有了日志，变动数据随时可以持久化，从优化磁盘I/O性能考虑，没有必要强制数据写入立即进行。
- **STEAL**：在事务提交前，允许变动数据提前写入则称为STEAL，不允许则称为NO-STEAL。从优化磁盘I/O性能考虑，允许数据提前写入，有利于利用空闲I/O资源，也有利于节省数据库缓存区的内存。

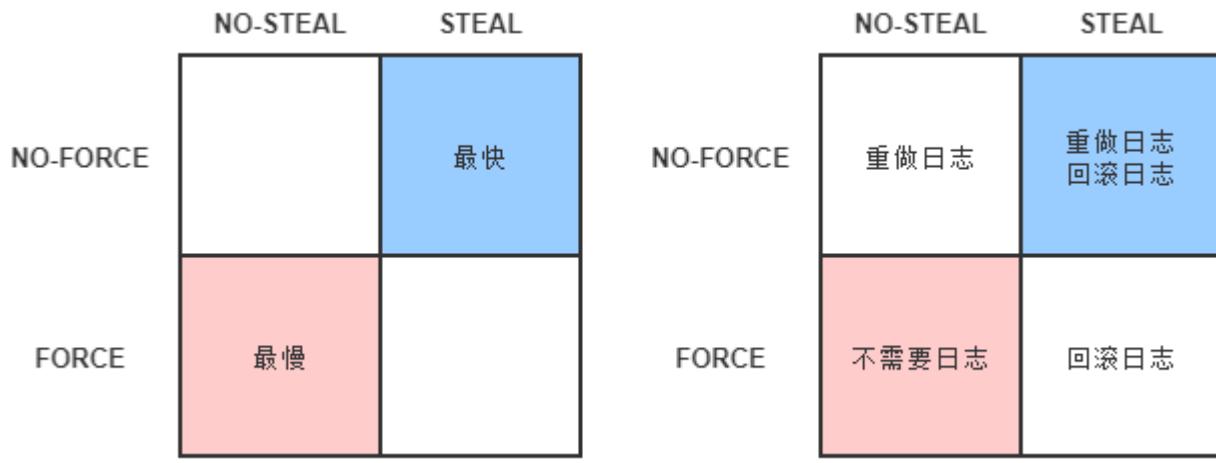
Commit Logging允许NO-FORCE，但不允许STEAL。因为假如事务提交前就有部分变动数据写入磁盘，那一旦事务要回滚，或者发生了崩溃，这些提前写入的变动数据就都成了错误。

Write-Ahead Logging允许NO-FORCE，也允许STEAL，它给出的解决办法是增加了另一种称为Undo Log的日志，当变动数据写入磁盘前，必须先记录Undo Log，写明修改哪个位置的数据、从什么值改成什么值，以便在事务回滚或者崩溃恢复时根据Undo Log对提前写入的数据变动进行擦除。Undo Log现在一般被翻译为“回滚日志”，此前记录的用于崩溃恢复时重演数据变动的日志就相应被命名为Redo Log，一般翻译为“重做日志”。由于Undo Log的加入，Write-Ahead Logging在崩溃恢复时会以此经历以下三个阶段：

- **分析阶段（Analysis）**：该阶段从最后一次检查点（Checkpoint，可理解为在这个点之前所有应该持久化的变动都已安全落盘）开始扫描日志，找出所有没有End Record的事务，组成待恢复的事务集合（一般包括Transaction Table和Dirty Page Table）。
- **重做阶段（Redo）**：该阶段依据分析阶段中产生的待恢复的事务集合来重演历史（Repeat History），找出所有包含Commit Record的日志，将这些日志修改的数据写入磁盘，写入完成后在日志中增加一条End Record，然后移除出待恢复事务集合。
- **回滚阶段（Undo）**：该阶段处理经过分析、重做阶段后剩余的恢复事务集合，此时剩下的都是需要回滚的事务（被称为Loser），根据Undo Log中的信息回滚这些事务。

重做阶段和回滚阶段的操作都应该设计为幂等的。为了追求高性能，以上三个阶段无可避免地会涉及到非常繁琐的概念和细节（如Redo Log、Undo Log的具体数据结构等），囿于篇幅限制，笔者并没有介绍这些内容，如感兴趣，阅读开头引用的那两篇论文是最佳的途径。Write-Ahead Logging是ARIES理论的一部分，整套ARIES拥有严谨、高性能等很多的优点，但这些也是以复杂性为代价的。数据库按照是否允许FORCE和STEAL可以产生共计四种组合，从优化磁盘I/O的角度看，NO-FORCE加STEAL组合的性能无疑是最高

的；从算法实现与日志的角度看NO-FORCE加STEAL组合的复杂度无疑是最高的。这四种组合与Undo Log、Redo Log之间的具体关系如下图所示：



FORCE和STEAL的四种组合关系

实现隔离性

这节我们来探讨数据库如何实现隔离性。隔离性保证了每个事务各自读、写的数据互相独立，不会彼此影响。只从定义上就能嗅出隔离性肯定与并发密切相关，如果没有并发，所有事务全都是串行的，那就不需要任何隔离，或者说这样的访问具备了天然的隔离性。但现实情况不可能没有并发，要在并发下实现串行的数据访问该怎样做？几乎所有程序员都会回答到：加锁同步呀！现代数据库均提供了以下三种锁：

- **写锁**（Write Lock，也叫做排他锁eXclusive Lock，简写为X-Lock）：如果数据有加写锁，就只有持有写锁的事务才能对数据进行写入操作，数据加持着写锁时，其他事务不能写入数据，也不能施加读锁。
- **读锁**（Read Lock，也叫做共享锁Shared Lock，简写为S-Lock）：多个事务可以对同一个数据添加多个读锁，数据被加上读锁后就不能再被加上写锁，所以其他事务不能对该数据进行写入，但仍然可以读取。对于持有读锁的事务，如果该数据只有一个事务加了读锁，那可以直接将其升级为写锁，然后写入数据。
- **范围锁**（Range Lock）：对于某个范围直接加排他锁，在这个范围内的数据不能被写入。如下语句是典型的加范围锁的例子：

```
SELECT * FROM books WHERE price < 100 FOR UPDATE;
```

请注意“范围不能被写入”与“一批数据不能被写入”的差别，即不要把范围锁理解成一组排他锁的集合。加了范围锁后，不仅无法修改该范围内已有的数据，也不能在该范围内新增或删除任何数据，后者是一组排他锁的集合无法做到的。

串行化访问提供了强度最高的隔离性，ANSI/ISO SQL-92¹中定义的最高等级的隔离级别便是 可串行化（Serializable）。可串行化 比较符合普通程序员对数据竞争加锁的理解，如果不考虑性能优化的话，对事务所有读、写的数据全都加上读锁、写锁和范围锁即可做到 可串行化（“即可”是简化理解，实际要分成Expanding和Shrinking两阶段去处理读锁、写锁与数据间的关系，称为Two-Phase Lock²，2PL）。但数据库不考虑性能肯定是不行的，并发控制理论³（Concurrency Control）决定了隔离程度与并发能力是相互抵触的，隔离程度越高，并发访问时的吞吐量就越低。现代数据库一定会提供除 可串行化 以外的其他隔离级别供用户使用，让用户调节隔离级别的选项，根本目的是让用户可以调节数据库的加锁方式，取得隔离性与吞吐量之间的平衡。

可串行化 的下一个隔离级别是 可重复读（Repeatable Read），可重复读 对事务所涉及到的数据加读锁和写锁，且一直持有至事务结束，但不再加范围锁。可重复读 比 可串行化 弱化的地方在于幻读问题⁴（Phantom Reads）。譬如我现在准备统计一下Fenix's Bookstore中售价小于100元的书有多少本，会执行以下SQL：

```
SELECT count(1) FROM books WHERE price < 100
```

根据前面对范围锁、读锁和写锁的定义可知，假如这条SQL语句在同一个事务中重复执行了两次，这两次执行之间恰好有另外一个事务在数据库插入了一本小于100元的书籍，那这两次重复执行的结果就会不一样，原因是 可重复读 没有范围锁来禁止在该范围内插入新的数据，这是一个事务遭到其他事务影响，隔离性被破坏的表现。

可重复读 的下一个隔离级别是 读已提交（Read Committed），读已提交 对事务涉及到的数据加的写锁会一直持续到事务结束，但加的读锁在查询操作完成后就马上会释放。读已提交 比 可重复读 弱化的地方在于不可重复读问题⁵（Non-Repeatable Reads）。譬如我要统计Fenix's Bookstore中售价小于100元的书有多少本，同样执行了两条SQL语句，在此之间，恰好另外一个事务修改了其中某一本书的价格，从90元涨价到110元，那这两次重复执行的结果就会不一样，原因是 读已提交 在数据缺乏贯穿整个事务周期的读锁，无法禁止读取过的数据发生变化。这也是一个事务遭到其他事务影响，隔离性被破坏的表现。

读已提交 的下一个级别是 读未提交 （ Read Uncommitted ）， 读未提交 对事务涉及到的数据只加写锁，会一直持续到事务结束，但完全不加读锁。 读未提交 比 读已提交 弱化的地方在于**脏读问题**（ Dirty Reads ）。譬如我觉得一本书从90元随便涨价到110元是损害消费者利益的行为，执行了一条UPDATE语句把价格改回了90元，在提交事务之前，同事过来告诉我这并不是随便涨价，是之前价格标错了，按90卖要亏本，于是我随即回滚了事务。不过，在我修改价格后这本书已经按90元的价格卖出了好几本。原因是 读未提交 在数据上完全不加读锁，这反而令它能读到其他事务加了写锁的数据（如果不能理解这句话，请再读一次写锁的定义，它禁止施加读锁，而不是禁止读取数据），导致事务未提交的数据也马上就能被其他事务所读到。这同样是一个事务遭到其他事务影响，隔离性被破坏的表现。

理论上还有更低的隔离级别，就是“完全不隔离”，即读、写锁都不加。 读未提交 会有脏读问题，但不会有脏写问题（ Dirty Write ，即一个事务的没提交之前的修改可以被另外一个事务的修改覆盖掉），脏写已经不单纯是隔离性上的问题了，它将导致事务的原子性都无法实现，所以一般谈论隔离级别时不将它纳入讨论范围它，而将 读未提交 视为是最低级的隔离级别。

以上四种隔离级别属于数据库的基础知识，多数大学的计算机课程应该都会讲到，可惜的是不少教材、资料将它们当作数据库的某种固有属性或设定来讲解，这导致很多同学只能对这些现象死记硬背。其实不同隔离级别以及幻读、脏读等问题都只是表面现象，是各种锁在不同加锁时间上组合应用所产生的结果，以锁为手段来实现隔离性才是根本的原因。

除了以锁来实现外，以上对四种隔离级别介绍还有一个共同特点，就是一个事务在读数据过程中，受另外一个写数据的事务影响而破坏了隔离性，针对这种“一个事务读+另一个事务写”的隔离问题，有一种名为“**多版本并发控制**”（ Multi-Version Concurrency Control , MVCC ）的无锁优化方案被主流的商业数据库广泛采用。MVCC是一种读取优化策略，它的“无锁”是指读取时不需要加锁。MVCC的基本思路是对数据库的任何修改都不会直接覆盖之前的数据，而是产生一个新版副本与老版本共存，以此达到读取时可以完全不加锁的目的。这句话里“版本”是关键词，你不妨将版本理解为数据库中每一行记录都存在两个看不见的字段：CREATE_VERSION和DELETE_VERSION，这两个字段记录的值都是事务ID（事务ID是一个全局严格递增的数值），然后：

- 数据被插入时：CREATE_VERSION记录插入数据的事务ID，DELETE_VERSION为空。

- 数据被删除时：DELETE_VERSION记录删除数据的事务ID，CREATE_VERSION为空。
- 数据被修改时：将修改视为“删除旧数据，插入新数据”，即先将原有数据复制一份，原有数据的DELETE_VERSION记录修改数据的事务ID，CREATE_VERSION为空。复制出来的新数据的CREATE_VERSION记录修改数据的事务ID，DELETE_VERSION为空。

此时，有另外一个事务要读取这些发生了变化的数据时，根据隔离级别来决定到底应该读取哪个版本的数据：

- 隔离级别是 可重复读：总是读取CREATE_VERSION小于或等于当前事务ID的记录，在这个前提下，如果数据仍有多个版本，则取最新（事务ID最大）的。
- 隔离级别是 读已提交：总是取最新的版本即可，即最近被Commit的那个版本的数据记录。

另外两个隔离级别都没有必要用到MVCC，读未提交 直接修改原始数据即可，其他事务查看数据的时候立刻可以查看到，根本无须版本字段。可串行化 本来的语义就要阻塞其他事务的读取操作，而MVCC是做读取时无锁优化的，自然就不会放到一起用。

MVCC是只针对“读+写”场景的优化，如果是两个事务同时修改数据，即“写+写”的情况，那就没有多少优化的空间了，加锁几乎是唯一可行的解决方案，稍微有点讨论余地的是加锁的策略是“乐观加锁”（Optimistic Locking）还是“悲观加锁”（Pessimistic Locking），这点还可以根据实际情况去商量一下。前面笔者介绍的加锁都属于悲观加锁策略，即认为如果不先做加锁再访问数据，就肯定会出现问题。相对的乐观加锁策略认为事务之间数据存在竞争是偶然情况，没有竞争才是普遍情况，这样就不应该一开始就加锁，而是应当出现竞争时再找补救措施。这种思路被称为“[乐观并发控制](#)”（Optimistic Concurrency Control，OCC），囿于字数原因就不再展开了，不过笔者提醒一句，不要迷信什么乐观锁要比悲观锁更快的说法，这纯粹看竞争的剧烈程度，如果竞争剧烈的话，乐观锁反而会更慢。

全局事务

与本地事务相对的是全局事务（Global Transactions），有一些资料中也将其称为外部事务（External Transactions），在本文中，全局事务被限定为一种适用于单个服务使用多个数据源场景的事务解决方案。请注意，理论上真正的全局事务并没有“单个服务”的约束，它本来就是DTP（[Distributed Transaction Processing](#)）模型中的概念，但本节所讨论的内容——一种在分布式环境中仍追求强一致性的事务处理方案，对于多节点而且互相调用彼此服务的场合（典型的就是现在的微服务）中是极不合适的，今天它几乎只实际应用于单服务多数据源的场合中，为了避免与后续介绍的放弃了ACID的弱一致性事务处理方式相互混淆，所以这里的全局事务所指范围有所缩减，后续涉及多服务多数据源的事务，笔者将称其为“分布式事务”。

1991年，为了解决分布式事务的一致性问题，[X/Open](#)组织（后来并入了[The Open Group](#)）提出了一套名为[X/Open XA](#)（XA是eXtended Architecture的缩写）的处理事务架构，其核心内容是定义了全局的事务管理器（Transaction Manager，用于协调全局事务）和局部的资源管理器（Resource Manager，用于驱动本地事务）之间的通讯接口。XA接口是双向的，能在一个事务管理器和多个资源管理器（Resource Manager）之间形成通信桥梁，通过协调多个数据源的一致动作，实现全局事务的统一提交或者统一回滚，现在我们在Java代码中还偶尔能看见的XADataSource、XAResource这些名字都源于此。

不过，XA并不是Java规范（XA提出那时还没有Java），而是一套通用技术规范，所以Java中专门定义了[JSR 907 Java Transaction API](#)，基于XA模式在Java语言中的实现了一套全局事务处理的标准，这也就是我们现在所熟知的JTA。JTA最主要的两个接口是：

- 事务管理器的接口：javax.transaction.TransactionManager。这套接口是给Java EE服务器提供容器事务（由容器自动负责事务管理）使用的，还提供了另外一套javax.transaction.UserTransaction接口，用于通过程序代码手动开启、提交和回滚事务。
- 满足XA规范的资源定义接口：javax.transaction.xa.XAResource，任何资源（JDBC、JMS等等）如果需要支持JTA，只要实现XAResource接口中的方法即可。

JTA原本是Java EE中的技术，一般情况下应该由JBoss、WebSphere、WebLogic这些Java EE容器来提供支持，但现在Bitronix[↗]、Atomikos[↗]和JBossTM[↗]（以前叫Arjuna）都以JAR包的形式实现了JTA的接口，称为JOTM（Java Open Transaction Manager），使得我们能够在Tomcat、Jetty这样的Java SE环境下也能使用JTA。

现在，我们对“[事务处理](#)”里的示例场景做另外一种假设：如果书店的用户、商家、仓库分别处于不同的数据库中，其他条件仍与之前相同，那情况会发生什么变化？如果我们以声明式事务来编码，那与本地事务看起来可能没什么区别（都是标个@Transactional注解而已），但不过以编程式事务来实现的话，写法就能看出差异，具体如下所示：

```
public void buyBook(PaymentBill bill) {  
    userTransaction.begin();  
    warehouseTransaction.begin();  
    businessTransaction.begin();  
    try {  
        userAccountService.pay(bill.getMoney());  
        warehouseService.deliver(bill.getItems());  
        businessAccountService.receipt(bill.getMoney());  
        userTransaction.commit();  
        warehouseTransaction.commit();  
        businessTransaction.commit();  
    } catch(Exception e) {  
        userTransaction.rollback();  
        warehouseTransaction.rollback();  
        businessTransaction.rollback();  
    }  
}
```

java

代码上看出目的是要做三次事务提交，但实际上代码是并不能这样写的，试想一下，如果在businessTransaction.commit()中出现错误，代码转到catch块中执行，此时userTransaction和warehouseTransaction已经完成提交，再调用rollback()方法也无济于事，这将导致一部分数据被提交，另一部分被回滚，整个事务的一致性也就无法保证了。为了解决这个问题，XA将事务提交拆分成为两阶段过程：

- 准备阶段（又叫做投票阶段）：在这一阶段，协调者询问所有参与的是否准备好提交，参与者如果已经准备好提交则回复Prepared，否则回复Non-Prepared。所谓的准备操作，对于数据库来说，其逻辑是在重做日志中记录全部事务提交操作所要做的内容，只是与本地事务提交时的区别是暂不写入最后一条Commit Record而已，相当于在做完数

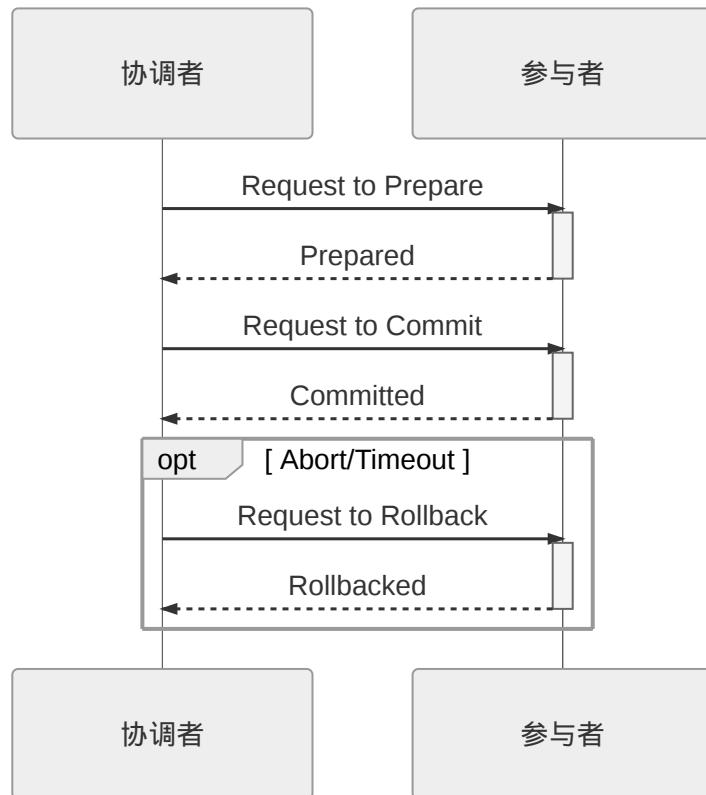
据持久化后并不立即释放隔离性，即仍继续持有锁，维持数据对其他非事务内观察者的隔离状态。

- 提交阶段（又叫做执行阶段）：协调者如果在上一阶段收到所有参与者回复的Prepared消息，则先自己在本地持久化事务状态为Commit，在此操作完成后向所有参与者发送Commit指令，所有参与者立即执行提交操作；否则，任意一个参与者回复了Non-Prepared消息，或任意一个参与者超时未回复，协调者将在自己完成事务状态为Abort持久化后，向所有参与者发送Abort指令，参与者立即执行回滚操作。对于数据库来说，这个阶段的提交操作应是很轻量快速的，仅仅是持久化一条Commit Record而已，只有收到Abort指令时，才需要清理已提交的数据，这可能是相对重负载操作。

以上这两个过程被称为“[两段式提交](#)”（2 Phase Commit，2PC）协议，而它能够成功保证一致性还要求有其他前提条件：

- 必须假设网络（在提交阶段的短时间内）是可靠的，即不会丢失消息或者传递错误的消息，XA的设计目标并不是解决诸如[拜占庭问题](#)的网络问题。两段式提交中投票阶段失败了可以补救（回滚），而提交阶段失败了无法补救（不再改变提交或回滚的结果，只能等崩溃的节点重新恢复），但此阶段耗时很短，这也是为了尽量控制网络风险的考虑。
- 必须假设因为网络、机器崩溃或者其他原因而导致失联的节点最终能够恢复，不会永久性地处于崩溃状态。由于在准备阶段已经写入了完整的重做日志，所以当失联机器一旦恢复，就能够从日志中找出已准备妥当但并未提交的事务数据，再而向协调者查询该事务的状态，确定下一步应该进行提交还是回滚操作。

请注意，上面所说的协调者、参与者通常都是数据库的角色，协调者一般是在参与者之间选举产生的，而应用服务器相对于数据库来说是客户端的角色。两段式提交的交互时序如下图所示：



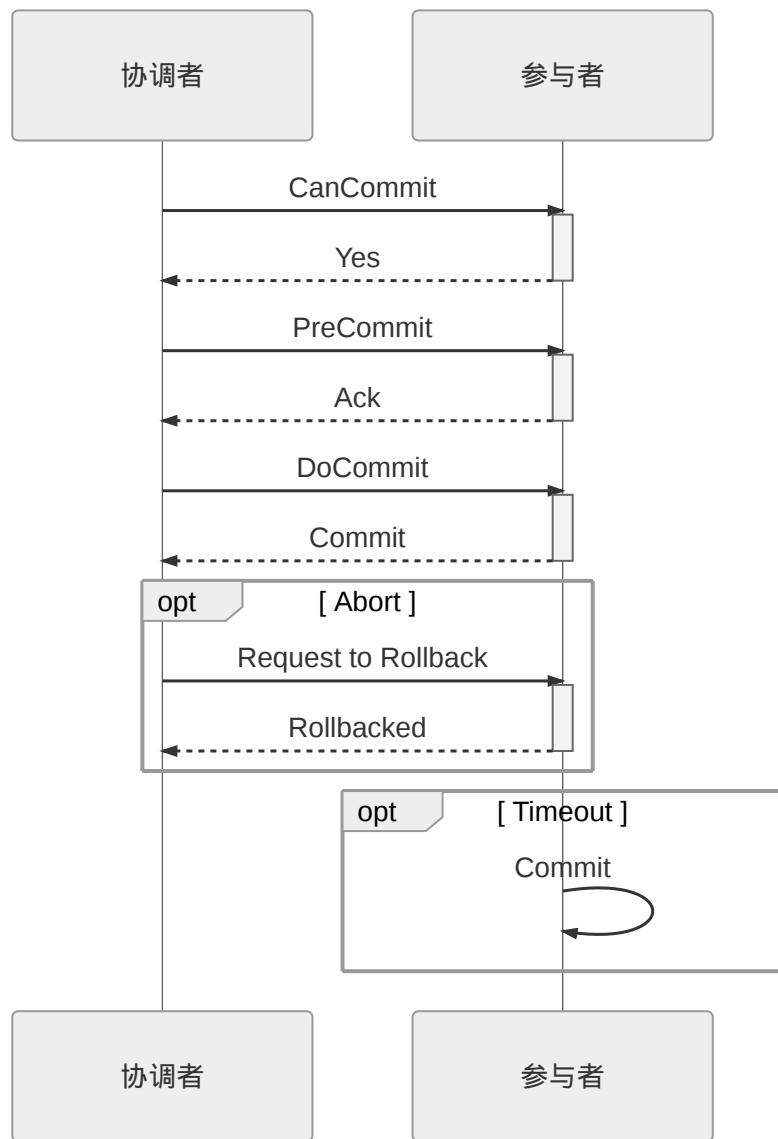
两段式提交原理简单，易于实现，但其缺点也是非常显著的：

- 单点问题**：协调者在两段提交中具有举足轻重的作用，协调者等待参与者回复时可以有超时机制，允许参与者宕机，但参与者等待协调者指令时无法做超时处理。一旦宕机的不是其中某个参与者，而是协调者的话，所有参与者都会受到影响，譬如，协调者没有正常发送Commit或者Rollback的指令，那所有参与者都将一直等待。
- 性能问题**：两段提交过程中，所有参与者相当于被绑定成为一个统一调度的整体，期间要经过两次远程服务调用，三次数据持久化（准备阶段写重做日志，协调者做状态持久化，提交阶段在日志写入Commit Record），整个过程将持续到参与者集群中最慢的那一个处理操作结束为止，这决定了两段式提交对性能影响通常都会比较大。
- 一致性风险**：前面已经提到，两段式提交的成立是有前提条件的，网络稳定性和宕机恢复能力的假设不成立时，仍可能出现一致性问题。宕机恢复能力这一点不必多谈，1985年Fischer、Lynch、Paterson提出了定理证明了如果宕机最后不能恢复，那就不存在任何一种分布式协议可以正确地达成一致性结果（被称为FLP不可能原理，它在分布式中是与CAP齐名的定理）。而网络稳定性带来的一致性风险是指：尽管提交阶段时间很短，但这仍是一段明确存在的危险期，如果协调者在发出准备指令后，根据收到各个参与者发回的信息确定事务状态是可以提交的，协调者会先持久化事物状态，并提交自己的事务，如果这时候网络忽然被断开，无法再通过网络向参与者发出Commit指令的话，

就会导致部分数据（协调者的）已提交，但部分数据（参与者的）还未提交（也没有回滚），产生了数据不一致的问题。

为了缓解两段式提交协议的头两点缺陷——即单点问题和性能问题，后续发展出了“三段式提交”（3 Phase Commit，3PC）协议。三段式提交把原本的两段式提交的准备阶段再细分为两个阶段，分别称为CanCommit、PreCommit，把的提交阶段称为DoCommit阶段。其中，新增的CanCommit是一个询问阶段，协调者让每个参与的数据库根据自身状态，评估该事务是否有可能顺利完成。将准备阶段一分为二的理由是这个阶段是重负载的操作，一旦协调者发出开始准备的消息，每个参与者都将马上开始写重做日志，它们所涉及的数据资源即被锁住，如果此时某一个参与者宣告无法完成提交，相当于大家都白做了一轮无用功。所以，增加一轮询问阶段，如果都得到了正面的响应，那事务能够成功提交的把握就比较大了，这意味着因某个参与者提交时发生崩溃而导致大家全部回滚的风险相对变小。因此，在事务需要回滚的场景中，三段式的性能通常是要比两段式好很多的，但在事务能够正常提交的场景中，两者的性能都依然很差（三段式的多了一次询问，还要稍微更差一些）

同样也是基于事务失败回滚概率变小的原因，三段式提交中，如果在PreCommit阶段之后发生了协调者宕机，参与者没有能等到DoCommit的消息的话，默认的操作策略将是提交事务（而不是回滚），这就相当于避免了协调者单点问题的风险。三段式提交的操作时序如下图所示。

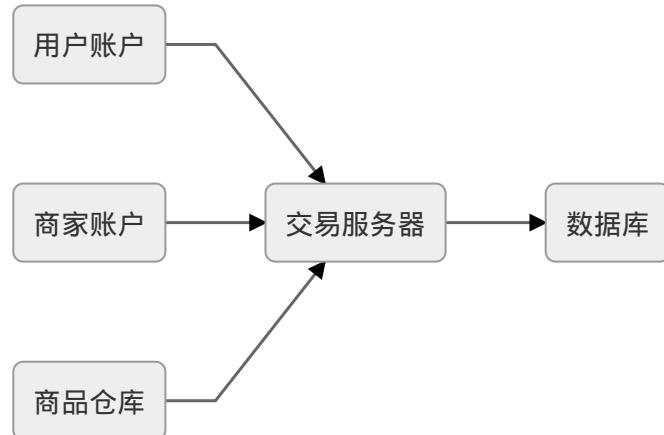


从以上过程可以看出，三段式提交对单点问题和回滚时的性能问题有所改善，但是它对一致性风险问题并未有任何改进，在这方面它面临的风险甚至反而是略有增加了的。譬如，进入PreCommit阶段之后，协调者发出的指令不是Ack而是Abort，而此时因网络问题，有部分参与者直至超时都未能收到协调者的Abort指令的话，这些参与者将会错误地提交事务，这就产生了数据一致性问题。

共享事务

与前面全局事务所指的单个服务使用多个数据源正好相反，共享事务（Share Transaction）是指多个服务使用同一个数据源。请注意此语境里“数据源”与“数据库”的区别，我们在部署应用集群时一种典型模式是将同一套程序部署到多个中间件服务器的节点，它们连接了同一个数据库，但每个节点配有自己的专属的数据源JNDI，所有节点的数据访问都是完全独立的，并没有任何交集，此时每个节点所采用的是简单的本地事务。而本节讨论的是多个服务之间会产生业务交集的场景，举个具体例子，在Fenix's Bookstore的[场景事例](#)中，假设用户账户、商家账户和商品仓库都存储于同一个数据库之中，但用户、商户和仓库每个领域都部署了独立的微服务，此时一次购书的业务操作将贯穿三个微服务，它们都要在数据库中修改数据。如果我们直接将不同数据源就视为是不同数据库，那上一节所讲的全局事务和下一节要讲的分布式事务都是可行的，不过，针对这种每个数据源连接的都是同一个数据库的特例，共享事务则有机会成为另一条可能提高性能、降低复杂度的途径（但更有可能是个伪需求，为何拆了微服务还要连同一个数据库？）。

一种理论可行的方案是直接让各个服务共享数据库连接，同一个服务进程中的不同持久化工具（JDBC、ORM、JMS等）要共享数据库连接并不困难，一些应用服务器，如WebSphere中也会有“[可共享连接](#)”的功能支持。但由于数据库连接的基础是网络连接，这是与IP地址绑定的，字面意义上的“不同服务节点能共享数据库连接”很难做到，所以这种方案里需要新增一个“交易服务器”的中间角色，无论是用户服务、商家服务还是仓库服务，它们都通过同一台交易服务器来与数据库打交道。如果你将交易服务器的对外接口实现为JDBC规范，那它完全可以视为是一个独立于各个服务的远程连接池或者数据库代理来看待，此时三个服务所发出的交易请求就有可能做到交由同一个数据库连接通过本地事务的方式完成。譬如，交易服务器根据不同服务节点传来的同一个事务ID，使用同一个数据库连接来处理跨越多个服务的交易事务。



之所以强调理论上，是因为这是与实际生产系统中的压力方向相悖的，一个集群中数据库往往才是压力最大而又最不容易伸缩拓展的重灾区，所以现实中只有类似[ProxySQL](#)、[MaxScale](#)这样用于对多个数据库实例做负载均衡的代理（其实用ProxySQL代理单个数据库，再启用Connection Multiplexing，其实已经挺接近于前面所提及的方案了），而几乎没有反过来代理一个数据库为多个应用提供事务协调的交易服务代理。这也是我说它更有可能是个伪需求的原因，连数据库都不拆分的话，你必须找到十分站得住脚的理由来向团队解释做微服务的价值是什么才行。

以上方案还有另外一种本质上是同样思路变种应用的形式：使用JMS服务器的来代替交易服务器，用户、商家、仓库的服务操作业务时，通过消息将所有对数据库的改动传送到JMS服务器，通过JMS来统一完成有事务保障的持久化操作。这被称作是“[单个数据库的消息驱动更新](#)”（Message-Driven Update of a Single Database）。“共享事务”的提法和这里所列的两种处理方式在实际应用中均不常见，鲜有采用这种方式的成功案例，笔者查询到的资料几乎都发源于十余年前的[这篇文章](#)，考虑到它并不契合于现在的技术趋势，这里也不花费更多的篇幅了。

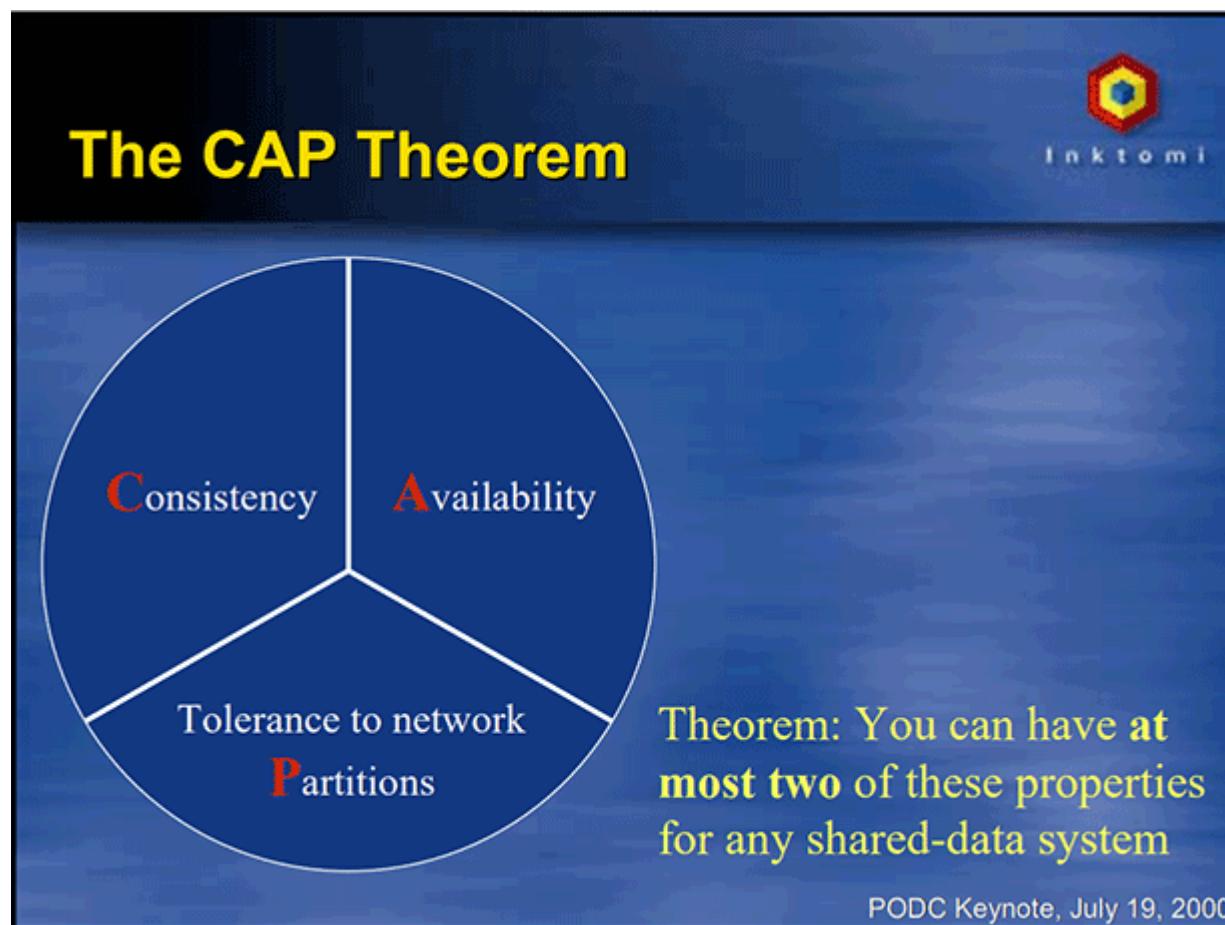
分布式事务

本节所说的分布式事务（Distributed Transactions）指的是多个服务同时访问多个数据源的事务处理机制，请注意它与[DTP模型](#)中“分布式事务”的差异，DTP模型所指的“分布式”是相对于数据源而言的，并不涉及服务，这部分内容已经在[“全局事务”](#)一节里进行过讨论。本节所指的“分布式”是相对于服务而言的，如果严谨地说，它更应该被称为“在分布式服务环境下的事务处理机制”。

曾经（在2000年以前），人们寄希望于XA的事务机制可以在本节所说的分布式环境中也能良好地应用，但这个美好的愿望今天已经被CAP理论彻底地击碎了，这节的话题就从CAP与ACID的矛盾说起。

CAP与ACID

CAP理论，也被称为Brewer理论，是在2000年7月，加州大学伯克利分校的Eric Brewer教授于“ACM分布式计算原理研讨会（PODC）”上所提出的一个猜想：

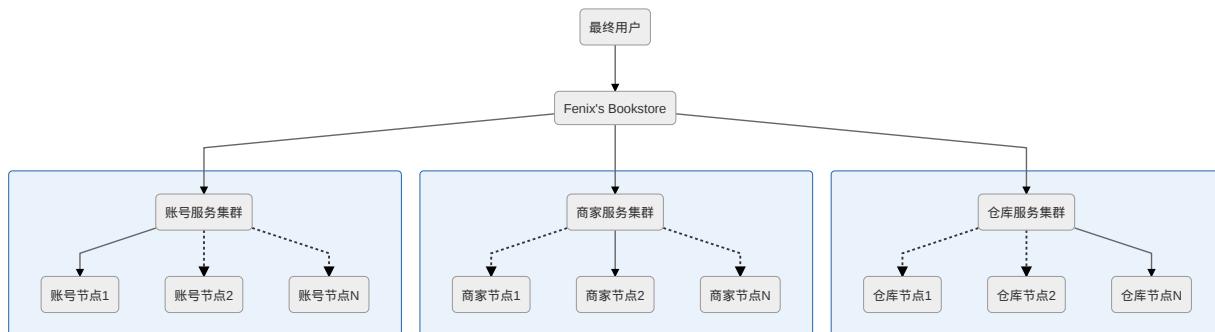


CAP理论原稿 (那时候还只是猜想)

2002年，麻省理工学院的Seth Gilbert和Nancy Lynch以严谨的数学推理上证明了CAP猜想。自此之后，CAP理论正式成为分布式计算领域所公认的著名定理。这个定理里描述了一个分布式的系统中，涉及到共享数据问题时，以下三个特性最多只能满足其二：

- **一致性 (Consistency)**：代表数据在任何时刻、任何分布式节点中所看到的都是没有矛盾的。这与前面所提的ACID中的C是相同的单词又有不同的定义（分别指Replication的一致性和数据库状态的一致性）。但分布式事务中，ACID的C要以满足CAP中的C为前提。
- **可用性 (Availability)**：代表系统**不间断地**提供服务的能力。
- **分区容忍性 (Partition Tolerance)**：代表分布式环境中部分节点因网络原因而彼此失联（即与其他节点形成“网络分区”）时，系统仍能**正确地**提供服务的能力。

单纯只列概念，CAP是比较抽象的，笔者仍以本章开头所列的事例场景来说明这三种特性对分布式系统来说将意味着什么。假设Fenix's Bookstore的服务拓扑如下图所示，一个来自最终用户的交易请求，将交由账号、商家和仓库服务集群中某一个节点来完成响应：



在这套系统中，每一个单独的服务节点都有着自己的数据库，假设某次交易请求分别由“账号节点1”、“商家节点2”、“仓库节点N”来进行响应。当用户购买一件价值100元的商品后，账号节点1首先应给该用户账号扣减100元货款，它在自己数据库扣减100元很容易，但它还要把这次交易变动告知节点2到N，以及确保能正确变更商家和仓库集群其他账号节点中的关联数据，此时将面临以下情况：

- 如果该变动信息没有及时同步给其他账号节点，将导致有可能发生用户购买另一商品时，被分配给到另一个节点处理，由于看到账户上有不正确的余额而错误地发生了原本无法进行的交易，此为一致性问题。
- 如果由于要把该变动信息同步给其他账号节点，必须暂时停止对该用户的交易服务，直至数据同步一致后再重新恢复，将可能导致用户在下一次购买商品时，因系统暂时无法提供服务而被拒绝交易，此为可用性问题。
- 如果由于账号服务集群中某一部分节点，因出现网络问题，无法正常与另一部分节点交换账号变动信息，那此时服务集群中无论哪一部分节点对外提供的服务都可能是不正确的，能否接受由于部分节点之间的连接中断而影响整个集群的正确性，此为分区容忍性。

以上还仅是涉及到了账号服务集群自身的CAP问题，对于整个Fenix's Bookstore站点来说，它更是面临着来自于账号、商家和仓库服务集群带来的CAP问题，譬如，用户账号扣款后，由于未及时通知仓库服务，导致另一次交易中看到仓库中有不正确的库存数据而发生超售。又譬如因涉及到仓库中某个商品的交易正在进行，为了同步用户、商家和仓库的交易变动，而暂时锁定该商品的交易服务，导致了的可用性问题，等等。

既然已有数学证明，我们就不去讨论为何CAP不可兼得，接下来直接分析如何权衡取舍CAP，以及不同取舍所带来的问题。

- 如果放弃分区容错性（CA without P），这意味着我们将假设节点之间通讯永远是可靠的。永远可靠的通讯在分布式系统中必定不成立的，这不是你想不想的问题，而是网络

分区现象始终会存在。在现实中，主流的RDBMS集群通常就是放弃分区容错性的工作模式，以Oracle的RAC集群为例，它的每一个节点均有自己的SGA、重做日志、回滚日志等，但各个节点是共享磁盘中的同一份数据文件和控制文件的，是通过共享磁盘的方式来避免网络分区的出现。

- 如果放弃可用性（CP without A），这意味着我们将假设一旦发生分区，节点之间的信息同步时间可以无限制地延长，此时问题相当于退化到前面“全局事务”中讨论的一个系统多个数据源的场景之中，我们可以通过2PC/3PC等手段，同时获得分区容错性和一致性。在现实中，除了DTP模型的分布式数据库事务外，著名的HBase也是属于CP系统，以它的集群为例，假如某个RegionServer宕机了，这个RegionServer持有的所有键值范围都将离线，直到数据恢复过程完成为止，这个时间通常会是很长的。
- 如果放弃一致性（AP without C），这意味着我们将假设一旦发生分区，节点之间所提供的数据可能不一致。AP系统目前是分布式系统设计的主流选择，因为P是分布式网络的天然属性，你不想要也无法丢弃；而A通常是建设分布式的目的，如果可用性随着节点数量增加反而降低的话，很多分布式系统可能就没有存在的价值了（除非银行这些涉及到金钱交易的服务，宁可中断也不能出错）。目前大多数NoSQL库和支持分布式的缓存都是AP系统，以Redis集群为例，如果某个Redis节点出现网络分区，那仍不妨碍每个节点以自己本地的数据对外提供服务，但这时有可能出现请求分配到不同节点时返回给客户端的是不同的数据。

行文至此，不知道你是否感受到一丝无奈，本章讨论的话题“事务”原本的目的就是获得“一致性”，而在分布式环境中，“一致性”却不得不成为了通常被牺牲、被放弃的那一项属性。但无论如何，我们建设信息系统，终究还是要保证操作结果（在最终被交付的时候）是正确的，为此，人们又重新给一致性下了定义，将前面我们在CAP、ACID中讨论的一致性称为“[强一致性](#)”（Strong Consistency），有时也称为“[线性一致性](#)”（Linearizability，通常是在讨论[共识算法](#)的场景中），而把牺牲了C的AP系统又要尽可能获得正确的结果的行为称为追求“弱一致性”，不过，如果单纯只说“弱一致性”那其实就是“不保证一致性”的意思……人类语言这东西真是博大精深。为此，在弱一致性中，人们又总结出了一种特例，被称为“[最终一致性](#)”（Eventual Consistency），它是指：如果数据在一段时间之内没有被另外的操作所更改，那它最终将会达到与强一致性过程相同的结果，有时候面向最终一致性的算法也被称为“[乐观复制算法](#)”。

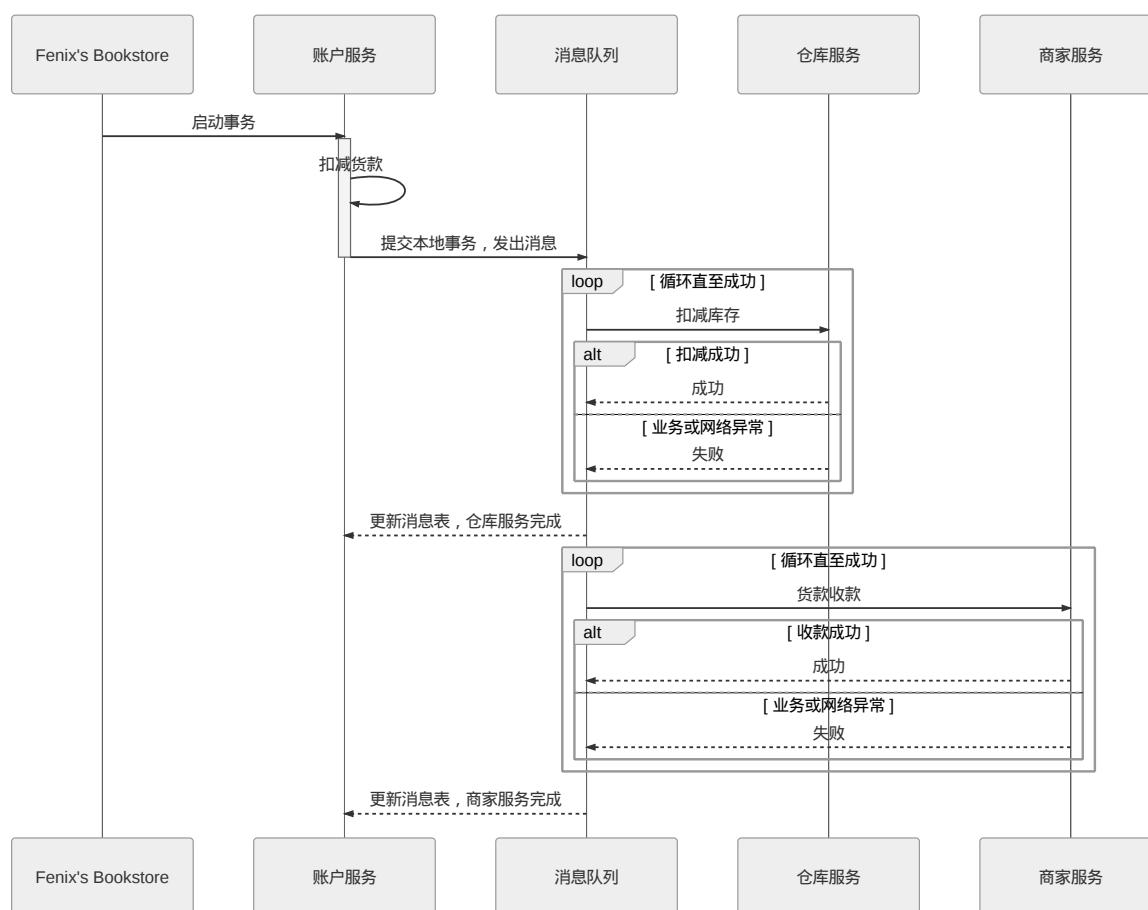
在本节讨论的主题“分布式事务”中，目标同样也不得不从前面的获得强一致性，降低为获得“最终一致性”，在这个意义上，其实“事务”一词的含义也已经被拓宽了，人们把之前追求

ACID的事务称为“刚性事务”，而把笔者下面将要介绍几种分布式事务的常见做法统称为“柔性事务”。

可靠事件队列

最终一致性的概念是eBay的系统架构师Dan Pritchett在2008年发表于ACM的论文《[Base: An Acid Alternative](#)》中提出的，该文中总结了另外一种独立于ACID获得的强一致性之外的、通过BASE来达成一致性目的的途径，最终一致性就是其中的“E”。BASE这提法比起ACID凑缩写的痕迹更重，不过有ACID vs BASE（酸 vs 碱）这个朗朗上口的梗，这篇文章传播得足够快，在这里笔者就不多谈BASE中的概念了，但这篇论文本身作为最终一致性的概念起源，并系统性地总结了一种在分布式事务的技术手段，还是非常有价值的。

我们继续以本章的事例场景来解释Dan Pritchett提出的“可靠事件队列”的具体做法，下图为操作时序：



1. 最终用户向Fenix's Bookstore发送交易请求：购买一本价值100元的《深入理解Java虚拟机》。

2. Fenix's Bookstore应该对用户账户扣款、商家账户收款、库存商品出库这三个操作有一个出错概率的先验评估，根据出错概率的大小来安排它们的操作顺序（这个一般体现在程序代码中，有一些大型系统也可能动态排序）。譬如，最有可能的出错的是用户购买了，但是不同意扣款，或者账户余额不足；其次是商品库存不足；最后商家收款，一般收款不会遇到什么意外。那顺序就应该是最容易出错的最先进行，即：账户扣款 → 仓库出库 → 商家收款。
3. 账户服务进行扣款业务，如扣款成功，则在自己的数据库建立一张消息表，里面存入一条消息：“事务ID：UUID，扣款：100元（状态：已完成），仓库出库《深入理解Java虚拟机》：1本（状态：进行中），某商家收款：100元（状态：进行中）”，注意，这个步骤中“扣款业务”和“写入消息”是依靠同一个本地事务写入自身数据库的。
4. 系统建立一个消息服务，定时轮询消息表，将状态是“进行中”的消息同时发送到库存和商家服务节点中去（可以串行地，即一个成功后再发送另一个，但在我门讨论的场景中没必要）。这时候可能产生以下几种可能的情况：
 1. 商家和仓库服务成功完成了收款和出库工作，向用户账户服务器返回执行结果，用户账户服务把消息状态从“进行中”更新为“已完成”。整个事务宣告顺利结束，达到最终一致性的状态。
 2. 商家或仓库服务有某个或全部因网络原因，未能收到来自用户账户服务的消息。此时，由于用户账户服务器中存储的消息状态一直处于“进行中”，所以消息服务器将在每次轮训的时候持续地向对应的服务重复发送消息。这个步骤可重复性决定了所有被消息服务器发送的消息都必须具备幂等性，通常的设计是让消息带上一个唯一的事务ID，以保证一个事务中的出库、收款动作只会被处理一次。
 3. 商家或仓库服务有某个或全部无法完成工作，譬如仓库发现《深入理解Java虚拟机》没有库存了，此时，仍然是持续自动重发消息，直至操作成功（譬如补充了库存），或者被人工介入为止。
 4. 商家和仓库服务成功完成了收款和出库工作，但回复的应答消息因网络原因丢失，此时，用户账户服务仍会重新发出下一条消息，但因消息幂等，所以不会导致重复出库和收款，只会导致商家、仓库服务器重新发送一条应答消息，此过程重复直至双方网络恢复。
 5. 也有一些支持分布式事务的消息框架，如RocketMQ，原生就支持分布式事务操作，这时候上述情况2、4也可以交由消息框架来保障。

以上这种靠着持续重试来保证可靠性的操作，在计算机中非常常见，它有个专门的名字叫做“[最大努力交付](#)”（Best-Effort Delivery），譬如TCP协议中的可靠性保障就属于最大努

力交付。而“可靠事件队列”有一种更普通的形式，被称为“最大努力一次提交”（Best-Effort 1PC），所指的就是将最有可能出错的业务以本地事务的方式完成后，通过不断重试的方式（不限于消息系统）来促使同个事务的其他关联业务完成。

TCC事务

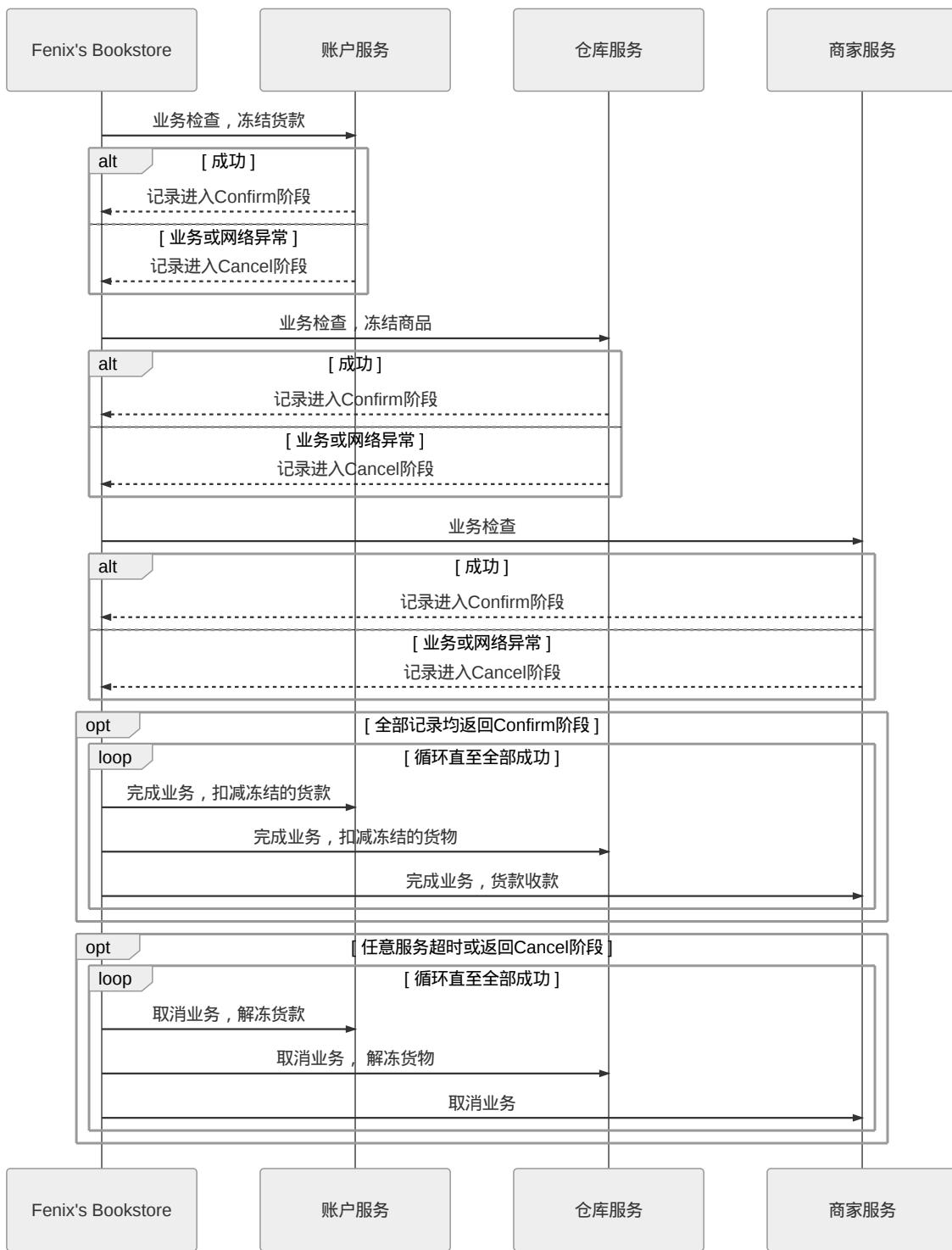
TCC是另一种常见的分布式事务机制，它是“Try-Confirm-Cancel”三个单词的缩写，是由数据库专家Pat Helland在2007年撰写的论文《Life beyond Distributed Transactions: an Apo state's Opinion》中提出。

前面介绍的可靠消息队列虽然能保证最终的结果是相对可靠的，过程也简单（相对于TCC来说），但整个过程完全没有任何隔离性可言，有一些业务中隔离性是无关紧要的，但有一些业务中缺乏隔离性就会带来许多麻烦。譬如我们的事例场景中，缺乏隔离性带来的一个显而易见的问题便是“超售”：完全有可能两个客户在短时间内都成功购买了同一件商品，而且他们各自购买的数量都不超过目前的库存，但他们购买的数量之和却超过了库存。如果这件事情处于刚性事务，且隔离级别足够的情况下是可以避免的，譬如，处理“第二类丢失更新的问题”（Second Lost Update）需要“可重复读”（Repeatable Read）的隔离剂别（这部分属于数据库本地事务方面的内容，就不再展开了），以保证后面提交的事务会因为无法获得锁而导致更新失败，但用可靠消息队列就无法做到这一点，这时候就可以考虑TCC方案了，它比较适合用于需要较强隔离性的分布式事务中。

TCC是一种业务侵入式较强的事务方案，它要求业务处理过程必须拆分为“预留业务资源”和“确认/释放消费资源”两个子过程。如同TCC的名字所示，它分为以下三个阶段：

- **Try**：尝试执行阶段，完成所有业务可执行性的检查（保障一致性），并且预留好全部需要用到的业务资源（保障隔离性）。
- **Confirm**：确认执行阶段，不进行任何业务检查，直接使用Try阶段准备的资源来完成业务处理。Confirm阶段可能会重复执行，需要满足幂等性。
- **Cancel**：取消执行阶段，释放Try阶段预留的业务资源。Cancel阶段可能会重复执行，需要满足幂等性。

按照我们的示例场景，TCC的执行过程应该是这样的：



1. 最终用户向Fenix's Bookstore发送交易请求：购买一本价值100元的《深入理解Java虚拟机》。
2. 创建事务，生成事务ID，记录在活动日志中，进入Try阶段：
 - 用户服务：检查业务可行性，可行的话，将该用户的100元设置为“冻结”状态，通知下一步进入Confirm阶段；不可行的话，通知下一步进入Cancel阶段。
 - 仓库服务：检查业务可行性，可行的话，将该仓库的1本《深入理解Java虚拟机》设置为“冻结”状态，通知下一步进入Confirm阶段；不可行的话，通知下一步进入Cancel

I阶段。

- 商家服务：检查业务可行性，不需要冻结资源。

3. 如果第2步所有业务均反馈业务可行，将活动日志中的状态记录为Confirm，进入Confirm阶段：

- 用户服务：完成业务操作（扣减那被冻结的100元）
- 仓库服务：完成业务操作（标记那1本冻结的书为出库状态，扣减相应库存）
- 商家服务：完成业务操作（收款100元）

4. 第3步如果全部完成，事务宣告正常结束，如果第3步中任何一方出现异常（业务异常或者网络异常），将根据活动日志中的记录，重复执行该服务的Confirm操作（即最大努力交付）。

5. 如果第2步有任意一方反馈业务不可行，或任意一方超时，将活动日志的状态记录为Cancel，进入Cancel阶段：

- 用户服务：取消业务操作（释放被冻结的100元）
- 仓库服务：取消业务操作（释放被冻结的1本书）
- 商家服务：取消业务操作（大哭一场后安慰商家谋生不易）

6. 第5步如果全部完成，事务宣告回滚结束，如果第5步中任何一方出现异常（业务异常或者网络异常），将根据活动日志中的记录，重复执行该服务的Cancel操作（即最大努力交付）。

由上述操作过程可见，TCC其实有点类似于2PC的准备阶段和提交阶段，但TCC是位于用户代码层面，而不是基础设施层面，这为它的实现带来了一定的灵活性，可以根据需要设计资源锁定的粒度。同时，这也带来了更高的开发成本和业务侵入性（主要影响到可控性和更换事务实现方案的成本），所以，通常我们并不会裸编码来做TCC，而是基于某些分布式事务中间件（譬如阿里开源的Seata）基础之上完成。

SAGA事务

TCC事务具有较强的隔离性，避免了“超售”的问题，而且其性能一般来说是本篇提及的几种柔性事务模式中最高的（只操作预留资源，几乎不会涉及到锁和资源的争用），但它仍不能满足所有的场景。TCC的主要限制是它的业务侵入性很强，这里并不是说它需要开发编码配合所带来的工作量，而更多的是指它所要求的技术可控性上的约束。譬如，把我们的事例场景修改如下：由于中国网络支付日益盛行，现在用户和商家在书店系统中可以选择不在开设账号，至少不会强求一定要从银行充值到系统中才能进行消费，可以直接在

购物时通过网络支付在银行账号中划转货款。这里面就给系统施加了限制，用户、商家的账户在银行的话，其操作权限和数据结构就不可能再随心所欲的地设计，通常也就无法完成冻结款项、解冻、扣减这样的操作（银行一般不会配合你的操作）。所以TCC中第一步Try阶段往往就已经无法施行。这时候我们就可以考虑一下采用另外一种柔性事务方案：SAGA事务（SAGA在英文中是“长篇故事、长篇记叙、一长串事件”的意思）。

SAGA事务模式的历史很久，最早源于1987年普林斯顿大学的Hector Garcia-Molina和Kenneth Salem在ACM发表的一篇论文《SAGA¹》（这就是论文的名字）。文中提出了一种如何提升“长时间事务”（Long Lived Transaction）运作效率的方法，大致思路是把一个大事务分解为可以交错运行的一系列子事务集合。原本SAGA目的是为了避免大事务长时间锁定数据库的资源，后来发展成将一个分布式环境中的大事务分解为一系列本地事务的设计模式。SAGA由两部分操作组成：

- 每个分布式事务对数据的操作，分解为N个子事务，命名为 $T_1, T_2, \dots, T_i, \dots, T_n$ 。每个子事务都应该是或者能被视为是原子行为。如果分布式事务能够正常提交，其对数据的影响（最终一致性）应与连续按顺序成功提交 T_i 等价。
- 为每一个子事务设计补偿动作，命名为 $C_1, C_2, \dots, C_i, \dots, C_n$ 。 T_i 与 C_i 满足以下条件：
 - T_i 与 C_i 都具备幂等性。
 - T_i 与 C_i 满足交换律（Commutative），即先执行 T_i 还是先执行 C_i ，其效果都是一样的。
 - C_i 必须能成功提交，不考虑 C_i 本身提交失败被回滚的情形，此时需要人工介入。

如果 T_1 到 T_n 均成功提交，那事务顺利完成，否则，要采取以下两种恢复策略之一：

- **正向恢复**（Forward Recovery）：如果 T_i 事务提交失败，则一直对 T_i 进行重试，直至成功为止（最大努力交付）。这种恢复方式不需要补偿，适用于事务最终都要成功的场景（譬如扣了款，就一定要给别人发货）。正向恢复的执行模式为： T_1, T_2, \dots, T_i （失败）， T_i （重试）， \dots, T_n 。
- **反向恢复**（Backward Recovery）：如果 T_i 事务提交失败，则一直执行 C_i 对 T_i 进行补偿，直至成功为止（最大努力交付）。这里要求 C_i 必须（持续重试后）执行成功。反向恢复的执行模式为： T_1, T_2, \dots, T_i （失败）， C_i （补偿）， \dots, C_2, C_1 。

与TCC相比，SAGA不需要为资源设计冻结状态和撤销冻结的操作，补偿操作往往要容易实现得多。譬如，前面提到的账户直接开设在银行的场景，从银行划转货款到Fenix's Bookstore系统中，这步是经由用户支付操作（扫码、U盾）来促使银行提供服务；如果后续业务操作失败，尽管我们无法要求银行撤销掉之前用户转账的操作，但是由Fenix's Bookstore系统将货款转回到用户账上作为补偿措施确是完全可行的。

SAGA必须保证所有子事务都得以提交或者补偿，但SAGA系统本身也有可能会崩溃，所以它必须设计与数据库类似的日志机制（被称为SAGA Log）以保证系统恢复后可以追踪到子事务的执行情况，譬如执行至哪一步或者补偿至哪一步了。另外，尽管补偿操作通常比冻结/撤销容易实现，但保证正向、反向恢复过程的能严谨地进行也需要花费不少的工夫（譬如通过服务编排、可靠事件队列等方式完成），所以，SAGA事务通常也不会完全裸编码来实现，一般也是在事务中间件的基础上完成，前面提到的Seata同样支持SAGA模式。

基于数据补偿来代替回滚的思路，可以应用在其他事务方案上，这个笔者就不开独立小节，放到这里一起来解释。举个例子，譬如阿里的GTS（Global Transaction Service，Seata由GTS开源而来）所提出的“[AT事务模式](#)”就是这样的一种应用。

从整体上看是AT事务是参照了XA两段提交协议实现的，但针对XA 2PC的缺陷，即在准备阶段必须等待所有数据源都返回成功后，协调者才能统一发出Commit命令而导致的木桶效应（所有涉及到的锁和资源都需要等待到最慢的事务完成后才能统一释放），设计了针对性的解决方案。大致的做法是在业务数据提交时自动拦截所有SQL，将SQL对数据修改前、修改后的结果分别保存快照，生成行锁，通过本地事务一起提交到操作的数据源中（相当于记录了重做和回滚日志）。如果分布式事务成功提交，那后续清理每个数据源中对应日志数据即可；如果分布式事务需要回滚，就根据日志数据自动产生用于补偿的“逆向SQL”。基于这种补偿方式，分布式事务中所涉及的每一个数据源都可以单独提交，然后立刻释放锁和资源。这种异步提交的模式，相比起2PC极大地提升了系统的吞吐量水平。而其代价就是大幅度地牺牲了隔离性，在缺乏隔离性的前提下，以补偿代替回滚并不一定是总能成功的。譬如，当本地事务提交之后、分布式事务完成之前，该数据被补偿之前又被其他操作修改过，即出现了脏写（Dirty Write），这时候一旦出现分布式事务需要回滚，就不可能再通过自动的逆向SQL来实现补偿，只能由人工介入处理了。

通常来说，脏写是一定要避免的（所有DBMS在最低的隔离级别上都仍然要加锁以避免脏写），实际上这种情况人工也很难进行有效处理。所以GTS增加了一个“全局锁”（Global L

ock) 的机制来实现写隔离 , 要求本地事务提交之前 , 一定要先拿到针对修改记录的全局锁后才允许提交 , 没有获得全局锁之前就必须一直等待 , 这避免了有两个分布式事务中包含的本地事务修改了同一个数据 , 从而避免脏写。在读隔离方面 , AT 事务默认的隔离级别是 Read Uncommitted , 这意味着可能产生脏读 (Dirty Read) 。读隔离也可以采用全局锁的方案解决 , 但直接阻塞读取的话 , 代价就非常大了 , 通常并不会这样做。由此可见 , 分布式事务中没有一揽子包治百病的解决办法 , 因地制宜地选用合适的事务处理方案才是唯一有效做法。

透明多级分流系统

奥卡姆剃刀原则

entities should not be multiplied without necessity

如无必要，勿增实体

—— Occam's Razor  , William of Ockham 

现代的企业级或互联网系统，“分流”是必须要考虑的设计，分流所使用手段数量之多、涉及场景之广，可能到了连它的开发者本身都未必能全部意识到程度。这听起来似乎并不合理，但笔者认为这恰好是优秀架构设计的一种体现，“分布广阔”源于“多级”，“意识到”谓之“透明”，也即是本章我们要讨论的主题**“透明多级分流系统”**（Transparent Multilevel Diversion System，这个词是笔者自己创造的，业内通常只提“Transparent Multilevel Cache”，但我们这里谈的并不仅仅涉及到缓存）的来由。

用户使用信息系统的过程中，请求从浏览器出发，在DNS的指引下找到系统的入口，经过网关、负载均衡器、缓存、服务集群等一系列设施，最后触及到末端存储于数据库服务器中的信息，然后再逐级返回到用户的浏览器之中。这其中要经过许许多多的技术部件。作为系统的设计者，我们应该意识到不同的设施、部件在系统中有各自不同的价值：

- 有一些部件位于客户端或网络的边缘，能够迅速响应用户的请求，避免给后方的I/O与CPU带来压力，典型如本地缓存、内容分发网络、反向代理等。
- 有一些部件处理能力能够线性拓展，易于伸缩，可以使用较小的代价堆叠机器来获得与用户数量相匹配的并发性能，应尽量作为业务逻辑的主要载体，典型如集群中能够自动扩缩的服务节点。
- 有一些部件稳定服务对系统运行有全局性的影响，要时刻保持着容错备份，维护着高可用性，典型如服务注册中心、配置中心。
- 也有一些设施是天生的单点部件，只能依靠升级机器本身的网络、存储和运算性能来提升处理能力，如位于系统入口的路由、网关或者负载均衡器（它们都可以做集群，但一

次网络请求中无可避免至少有一个是单点的部件）、位于请求调用链末端的传统关系数据库等，都是典型的容易形成单点部件。

对系统进行流量规划时，我们应该充分理解这些部件的价值差异，有两个简单、普适的原则能指导我们进行设计：一个原则是尽可能减少单点部件，如果某些单点是无可避免的，则应尽最大限度减少到达单点部件的流量。举个例子，用户请求在系统中往往会有多个部件都能够处理，譬如要获取一张图片，浏览器缓存、CDN、反向代理、Web服务器、文件服务器、数据库都有可能提供这张图片。恰如其分地引导请求分流至最合适的组件中，避免绝大多数流量汇集到单点部件（如数据库），同时仍能够（或者在绝大多数时候）保证处理结果的准确性，仍能在单点系统出现故障时自动而迅速地实施补救措施，这便架构设计中多级分流的原则。

缓存、节流、主备、负载均衡等措施，是为了达成该目标所采用的工具与手段，而高可用架构、高并发架构则是通过该原则所获得的价值。许多介绍架构设计的资料中，会以“高可用、高并发架构”为主题来讲解这一部分内容。在本文档中，笔者选择以流量流经的部件为脉络，以“透明多级分流系统”为题介绍，这两者实质上内容是一样的。按从前（用户端）到后（服务端）的顺序，我们会讨论以下设施的运作和原理：

- **客户端缓存** (Client Cache)：HTTP协议的无状态性决定了它必须依靠客户端缓存来解决网络传输效率上的缺陷。
- **域名解析** (DNS Lookup)：DNS也许是全世界最大、使用最频繁的信息查询系统，如果没有适当的分流机制，DNS将会成为整个网络的瓶颈。
- **传输链路** (Transmission Optimization)：今天的传输链路优化原则，在若干年后的未来再回头看它们时，其中多数已经成了奇技淫巧，有些甚至成了反模式。
- **内容分发网络** (Content Distribution Network)：CDN是一种十分古老而又十分透明，没什么存在感的分流系统，许多人都说听过它，但甚少人真正去了解过它。
- **负载均衡** (Load Balancing)：调度后方的多台机器，以统一的接口对外提供服务，承担此职责的技术组件被称为“负载均衡”。
- **缓存** (Cache)：软件开发中的缓存并非多多益善，它有收益，也有风险。

同时也应当说明的是另一个，可能是更关键重要的原则：奥卡姆剃刀原则。作为一个设计者，你对以上讨论的多级分流的手段应该有全面的理解与充分的准备，但这些设施并不是越多越好。实际构建系统时，应当在有明确需求，真正有必要的时候再去考虑部署它们，不是每一个系统都需要时刻高并发高可用的保障，根据系统的用户量、峰值流量和团队本

身的技术与运维能力出发来考虑如何布置这些设施才是合理的做法，能满足需求的前提下，**最简单的系统就是最好的系统。**

客户端缓存

客户端缓存 (Client Cache)

HTTP协议的无状态性决定了它必须依靠客户端缓存来解决网络传输效率上的缺陷。

浏览器的缓存机制几乎是在万维网刚刚出现就已经存在，在HTTP协议设计之初，便确定了服务端与客户端之间“无状态”（Stateless）的交互原则，即要求每次请求是独立的，每次请求无法感知和依赖另一个请求的存在，这既简化了HTTP服务器的设计，也为其水平扩展能力留下了广袤的空间。但无状态并不只有好的一面，由于每次请求都是独立的，服务端不保存此前请求的状态和资源，所以也不可避免地导致其携带有重复的数据，造成网络性能降低。HTTP协议对此的解决方案就是客户端缓存，在HTTP从1.0到最新2.0版本的每次演进中，都提出过现在被称为“状态缓存”、“强制缓存”（许多资料中简称为“强缓存”）和“协商缓存”的缓存机制。

其中，状态缓存是指不经过服务器，客户端直接根据缓存信息对目标网站的状态判断，以前只有301/Moved Permanently（永久重定向）这一个；后来在[RFC6797](#)中增加了HSTS（HTTP Strict Transport Security）机制，用于避免依赖301/302跳转HTTPS时可能产生的降级中间人劫持（详细可见安全架构中的“[传输](#)”），这也属于另一种状态缓存。由于状态缓存所涉内容就只有这一点，后续我们就只聚焦于强制缓存与协商缓存两种机制。

强制缓存

只要是缓存，几乎都不可避免地会遇到一致性的问题。强制缓存对一致性处理就如它的名字一样，显得十分的直接粗暴，假设在某个时间点（譬如10分钟）之内，资源的内容和状态一定不会被改变，因此客户端可以无需经过任何浏览器请求，在该时间点来临前一直持有和使用该资源的本地缓存副本。

根据约定，强制缓存在用户在浏览器输入地址、页面链接跳转、新开窗口、前进/后退中均可生效，但在使用F5刷新页面时应当失效。有以下两类HTTP Header可以实现强缓存：

- **Expires**：Expires是HTTP/1.0协议中提供的Header（当然，在HTTP/1.1中同样存在），后面跟随一个截至时间参数。当服务器返回某个资源时带有该Header的话，意味着服务器承诺截止时间之前资源不会发生变动，浏览器可直接缓存该数据，不再重新发请求，示例：

```
HTTP/1.1 200 OK
Expires: Wed, 8 Apr 2020 07:28:00 GMT
```

Expires是HTTP协议最初版本的缓存机制，设计非常直观易懂，但考虑得并不够周全，它至少存在以下显而易见的问题：

- 受限于客户端的本地时间。譬如，客户端修改了本地时间，可能会造成缓存提前失效或超期持有。
- 无法处理涉及到用户身份的私有资源，譬如，某些资源被登录用户缓存在自己的浏览器上是合理的，但如果被CDN服务器缓存起来，则可能被其他未认证的用户所获取。
- 无法描述“不缓存”的语义。譬如，浏览器为了提高性能，往往会自动在当次会话中缓存某些MIME类型的资源，在HTTP/1.0的服务器中就缺乏手段强制浏览器不允许缓存某个资源。以前为了实现这类功能，通常不得不使用Script脚本，在资源后面增加时间戳（如“xx.js?t=1586359920”）来保证每次资源都会重新获取。

关于“不缓存”的语义，在HTTP/1.0中其实设计了“Pragma: no-cache”来实现，但Pragma在HTTP响应中的行为没有确切描述，随后就被HTTP/1.1中出现过的Cache-Control所替代，现在，尽管主流浏览器通常都会支持Pragma，但实际并没有什么使用价值了。

- **Cache-Control**：Cache-Control是HTTP/1.1协议中定义的强制缓存Header，它的语义比起Expires来说就丰富了很多，如果Cache-Control和Expires同时存在，并且语义存在冲突（Expires与max-age / s-maxage冲突）的话，必须以Cache-Control为准。Cache-Control的示例如下：

```
HTTP/1.1 200 OK
Cache-Control: max-age=600
```

Cache-Control在客户端的请求头或服务器的响应头中都可以使用，它定义了一系列的参数，且允许扩展（不在标准RFC协议中，由浏览器自行支持），其标准的参数主要包括有：

- max-age / s-maxage：max-age后面跟随一个以秒为单位的数字，表明相对于请求时间（Date Header中也会注明请求时间），多少秒以内缓存是有效的，资源不需要重新从服务器中获取。相对时间避免了Expires中采用的绝对时间可能受客户端时钟影响的尴尬。s-maxage中的s是“Share”的缩写，意味“共享缓存”（即被CDN、代理等持有的缓存）有效时间，用于提示CDN这类服务器如何对缓存进行失效。
- public / private：指明是否涉及到用户身份的私有资源，如果是public，着可以被代理、CDN等缓存，如果是private，着只能由客户端进行私有缓存。
- no-cache / no-store：no-cache指明该资源不应该被缓存，哪怕是同一个会话中对同一个URL地址的请求，也必须从服务端获取（但协商缓存机制依然是生效的）；no-store不强制会话中相同URL资源的重复获取，但禁止浏览器、CDN等以任何形式保存该资源。
- no-transform：禁止资源被任何形式地修改。譬如，某些CDN、透明代理支持自动GZIP压缩图片或文本，以提升网络性能，而no-transform就禁止了这样的行为，它要求Content-Encoding、Content-Range、Content-Type均不允许进行任何形式的修改。
- min-fresh / only-if-cached：这两个参数是仅用于客户端的请求Header。min-fresh后继跟随一个以秒为单位的数字，用于建议服务器能返回一个不少于该时间的缓存资源（即包含max-age且不少于min-fresh的数字）。only-if-cached表示要求客户端要求不发送网络请求，只使用缓存来进行响应，若缓存不能命中，就直接返回503/Service Unavailable错误。
- must-revalidate / proxy-revalidate：must-revalidate表示在资源过期后，一定需要从服务器中进行验证（即超过了max-age的时间，就等同于no-cache的行为），proxy-revalidate用于提示代理、CDN等缓存服务，语义与must-revalidate一致。

协商缓存

强制缓存是基于时效性的，但无论是人还是服务器，其实多数情况下都并没有什么把握去承诺某项资源多久不会发生变化。另外一种基于变化检测的缓存机制，在一致性上会有比强制缓存更好的表现，但需要一次变化检测的交互开销，性能上就会略差一些，这种基于检测的缓存机制，通常被称为“协商缓存”。另外，应注意在HTTP中协商缓存与强制缓存并

没有排他性，这两套机制是并行工作的，譬如，当强制缓存存在时，直接从强制缓存中返回资源，无需进行变动检查；而当强制缓存超过时效，或者被禁止（no-cache / must-revalidate），协商缓存仍可以正常地工作。协商缓存主要有根据资源的修改时间或根据资源唯一标识是否发生变化来进行变动检查的机制，这都是靠一组成对出现的请求、响应Header来实现的：

- **Last-Modified和If-Modified-Since**：Last-Modified是服务器的响应Header，用于告诉客户端这个资源的最后修改时间。对于带有这个Header的资源，当客户端需要在此请求时，会通过If-Modified-Since把之前收到的资源最后修改时间发送回服务端。如果此时服务端发现资源在该时间后没有被修改过，就只要返回一个304/Not Modified的响应即可，无需附带消息体，如下所示：

```
HTTP/1.1 304 Not Modified
Cache-Control: public, max-age=600
Last-Modified: Wed, 8 Apr 2020 15:31:30 GMT
```

如果此时服务端发现资源在该时间之后有变动，就会返回200/OK的完整响应，在消息体中包含最新的资源，如下所示：

```
HTTP/1.1 200 OK
Cache-Control: public, max-age=600
Last-Modified: Wed, 8 Apr 2020 15:31:30 GMT

Content
```

- **Etag和If-None-Match**：Etag是服务器的响应Header，用于告诉客户端这个资源的唯一标识（HTTP服务器可以根据自己的意愿来选择如何生成这个标识，譬如Apache服务器的Etag值，默认是对文件的索引节点（INode），大小（Size）和最后修改时间（MTime）进行哈希计算后得到的），对于带有这个Header的资源，当客户端需要在此请求时，会通过If-None-Match把之前收到的资源唯一标识发送回服务端。如果此时服务端计算后发现资源的唯一标识与上传回来的一致，说明资源没有被修改过，就只要返回一个304/Not Modified的响应即可，无需附带消息体，如下所示：

```
HTTP/1.1 304 Not Modified
Cache-Control: public, max-age=600
```

```
Last-Modified: Wed, 8 Apr 2020 15:31:30 GMT
```

如果此时服务端发现资源的唯一标识有变动，就会返回200/OK的完整响应，在消息体中包含最新的资源，如下所示：

```
HTTP/1.1 200 OK
Cache-Control: public, max-age=600
Last-Modified: Wed, 8 Apr 2020 15:31:30 GMT
```

```
Content
```

Etag是HTTP中一致性最强的缓存机制，譬如，Last-Modified标注的最后修改只能精确到秒级，如果某些文件在1秒钟以内，被修改多次的话，它将不能准确标注文件的修改时间；又或者如果某些文件会被定期生成，可能内容并没有任何变化，但Last-Modified却改变了，导致文件无法有效使用缓存，这些情况Last-Modified都有可能产生一致性问题，只能使用Etag解决。

Etag却又是HTTP中性能最差的缓存机制，体现在每次请求时，服务端都必须对资源进行哈希计算，这比起简单获取一下修改时间，开销要大了很多。Etag和Last-Modified是允许一起使用的，服务器会优先验证Etag，在Etag一致的情况下，再去对比Last-Modified，这是为了防止有一些HTTP服务器未将文件修改日期纳入哈希范围内。

到这里为止，HTTP的协商缓存机制已经能很好地处理通过URL获取单个资源的场景，“单个资源”是什么意思？在HTTP协议的设计中，一个URL地址有可能能够提供多份不同版本的资源，譬如，一段文字的不同语言版本，一个文件的不同编码格式版本，一份数据的不同压缩方式版本，等等。HTTP协议设计了Accept*（Accept、Accept-Language、Accept-Charset、Accept-Encoding）的一套请求Header和对应的Content-*（Content-Language、Content-Type、Content-Encoding）的响应Header，这被称为HTTP的内容协商机制。与之对应的，对于一个URL能够获取多个资源的场景中，缓存也同样也需要有明确的标识来获知根据什么内容来对同一个URL返回给用户正确的资源。这个就是Vary Header的作用，Vary后面可以跟随其他Header的名字，譬如：

```
HTTP/1.1 200 OK
Vary: Accept, User-Agent
```

以上说明应该根据MINE类型和浏览器类型来缓存资源，获取资源时也需要根据请求头中对应的字段来筛选出适合的资源版本。

根据约定，协商缓存不仅在用户在浏览器输入地址、页面链接跳转、新开窗口、前进/后退中生效，而且在使用F5刷新页面时也同样是生效的，只有用户强制刷新（Ctrl+F5）或者禁用缓存（譬如在DevTools中设定）时才会失效，此时客户端向服务端发出的请求会自动带有“Cache-Control: no-cache”。

域名解析

域名缓存 (DNS Lookup)

DNS也许是全世界最大、使用最频繁的信息查询系统，如果没有适当的分流机制，DNS将会成为整个网络的瓶颈。

我们都知道DNS的作用是将便于人类理解的域名地址转换为便于计算机处理的IP地址，也许你会觉得好笑：笔者在接触计算机网络的开头一段不短的时间里面，都把DNS想像成一个部署在全世界某个神秘机房中的大型电话本式的翻译服务。后来，当笔者第一次了解到DNS的工作原理，并得知世界根域名服务器的ZONE文件只有2MB大小，甚至可以打印出来物理备份的时候，对DNS系统的设计是非常惊叹的。域名解析这个话题同样涉及缓存等因素，虽然它并不算本篇讨论的重点，但其本身就是堪称示范性的透明多级分流系统，很值得我们借鉴。

假设我们访问域名：www.icyfenix.com.cn，DNS并不是一次性地将“www.icyfenix.com.cn”解析成IP地址的，这需要经历一个递归解析的过程。首先DNS会将域名还原为“www.icyfenix.com.cn.”，注意最后多了一个点“.”，它是“.root”的含义（早期的域名必须带有这个点）DNS才能够正确解析，如今DNS服务器已经可以自动补上结尾的点号），然后开始如下过程：

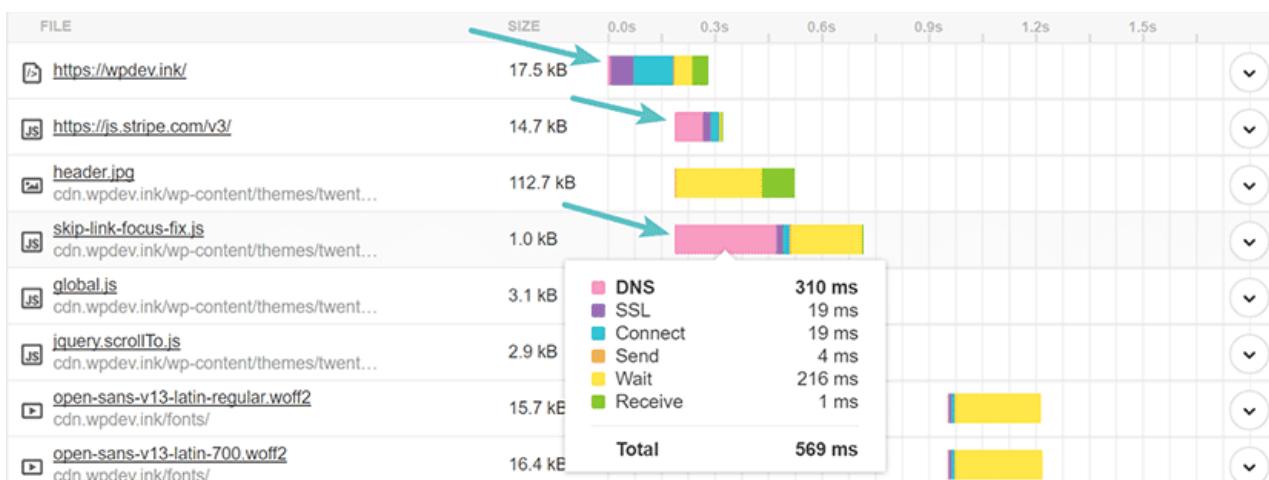
1. 客户端检查本地DNS缓存，查看是否存在并存活着的该域名的地址记录，DNS是以存活时间（Time to Live，TTL）来衡量缓存的存活情况的。后续每一级DNS查询的过程都会有类似的缓存查询操作，将不再重复叙述。
2. 客户端将地址发送给本机系统中设置的本地DNS（Local DNS，这个服务器可以通过手工设置，在路由做DHCP分配时或者在拨号时从PPP服务器中也会自动获取到）。
3. 本地DNS收到查询后，会按照“是否有www.icyfenix.com.cn的权威服务器”→“是否有www.icyfenix.com的权威服务器”→“是否有com.cn的权威服务器”→“是否有cn的权威服务器”的顺序，查询自己的地址记录，如果都没有查询到，就会一直找到最后点号代表的根域名服务器为止。这里涉及了两个名词：

- **权威域名服务器** (Authoritative DNS) : 是指负责翻译指定域名的DNS服务器 , “权威”意味着指定域名应该翻译出怎样的结果是由它来决定。DNS翻译域名时无需像查电话本一样刻板地机械翻译 , 根据来访机器、网络链路、服务内容等各种信息 , 可以玩出很多花样。
- **根域名服务器** (Root DNS) 是指固定的、无需查询的 (可以默认为已内置) **顶级域名** (Top-Level Domain) 服务器。全世界一共有13个根域名服务器 (但并不是13台 , 每一个根域名都通过任播的方式建立了一大群镜像 , 根据维基百科的数据 , 迄今已经超过1000台根域名服务器的镜像了) 。13这个数字是由于DNS主要采用UDP传输协议 (在需要稳定性保证的时候也可以采用TCP) 来进行数据交换 , 未分片的UDP数据包在IPv4下最大有效值为512字节 , 由此而来的限制。

4. 我们假设本地DNS是新开张的 , 上述权威服务器的记录它都没有 , 一直查到根域名服务器后 , 它将会得到“cn的权威服务器”的记录 , 然后通过“cn的权威服务器” , 得到“com.cn的权威服务器” , 以此类推 , 最后找到“www.icyfenix.com.cn的权威服务器”。
5. 通过“www.icyfenix.com.cn的权威服务器” , 查询www.icyfenix.com.cn的地址记录 (有RFC定义的地址记录有数十种类型) , 譬如IPv4下的IP地址为A记录 , IPv6下的AAAA记录、主机别名CNAME记录 , 等等) , 选择一条合适的返回给客户端。

一个域名可以配置多条不同的A记录 , 此时权威服务器可以根据自己的策略来进行选择。一种典型的应用是智能线路 : 根据访问者所处的不同地区 (譬如华北、华南、东北、港澳台、国外) 、不同服务商 (譬如电信、联通、移动) 等因素来确定返回的A记录。

DNS系统多级分流的设计使得DNS系统能够经受住全球网络流量不间断的冲击 , 但也并非全无缺点。譬如 , 当极端情况 (各级服务器均无缓存) 下的域名解析可能导致后续递归的多次查询而显著影响响应速度 , 譬如下图所示。



首次DNS请求耗时 (图片来自网络)

专门有一种被称为“DNS预取”（ DNS Prefetching ）的前端优化手段：如果网站后续要使用来自于其他域的资源，那就在网页加载时便生成一个link请求，促使浏览器对该域名进行预解释，譬如下面所示：

```
<link rel="dns-prefetch" href="//domain.not-icyfenx.cn">
```

html

而另一种可能更严重的缺陷是DNS的分级查询意味着每一级都有可能受到中间人攻击的威胁，产生被劫持的风险。要攻陷位于递归链条顶层的（譬如根域名服务器，cn权威服务器）服务器和链路是非常困难的，但很多位于递归链底层的、本地运营商的Local DNS服务器的安全防护则相对松懈，甚至不少地区的运行商自己就会进行劫持，专门返回一个错的IP，在这个IP上代理用户请求，以便给特定资源（主要是HTML）注入广告，以此牟利。

为此，最近几年出现了另一种新的DNS应用形式：[HTTPDNS](#)（也称为DNS over HTTP S，DoH）。它将DNS服务开放为一个HTTPS服务，替代基于UDP传输协议的DNS域名解析，直接从权威DNS或者可靠Local DNS获取解析数据，从而绕过传统Local DNS。这种做法的好处是避免了底层的域名劫持（遇到顶层劫持是往往是政府行为，这是没办法的），能够有效解决Local DNS不可靠导致的域名生效缓慢、来源IP不准确产生的智能线路切换错误等问题。

传输链路

传输链路优化 (Transmission Optimization)

今天的传输链路链路优化原则，在若干年后的未来再回头看它们时，其中多数已经成了奇技淫巧，有些甚至成了反模式¹。

在开始本节的讨论前，笔者先列一些在网络上很容易就能找到的，对Web进行链路性能优化的原则（譬如[雅虎YSlow23条规则](#)），这些原则在今天大多仍是（暂时）有一定价值的，至少也算是曾经（可能现在也还算是）广泛地流行过，但大概率在若干年后的未来再回头看它们时，其中多数已经成了奇技淫巧，有些甚至成了反模式。趁着当今的Web在传输链路这一块正处于新老交替之际，我们来说一下两代HTTP协议下的链路优化的问题。

1. 利用客户端缓存：缓存总是有益的，这点第一节中详细介绍过，本节不再涉及。
2. 减少请求数量：请求每次都需要建立通信链路进行数据传输，这些开销很昂贵，减少请求数量可有效的提高访问性能。
 - 雪碧图 ([CSS Sprites](#))
 - CSS、JS文件合并/内联 (Concatenation / Inline)
 - 分段文档 ([Multipart Document](#))
 - 媒体（图片、音频）内联 ([Data Base64 URI](#))
 - 异步请求合并 (Batch Ajax Request)
 -
3. 扩大并发请求数：现代浏览器一般对每个域名支持6个（IE为8-13个）并发请求，如果希望更快地加载大量图片或其他资源，需要进行域名分片（Domain Sharding），将图片同步到不同主机或者同一个主机的不同域名上（YSlow：Split Components Across Domains）。
4. 避免页面重定向：当页面发生了重定向，就会延迟整个文档的传输。在HTML文档到达之前，页面中不会呈现任何东西，降低了用户体验。
5. 按重要性调节资源优先级：将重要的、马上就要使用的、对客户端展示影响大的资源，放在HTML的头部，以便优先下载。

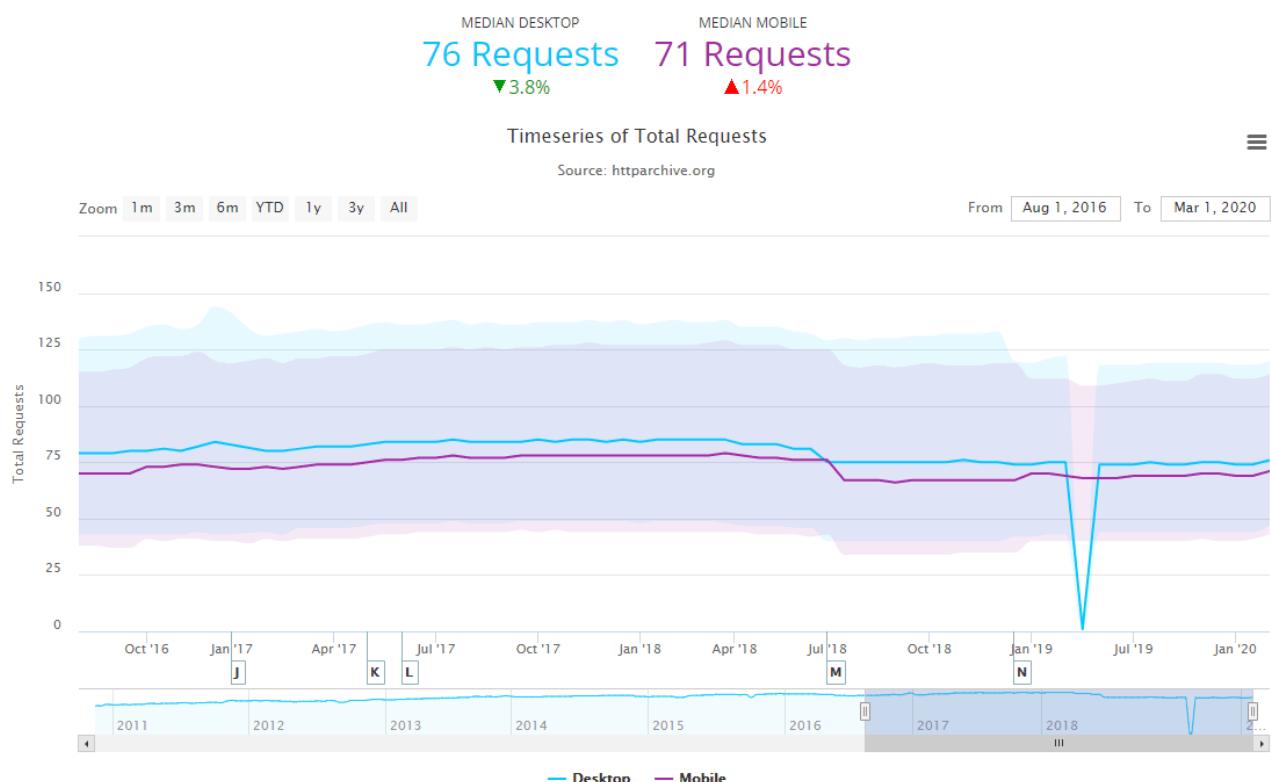
6. 启用压缩传输：启用压缩能够大幅度减少需要在网络上传输内容的大小，节省网络流量。

7.

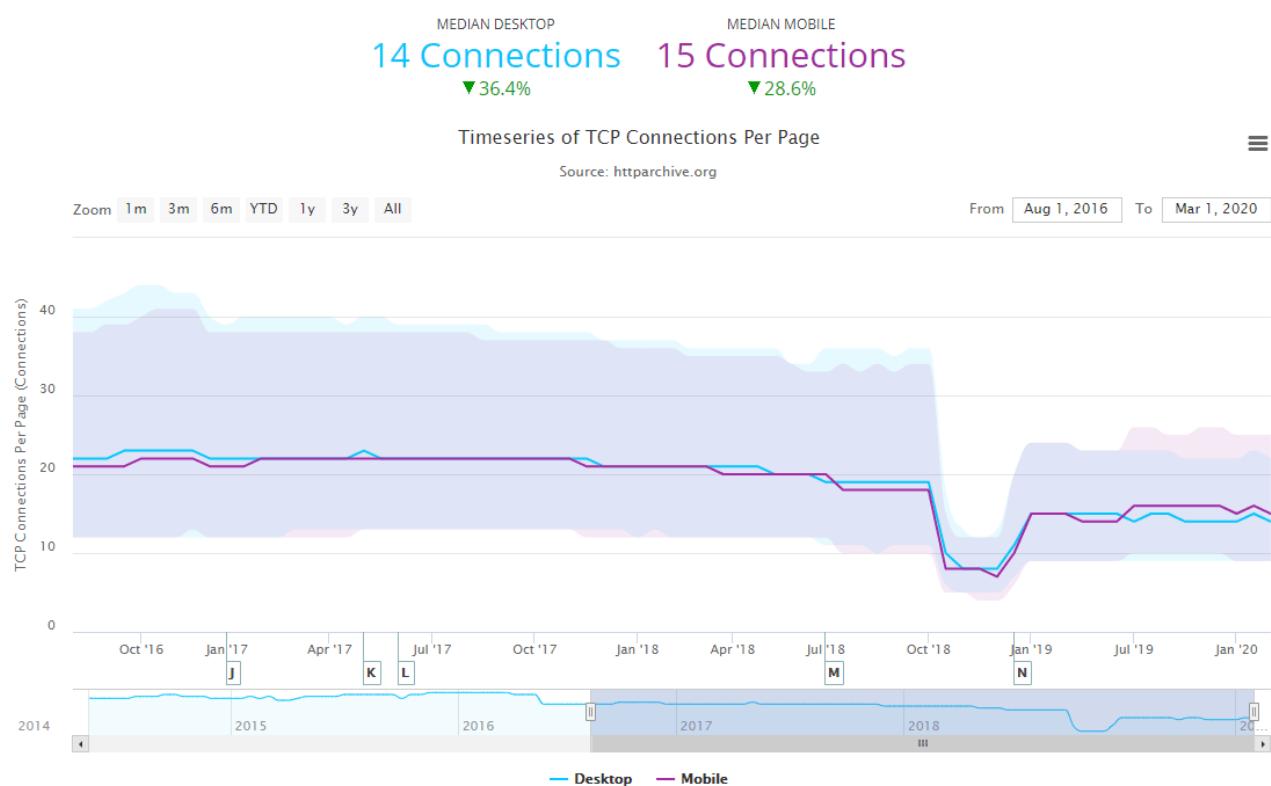
如同之前介绍客户端缓存时提到的那样，HTTP要得到无状态的好处，就必须相应承受网络效率降低的代价。在其他方面，HTTP协议设计和应用中也经历过了类似的权衡取舍，现在看来那些需要用户去优化的内容，往往都是当时技术现状下权衡取舍的结果。我们就从优化原则中条目最多的针对HTTP请求数量的措施说起。

连接数优化

我们都知道HTTP是基于TCP协议的，必须在[TCP三次握手](#)完成之后才能进行数据传输，这是一个通常以“百毫秒”为计时尺度的事件；此外，TCP还有[慢启动](#)的特性，使得刚刚建立连接时传输速度是最低的，后面再逐步加快直至稳定。由于TCP协议本身是面向于长时间、大数据传输来设计的，在长时间尺度下，它连接建立的成本高昂才不至于成为瓶颈，它的稳定性和可靠性的优势才能展现出来，那显然HTTP over TCP这种搭配，在目标倾向于上就多少产生了一些矛盾，以至于HTTP/1.x时代，大量短而小的TCP连接确实造成了网络性能的瓶颈。为了缓解HTTP在这个问题上的缺陷，聪明的程序员们一面致力于减少发出的请求数量，另外一方面也致力于增加客户端到服务端的连接数量，就是上面2、3点所提到的优化措施。这些Tricks的确减少消耗TCP连接数量，下面两张图片是来自于[HTTP Archive](#)对最近五年来数百万个URL地址采样得出的结论，页面平均请求没有改变的情况下，TCP连接在持续地下降（当然，后面说的HTTP/2其实占了很大功劳）。



HTTP平均请求数量，70余个，没有明显变化



TCP连接数量，约15个，有明显下降趋势

但是，上述这些节省TCP连接的优化措施但也带来了诸多不良的副作用：

- 如果你用CSS Sprites将多张图片合并，意味着任何场景下哪怕只用到其中一张小图，也必须完整加载整个大图片；任何场景下哪怕一张小图要进行修改，都会导致整个缓存失

效，类似地，样式、脚本等其他文件的合并也会造成同样的问题。

- 如果你使用了媒体内嵌，除了要承受Base64编码导致传输容量膨胀1/3的代价外（Base64以8 bit表示6 bit数据），也将无法有效利用缓存。
- 如果你合并了异步请求，这就会导致所有请求返回时间都受最慢的那个请求的拖累，整体响应速度下降。
- 如果你把图片放到不同子域下面，将会导致更大的DNS解析负担，而且浏览器对两个不同子域下的同一图片必须持有两份缓存，也使得缓存效率的下降。
-

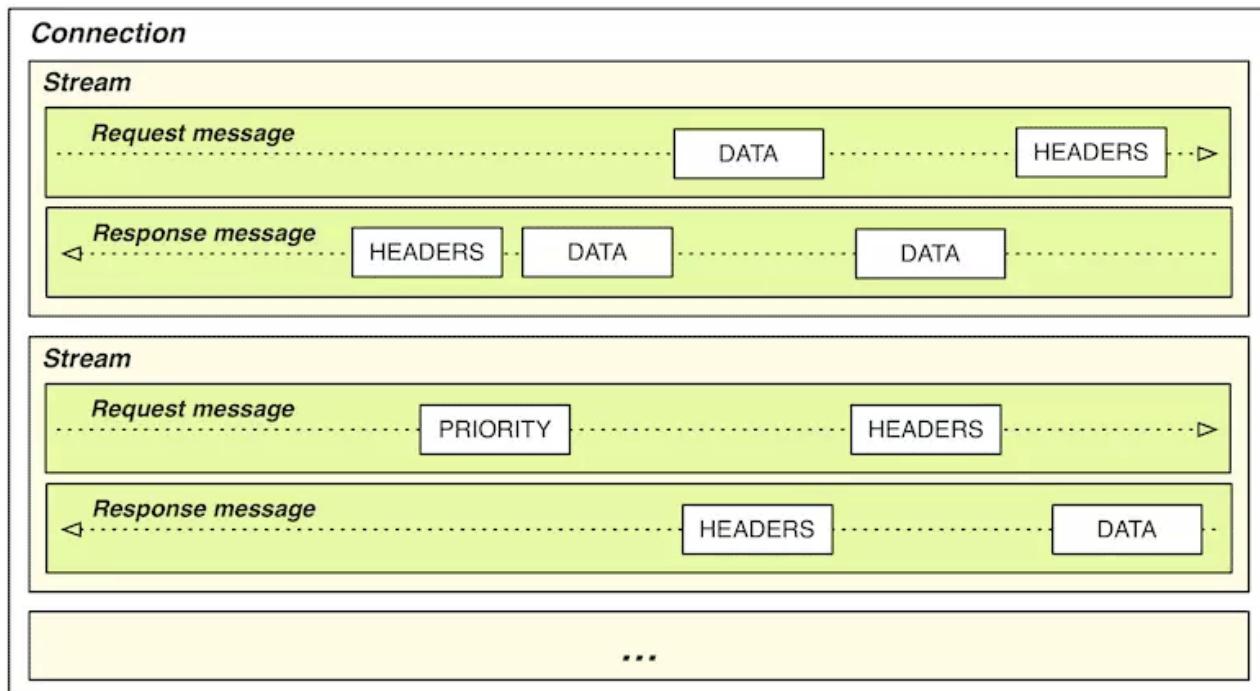
由此可见，一旦技术根基上出现的缺陷，依赖使用者通过各种Tricks去解决，无论如何都难以摆脱“两害相权取其轻”的权衡困境，否则这就不是Tricks而是会成为一种标准的设计模式了。

在另一方面，HTTP的设计者们并非没有尝试过在基础设施层面去解决连接成本过高的问题，即使是HTTP协议的最初版本（指HTTP/1.0，忽略非正式的HTTP/0.9版本）也是支持（不是默认，HTTP/1.1中变为默认）连接复用的，即今天大家所熟知的[持久连接](#)（Persistent Connection）或者叫连接[Keep-Alive机制](#)。其大致原理是让客户端可以对一个域名长期持有一个（或多个）TCP连接，在客户端维护一个FIFO队列，每次取完数据（如何在不断开连接下判断取完数据将会放到稍后压缩部分去讨论）之后不断开连接，以便下一个资源需要获取时备用，避免创建TCP连接的成本。而在2014年，IETF发布的[RFC 7230](#)中提出了名为“[HTTP管道](#)”（HTTP Pipelining）复用技术试图在服务端也建立类似的队列，以进一步提高效率，客户端一次过将所有请求发给服务端，由服务端来管理队列的话，可以保证队列中两项工作之间没有空隙，甚至可能进行并行化处理，提升了服务端的效率。不过，HTTP管道需要多方共同支持，推广得并不算成功。

不幸的是，连接复用仍然存在它的副作用，最主要的一项副作用是“[队首阻塞](#)”（Head-of-Line Blocking）问题，请设想以下场景：浏览器有10个资源需要从服务器中获取，此时它将10个资源放入队列，入列顺序只能是按照浏览器预见这些资源的先后顺序来决定的。但如果这10个资源中的第1个就让服务器陷入长时间运算状态那会怎样？当它的请求被发送到服务端之后，服务端开始计算，而运算结果出来之前TCP连接中并没有任何数据返回，此时后面9个资源都必须阻塞等待。无论队列维护在服务端还是客户端，其实都无法解决这个问题，因为服务端虽然很可能可以并行处理另外9个请求（譬如第一个是复杂运算请求，消耗CPU资源，第二个是数据库访问，消耗数据库资源，第三个是访问某张图片，消耗磁盘I/O资源，等等，这就很适合并行），但处理结果却无法发回给客户端，服务端既不能哪个

请求先完成就返回哪个，更不可能将所有要返回的资源混杂到一起交叉传输……显然，TCP连接带来的问题，本质上是传输链路上的问题，无论在服务端还是客户端，涉及到传输方面都显得无能为力。

队首阻塞问题一直持续到第二代的HTTP协议，即HTTP/2发布后才算是被比较完美地解决。在HTTP/1.x中，“请求”就是传输过程中最小粒度的信息单位了，所以如果将多个请求切碎，再混杂在一块传输，客户端势必难以分辨重组出有效信息。而在HTTP/2中，帧（Frame）才是最小粒度的信息单位，它可以用来描述各种数据，譬如请求的Header、Body，或者用来做控制标识，譬如打开流、关闭流。这里说的流（Stream）是一个逻辑数据通道的概念，每个帧都附带有一个流ID以标识这个帧属于哪个流。这样，在同一个TCP连接中传输的多个数据帧就可以根据流ID轻易区分出来，在客户端毫不费力地将不同流中的数据重组出HTTP的请求、响应报文来。这项设计是HTTP/2的重点技术特征之一，被称为[HTTP/2 多路复用](#)（HTTP/2 Multiplexing）



HTTP2的多路复用（图片来自：<https://hpbn.co/http2>）

有了多路复用的支持，HTTP/2就可以对每个域名只维持一个TCP连接（One Connection Per Origin），既减轻了服务器的连接压力，开发者也不用去考虑域名分片这种事情来突破浏览器对每个域名最多6个连接数限制了。而更重要的是，没有了TCP连接数的逼迫，所有通过合并/内联文件（无论是图片、样式、脚本）以减少请求数的需求就不再成立了，甚至反而是徒增副作用的反模式了——可能还有人会反驳说：不至于吧，减少请求数量，不是至少还减少了传输中耗费的Header吗？先得承认一个事实，在HTTP协议中，Header的成

本所占的比重相当的大，以至于在HTTP/2中需要专门考虑如何进行Header压缩的问题。但是，以下几个因素导致了通过合并资源文件减少请求数，对节省Header成本也几乎没有帮助：

- Header的传输成本在Ajax（尤其是只返回少量数据的请求）请求中可能是比重很大的开销，但在图片、样式、脚本这些静态资源的请求中，通常并不占主要。
- 在HTTP/2中Header压缩的原理是基于字典编码的信息复用，简而言之是同一个连接上产生的请求和响应越多，动态字典积累得越全，头部压缩效果也就越好。所以HTTP/2是单域名单连接的机制，合并资源和域名分片反而对性能提升不利。
- 与HTTP/1.x相反，HTTP/2本身反而变得更适合传输小资源了，譬如传输1000张10K的小图，HTTP/2要比HTTP/1.x快，但传输10张1000K的大图，则应该HTTP/1.x会更快。这一方面是TCP连接数量（相当于多点下载）的影响，更多的是由于TCP协议丢包重传机制导致的，一个丢失的TCP包会导致所有的流都必须等待这个包重传成功，这个问题就是HTTP/3.0要解决的目标了。因此，把小文件合并成大文件，在HTTP/2下是毫无好处的。

传输压缩

我们接下来再花一点点篇幅来讨论链路优化中除了缓存、连接之外另一个主要话题：压缩。很多人都知道HTTP协议是支持[GZip](#)压缩的，由于HTTP传输的主要内容，譬如HTML、CSS、Script等，都是文本数据，对于这些文本数据启用压缩的收益是非常高的，传输量一般会降至原有的20%左右。而对于那些不适合压缩的资源，Web服务器则能根据MIME类型来自动判断是否对响应进行压缩，这样，已经采用过压缩算法存储的资源，如JPEG、PNG图片，便不会被二次压缩，空耗性能。

不过，大概就没有多少人想过压缩与之前提到的用于节约TCP的持久连接机制是存在一些冲突的。在古代，服务器处理能力还很差的时候，通常是把静态资源先预先压缩为.gz文件的形式存放起来，当客户端可以接受压缩版本的资源时（请求的Header中包含Accept-Encoding: gzip）就返回压缩后的版本（响应的Header中包含Content-Encoding: gzip），否则就返回未压缩的原版，这种方式被称为“[静态预压缩](#)”（Static Pre-compression）。而现代的Web服务器处理能力有了大幅提升，已经没有人再采用麻烦的预压缩方式了，都是由服务器对符合条件的请求将在输出时进行“[即时压缩](#)”（On-The-Fly Compression），整个压缩过程全部在内存的数据流中完成，不必等资源压缩完成再返回响应，这样可以显著

提高“首字节时间”（ Time To First Byte , TTFB ） , 改善Web性能体验。而这个过程中唯一不好的地方就是服务器再没有办法给出Content-Length这个响应Header了 , 因为输出Header时服务器还不知道压缩后资源的确切大小。

到这里 , 大家想明白即时压缩与持久链接的冲突在哪了吗 ? 持久链接机制不再依靠TCP连接是否关闭来判断资源请求是否结束 , 它会重用同一个连接以便向同一个域名请求多个资源 , 这样 , 客户端就必须要有除了关闭连接之外的其他机制来判断一个资源什么时候算传递完毕 , 这个机制最初 (在HTTP/1.0时) 就只有Content-Length , 即靠着请求头中明确给出资源的长度 , 传输到达该长度即宣告一个请求响应的结束。由于启用即时压缩后就无法给出Content-Length了 , 如果是HTTP/1.0的话 , 持久链接和即时压缩只能二选其一 (HTTP/1.0中两者默认都是不开启的) 。其实Content-Length的缺陷不仅仅在于即时压缩这一种场景 , 譬如对于动态内容 (Ajax、 PHP、 JSP等输出) , 服务器也同样无法事项得知Content-Length。

HTTP/1.1版本中修复了这个缺陷 , 增加了另一种“分块传输编码” (Chunked Transfer Encoding) 的资源结束判断机制 , 解决Content-Length与持久链接的冲突问题。分块编码原理相当简单 : 在响应Header中加入“Transfer-Encoding: chunked”之后 , 就代表这个响应报文将采用分块编码。此时 , 报文中的Body需要改为用一系列“分块”来传输。每个分块包含十六进制的长度值和对应长度的数据内容 , 长度值独占一行 , 数据从下一行开始。最后以一个长度值为0的分块来表示资源结束。举个例子 (来自于前面维基百科中的页面 , 为便于观察 , 只分块 , 未压缩) :

```
HTTP/1.1 200 OK
Date: Sat, 11 Apr 2020 04:44:00 GMT
Transfer-Encoding: chunked
Connection: keep-alive

25
This is the data in the first chunk

1C
and this is the second one

3
con

8
sequence
```

0

根据分块长度可知，前两个分块包含显式的回车换行符（CRLF，即\r\n字符）

```
"This is the data in the first chunk\r\n"      (37 字符 => 十六进制:  
0x25)  
"and this is the second one\r\n"                (28 字符 => 十六进制:  
0x1C)  
"con"                                         (3 字符 => 十六进制:  
0x03)  
"sequence"                                    (8 字符 => 十六进制:  
0x08)
```

所以解码后的内容为：

```
This is the data in the first chunk  
and this is the second one  
consequence
```

一般来说，Web服务器给出的数据分块大小是一致的（但并不强制），而不是如例子中那样随意。HTTP/1.1通过分块传输解决了即时压缩与持久连接并存的问题，到了HTTP/2，由于多路复用和单域名单连接的设计，已经无需再刻意强去提久链接机制了，但数据压缩仍然有节约传输带宽的重要价值。

内容分发网络

内容分发网络 (Content Distribution Network)

CDN是一种十分古老而又十分透明，没什么存在感的分流系统，许多人都说听过它，但真正了解过它的人却很少。

前面几个小节介绍了缓存、域名解析、链路优化，这节我们来讨论它们的一个经典的综合运用案例：内容分发网络 (Content Distribution Network, CDN)。

CDN是一种十分古老的应用，以至于笔者相信阅读本文的受众至少有八、九成应该对它有不同程度的了解的——起码是听说过它的名字的。如果把某个互联网系统比喻为一家开门营业的企业，那CDN就是它遍布世界各地的分支销售机构，客户要买一块CPU就订机票飞到美国加州Intel总部去那肯定是不合适的，到本地电脑城找个装机铺才是正常人类的做法，CDN就相当于电脑城那吆喝着CPU三十块钱一斤的本地经销商。

CDN又是一种十分透明的应用，以至于笔者相信阅读本文的受众至少有八、九成应该对它是如何为互联网站点分流、对它的工作原理并没有什么系统性的概念——起码没有自己亲自使用过。如果抛却其他影响服务质量的因素，仅从网络角度看，一个互联网系统的速度快慢取决于以下四点因素：

1. 网站服务器接入网络运营商的链路所能提供的出口带宽。
2. 用户客户端接入网络运营商的链路所能提供的入口带宽。
3. 从网站到用户之间经过的不同运营商之间互联节点的带宽，一般来说两个运营商之间只有固定的若干个点是互通的，所有跨运营商之间的交互都要经过这些点。
4. 从网站到用户之间的物理链路传输时延。打游戏的同学都清楚，ping比流量更重要。

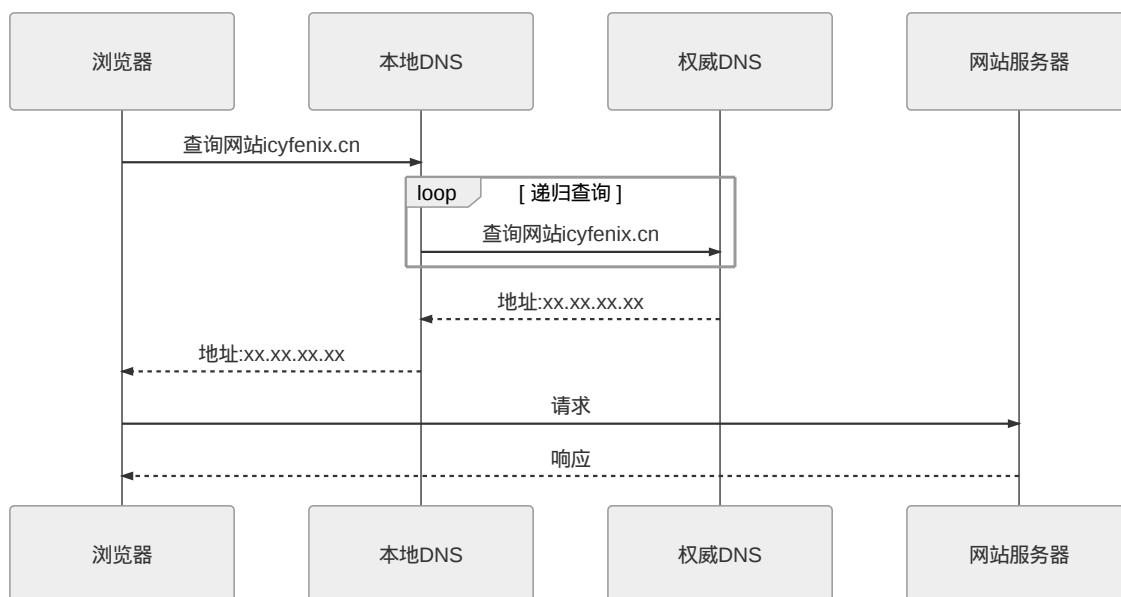
以上四个网络问题，除了第二个只能由用户掏腰包装个更好的宽带才能够解决之外，其余三个都能通过内容分发网络来显著改善的。一个工作良好的CDN，能为互联网系统解决跨运营商、跨地域物理距离所导致的时延问题，能为网站流量带宽起到分流、减负的作用。

如果不是有遍布全国的阿里云CDN网络支持，哪怕把整个杭州所有市民上网的权力都剥夺，带宽全部让给淘宝，恐怕也撑不住双十一全国乃至全球用户的疯狂围攻。

CDN的工作过程，主要涉及到路由解析、内容分发、负载均衡（由于后面专门有一节讨论负载均衡的内容，所以这部分在CDN中就暂不涉及）和所能支持的应用内容四个方面，下面我们就逐一了解。

路由解析

根据我们在第二节中对DNS系统的介绍，一个未使用CDN的用户访问网站的过程应该是这样的：



以上时序所表达的内容跟第二节中讲述的没有差异，这里仅列作对比，不再赘述。下面分析使用了CDN的DNS查询过程之前，我们先来看一段对本站进行DNS查询的实际应答。通过dig或者host命令，可以很方便地得到DNS服务器的返回结果（结果中头4个IP的地址是我手工加入的，后面的就不一个一个查了），如下所示：

```

$ dig icyfenix.cn
sh

; <>> DiG 9.11.3-1ubuntu1.8-Ubuntu <>> icyfenix.cn
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60630
;; flags: qr rd ra; QUERY: 1, ANSWER: 17, AUTHORITY: 0, ADDITIONAL: 1

```

```

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;icyfenix.cn.           IN      A

;; ANSWER SECTION:
icyfenix.cn.        600    IN      CNAME
icyfenix.cn.cdn.dnsv1.com.
icyfenix.cn.cdn.dnsv1.com. 599    IN      CNAME
4yi4q4z6.dispatch.spcdntip.com.
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      101.71.72.192      #浙江宁波市
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      113.200.16.234      #陕西省榆林市
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      116.95.25.196      #内蒙古自治区呼和浩特市
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      116.178.66.65      #新疆维吾尔自治区乌鲁木齐市
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      118.212.234.144
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      211.91.160.228
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      211.97.73.224
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      218.11.8.232
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      221.204.166.70
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      14.204.74.140
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      43.242.166.88
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      59.80.39.110
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      59.83.204.12
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      59.83.204.14
4yi4q4z6.dispatch.spcdntip.com. 60    IN      A      59.83.218.235

;; Query time: 74 msec
;; SERVER: 127.0.0.53#53(127.0.0.53)
;; WHEN: Sat Apr 11 22:33:56 CST 2020
;; MSG SIZE  rcvd: 152

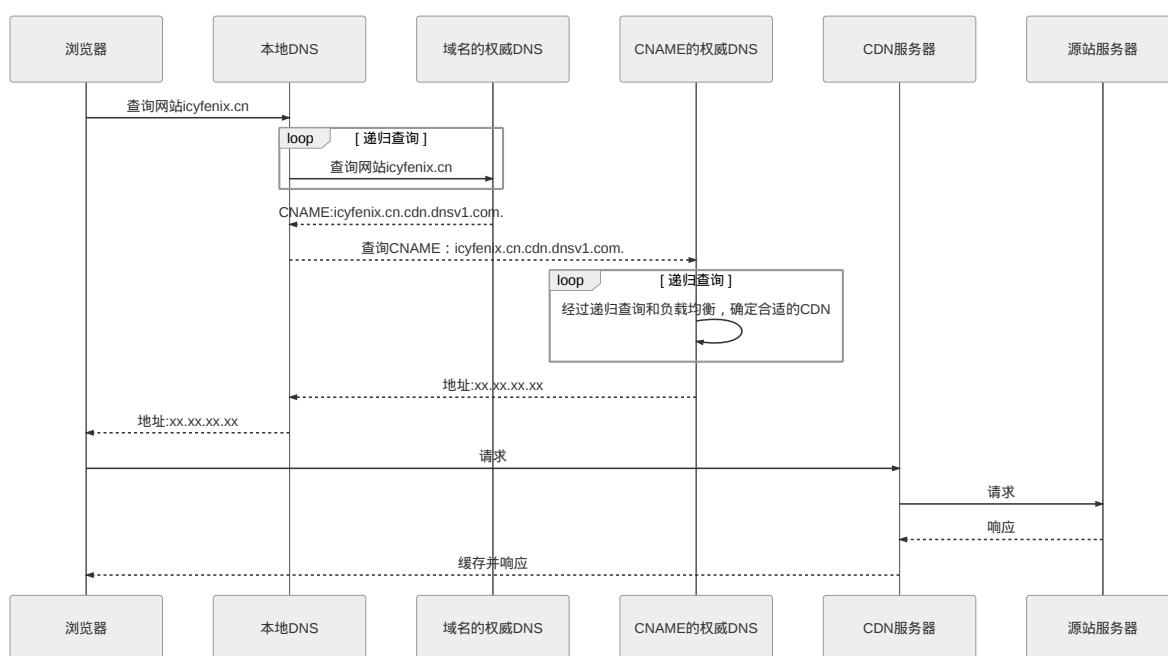
```

根据以上信息，查询“icyfenix.cn.”的查询结果首先返回了一个CNAME记录（icyfenix.cn.cdn.dnsv1.com.），递归查询该CNAME时候，返回了另一个看起来更奇怪的CNAME（4yi4q4z6.dispatch.spcdntip.com.），最后，这个CNAME返回了十几个位于全国不同地区的A记录，很明显，那些A记录就是存有本站缓存的CDN节点。CDN路由解析的工作过程是：

1. 架设好服务器后，将服务器的IP地址在你的CDN服务商上注册为“源站”，注册后你会得到一个CNAME，即本例中的“icyfenix.cn.cdn.dnsv1.com.”。

2. 将得到的CNAME在你购买域名的DNS服务商上注册为一条CNAME记录。
3. 当发生一次未命中缓存的DNS查询时，域名服务商解析出CNAME后，返回给本地DNS，至此之后链路解析的主导权就开始由CDN的调度服务接管了。
4. 本地DNS查询CNAME时，能解析该CNAME的权威服务器只有CDN服务商的权威DNS，该DNS会根据一定的均衡策略和参数，如拓扑结构、容量、时延等，在全国各地能提供服务的CDN节点中挑选一个适合的，将它的IP返回给本地DNS。
5. 浏览器从本地DNS拿到IP，将该IP当作源站服务器来进行访问，此时该IP的CDN服务上可能有，也可能没有缓存源站的资源，这点将在稍后“内容分发”部分讨论。
6. CDN代替源站向用户提供所需的资源。

以上步骤反映在时序图上，将如下图所示：



内容分发

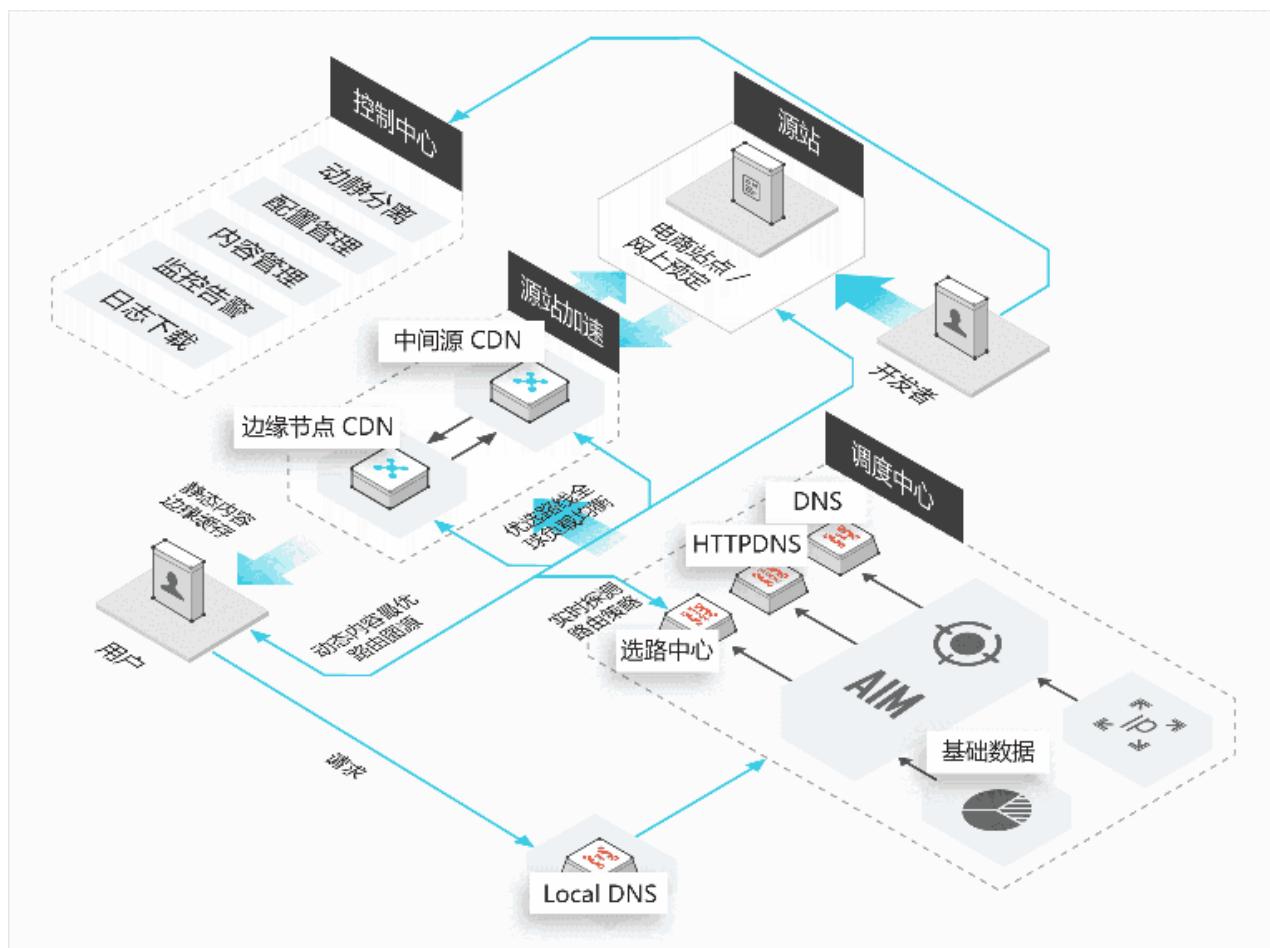
通过智能化的路由解析，CDN节点实现了无论是对用户端还是服务端，都完全透明地中间接管了用户向服务发出的资源请求，后面随之而来的下一个问题是CDN中必须缓存有用户想要请求的资源，然后才能代替源站来满足用户的资源请求。这包括了两个大的问题：“如何获取源站资源”和“如何管理（更新）资源”。

CDN获取源站资源的过程被称为“内容分发”，目前主要有两种主流的内容分发方式：

- **主动分发（Push）**：分发由源站主动发起，将内容从源站或者其他资源库推送到各边缘的CDN节点上。这个推送的过程没有什么技术标准，可以采用任何传输方式（HTTP、F

TP、P2P，等等）、任何推送策略（满足特定条件、定时、人工，等等）、任何推送时间，只要与后面说的更新策略先匹配即可。由于主动分发（通常）需要源站、CDN服务双方提供程序API接层面的配合，它并不是透明的，一般用于系统预载大量数据资源。譬如双十一之前一段时间内，淘宝、京东等各个网络商城就会开始把未来活动中需要用到的资源推送到CDN上，特别常用的资源甚至会直接缓存到你的手机APP、你浏览器的localStorage上。

- **被动回源 (Pull)**：回源由用户访问触发，由CDN服务发起。当某资源第一次被用户访问的时候，CDN会实时从源站获取，这时资源的响应时间可粗略认为是资源从源站到CDN的时间加上CDN到用户的时间之和，因此，被动回源的首次访问通常是比较慢（但并不一定比你直接访问源站慢）的，不适合应用于大量的数据资源。但被动回源可以做到完全透明，不需要源站在程序上做任何的配合，使用起来较为方便，这种是小型站点使用CDN服务的主流选择（不是自建CDN，购买阿里云、腾讯云的CDN服务，多数就是这种方式）。



阿里云官网上的的CDN介绍

对于“CDN如何管理（更新）资源”这个问题，同样没有统一的标准可言，尽管在HTTP协议中关于缓存的Header定义中确实是有对CDN这类共享缓存的一些指引性参数（如Cache-C

ontrol的s-maxage），但是否要遵循，完全取决于CDN本身策略。更令人无奈的是，由于大多数网站的开发和运维本身并不了解HTTP缓存机制，所以CDN如果完全照着HTTP Header来控制缓存失效和更新，效果也会相当的差，还可能引发其他问题。因此，CDN缓存的管理并不存在通用的准则。

CDN应用

CDN是为了快速分发静态资源而设计的，但今天的CDN所能做的事情已经远远超越了开始建设时的目标，这部分无法展开逐一细说，只对现在CDN可以做的事情简要列举，以便有个总体认知：

- 加速静态资源：这是CDN本职工作。
- 安全防御：CDN在广义上可以视作你网站的堡垒机，源站只对CDN提供服务，由CDN来对外界其他用户服务，这样恶意攻击者就不容易直接威胁源站，CDN对防御某些攻击手段，如[DDoS攻击](#)的尤其有效。但需注意，将安全都寄托在CDN上本身是不安全的，一旦源站真实IP被泄漏，就会面临很高风险。
- 协议升级：不少CDN提供商都同时对接（代售CA的）SSL证书服务，可以实现源站是HTTP的，而对外开放的网站是基于HTTPS的。同理，可以实现源站到CDN是HTTP/1.x协议，CDN提供的外部服务是HTTPS/2.0协议、实现源站是基于IPv4网络的，CDN提供的外部服务支持IPv6网络，等等。
- 状态缓存：第一节介绍缓存时简要提到了一下状态缓存，CDN不仅可以缓存源站的资源，还可以缓存源站的状态，譬如源站的301/302转向，可以缓存起来，让客户端直接跳转、可以通过CDN开启HSTS、可以通过CDN进行[OCSP装订](#)加速SSL证书访问，等等。有一些情况下甚至可以配置CDN对任意状态码（譬如404）进行一定时间的缓存，以减轻源站压力，但这个操作应当慎重。
- 修改资源：CDN可以在返回资源给用户的时候修改它的任何内容，以实现不同的目的。譬如，可以对源站未压缩的资源自动压缩并修改Content-Encoding，以节省用户的网络带宽消耗、可以对源站未启用客户端缓存的内容加上缓存Header，以启用客户端缓存，可以修改[CORS](#)的相关Header，将源站不支持跨域的资源提供跨域能力，等等。
- 访问控制：CDN可以实现IP黑/白名单，根据不同的来访IP提供不同的响应结果，根据IP的访问流量来实现QoS控制、根据HTTP的Referer来实现防盗链，等等。
- 注入功能：CDN可以在不修改源站代码的前提下，为源站注入各种功能，下图是国际CDN巨头CloudFlare提供的Google Analytics、PACE、Hardenize等第三方应用，均无需

修改源站任何代码。

1. Install your first app

We recommend getting started by installing one of these great apps. It only takes a few

[Install](#)



Drift

Automatically turn your website traffic into qualified sales me

[Install](#)



Google Analytics

Deep insights into your site's traffic to improve user retention

[Install](#)



Hardenize

Security reports that help safeguard your site from attacks.

[Install](#)



PACE

A loading bar for your site to make it feel faster while improv

- 绕过某些不存在的网络措施，这也是在国内申请CDN也必须实名备案的原因，就不细说了。
-

负载均衡

负载均衡 (Load Balancing)

调度后方的多台机器，以统一的接口对外提供服务，承担此职责的技术组件被称为“负载均衡”。

互联网早期，业务流量比较小并且业务逻辑比较简单，单台服务器便可以满足基本的需求，但时至今日，互联网也好，企业也好，一般实际用于生产的系统，几乎都离不开集群了。信息系统不论是分布式单体架构还是微服务架构，不论是为了实现高可用还是为了获得高性能，都需要使用到多台机器来扩展服务能力，用户的请求不管连接到哪台机器上，都能得到相同的处理。另一方面，如何构建、调度服务集群这种事情，又应当对用户一侧保持足够的透明，即使用户的请求背后是由一千台、一万台机器来共同响应的，也并非用户所关心之事，他需记住的只有一个域名地址而已。调度后方的多台机器，以统一的接口对外提供服务，承担此职责的技术组件被称为“负载均衡” (Load Balancing)

真正大型系统的负载均衡过程往往是多级的。譬如，在各地建有多个机房，或机房有不同网络链路入口的大型互联网站，会从DNS解析开始，通过“域名” → “CNAME” → “负载调度服务” → “就近的数据中心入口”，先将来访地用户根据IP（或者其他条件）分配到一个合适的数据中心中，然后才到后续将要讨论的各式负载均衡。在DNS层面的负载均衡与前一节介绍的CDN，在工作原理上是类似的，其差别只是数据中心能提供的不仅有缓存，而是全方位的服务能力。由于这种方式此前已经详细介绍过，后续我们所讨论的“负载均衡”将聚焦于网络请求进入数据中心入口之后的其他级别的负载均衡。

无论在网关内部建立了多少级的负载均衡，从形式上来说都可以分为两种：四层负载均衡和七层负载均衡。在详细介绍它们是什么、如何工作之前，我们先来建立两个大致的、概念性的印象：

- 四层负载均衡优势是性能高，七层负载均衡的优势功能强。
- 做多级混合负载均衡，通常应是低层的负载均衡在前，高层的负载均衡在后（想一想为什么？）。

我们所说的“四层”、“七层”，指的是经典的[OSI七层模型](#)中第四层传输层和第七层应用层，下表是来自于维基百科上对OSI七层模型的介绍，这部分属于网络基础知识，笔者就不多解释了（英文也偷懒不翻译了）。请注意到表中各个名词特意保留的超链接，后面我们会多次使用到这张表，如你对网络知识并不是特别了解的，可通过这些连接获得进一步的资料。下面我们直接从四层负载均衡具体工作过程开始说起。

	Layer	Protocol Data Unit	Function
7	Application	Data	High-level APIs, including resource sharing, remote file access
6	Presentation	Data	Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption
5	Session	Data	Managing communication sessions, i.e., continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes
4	Transport	Segment	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing
3	Network	Packet	Structuring and managing a multi-node network, including addressing, routing and traffic control
2	Data link	Frame	Reliable transmission of data frames between two nodes connected by a physical layer
1	Physical	Symbol	Transmission and reception of raw bit streams over a physical medium

现在所说的四层负载均衡是多种工作模式的统称，“四层”的意思是说这些工作模式的共同特点是都维持着同一个TCP连接，而不是说它工作在第四层。事实上，这些模式其实都是工作在二层（数据链路层，改写MAC地址）和三层（网络层，改写IP地址）上的，单纯只处理第四层（传输层，可以改写TCP、UDP等协议的内容和端口）的数据无法做到负载均衡的转发，因为OSI的下三层是媒体层（Media Layers），上四层是主机层（Host Layers），既然流量都已经到达目标主机上了，也就谈不上什么流量转发，最多只能做代理了。但出于习惯和方便，现在几乎所有的资料都把它们统称为四层负载均衡，笔者也同样称呼

它为四层负载均衡，在这里提示一下，以免读者在某些资料上看见“二层负载均衡”、“三层负载均衡”这样的表述，误以为和这里说的“四层负载均衡”是一类意思。下面笔者介绍几种常见的四层负载均衡的工作模式。

数据链路层负载均衡

参考上面OSI模型的表格，数据链路层传输的内容是数据帧（Frame），譬如以常见的太网帧、ADSL宽带的PPP帧等。我们讨论的上下文场景里，目标肯定就是以太网帧，按照IEE E 802.3标准，以最典型的1500 Bytes MTU的以太网帧结构为例：

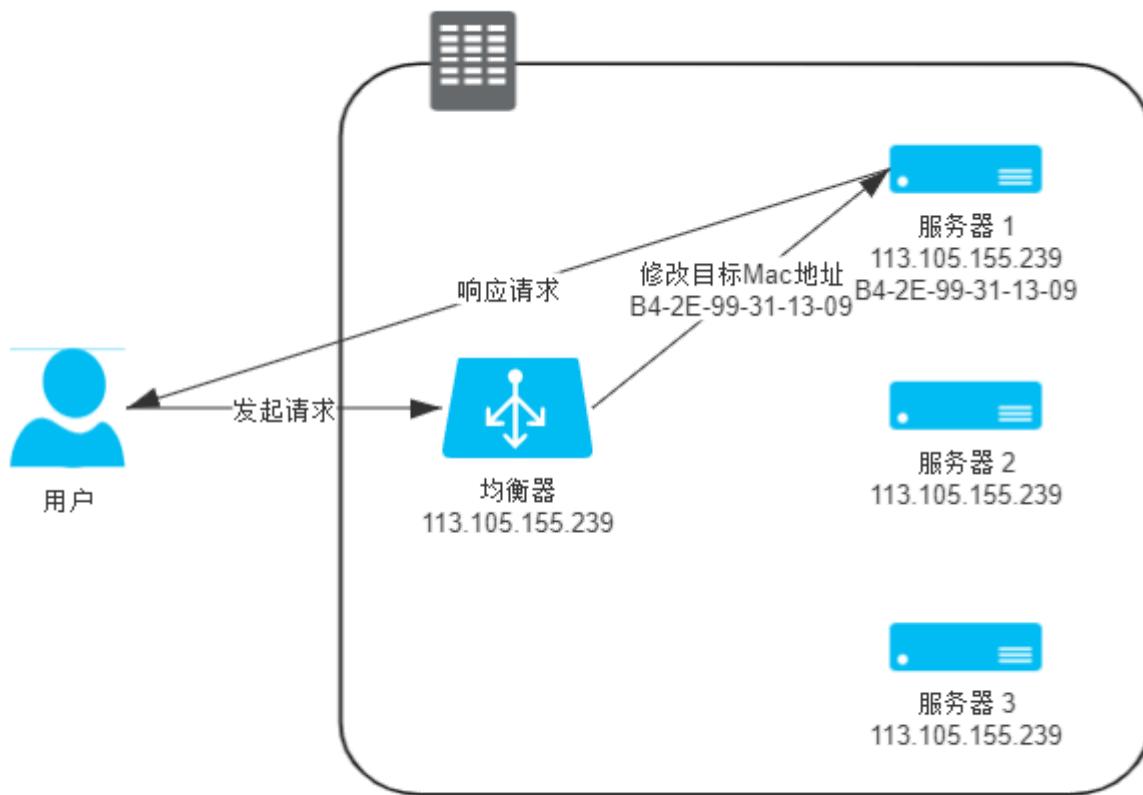
前导码	帧开始符	MAC目标地址	MAC源地址	802.1Q <u>标签(可选)</u>	以太类型	有效负载	冗余校验	帧间距
101 010 10 7 Bytes	101 010 11 1 Bytes	6 Bytes	6 Bytes	(4 Bytes)	2 Bytes	1500 Bytes	4 Bytes	12 Bytes

帧中其他结构项的含义我们可以不去理会，只需注意到“MAC目标地址”和“MAC源地址”两项即可。我们知道每一块网卡都有独立的MAC地址，以太帧上这两个地址告诉了交换机，此帧应该是从连接在交换机上的哪个端口的网卡发出，送至哪块网卡的。

数据链路层负载均衡所做的事情，是修改请求的数据帧中的MAC目标地址，让用户原本是发送给负载均衡器的请求的数据帧，被二层交换机根据新的MAC目标地址转发到服务器集群中对应的服务器（后文称为“真实服务器”，Real Server）的网卡上，这样真实服务器就获得了一个原本目标并不是发送给它的请求。

由于二层负载均衡器在转发请求过程中只修改了帧的MAC目标地址，不涉及更上层协议（没有修改Payload的数据），所以在更上层（第三层）看来，所有数据都是未曾被改变过的。由于第三层的数据包（IP数据包，下节会介绍）中包含了源（客户端）和目标（均衡器）的IP地址，只有真实服务器保证自己的IP地址与数据包中的目标IP地址一致，这个数据包才能被正确处理。因此，使用这种负载均衡模式时，将会把真实物理服务器集群所有机器的虚拟IP地址（Virtual IP Address，VIP）配置成与负载均衡服务器虚拟IP一样，这样经转发后的数据就能在真实服务器中顺利地使用。也正是因为实际处理请求的真实物理

服务器IP和数据请求中的目的IP是一致的，所以响应结果便不需要通过负载均衡服务器进行地址交换，可将响应结果的数据包直接从真实服务器返回给用户，避免负载均衡服务器网卡带宽成为瓶颈，数据链路层的负载均衡效率也是最高的。整个请求到响应的过程如下图所示：



数据链路层负载均衡

上述只有请求经过负载均衡器，而服务响应无需从负载均衡器原路返回的工作模式，整个请求、转发、响应的链路形成一个“三角关系”，所以这种负载均衡模式也常被很形象地称为“三角传输模式”（Direct Server Return，DSR），也有叫“单臂模式”（Single Legged Mode）或者“直接路由”（Direct Routing）。

虽然数据链路层负载均衡效率很高，但它并不能适用于所有的场合，除了那些需要感知应用层协议信息的负载均衡场景它无法胜任外（所有的四层负载均衡器都无法胜任，将在后续介绍七层均衡器时一并解释），它在网络一侧受到的约束也很大。它直接改写目标MAC地址的工作原理决定了它与真实的服务器的通讯必须是二层可达的，说白了就是必须在同一个子网当中，无法跨VLAN。优势（效率高）和劣势（不能跨子网）决定了数据链路层负载均衡最适合用来做数据中心的第一级（这里只谈负载均衡设备，并没有把ECMP等价路由算进去）负载均衡。

网络层负载均衡

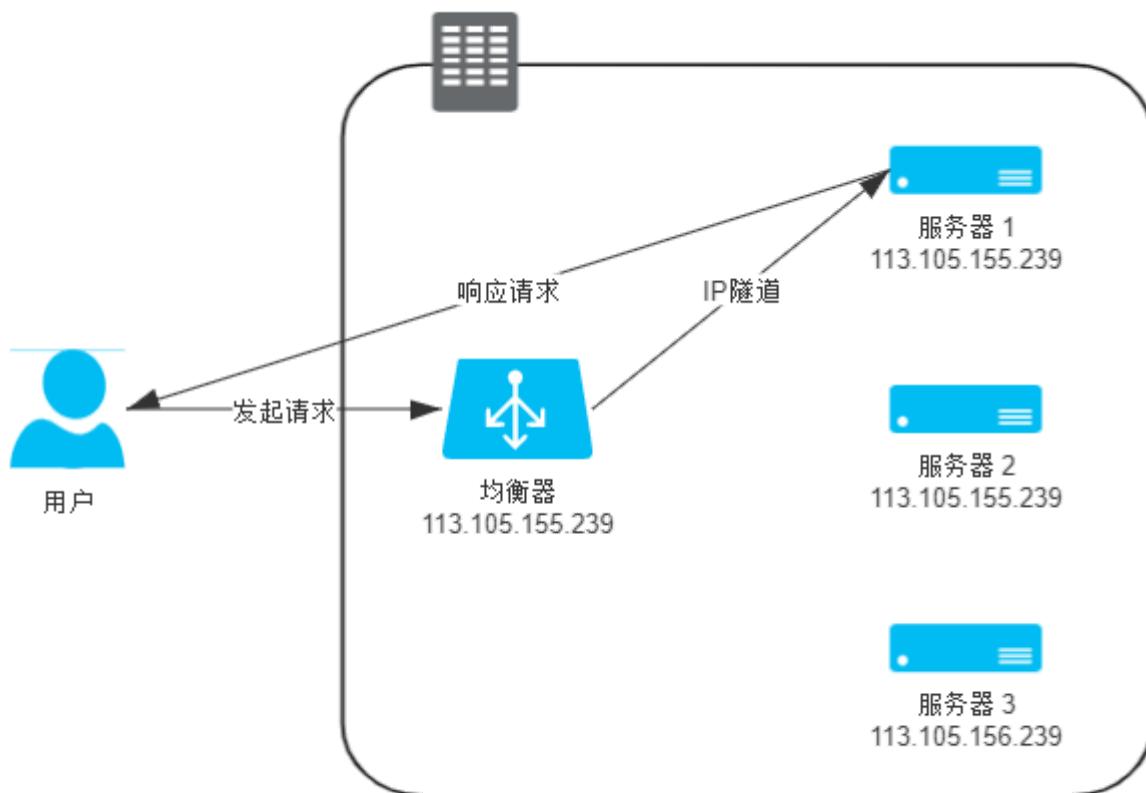
根据OSI七层模型，在第三层网络层传输的就是分组数据包（Packet），这是一种在分组交换网络（Packet Switching Network，PSN）中传输的结构化数据单位。以IP协议为例，一个IP数据包由Header和Payload两部分组成，其中Header长64 Bytes，其中包括了24 Bytes的固定数据和最长不超过40 Bytes的可选数据组成。按照IPv4标准，一个典型的分组数据包的Header部分如下表所示：

长度	存储信息
0-4 Bytes	版本号（4 Bits）、首部长度（4 Bits）、分区类型（8 Bits）、总长度（16 Bits）
5-8 Bytes	报文计数标识（16 Bits）、标志位（4 Bits）、片偏移（12 Bits）
9-12 Bytes	TTL生存时间（8 Bits）、上层协议代号（8 Bits）、首部校验和（16 Bits）
13-20 Bytes	源地址（32 Bits）
21-24 Bytes	目标地址（32 Bits）
25-64 Bytes	可选字段和空白填充

同样，我们对于表格中其他信息无需过多关心，只要知道在IP分组数据包的头部带有源和目标的IP地址即可。源和目标IP地址代表了数据是从分组交换网络中哪台机器发送到哪台机器，那我们就可以用之前同样的思路，通过改变这里面的地址来实现数据包的转发。有两种很容易想到的修改方式，其一是保持原来的数据包不变，新创建一个数据包，把原来数据包的Header和Payload整体作为另一个新的数据包的Payload，在这个新数据包的Header中写入真实服务器的IP作为目标地址，再把这个数据包发送出去。经过三层交换机的转发，真实服务器收到数据包后，要在接收入口处设计一个针对性的拆包机制，把由负载均衡器加入的那层Header扔掉，还原出原来的数据包来使用。这样，这台服务器就同样拿到了一个原本不是发给它（目标IP不是它）的数据包，达到了流量转发的目的。估计是那时

候还没有流行起“[禁止套娃](#)”的梗，所以设计者给这种“套娃式”的传输起名叫做“[IP隧道](#)”（ IP Tunnel ）传输，也还是相当的形象。

尽管由于要封装新的数据包，IP隧道的转发模式比起直接路由模式效率会有所下降，但由于并没有修改原有数据包中的任何信息，所以IP隧道的转发模式仍然具备三角传输的特性，即负载均衡器转发来的请求，可以由真实服务器直接应答，无需在经过均衡器原路返回。而且由于IP隧道工作在网络层，所以可以跨越VLAN，因此摆脱了直接路由模式中网络侧的约束。此模式从请求到响应的过程如下图所示：

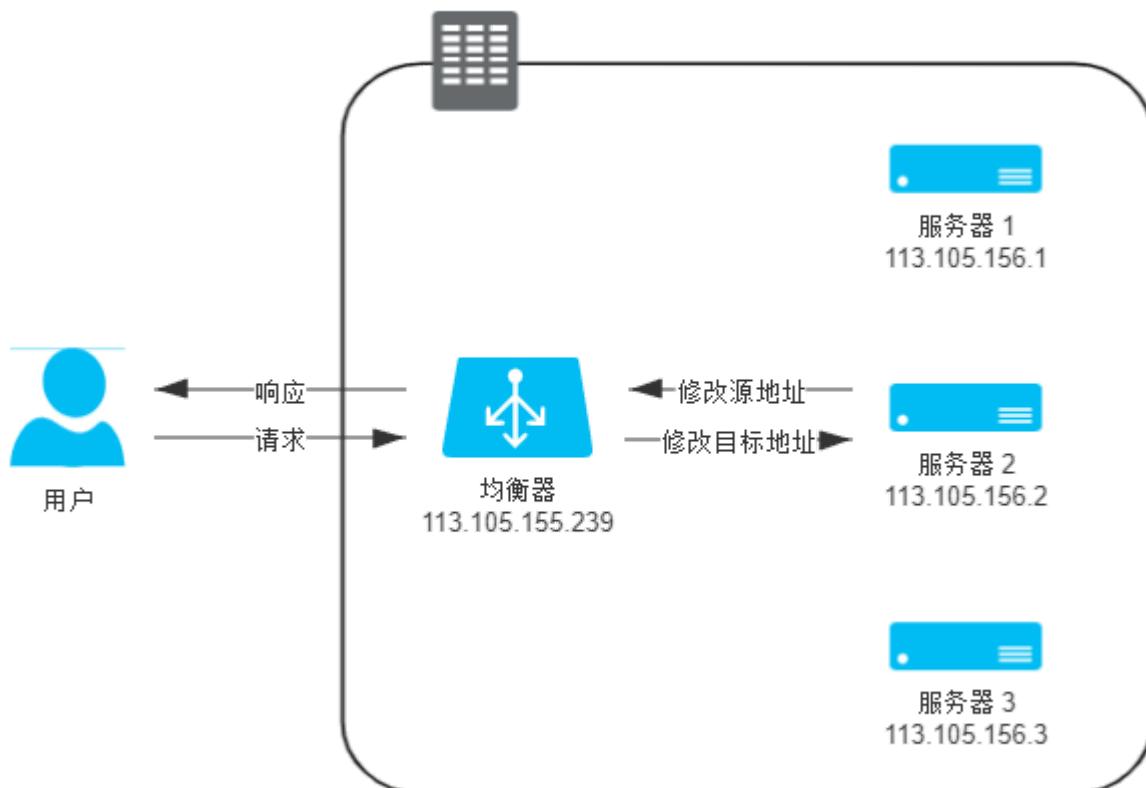


IP隧道模式的负载均衡

而这种方式的缺点是它要求真实服务器必须得支持[IP隧道协议](#)（ IP Encapsulation ）协议，就是它得会自己拆包扔掉一层Header，这个其实并非什么大问题，现在几乎所有的*nix系统都支持IP隧道。而另外一个问题是这种模式仍必须通过专门的配置，保证所有的真实服务器与均衡器有着相同的虚拟IP地址，因为回复该数据包时，需要使用这个虚拟IP作为响应数据包的源地址，这样客户端收到这个数据包时才能正确解析。这个限制就比较讨厌了，它与“透明”这个原则有冲突。

对服务器进行虚拟IP的配置并不是在任何情况下都可行的，尤其是当有好几个服务共用一台物理服务器的时候。此时就需要考虑另一种改变目标数据包的方式：直接把数据包Head

er中的目标地址改掉。这样原本由用户发给均衡器的数据包，也会被三层交换机转发到真实服务器手上，而且因为没有经过IP隧道的额外包装，也就不再需要再拆包了。但现在问题是由于这种模式修改了目的IP地址才到达真实服务器的，如果真实服务器直接将应答包发回给客户端的话，这个应答数据包的源IP是真实服务器的IP，也即是均衡器修改后的那个IP地址，客户端肯定就无法正常处理这个应答。因此，只能让应答流量继续回到负载均衡，负载均衡把应答包的源IP改回自己的IP再发到客户端，这样才可以保证正常通信。如果你对网络知识有了解的话，肯定会觉得这种处理似曾相识，没错，这不就是在家里、公司、学校上网时，由一台路由器带着一群内网机器上网的“[网络地址转换](#)”（ Network Address Translation , NAT ）操作吗？此时，负载均衡器就是充当了家里、公司、学校的上网路由器的作用，所以，只要机器将自己的网关地址设置为均衡器地址，就无需再进行任何设置了。这种负载均衡的模式的确就被称为是NAT模式，此模式从请求到响应的过程如下图所示：



NAT模式的负载均衡

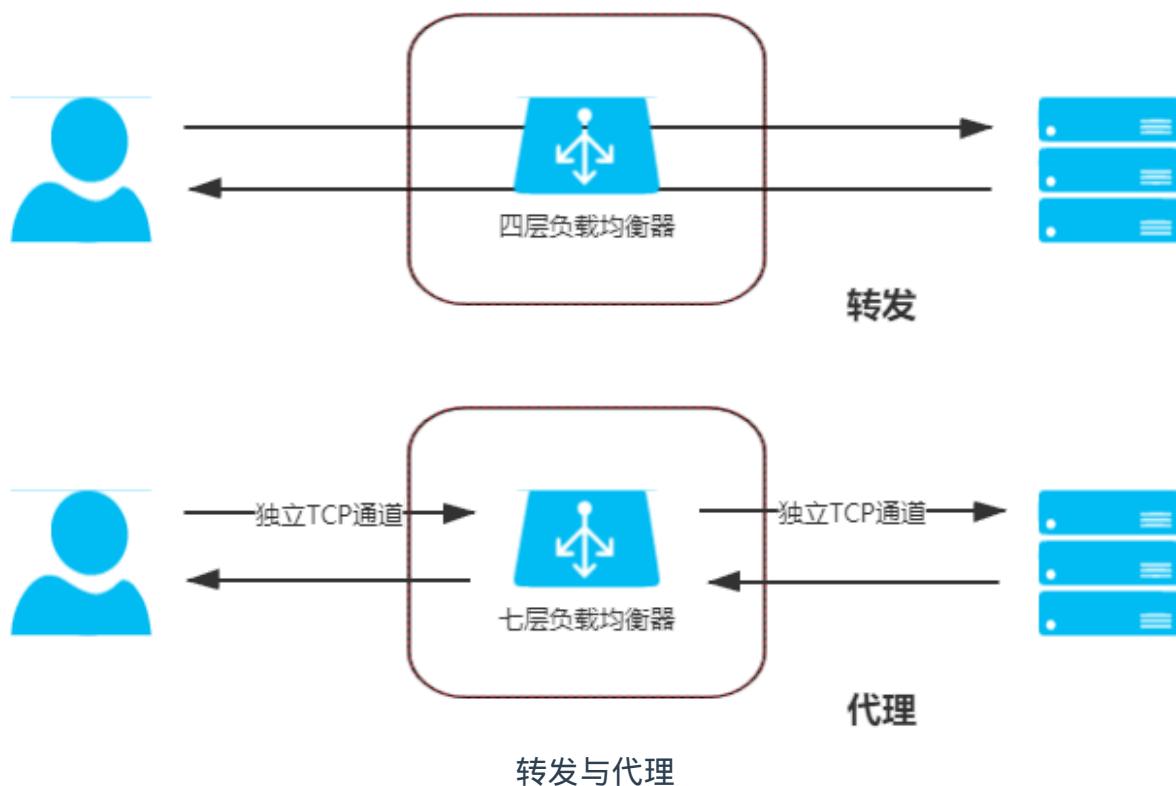
NAT模式的负载均衡会带来较大的（比起直接路由和IP隧道模式，往往是数量级上的下降）性能损失，这点是显而易见的，由负载均衡器代表整个服务集群来进行应答，各个服务器的响应数据都会互相争抢均衡器的出口带宽，这就好比在家里用NAT上网的话，如果

由人在下载，你打游戏可能就会觉得卡是一个道理，此时整个系统的瓶颈很容易就出现在负载均衡器上。

还有一种更加彻底的NAT模式，均衡器在转发时，不仅修改目标IP地址，连源IP地址也一起改了，源地址就改成均衡器自己的IP，称作Source NAT（SNAT）。这样好处是真实服务器连网关都无需配置了，可以让应答流量经过正常的三层路由回到负载均衡器上，做到了彻底的透明。但是缺点是由于做了SNAT，真实服务器处理请求时就无法拿到客户端的IP地址，在真实服务器的视角看来，所有的流量都来自于负载均衡器，这样有一些需要根据目标IP进行控制的业务逻辑就无法进行了。

应用层负载均衡

前面介绍的四层负载均衡工作模式都属于“转发”，即直接将承载着TCP报文的底层数据（IP数据包或以太帧）转发到真实服务器上，此时客户端到响应请求的真实服务器维持着同一条TCP通道。但工作在四层之后的负载均衡模式就无法再进行转发了，只能进行代理，转发与代理的区别如下图所示：



“代理”这个词，根据“哪一方能感知到”的原则，可以分为“正向代理”、“反向代理”和“透明代理”三类。正向代理就是我们通常简称的代理，指在客户端设置的、代表客户端与服务器通

讯的代理服务，它是客户端可知，而对服务器透明的。反向代理是指在设置在服务器这一侧，代表真实服务器来与客户端通讯的代理服务，此时它对客户端来说是透明的。至于透明代理是指对双方都透明的，配置在网络中间设备上（譬如，架设在路由器上的透明翻墙代理）的代理服务。

根据以上定义，很显然，七层负载均衡器它就属于一种反向代理，如果只论网络性能，七层均衡器肯定是无论如何比不过四层均衡器的，它比四层均衡器至少多一轮TCP握手，有着跟NAT模式一样的带宽问题，而且通常要耗费更多的CPU（因为可用的解析规则远比四层丰富），所以如果用七层均衡器去做下载站、视频站这种流量应用是不合适的（起码不能作为第一级均衡器）。但是，如果网站的性能短板并不在于网络性能，要论整个服务集群对外所体现出来的服务性能，七层均衡器就有它的用武之地了。这里面七层均衡器的底气就是来源于它工作在应用层，可以感知应用层通讯的内容，往往能够做出更明智的决策，玩出更多的花样来。

举个生活中的例子，四层均衡器就像银行的自助排号机，转发效率高且不知疲倦，每一个达到银行的客户根据排号机的顺序，选择对应的窗口接受服务；而七层均衡器就像银行大堂经理，他会先确认客户需要办理的业务，再安排排号。这样办理理财、存取款等业务的客户，会根据银行内部资源得到统一协调处理，加快客户业务办理流程，有一些无需柜台办理的业务，甚至大堂经理直接就可以解决了（譬如，反向代理的静态资源缓存）。

相信代理的工作模式大家应该是比较熟悉的，因此关于七层均衡器如何工作的就不作详细介绍了，笔者列举了一些七层代理可以实现的功能，以便读者对它“功能强大”有个直观的感受：

- 前面介绍CDN应用时，所有CDN可以做的缓存类工作（就是除去CDN就近返回这种优化链路的工作外），七层均衡器都可以实现，譬如静态资源缓存、协议升级、安全防护、访问控制，等等。
- 七层均衡器可以实现更智能化的路由。譬如，根据Session路由，以实现亲和性的集群；根据URL路由，实现专职化服务（如Kubernetes Ingress均衡器就属于这类）；甚至根据用户身份路由，实现对部分用户的特殊服务（如某些站点的贵宾服务器），等等。
- 某些安全攻击可以由七层均衡器来解决，譬如一种常见的DDoS手段是SYN Flood攻击，即攻击者控制众多客户端，使用虚假IP地址对同一目标大量发送SYN报文。从技术原理上看，由于四层均衡器无法感知上层协议的内容，这些SYN攻击都会被转发到后端

的真实服务器上；而七层均衡器下这些SYN攻击自然在负载均衡设备上就截止，不会影响到后面服务器的正常运行。类似地，可以在七层均衡器上设定多种策略，譬如过滤特定报文，以防御如SQL注入等应用层面的特定攻击手段。

- 多数微服务中的链路治理措施，都需要在七层中进行，譬如服务降级、熔断、异常注入等等。譬如，一台服务器只有出现物理层面或者系统层面的故障，导致无法应答TCP请求才能被四层均衡器所感知，进而剔除出服务集群，如果一台服务器一直在报500错，那四层均衡器对此是完全无能力的，只能由七层均衡器来解决。

均衡器实现与策略

负载均衡的两大职责是“选择谁来处理用户请求”和“将用户请求转发过去”。到此我们仅介绍了后者，即请求的转发或代理过程。前者是指均衡器所采取的均衡策略，这一块较为接近于实现细节，笔者就不展开了，常见的均衡策略有：

- **轮循均衡** (Round Robin)：每一次来自网络的请求轮流分配给内部中的服务器，从1至N然后重新开始。此种均衡算法适合于服务器组中的所有服务器都有相同的软硬件配置并且平均服务请求相对均衡的情况。
- **权重轮循均衡** (Weighted Round Robin)：根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。譬如：服务器A的权值被设计成1，B的权值是3，C的权值是6，则服务器A、B、C将分别接受到10%、30%、60%的服务请求。此种均衡算法能确保高性能的服务器得到更多的使用率，避免低性能的服务器负载过重。
- **随机均衡** (Random)：把来自客户端的请求随机分配给内部中的多个服务器，在数据足够大的场景能达到一个均衡分布。
- **权重随机均衡** (Weighted Random)：此种均衡算法类似于权重轮循算法，不过在处理请求分担时是个随机选择的过程。
- **一致性哈希均衡** (Consistency Hash)：根据请求中某一些数据（可以是MAC、IP地址，也可以是高层协议中的某些信息）作为特征值来计算需要落在的结点上，可以保证一个同一个特征值一定落在相同的服务器上。一致性的意思是保证当服务集群某个真实服务器出现故障，只影响该服务器的哈希，而不会导致整个服务集群的哈希键值重新分布。
- **响应速度均衡** (Response Time)：负载均衡设备对内部各服务器发出一个探测请求（例如Ping），然后根据内部中各服务器对探测请求的最快响应时间来决定哪一台服务

器来响应客户端的服务请求。此种均衡算法能较好的反映服务器的当前运行状态，但最快响应时间仅仅指的是负载均衡设备与服务器间的最快响应时间，而不是客户端与服务器间的最快响应时间。

- **最少连接数均衡 (Least Connection)**：客户端的每一次请求服务在服务器停留的时间可能会有较大的差异，随着工作时间加长，如果采用简单的轮循或随机均衡算法，每一台服务器上的连接进程可能会产生极大的不同，并没有达到真正的负载均衡。最少连接数均衡算法对内部中需负载的每一台服务器都有一个数据记录，记录当前该服务器正在处理的连接数量，当有新的服务连接请求时，将把当前请求分配给连接数最少的服务器，使均衡更加符合实际情况，负载更加均衡。此种均衡策略适合长时处理的请求服务，如FTP。
-

负载均衡器的实现有“软件均衡器”和“硬件均衡器”两类。在软件均衡器方面，又分为直接建设在操作系统内核的均衡器和应用程序形式的均衡器两种。前者的代表是LVS (Linux Virtual Server)，后者的代表有Nginx、HAProxy、KeepAlived等，前者性能会更好（数据包从网卡开始到达应用为止这段路径，会经过层层的协议处理和运行钩子的过程），后者选择广泛，使用方便。在硬件均衡器方面，往往会直接采用[应用专用集成电路](#) (Application Specific Integrated Circuit , ASIC) 来实现，有专用处理芯片的支持，避免操作系统层面的损耗，得以达到最高的性能。这类的代表是F5和A10公司的负载均衡产品。

缓存

缓存 (Cache)

软件开发中的缓存并非多多益善，它有收益，也有风险。

在引入缓存之前，第一件事情是确认你的系统是否真的需要缓存。很多人会有意无意地把硬件里那种常用于区分不同产品档次、“多多益善”的缓存（如CPU L1/2/3缓存、磁盘缓存，等等）代入到软件开发中去，实际上这两者差别很大，在软件开发中引入缓存的负面作用要明显大于硬件的缓存：从开发角度来说，引入缓存会提高系统复杂度，因为你要考虑缓存的失效、更新、一致性等问题（硬件缓存也有这些问题，只是不需要由你去考虑，主流的ISA都没有提供任何直接操作缓存的汇编指令，而是由MESI¹这类缓存协议去规范处理）；从运维角度来说，缓存会掩盖掉一些缺陷，让问题在更久的时间以后，出现在距离发生现场更远的位置上；从安全角度来说，缓存可能泄漏某些保密数据，也是容易受到攻击的薄弱点。说服你引入缓存的理由，总结起来无外乎以下两种：

- 为缓解CPU压力而做缓存：譬如把方法运行结果存储起来、把原本要实时计算的内容提前算好、把一些公用的数据进行复用，这可以节省CPU算力，顺带提升响应性能。
- 为缓解I/O压力而做缓存：譬如把原本对网络、磁盘等较慢介质的读写访问变为对内存等较快介质的访问，将原本对单点部件（如数据库）的读写访问变为到可扩缩部件（如缓存中间件）的访问，顺带提升响应性能。

请注意，缓存虽然是典型以空间换时间提升性能的手段，但它的出发点是缓解CPU和I/O资源在峰值流量下的压力，“顺带”而非“专门”地提升响应性能。这里的言外之意是如果可以通过增强CPU、I/O本身的性能（譬如扩展服务器的数量）来满足需要的话，那升级硬件往往是更好的解决方案，即使需要你掏腰包花费一些额外成本，也往往要优于引入缓存后可能带来的风险。

缓存属性

有不少软件系统最初的缓存功能是以HashMap或者ConcurrentHashMap为起点开始的演进的。当开发人员发现系统中某些资源的构建成本比较高，而这些资源又有被重复使用的可能性，那很自然就会产生“循环再利用”的想法，将它们放到Map容器中，下次需要时取出重用，避免重新构建，这种原始朴素的复用就是最基本的缓存了。不过，一旦我们专门把“缓存”看作一项技术基础设施，一旦它有了通用、高效、可统计、可管理等方面的需求，其中要考虑的因素就变得复杂起来。通常，我们设计或者选择缓存至少会考虑以下四个维度的属性：

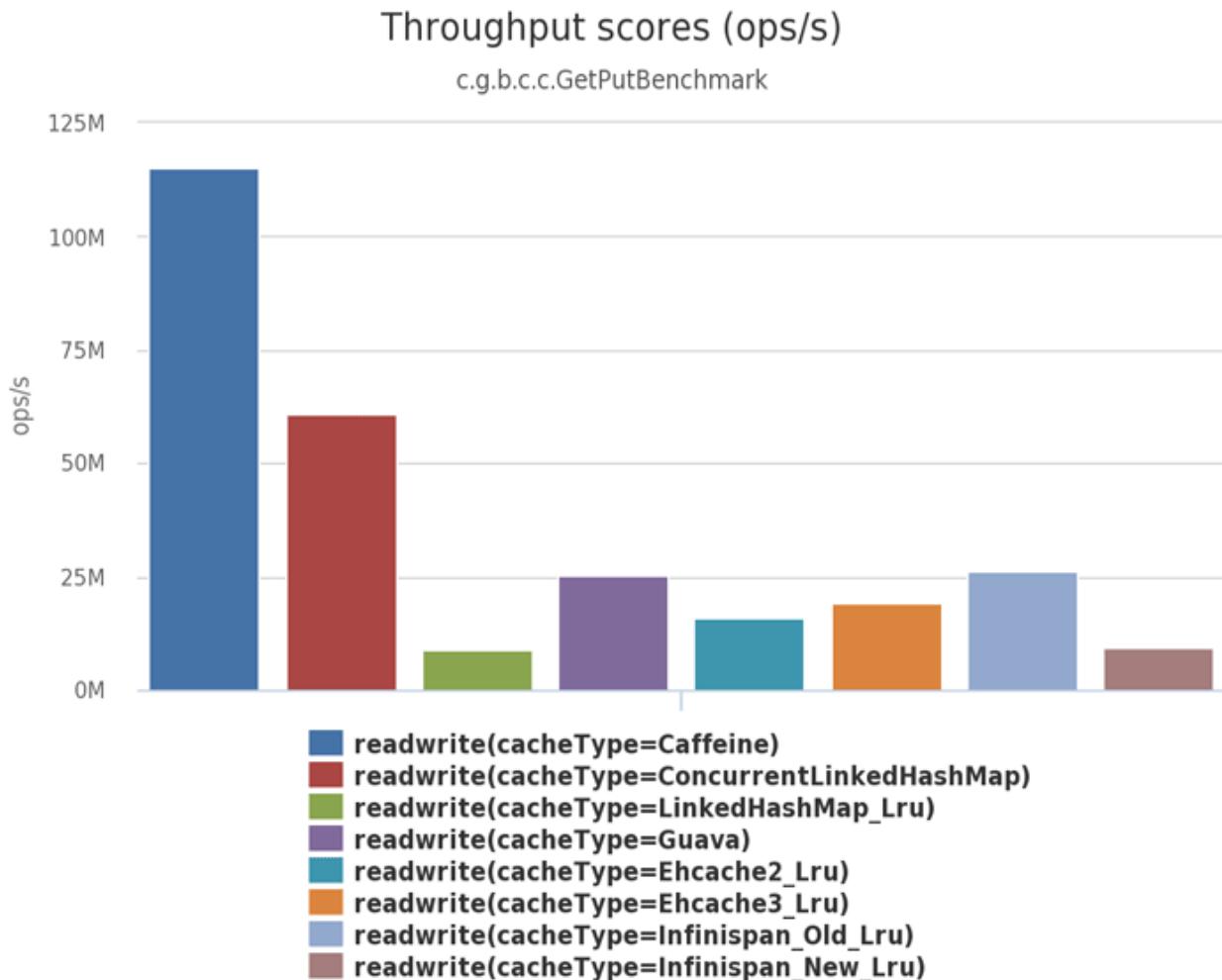
- **吞吐量**：缓存的吞吐量使用OPS值（每秒操作数，Operations per Second，ops/s）来衡量，反映了对缓存进行并发读、写操作的效率，即缓存本身的工作效率高低。
- **命中率**：缓存的命中率即成功从缓存中返回结果次数与总请求次数的比值，反映了引入缓存的价值高低（命中率越低，引入缓存的收益越小，价值越低）。
- **扩展功能**：缓存除了基本读写功能外，还提供哪些额外的管理功能，譬如最大容量、失效时间、失效事件、命中统计，等等。
- **分布式支持**：缓存可分为“进程内缓存”和“分布式缓存”两大类，前者只为节点本身提供服务，无网络访问操作，速度快但缓存的数据不能在各个服务节点中共享，后者则相反。

吞吐量

缓存的吞吐量只在并发场景中才有统计的意义，因为即使是最原始的以HashMap实现的缓存，访问效率也已经是常量时间复杂度（即O(1)，其中涉及到碰撞、扩容等场景的处理属于数据结构基础，这里就不展开）。但HashMap并不是线程安全的容器，如果要让它在多线程并发下能正确地工作，就要用Collections.synchronizedMap进行包装，这相当于给Map接口的所有访问方法都自动加全局锁；或者改用ConcurrentHashMap来实现，这相当于给Map的访问分段加锁（从JDK 8起已取消分段加锁，改为CAS+Synchronized锁单个元素）。无论采用怎样的实现方法，线程安全措施都会带来一定的吞吐量损失。

如果只比较吞吐量，完全不去考虑命中率、淘汰策略、缓存统计、过期等功能该如何实现，那不必选择，JDK 8改进之后的ConcurrentHashMap基本上就是吞吐量最高的缓存容器了。可是很多场景里，以上提及的功能至少有部分一两项是必须的，不能不考虑。根据Caffeine给出的一组目前业界主进程中流缓存实现方案（包括有Caffeine、ConcurrentLinkedHashMap、LinkedHashMap、Guava Cache、Ehcache和Infinispan Embedded）的对比，它们在8线程、75%读操作、25%写操作下的吞吐量表现Benchmarks¹来看，各种缓

存方案的性能差异还是十分明显的，最高与最低的相差了有一个数量级，具体如下图所示。



8线程、75%读、25%写的吞吐量比较（图片来自[Caffeine](#)）

这种并发读写的场景中，吞吐量受多方面因素的共同影响，譬如，怎样设计数据结构以尽可能避免数据竞争，存在竞争风险时怎样处理同步（主要有使用锁实现的悲观同步和使用CAS实现的乐观同步）、如何避免伪共享现象（False Sharing，这也算是典型缓存提升开发复杂度的例子）发生，等等。其中第一点尽可能避免竞争是最关键的，无论如何实现同步都不会比直接无需同步更快，笔者下面以Caffeine为例，介绍一些缓存如何避免竞争、提高吞吐量的方案。

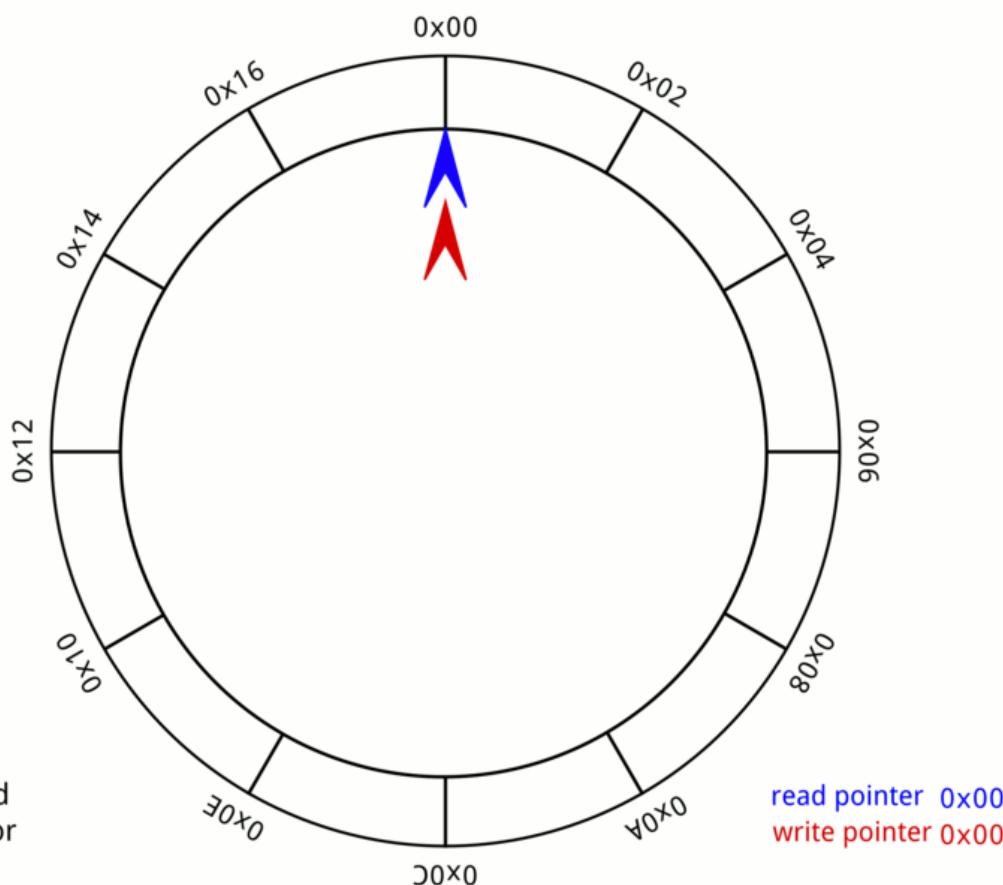
缓存中最主要的数据竞争源于读取数据的同时，也会伴随着对数据状态的写入操作，譬如读取时要更新数据的最近访问时间和访问计数器的状态（为了追求高效，可能不会记录时间和次数，如通过调整链表顺序来表达时间先后，通过Sketch结构来表达热度高低），以实现稍后要讲解的淘汰策略；又或者读取时要判断数据的超期时间等信息，以实现失效重加载等稍后要讲解的其他扩展功能。对以上伴随读取操作而来的状态维护，有两种可选择

的处理思路，一种是以Guava Cache为代表的同步处理机制，即在访问数据时一并完成缓存淘汰、统计、失效等状态变更操作，通过分段加锁等优化手段来尽量减少竞争。另一种是以Caffeine为代表的异步日志提交机制，这种机制参考了经典的数据库设计理论，将对数据的读、写过程看作是日志（日志即对数据的操作指令）的提交过程。尽管日志也涉及到写入操作，有并发的数据变更就必然面临锁竞争，但异步提交的日志已经将原本在Map内的锁转移到日志的追加写操作上。

在Caffeine的实现中，设有专门的[环形缓存区](#)（Ring Buffer，也常称作Circular Buffer）来记录由于数据读取而产生的状态变动日志。为进一步减少竞争，Caffeine给每条线程（对线程取Hash，哈希值相同的使用同一个缓冲区）都设置一个专用的环形缓冲。

额外知识：环形缓冲

所谓环形缓冲，并非Caffeine的专有概念，它是一种拥有读、写两个指针的数据复用结构，在计算机科学中有非常广泛的应用。举个具体例子，譬如一台计算机通过键盘输入，并通过CPU读取“HELLO WIKIPEDIA”这个长14字节的单词，通常需要一个至少14字节以上的缓冲区才行。但如果是环形缓冲结构，读取和写入就应当一起进行，在读取指针之前的位置均可以重复使用，理想情况下，只要读取指正不落后于写入指针一整圈，这个缓冲区就可以持续工作下去，能容纳无限多个新字符。否则，就必须阻塞写入操作去等待读取消空缓冲区。



环形缓存区工作原理 (图片来自[维基百科](#))

从Caffeine读取数据时，数据本身会在其内部的ConcurrentHashMap中直接返回，而数据的状态信息变更就存入环形缓冲中，由后台线程异步处理。如果异步处理的速度跟不上状态变更的速度，导致缓冲区满了，那此后接收的状态的变更信息就会直接被丢弃掉，直至缓冲区重新富余。通过环形缓冲和容忍有损失的状态变更，Caffeine大幅降低了由于数据读取而导致的垃圾收集和锁竞争，因此Caffeine的读取性能几乎与ConcurrentHashMap的读取性能相同。

向Caffeine写入数据时，将使用传统的有界队列（ArrayQueue）来存放状态变更信息，写入带来的状态变更是无损的，不允许丢失任何状态，这是考虑到许多状态的默认值必须通过写入操作来完成初始化，因此写入会有一定的性能损失。根据Caffeine官方给出的数据，相比ConcurrentHashMap，Caffeine在写入时大约会慢10%左右。

命中率与淘汰策略

有限的物理存储决定了任何缓存的容量都不可能是无限的，所以缓存需要在消耗空间与节约时间之间取得平衡，这要求缓存必须能够自动或者由人工淘汰掉缓存中的低价值数据，

由人工管理的缓存淘汰主要取决于开发者如何编码，不能一概而论，这里我们只讨论由缓存自动进行淘汰的情况。我们所说的“缓存如何自动地实现淘汰低价值目标”，现在被称为缓存的 淘汰策略（也常叫做替换策略或者清理策略）。

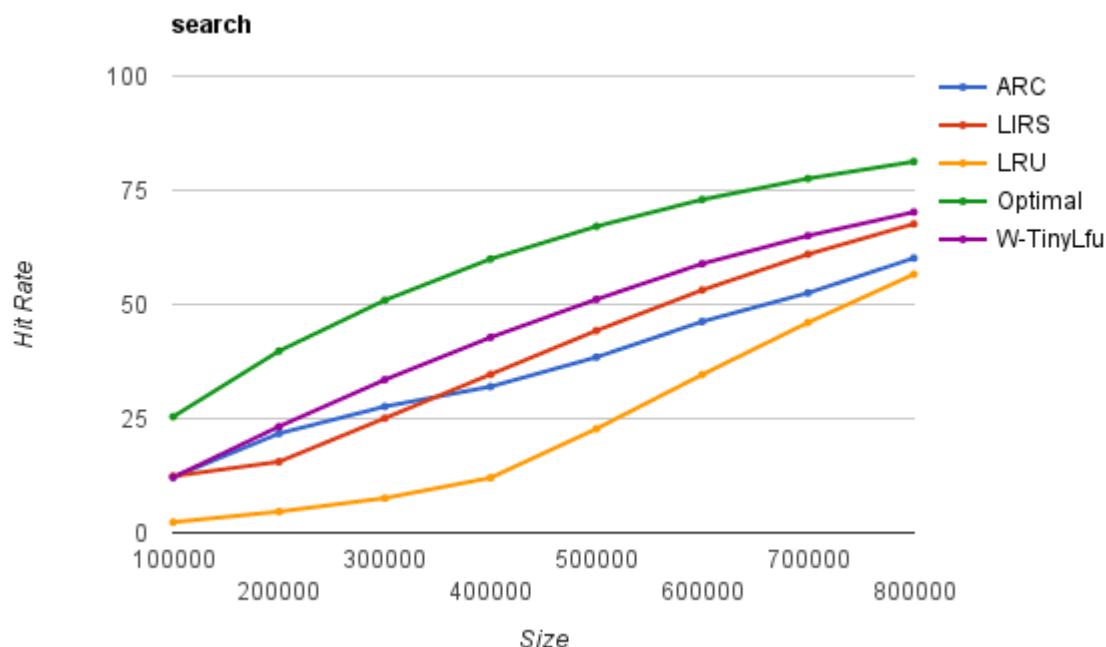
缓存实现自动淘汰低价值数据的容器之前，首先要定义怎样的数据才算是“低价值”？由于缓存的通用性，这个问题的答案一定是与具体业务逻辑是无关的，只能从缓存工作过程收集到的统计结果来确定数据是否有价值，通用的统计结果包括但不限于数据何时进入缓存？被使用过多少次？最近什么时候被使用？等等。由此，一旦确定选择何种统计数据，及如何通用地、自动地判定缓存中每个数据价值高低，也相当于决定了缓存的淘汰策略是如何实现的。最基础的有以下三种淘汰策略的实现方案：

- **FIFO** (First In First Out)：优先淘汰最早进入被缓存的数据。FIFO实现很简单，但一般来说它并不是优秀的淘汰策略，越是频繁被用到的数据，往往会被存入缓存之中。如果采用这种淘汰策略，很可能会大幅降低缓存的命中率。
- **LRU** (Least Recent Used)：优先淘汰最久未被使用访问过的数据。LRU通常会采用HashMap加LinkedList双重结构（如LinkedHashMap）来实现，以HashMap来提供访问接口，保证常量时间复杂度的读取性能，以LinkedList的链表元素顺序来表示数据的时间顺序，每次缓存命中时把返回对象调整到LinkedList开头，每次缓存淘汰时从链表末端开始清理数据。对大多数的缓存场景来说，LRU都明显要比FIFO策略合理，尤其适合用来处理短时间内频繁访问的热点对象。但相反，它的问题是如果一些热点数据在系统中经常被频繁访问，但最近几分钟因为某种原因未被访问过，此时这些热点数据依然要面临淘汰的命运，LRU依然可能错误淘汰价值更高的数据。
- **LFU** (Least Frequently Used)：优先淘汰最不经常使用的数据。LFU会给每个数据添加一个访问计数器，每访问一次就加1，需要淘汰时就清理计数器数值最小的那批数据。LFU可以解决上面LRU中热点数据间隔一段时间不访问就被淘汰的问题，但同时它又引入了两个新的问题，一个是需要对每个缓存的数据专门去维护一个计数器，每次访问都要更新，在上一节“吞吐量”里解释了这会带来高昂的开销；另一个问题是不便于处理随时间变化的热度变化，譬如某个曾经频繁访问的数据现在不需要了，它也很难自动被清理出缓存。

缓存淘汰策略直接影响缓存的命中率，没有一种策略是完美的、能够满足全部系统所需的。不过，随着淘汰算法的发展，近年来的确出现了许多相对性能要更好的，也更为复杂的新算法。以LFU分支为例，针对它存在的两个问题，近年来提出的TinyLFU和W-TinyLFU算法就往往会有更好的效果。

- **TinyLFU** (Tiny Least Frequently Used) : TinyLFU是LFU的改进版本。为了缓解LFU每次访问都要修改计数器所带来的性能负担，TinyLFU会首先采用Sketch对访问数据进行分析，所谓Sketch是指用少量的样本数据来估计全体数据的特征，这种做法显然牺牲了准确性，但是只要样本数据与全体数据具有相同的概率分布，Sketch得出的结论仍不失为一种高效与准确之间权衡的有效结论。借助[Count–Min Sketch](#)算法（可视为布隆过滤器的一种等价变种结构），TinyLFU可以用相对小得多的记录频率和空间来近似地找出缓存中的低价值数据。为了解决LFU不便于处理随时间变化的热度变化问题，TinyLFU采用了基于“滑动时间窗”（在“流量控制”中我们会更详细地介绍这种算法）的热度衰减算法，简单理解就是每隔一段时间，便会把计数器的数值减半，以此解决“旧热点”数据难以清除的问题。
- **W-TinyLFU** (Windows-TinyLFU) : W-TinyLFU又是TinyLFU的改进版本。TinyLFU在实现减少计数器维护频率的同时，也带来了无法很好地应对稀疏突发访问的问题，所谓稀疏突发访问是指有一些绝对频率较小，但突发访问频率很高的数据，譬如某些运维性质的任务，也许一天、一周只会在特定时间运行一次，其余时间都不会用到，此时TinyLFU就很难让这类元素通过Sketch的过滤，因为它们无法在运行期间积累到足够高的频率。应对短时间的突发访问是LRU的强项，W-TinyLFU就结合了LRU和LFU两者的优点，从整体上看是它是LFU策略，从局部实现上看又是LRU策略。具体做法是将新记录暂时放入一个名为Window Cache的前端LRU缓存里面，让这些对象可以在Window Cache中累积热度，如果能通过TinyLFU的过滤器，再进入名为Main Cache的主缓存中存储，主缓存根据数据的访问频繁程度分为不同的段（LFU策略，实际上W-TinyLFU只分了两段），但单独某一段局部来看又是基于LRU策略去实现的（称为Segmented LRU）。每当前一段缓存满了之后，会将低价值数据淘汰到后一段中去存储，直至最后一段也满了之后，该数据就彻底清理出缓存。

仅靠以上简单的、有限的介绍，你不一定能完全理解TinyLFU和W-TinyLFU的工作原理，但肯定能看出这些改进算法比起原来基础版本的LFU要复杂了许多。有时候为了取得理想的效果，采用较为复杂的淘汰策略是不得已的选择，Caffeine官方给出的W-TinyLFU以及另外两种高级淘汰策略[ARC](#) (Adaptive Replacement Cache) 、[LIRS](#) (Low Inter-Reference Recency Set) 与LFU之间的对比，如下图所示：



几种淘汰算法在搜索场景下的命中率对比（图片来自[Caffeine](#)）

在搜索场景中，三种高级策略的命中率较为接近于理想曲线（Optimal），而LRU则差距最远，在数据库、网站、分析类等应用场景中，这几种策略之间的绝对差距不尽相同，但相对排名基本上没有改变，最基础的淘汰策略的命中率是最低的。对其他缓存淘汰策略感兴趣的读者可以参考维基百科中对[Cache Replacement Policies](#)的介绍。

扩展功能

一般来说，一套标准的Map接口（或者来自[JSR 107](#)的javax.cache.Cache接口）就可以满足缓存访问的基本需要，不过在“访问”之外，专业的缓存往往还会提供很多额外的功能。笔者简要列举如下：

- 加载器：许多缓存都有“CacheLoader”之类的设计，加载器可以让缓存从只能被动存储外部放入的数据，变为能够主动通过加载器去加载指定Key值的数据，加载器也是实现自动刷新功能的基础前提。
- 淘汰策略：有的缓存淘汰策略是固定的，也有一些缓存能够支持用户自己根据需要选择不同的淘汰策略。
- 失效策略：要求缓存的数据在一定时间后自动失效（移除出缓存）或者自动刷新（使用加载器重新加载）。
- 事件通知：缓存可能会提供一些事件监听器，让你在数据状态变动（如失效、刷新、移除）时进行一些额外操作。有的缓存还提供了对缓存数据本身的监视能力（Watch功能）。

- 并发级别**：对于通过分段加锁来实现的缓存（以Guava Cache为代表），往往会提供并发级别的设置。可以简单将其理解为缓存内部是使用多个Map（分段）来存储数据的，并发级别就用于计算出使用Map的数量。如果将这个参数设置过大，会引入更多的Map，需要额外维护这些Map而导致更大的时间和空间上的开销；如果设置过小，又会导致在访问时产生线程阻塞，因为多个线程更新同一个ConcurrentMap时会产生锁竞争。
- 容量控制**：缓存通常都支持指定初始容量和最大容量，初始容量目的是减少扩容频率，这与Map接口本身的初始容量含义是一致的。最大容量类似于控制Java堆的-Xmx参数，当缓存接近最大容量时，会自动清理掉低价值的数据。
- 引用方式**：支持将数据设置为软引用或者弱应用，提供引用方式的设置是为了将缓存与Java虚拟机的垃圾收集机制联系起来。
- 统计信息**：提供诸如缓存命中率、平均加载时间、自动回收计数等统计。
- 持久化**：支持将缓存的内容存储到数据库或者磁盘中，进程内缓存提供持久化功能的作用不是太大，但分布式缓存大多都会考虑提供持久化功能。

至此已简要介绍了缓存的三项属性：吞吐量、命中率和扩展功能，笔者将几款主流进程中缓存方案整理成下表，供读者参考。

	ConcurrentHashMap	Ehcache	Guava Cache	Caffeine
访问性能	最高	一般	良好	优秀 接近于ConcurrentHashMap
淘汰策略	无	支持多种淘汰策略 FIFO、LRU、LFU等	LRU	W-TinyLFU
扩展功能	只提供基础的访问接口	并发级别控制 失效策略 容量控制 事件通知 统计信息	大致同左	大致同左

分布式缓存

相比起缓存数据在进程内存中读写的速度，涉及网络I/O后，由网络传输、数据复制、序列化和反序列化等操作所导致的延迟要比内存访问高得多，所以对分布式缓存来说，处理与网络有相关的操作是对吞吐量影响更大的因素，往往也是比淘汰策略、扩展功能更重要的关注点，这决定了尽管也有Ehcache、Infinispan这类能同时支持分布式部署和进程内嵌部署的缓存方案，但通常进程内缓存和分布式缓存选型时会有完全不同的候选对象及考察点。我们决定使用哪种分布式缓存前，首先必须确认自己需求是什么？

- 从访问的角度来说，如果频繁更新但甚少读取的数据，正常来说是不会有人把它拿去做缓存的，这是自讨麻烦。对于甚少更新但频繁读取的数据，理论上更适合做复制式缓存；对于更新和读取都较为频繁的数据，理论上就更适合做集中式缓存。笔者简要介绍这两种分布式缓存形式的差别与代表：

- **复制式缓存**：复制式缓存可以看作是“能够支持分布式的进程内缓存”，它的工作原理与Session复制类似。缓存中所有数据在分布式集群的每个节点里面都存在有一份副本，读取数据时无需网络访问，直接从当前节点的进程内存中返回，理论上可以做到与进程内缓存一样高的读取性能；当数据发生变化时，就必须遵循复制协议，将变更同步到集群的每个节点中，复制性能随着节点的增加呈现平方级下降，变更数据的代价十分高昂。

复制式缓存的代表是JBossCache，这是JBoss针对企业级集群设计的缓存方案，支持JTA事务，依靠JGroup进行集群节点间数据同步。以JBossCache为典型的复制式缓存曾有一段短暂的兴盛期，但今天基本上已经很难再见到使用这种缓存形式的大型信息系统了，JBossCache被淘汰的主要原因是写入性能实在差到不堪入目的程度，如果对它没有足够了解的话，稍有不慎就要被埋进坑里。它在小规模集群中同步数据尚可接受，但在大规模集群下，很容易就因网络同步的速度跟不上写入速度，进而导致在内存中累计大量待重发对象，最终引发OutOfMemory崩溃。

为了缓解复制式同步的写入效率问题，JBossCache的继任者Infinispan提供了另一种分布式同步模式（该同步模式的名字就叫做“分布式”），允许用户配置数据需要复制的副本数量，譬如集群中又有八个节点，可以要求数据只复制四份副本，此时缓存的容量相当于是传统复制模式的一倍，如果访问的数据在本地缓存中没有副本，Infinispan完全有能力感知网络的拓扑结构，知道应该到哪些节点中寻找。

- **集中式缓存**：集中式缓存是目前分布式缓存的主流形式，集中式缓存的读、写都需要网络访问，其好处是不会随着集群节点数量的增加而产生额外的负担，其坏处自然是读、写都不再可能达到进程内缓存那样的性能。

集中式缓存还有一个必须提及的特点是与使用缓存的应用分处在独立的进程空间中，

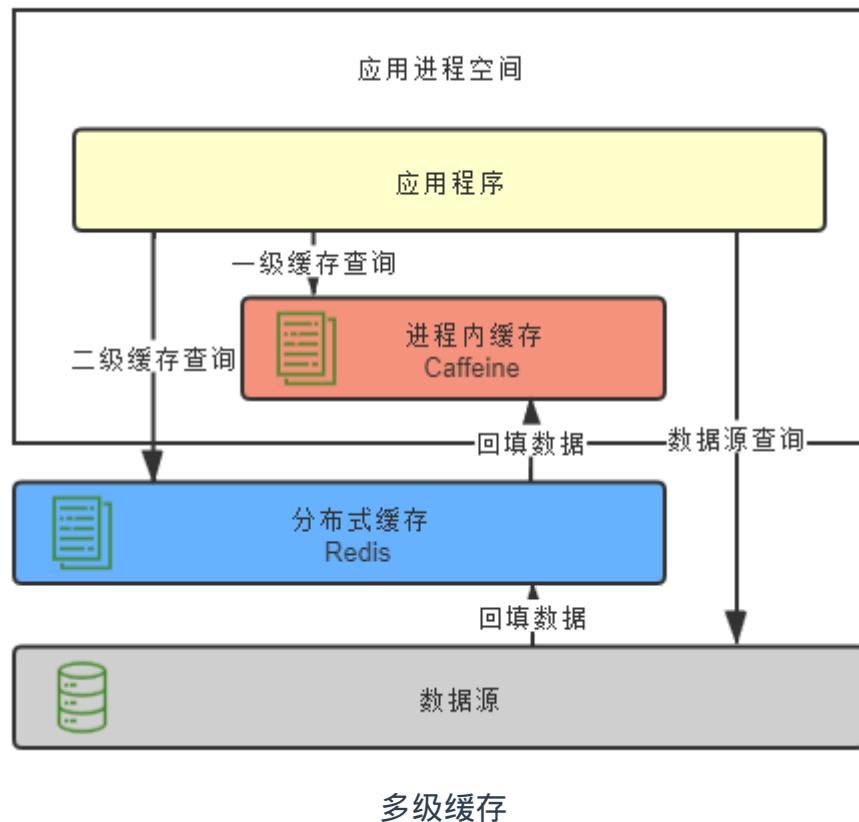
其好处是它能够为异构语言提供服务，譬如用C语言编写的Memcached完全可以毫无障碍地为Java语言编写的应用提供缓存服务；其坏处是如果要缓存对象等复杂类型的话，基本上就只能靠序列化来支撑具体语言的类型系统（支持Hash类型的缓存，可以部分模拟对象类型），不仅有序列化的成本，还很容易导致传输成本也显著增加。举个例子，假设某个有100个字段的大对象变更了其中一个字段的值，通常缓存也不得不把整个对象所有内容重新序列化传输出去才能实现更新，一般集中式缓存更提倡直接缓存原始数据类型。相比之下，JBossCache通过它的字节码自审（Introspection）功能和树状存储结构（TreeCache），做到了自动跟踪、处理对象的部分变动，修改了对象中哪些字段的数据，就只会同步对象中真正变更那部分。

如今Redis广为流行，基本上已经打败了Memcached及其他集中式缓存框架，成为集中式缓存的首选，甚至可以说成为了分布式缓存的实质上的首选，几乎到了不必管读取、写入哪种操作更频繁，都可以无脑上Redis的程度。也因此，之前说到哪些数据适合用复制式缓存、哪些数据适合集中式缓存时，笔者都在开头加了个拗口的“理论上”。尽管Redis最初设计的本意是NoSQL数据库而不是专门用来做缓存的，可今天它确实已经成为许多分布式系统中无可或缺的基础设施，广泛用作缓存的实现方案。

- 从数据一致性角度说，缓存本身也有集群部署的需求，理论上你应该认真考虑一下是否能接受不同节点取到的缓存数据有可能存在差异。譬如刚刚放入缓存中的数据，另外一个节点马上访问发现未能读到；刚刚更新缓存中的数据，另外一个节点访问在短时间类读取到的仍是旧的数据，等等。根据分布式缓存集群是否能保证数据一致性，可以将它分为AP和CP两种类型（在[“分布式事务”](#)中已介绍过CAP各自的含义）。此处又一次出现了“理论上”，是因为我们实际开发中通常不太会把追求强一致性的数据使用缓存来处理（可以但没必要，可类比MESI等缓存一致性协议）。譬如，Redis集群就是典型的AP式，有着高性能高可用等特点，却并不保证强一致性。而能够保证强一致性的ZooKeeper、Doozerd、Etcd等分布式协调框架，通常不会有人将它们当为“缓存”来使用，这些分布式协调框架的吞吐量相对Redis来说是非常有限的，不过它们倒是常与Redis和其他分布式缓存搭配工作，用来实现其中的通知、协调、队列、分布式锁等功能。

分布式缓存与进程内缓存各有所长，也有各有局限，它们是互补而非竞争的关系，如有需要，完全可以同时把进程内缓存和分布式缓存互相搭配，构成透明多级缓存（Transparent Multilevel Cache，TMC）。先不考虑“透明”的话，多级缓存是很好理解的，使用进程内缓存做一级缓存，分布式缓存做二级缓存，如果能在一级缓存中查询到结果就直接返回，否则便到二级缓存中去查询，将二级缓存中的结果回填到一级缓存，以后再访问该数据就没

有网络请求了。如果二级缓存也查询不到，就发起对最终数据源的查询，将结果回填到一、二级缓存中去。



尽管多级缓存结合了进程内缓存和分布式缓存的优点，但它的代码侵入性较大，需要由开发者承担多次查询、多次回填的工作，也不便于管理，如超时、刷新等策略都要设置多遍，数据更新更是麻烦，很容易会出现各个节点的一级缓存、以及二级缓存里数据互相不一致的问题。必须“透明”地解决以上问题，多级缓存才具有实用的价值。一种常见的方案是变更以分布式缓存中的数据为准，访问以进程内缓存的数据为先。大致做法是当数据发生变动时，在集群内发送推送通知（简单点的话可采用Redis的PUB/SUB，求严谨的话引入ZooKeeper或Etcd来处理），让各个节点的一级缓存自动失效掉相应数据。当访问缓存时，提供统一封装好的一、二级缓存联合查询接口，接口外部是只查询一次，接口内部自动实现优先查询一级缓存，未获取到数据再自动查询二级缓存的逻辑。

缓存风险

本文开篇就提到，缓存不是多多益善，它属于有利有弊，要真正到必要时候才考虑的解决方案。这节就介绍使用缓存的注意事项、常见的风险，以及应对的办法。

缓存穿透

缓存的目的是为了缓解CPU或者I/O的压力，譬如对数据库做缓存，大部分流量都从缓存中直接返回，只有缓存未能命中的数据请求才会流到数据库中，数据库压力自然就减小了。但是如果查询的数据在数据库中根本不存在的话，缓存里自然也不会有，这类请求的流量每次都不会命中，每次都会触及到末端的数据库，缓存就起不到缓解压力的作用了，这种查询不存在数据的现象被称为 **缓存穿透**。

缓存穿透有可能是业务逻辑本身就存在的固有问题，也有可能是被恶意攻击的所导致，为了解决缓存穿透，通常会采取下面两种办法：

1. 对于业务逻辑本身就不避免的缓存穿透，可以约定在一定时间内对返回为空的Key值依然进行缓存（注意是正常返回但是结果为空，不要把抛异常的也当作空给缓存了），使得在一段时间内缓存最多被穿透一次。如果后续业务在数据库中对该Key值插入了新记录，那应当在插入之后主动清理掉缓存的Key值。如果业务时效允许的话，也可以将对缓存设置一个较短的超时时间来自动处理。
2. 对于恶意攻击导致的缓存穿透，可以在缓存之前设置一个布隆过滤器来解决。所谓恶意攻击是指请求者刻意构造数据库中肯定不存在的Key值，然后发送大量请求进行查询。布隆过滤器是用最小的代价来判断某个元素是否存在于某个集合的办法。如果布隆过滤器给出的判定结果是请求的数据不存在，那就直接返回即可，连缓存都不必去查。虽然维护布隆过滤器本身需要一定的成本，但比起攻击造成的资源损耗仍然是值得的。

缓存击穿

我们都知道缓存的基本工作原理是首次从真实数据源加载数据，完成加载后回填入缓存，以后其他相同的请求就从缓冲中获取数据，缓解数据源的压力。如果缓存中某些热点数据忽然因某种原因失效了（典型的由于超期而失效），此时又有多个针对该数据的请求同时发送过来，这些请求将全部未能命中缓存，都到达真实数据源中去，导致其压力剧增，这种现象被称为 **缓存击穿**。要避免缓存击穿问题，通常会采取下面的两种办法：

1. 加锁同步，以请求该数据的Key值为锁，使得只有第一个请求可以流入到真实的数据源中，其他线程采取阻塞或重试策略。如果是进程内缓存出现的问题加普通互斥锁即可，

如果是分布式缓存中出现的问题就加分布式锁，这样数据源就不会同时收到大量针对同一个数据的请求了。

2. 热点数据由代码来手动管理，缓存击穿是仅针对热点数据被自动失效才引发的问题，对于这类数据，可以直接由开发者通过代码来有计划地完成更新、失效，避免由缓存的策略自动管理。

缓存雪崩

缓存击穿是针对单个热点数据失效，由大量请求击穿缓存而给真实数据源带来压力。有另一种可能是更普遍的情况，不需要是针对单个热点数据的大量请求，而是由于大批不同的数据在短时间内一起失效，导致了这些数据的请求都击穿了缓存到达数据源，同样令数据源在短时间内压力剧增。

出现这种情况，往往是系统有专门的缓存预热功能，也可能大量公共数据是由某一次冷操作加载的，这样都可能出现由此载入缓存的大批数据具有相同的过期时间，在同一时刻一起失效。还有一种情况是缓存服务由于某些原因崩溃后重启，此时也会造成大量数据同时失效，这种现象被称为 **缓存雪崩**。要避免缓存雪崩问题，通常会采取下面的三种办法：

1. 提升缓存系统可用性，建设分布式缓存的集群。
2. 启用透明多级缓存，各个服务节点一级缓存中的数据通常会具有不一样的加载时间，也就分散了它们的过期时间。
3. 将缓存的生存期从固定时间改为一个时间段内的随机时间，譬如原本是一个小时过期，那可以缓存不同数据时，设置生存期为55分钟到65分钟之间的某个随机时间。

缓存污染

缓存污染 是指缓存中的数据与真实数据源中的数据不一致的现象。尽管笔者在前面是说过缓存通常不追求强一致性，但这显然不能等同于缓存和数据源间连最终的一致性都可以不要求了。

缓存污染多数是由开发者更新缓存不规范造成的，譬如你从缓存中获得了某个对象，更新了对象的属性，但最后因为某些原因，譬如后续业务发生异常回滚了，最终没有成功写入到数据库，此时缓存的数据是新的，数据库中的数据是旧的。为了尽可能的提高使用缓存时的一致性，已经总结不少更新缓存可以遵循设计模式，譬如Cache Aside、Read/Write T

hrough、Write Behind Caching等等。其中最简单，也是成本最低的Cache Aside模式是指：

- 读数据时，先读缓存，缓存没有的话，再读数据源，然后将数据放入缓存，再响应请求。
- 写数据时，先写数据源，然后失效（而不是更新）掉缓存。

读数据方面一般没什么出错的余地，但是写数据时，就有必要专门强调两点：一是先后顺序是先数据源后缓存。试想一下，如果采用先失效缓存后写数据源的顺序，那一定存在一段时间缓存已经删除完毕，但数据源还未修改完成，此时新的查询请求到来，缓存未能命中，就会直接流到真实数据源中。此时请求读到的数据依然是旧数据，随后又重新回填到缓存中。当数据源的修改完成后，结果就成了数据在数据源中是新的，在缓存中是老的，两者就会有不一致的情况。另一点是应当失效缓存，而不是去尝试更新缓存，这很容易理解，如果去更新缓存，更新过程中数据源又被其他请求再次修改的话，缓存又要面临处理多次赋值的复杂时序问题。所以直接失效缓存，等下次用到该数据时自动回填，期间无论数据源中的值被改了多少次都不会造成任何影响。

Cache Aside模式依然是不能保证在一致性上绝对不出问题的，否则就无需设计出Paxos这样复杂的共识算法了。典型的出错场景是如果某个数据是从未被缓存过的，请求会直接流到真实数据源中，如果数据源中的写操作发生在查询请求之后，结果回填到缓存之前，也会出现缓存中回填的内容与数据库的实际数据一致的情况。但这种情况的概率是很低的，Cache Aside模式仍然是以低成本更新缓存，并且获得相对可靠结果的解决方案。

安全架构

即使只限定在“软件架构设计”这个语境下，系统安全仍然是一个很大的话题。我们谈论的计算机系统安全，远不仅指“防御系统被黑客攻击”这样狭隘的“安全”。架构安全性至少应包括了（不限于）以下这些问题的具体解决方案：

- **认证** (Authentication) : 系统如何正确分辨出操作用户的真实身份？
- **授权** (Authorization) : 系统如何控制一个用户该看到哪些数据、能操作哪些功能？
- **凭证** (Credentials) : 系统如何保证它与用户之间的承诺是双方当时真实意图的体现，是准确、完整且不可抵赖的？
- **保密** (Confidentiality) : 系统如何保证敏感数据无法被包括系统管理员在内的内外部人员所窃取、滥用？
- **传输** (Transport Security) : 系统如何保证通过网络传输的信息无法被第三方窃听、篡改和冒充？
- **验证** (Verification) : 系统如何确保提交到每项服务中的数据是合乎规则的，不会对系统稳定性、数据一致性、正确性产生风险？

上面这些安全相关的问题，解决起来确实是既繁琐复杂，又难以或缺。值得庆幸的是这一部分内容基本上都是与具体系统、具体业务无关的通用性问题、这意味着它们会存在着业界通行的，已被验证过是行之有效的解决方案，乃至已经形成某一些行业标准，不需要我们自己从头去构思如何解决。后面我们将会通过标准的方案，逐一探讨以上问题的主流处理方法。

还有其他一些安全相关的内容，主要由管理、运维、审计方面负责，尽管软件架构也需要配合参与，但不列入本文的讨论范围之中，譬如：安全审计、系统备份与恢复、防治病毒、信息系统安全法规与制度、计算机防病毒制度、保护私有信息规则，等等。

认证

认证 (Authentication)

系统如何正确分辨出操作用户的真实身份？

认证 (Authentication)、授权 (Authorization) 和凭证 (Credentials) 可以说是一个系统中最基础的安全设计，再简陋的系统大概也很难省略掉“用户登录”功能，系统为用户提供服务前，总是希望先弄清楚“你是谁？”（认证），“能干什么？”（授权）以及“如何证明？”（凭证）这三个基本问题的答案。另一方面，认证、授权与凭证这三个基本问题，又并不如部分人所认为的那样，只是“系统登录”功能，校验一下用户名、密码是否正确这么简单。因为账户信息作为一种必须保障安全和隐私，又同时要兼顾各个系统模块中共享访问的基础主数据，它的存储、管理与使用都面临一系列复杂的问题。对于某些大规模的信息系统，账户管理往往要由专门的基础设施，如微软的活动目录 (Active Directory, AD) 或者轻量目录访问协议 (Lightweight Directory Access Protocol, LDAP) 甚至是基于区块链技术去完成。笔者准备使用三节篇幅，介绍互联网系统和企业级系统是如何实现认证、授权与凭证的，涉及到哪些行业规范和标准。

首先澄清一个可能很多人都会有的先入为主的观念：尽管“认证”是解决“你是谁？”的问题，但这里的“你”并不是一定是指个人（听着像是骂人的话），也有可能是外部代码，即第三方的类库或者服务。最初对代码认证的重视程度甚至高于对最终用户的认证，譬如最早的Java系统里，安全中的“认证”就是特指“代码级安全”（你是否信任要在你的电脑中运行的代码），这是由Java早期的主要应用形式——Java Applets所决定的：类加载器从远端下载一段Java代码（严谨地说是字节码），以Applets的形式在用户的浏览器中运行，由于Java的语言操作计算机资源的能力要远远强于JavaScript，因此当然要保证这些代码不会损害用户的计算机，否则谁都不敢去用。这一阶段的安全观念催生了现在仍然存在于Java技术体系中的“安全管理器” (java.lang.SecurityManager)、“代码权限许可” (java.lang.RuntimePermission) 等概念。如今，对外部类库和服务的认证需求依旧旺盛，但相比起稍后介绍的百花齐放式的最终用户认证来说，代码认证的研究方向已经很固定，基本上都统一到证

书签名上。本节我们讨论的范围只针对最终用户认证，代码认证会安排在“分布式的基石”中的“[服务安全](#)”中去讲解。

世纪之交，Java迎来了Web时代的辉煌，互联网迅速兴起促使Java进入了快速发展时期。这时候，基于HTML和JavaScript的超文本Web应用迅速盖过了“Java 2时代”之前的Java Applets应用，B/S系统对最终用户认证的需求使得“安全认证”的重点逐渐从“代码级安全”转向为“用户级安全”（你是否信任正在操作的用户）。在1999年，随J2EE 1.2（它是J2EE的第一个版本，初始版本号直接就是1.2）所发布的Servlet 2.2中增加了一系列用于认证的API，主要包括两部分内容：

- 在标准上，添加了四种内置的（不可扩展的）认证方案，即Client-Cert、Basic、Digest和Form。
- 在实现上，添加了与认证和授权相关的一套程序接口，譬如HttpServletRequest::isUserInRole()、HttpServletRequest::getUserPrincipal()等。

原本一项发布超过20年的老旧技术，应该没有多少专门提起的必要，笔者之所以在这里专门引用这件事，是想从它标准和实现的两个改进点中引出一个系统安全的经验原则：以标准规范为指导、以标准接口去实现。安全涉及的问题很复杂，解决方案却也相当的成熟，对于99%的系统来说，在安全上不去做轮子，不去想发明创造，严格遵循标准就是最恰当的安全。本文也顺着此思路展开，分别介绍业界中认证的标准规范的做法，以及在Java程序中落地实现的方法。

认证的标准

引用J2EE 1.2对安全的改进还有另一个原因，它内置支持的Basic、Digest、Form和Client-Cert四种认证方案很有代表性，刚好分别覆盖了通讯信道、协议和内容层面的认证，这三种层面认证的含义和应用场景笔者列举如下：

- 通讯信道上的认证：你和我建立通讯连接之前，要先证明你是谁。在网络传输（Network）场景中的典型是基于SSL/TLS传输安全层的认证。
- 通讯协议上的认证：你请求获取我的资源之前，要先证明你是谁。在互联网（Internet）场景中的典型是基于HTTP协议的认证。
- 通讯内容上的认证：你使用我提供的服务之前，要先证明你是谁。在万维网（World Wide Web）场景中的典型是基于Web内容的认证。

关于第一点信道上的认证，由于内容较多，又与后续介绍微服务安全方面的话题关系密切，将会独立放到本章的“[传输](#)”里（而且J2EE中的Client-Cert其实并不是用于TLS的，以它引出TLS并不合适）。在本节中，我们将会了解基于通讯协议和内容的两种认证方式。

HTTP认证

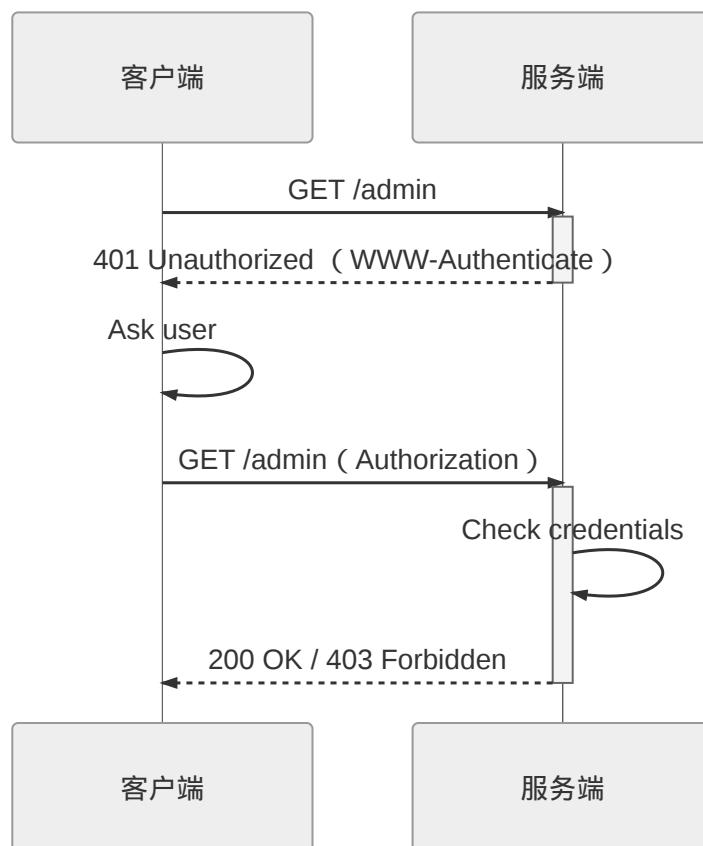
前面已经提前用到了一个名词 认证方案（Authentication Schemes），它就是指“生成能够证明用户身份的凭证的某种方法”，概念来源于HTTP协议的 认证框架（Authentication Framework）。互联网工程任务组（Internet Engineering Task Force，IETF）在[RFC 7235](#)中定义了HTTP协议的通用认证框架，要求所有支持HTTP协议的服务器，当未授权的用户意图访问服务端保护区域的资源时，应返回401 Unauthorized的状态码，同时应在响应报文头里附带以下两个Header项之一（分别代表网页认证和代理认证），告知客户端应该采取何种方式生成能够代表访问者身份的凭证信息：

```
WWW-Authenticate: <认证方案> realm=<保护区域的描述信息>
Proxy-Authenticate: <认证方案> realm=<保护区域的描述信息>
```

接收到该响应后，客户端必须遵循服务端指定的认证方案，在请求资源的报文头中加入身份凭证信息，服务端核实通过后才会允许该请求正常返回，否则将返回403 Forbidden。请求头报文应包含以下Header项之一：

```
Authorization: <认证方案> <凭证内容>
Proxy-Authorization: <认证方案> <凭证内容>
```

综合以上请求、响应的步骤的介绍，HTTP认证框架的工作流程如以下时序所示：



HTTP认证框架的设计意图是希望能把“身份认证”的目的与“具体如何认证”的实现分离开来，无论客户端通过生物信息（指纹、人脸）、用户密码、证书（U盾、数字证书）来生成凭证，均可归为是某种关于如何生成凭证的具体实现，均可包容在HTTP协议预设的框架之内。

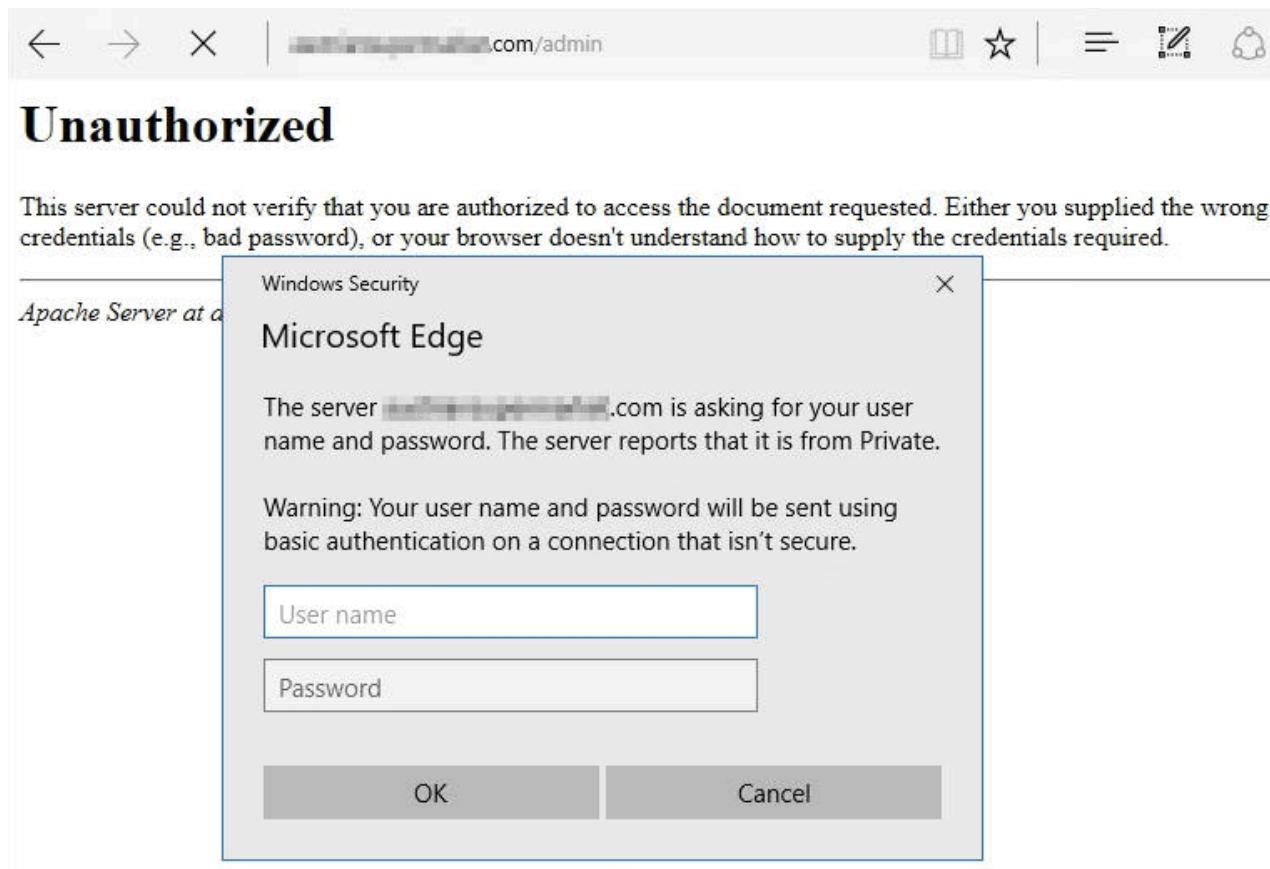
以上概念性的介绍可能还是有些抽象，笔者以最基础的认证方案 `HTTP Basic Authentication` 为例，这是一种主要以演示为目的认证方案（在一些不要求安全性的场合也有实际应用，譬如你家的路由器登录很可能就是这种认证方式）。Basic认证生成凭证的方法是让用户输入用户名和密码，然后经过Base64编码“加密”后作为身份凭证。譬如请求资源"GET /admin"时，浏览器收到服务端的如下响应：

```

HTTP/1.1 401 Unauthorized
Date: Mon, 24 Feb 2020 16:50:53 GMT
WWW-Authenticate: Basic realm="example from icyfenix.cn"

```

此时，浏览器需要询问最终用户，要求提供用户名和密码，会弹出类似下图所示的HTTP Basic认证窗口：



HTTP Basic Authentication

当用户输入了密码信息，譬如输入用户名“icyfenix”，密码“123456”，浏览器会将字符串“icyfenix:123456”编码为“aWN5ZmVuaXg6MTIzNDU2”，然后发送回服务端，如下所示：

```
GET /admin HTTP/1.1
Authorization: Basic aWN5ZmVuaXg6MTIzNDU2
```

由于Base64只是一种编码方式，并非任何形式的加密，Basic认证的风险是显而易见的，因此说它是一种带演示性质的认证方案。除Basic认证外，IETF还定义了很多种可用于生产的认证方案，譬如：

- **Digest** : RFC 7616，HTTP摘要认证，可视为Basic认证的改良版本，针对Base64明文发送的风险，Digest认证把用户名和密码加盐（一个被称为Nonce的变化值）后再通过MD5/SHA等哈希算法取摘要发送出去。可是，无论客户端如何加密，在面临中间人攻击时依然存在不小的安全风险，“[保密](#)”一节中我们将具体讨论这方面的问题。
- **Bearer** : RFC 6750，基于OAuth 2规范来完成认证，OAuth2是一个同时涉及到认证与授权的协议，我们将在下一节“[授权](#)”中详细介绍OAuth 2。

- **HOBA** : RFC 7486 [↗](#) , HTTP Origin-Bound Authentication的缩写，一种基于数字签名的认证。

HTTP认证框架中认证方案是允许自行扩展的，不要求一定要在RFC中定义，只要客户端（User Agent，这里就不一定是浏览器了）能够识别该方案即可。很多厂商也加入了自己的认证方式，譬如：

- **AWS4-HMAC-SHA256**：相当简单粗暴的名字，就是亚马逊AWS基于HMAC-SHA256哈希算法的认证。
- **NTLM / Negotiate**：这是微软公司NT LAN Manager (NTLM) 用到的两种认证方式。
- **Windows Live ID**：这个顾名思义，无需解释。
- **Twitter Basic**：一个不存在的网站所改良的HTTP基础认证。
-

Web认证

HTTP认证框架支持可插拔（Pluggable）的认证方案，本希望能够支持多种应用场景，但目前信息系统中，尤其是系统对终端用户的认证场景中，直接采用HTTP认证框架的比例并其实很小。原因仔细想一下就能明白：HTTP是“超文本传输协议”，首要任务是把资源从服务端传输到客户端，至于资源具体是什么内容，完全是由客户端自行解析驱动的。以HTTP为基础的认证只能面向传输协议而不是具体传输内容来设计，你查看一张图片、下载一个文件、浏览一个HTML页面、或者访问一个复杂的信息系统，都依赖同一套认证框架来支持。如果我从服务器中下载文件，弹个对话框让我登录或许还是可接受的；如果我访问信息系统，身份认证是在系统提供的“登录”功能中完成的，是由提供服务具体内容的信息系统，而不是由HTTP服务器来进行认证，这才是目前信息系统主流的认证方式，通常被称为“表单认证”（Form Authentication）。

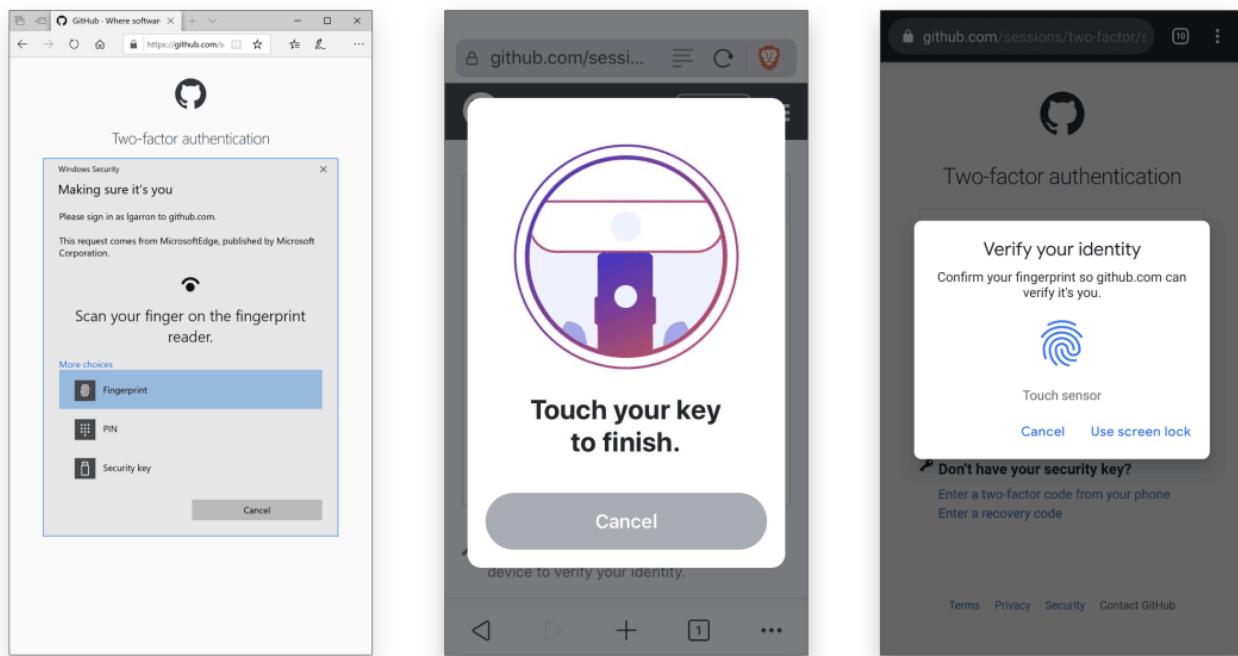
直至2019年以前，表单认证都没有什么行业标准可循，表单中的用户字段、密码字段、验证码字段、是否要在客户端加密、采用何种方式加密、接受表单的服务地址是什么等等，都直接由服务端与客户端的开发者自行协商决定。“没有标准的约束”反倒成了表单认证的一大优点，表单认证允许我们做出五花八门的页面，各种程序语言、框架或开发者本身都可以自行决定认证的全套交互细节。

可能你还记得开篇中说的“遵循规范、别造轮子就是最恰当的安全”，这里又将表单认证的高自由度说成是一大优点，初听来有矛盾。细想却并非如此，我们提倡用标准规范去解决

安全领域的共性的问题，并不应该与界面是否美观合理、操作流程是否灵活便捷这些应用需求对立起来。譬如，需要支持密码或扫码等多种登录方式、需要引入图形验证码来驱逐爬虫与机器人、需要在登录表单提交之前进行必要的表单校验等等，这些不可能定义在任何规范上，却很合理的应用需求应当被满足。同时，如何控制权限保证不产生越权操作、如何传输信息保证内容不被窃听篡改、如何加密敏感内容保证即使泄漏也不被逆推出明文等等，这些问题都有通行的解决方案，明确定义在规范中的要求也应当被遵循。到了具体实现层面，表单认证与HTTP认证完全可以结合使用，以Fenix's Bootstore的登录功能为例，页面表单是一个自行设计的Vue.js页面，但认证的交互过程遵循了OAuth 2规范的密码模式来完成。

2019年3月，万维网联盟（World Wide Web Consortium，W3C）批准了由[FIDO](#)（Fast Identity Online，一个安全、开放、防钓鱼、无密码认证标准的联盟）领导的第一份Web认证的标准“[WebAuthn](#)”，这里也许又有一些思维严谨的读者会感到矛盾与奇怪，刚才不是才说了Web表单长什么样、要不要验证码、登录表单是否在客户端校验等等是不可能定义在规范上的吗？的确如此，所以WebAuthn彻底抛弃了传统的密码登陆方式，改为直接采用生物识别（指纹、人脸、虹膜、声纹）或者实体密钥（以USB、蓝牙、NFC连接的物理密钥容器）来作为身份凭证，从根本上杜绝了输入错误（校验需求）和机器人模拟（验证码需求）等问题。

WebAuthn是相对比较复杂的认证协议，在阅读接下来的讲解之前，笔者建议如果你的设备和环境允许（硬件基本都没问题，用带有TouchBar的MacBook或者其他支持指纹、Face ID验证的手机即可。软件的话，直至iOS13.6，iPhone仍未支持，但Android和MacOS中的Chrome已经可以使用），先在[GitHub网站](#)的2FA认证上实际体验一下通过WebAuthn进行两段式登陆认证，然后再继续阅读后面的内容。

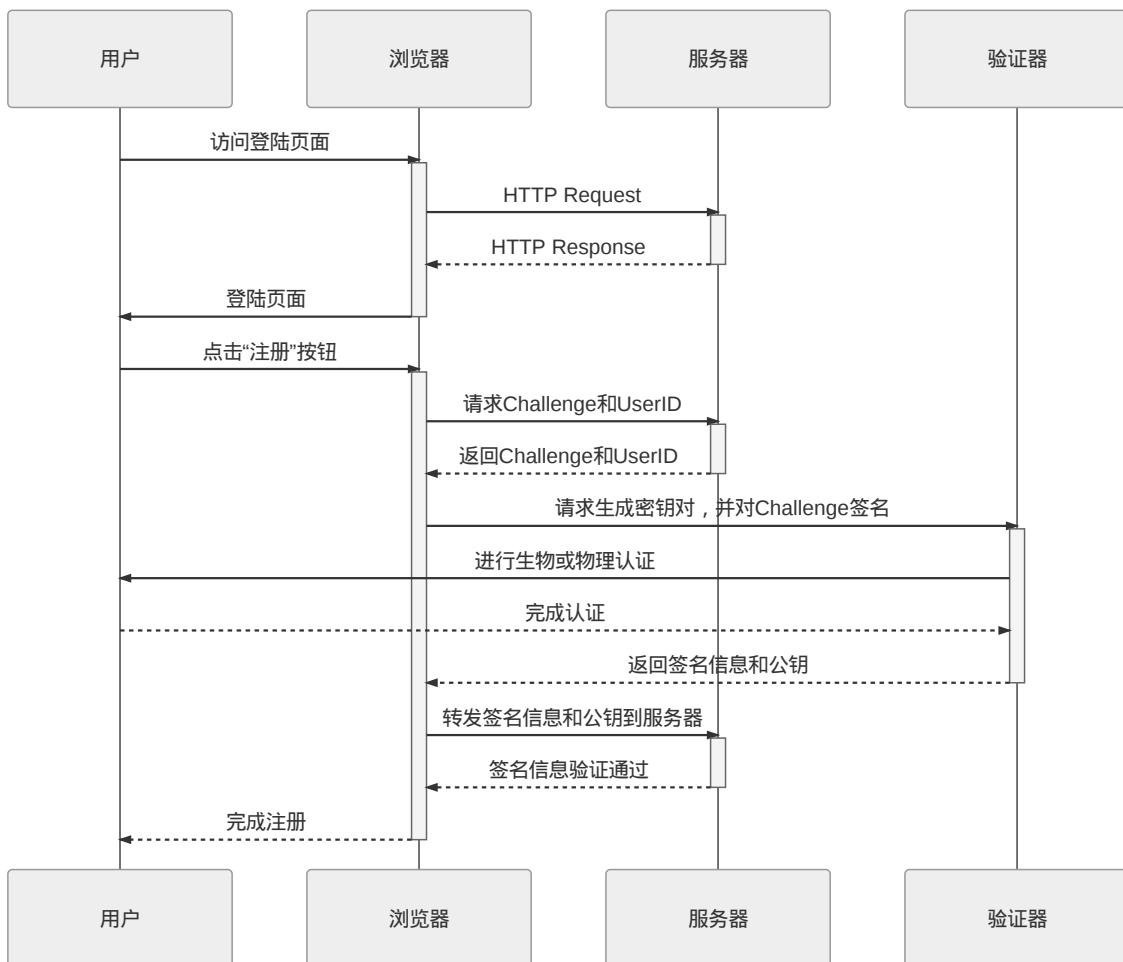


GitHub在不同浏览器上使用WebAuthn登陆

WebAuthn涵盖了“注册”与“认证”两个流程，先来说注册。注册大致可以分为以下步骤：

1. 用户进入系统的注册页面，这个页面的格式、内容和用户注册时需要填写的信息均不包含在WebAuthn标准的范围内。
2. 当用户填写完信息，点击“注册”按钮后，服务端暂存用户提交的数据，生成一个随机字符串（规范中称为Challenge）和用户的UserID（在规范中称作凭证ID）。
3. 浏览器的WebAuthn API接收到Challenge和UserID，把这些信息发送给验证器（Authenticator，可以理解为你机器上TouchBar、FaceID等认证设备的统一接口）。
4. 验证器提示用户进行验证，如果你机器支持多种认证设备，还会提示用户选择一个想要使用的设备。验证的结果是生成一个密钥对（公钥和私钥），验证器自己存储好私钥、用户信息以及域名。然后使用私钥对Challenge进行签名，并将签名结果、UserID和公钥一起返回给浏览器。
5. 浏览器将验证器返回的结果转发给服务器。
6. 服务器核验信息，检查UserID与之前发送的是否一致，并用公钥解密后得到的结果与之前发送的Challenge是否一致，一致即表明注册通过，服务端存储该UserID对应的公钥。

以上步骤的时序如下图所示：



登陆流程与注册流程很类似，如果你理解了注册流程，登陆就比较简单了。登陆大致可以分为以下步骤：

1. 用户访问登陆页面，填入用户名后即可点击登陆按钮。
2. 服务器返回随机字符串Challenge、用户UserID。
3. 浏览器将Challenge和UserID转发给验证器。
4. 验证器提示用户进行认证操作。由于在注册阶段验证器已经存储了该域名的私钥和用户信息，所以如果域名和用户都相同的话，就不需要生成密钥对了，直接以存储的私钥加密Challenge，然后返回给浏览器。
5. 服务端接收到浏览器转发来的被私钥加密的Challenge，以此前注册时存储的公钥进行解密，如果解密成功则宣告登录成功。

因为登陆流程与注册流程的步骤是基本一致的，笔者就不单独画登录的时序图了。WebAuthn采用非对称加密的公钥、私钥替代传统的密码，这是非常理想的认证方案，私钥是保密的，只有验证器需要知道它，用户本身都不需要得知，也就没有人为泄漏的可能；公钥是公开的，可以被任何人看到或存储。公钥可用于验证私钥生成的签名，但不能用来签

名，因此除了得知私钥外，没有其途径能够生成可被公钥验证为有效的签名，这样服务器就可以通过公钥是否能够解密来判断最终用户的身份是否合法。

WebAuthen还一揽子地解决了传统密码在网络传输上的风险，在“[保密](#)”一节中我们会讲到无论密码是否客户端进行加密、如何加密，对防御中间人攻击来说都是没有意义的。更值得夸赞的是WebAuthen为登录过程带来极大的便捷性，不仅注册和验证的用户体验十分优秀，而且彻底避免了用户在一个网站上泄漏密码，所有使用相同密码的网站都收到攻击的问题，这个优点使得用户无需再为每个网站想不同的密码。当前的WebAuthen还很年轻，普及率暂时还很有限，但笔者相信几年之内它必定会发展成Web认证的主流方式，被大多数网站和系统所支持。

认证的实现

了解过业界标准的认证规范以后，这节我们简要地介绍一下在Java技术体系内、在Fenix's Bookstore中具体是如何去实现安全认证的。Java其实也有自己的认证规范，第一个系统性的Java认证规范发布于Java 1.3时代，Sun公司提出了同时面向与代码级安全和用户级安全的认证授权服务[JAAS](#)（ Java Authentication and Authorization Service，1.3处于扩展包中，1.4纳入标准包），尽管JAAS已经开始照顾了最终用户的认证，但相对而言规范中代码级安全仍然是占更重要的地位。可能今天用过甚至是听过JAAS的Java程序员都已经不多了，但是这个规范提出了很多在今天仍然活跃于主流Java安全框架中的概念，譬如用户叫做“Subject / Principal”、密码存在“Credentials”之中、登录后从安全上下文“Context”中获取状态等常见设计都可以追溯到这一时期所定下的API：

- LoginModule (javax.security.auth.spi.LoginModule)
- LoginContext (javax.security.auth.login.LoginContext)
- Subject (javax.security.auth.Subject)
- Principal (java.security.Principal)
- Credentials (javax.security.auth.Destroyable、 javax.security.auth.Refreshable)

JAAS开创了这些沿用至今的安全概念，但规范本身实质上并没有得到广泛的应用，笔者认为有两大原因，一方面是由于JAAS同时面向代码级和用户级的安全机制，使得它过度复杂化，难以推广。在这个问题上Java社区一直有做持续的增强和补救，譬如Java EE 6中的JASPI、Java EE 8中的EE Security：

- JSR 115 : Java Authorization Contract for Containers (JACC)
- JSR 196 : Java Authentication Service Provider Interface for Containers (JASPI)
- JSR 375 : Java EE Security API (EE Security)

而另一方面，可能是更重要的一个原因是在21世纪的第一个十年里，以EJB为代表的容器化J2EE与以“Without EJB”为口号、以Spring、Hibernate等为代表的轻量化企业级开发框架发生了激烈的竞争，结果是后者获得了全面胜利。这个结果使得依赖于容器安全的JAAS无法得到大多数人的认可。在今时今日，实际活跃于Java安全领域的是两个私有的（私有的意思是不由JSR所规范的，即没有java/javax.*作为包名的）的安全框架：[Apache Shiro](#) 和[Spring Security](#)。

相较而言，Shiro使用更为便捷易用，而Spring Security的功能则要复杂强大一些。Fenix's Bookstore（无论是单体架构还是微服务架构）选择了Spring Security作为安全框架，这个选择与功能、性能之类的考虑都没什么关系，就只是因为Spring Boot/Cloud全家桶的原因。从目标上看，以上两个安全框架都解决的问题都很类似，大致包括以下四类：

- 认证功能：以HTTP协议中定义的各种认证、表单等认证方式确认用户身份，这是本节的主要话题。
- 安全上下文：用户获得认证之后，要开放一些接口，让应用可以得知该用户的基本资料、用户拥有的权限、角色等。
- 授权功能：授权在代码实现的角度来看主要就是访问控制（Access Control），但授权从标准的角度看仍然许多值得讨论的话题，这部分内容会放到“[授权](#)”去介绍。
- 密码的存储与验证：密码是烫手的山芋，无论存储、传输还是验证都很麻烦，我们会放到“[保密](#)”去具体讨论。

授权

授权 (Authorization)

系统如何控制一个用户该看到哪些数据、能操作哪些功能？

“授权”这个行为通常伴随着“认证”、“审计”、“账号”共同出现，并称为AAAA (Authentication、Authorization、Audit、Account，一些领域也有把Account解释为计费的意思)。授权行为在程序中其实非常普遍，我们给一个类、一个方法设置范围控制符 (public、protected、private、<Package>)，这其实也是一种授权 (访问控制) 行为。授权涉及到了两个相对独立的问题：

- 确保授权的过程可靠：对于单一系统来说，授权的过程是比较容易做到可控的，以前很多语境上提到授权，实质上讲的都是访问控制，理论上两者是应该分开的。而在涉及多方的系统中，授权过程就是一个必须严肃对待的问题：如何既让第三方系统能够访问到所需的资源，又能保证其不泄露用户的敏感数据？现在常用的多方授权协议主要有OAuth2和SAML 2.0（注意这两个协议涵盖的功能并不是直接对等的）。
- 确保授权的结果可控：授权的结果往往是用于对程序功能或者资源的访问控制 (Access Control)，形成理论的权限控制模型有：[自主访问控制](#) (Discretionary Access Control , DAC)、[强制访问控制](#) (Mandatory Access Control , MAC)、[基于属性的权限验证](#) (Attribute-Based Access Control , ABAC) 还有最为常用，也相对通用的是[基于角色的权限模型](#) (Role-Based Access Control , RBAC)。

由于篇幅原因，在这个小节里我们只介绍（将要）使用到的，也是最常用到的RBAC和OAuth2。先来说较为简单的RBAC。

RBAC

所有的访问控制模型，实质上都是在解决同一个问题：“谁 (User) ”拥有什么“权限 (Authority) ”去操作哪些“资源 (Resource) ”

这个问题看起来并不难，最直观的解决方案就是在用户对象上，设定一些操作权限，在使用资源时，检查是否有对应的操作权限即可。是的，请不要因太过简单直接而产生疑惑，很多著名的安全框架，譬如Spring Security的访问控制本质上就是这么做的。不过，这种把操作权限直接关联在用户身上的简单设计，在复杂系统上确实会导致比较繁琐的操作。试想一下，如果某个系统涉及到成百上千的资源，又有成千上万的用户，要为每个用户分配合适的权限将带来务必庞大的操作量和极高的出错概率，这也即是RBAC所要解决的问题。

为了避免对每一个用户设定权限，RBAC将权限从用户身上剥离，改为绑定到“**角色（Role）**”上，一种我们常见的RBAC应用就是操作系统权限中的“用户组”，这就是一种角色。用户可以隶属与一个或者多个角色，某个角色中也会包含有多个用户，角色之间还可以有继承性（父、子角色的权限继承，RBAC-1）。这样，资源的操作就只需按照有限且相对固定的角色去分配操作权限，而不去面对随时会动态增加的用户去分配。当用户的职责发生变化时，在系统中就体现为改变他所隶属的角色，譬如将“普通用户角色”改变“管理员角色”，就可以迅速完成其权限的调整，降低了权限分配错误的风险。RBAC的主要元素之间的关系可以以下图来表示：



上图中出现了一个新的名词“**许可（Permission）**”。所谓的许可，就是抽象权限的具象化体现。权限在系统中的含义应该是“允许何种**操作**作用于哪些**数据**之上”，这个即为“许可”。举个具体的例子，譬如某个文章管理系统的UserStory中，与访问控制相关的Backlog可能会是这样描述的：

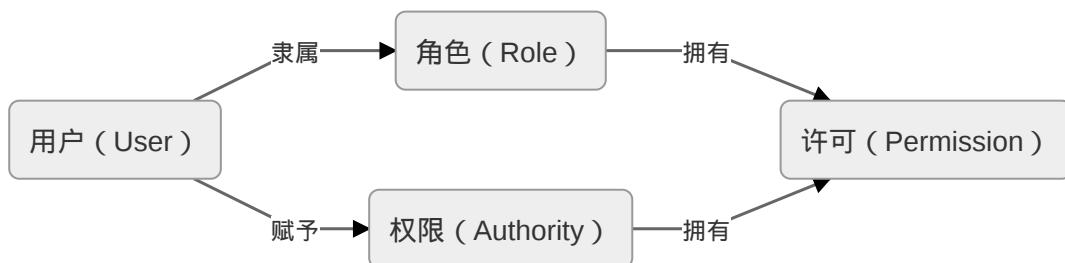
Backlog :

周同学（User）是某SCI杂志的审稿人（Role），职责之一是在系统中审核论文（Authority）。在审稿过程（Session）中，当他认为某篇论文（Resource）达到了可以公开发表标准时，就会在后台点击通过按钮（Operation）来完成审核。

以上，“给论文点击通过按钮”就是一种许可（Permission），它是“审核论文”这项权限（Authority）的具象化体现。

与微服务架构中的完全遵循RBAC进行访问控制的Kubernetes不同，我们在单体架构中使用的Spring Security参考了但并没有完全按照RBAC来进行设计。Spring Security的设计里

用户和角色都可以拥有权限，譬如在HttpSecurity对象上，就同时有着hasRole()和hasAuthority()方法，可能有不少刚接触的人会疑惑，混淆它们之间的关系。在Spring Security的访问控制模型可以认为是下图所示这样的：



站在代码实现的角度来看，Spring Security中Role和Authority的差异很小，它们共同存储在同一位置，唯一的差别仅是Role会在存储时自动带上“ROLE_”前缀（可以配置的）罢了。

但在使用者的角度来看，Role和Authority的差异可以很大，你可以执行决定你的系统中到底Permission只能对应到角色身上，还是可以让用户也拥有某些角色中没有的权限。这个观点，在Spring Security自己的文档上说的很清楚：这取决于你自己如何使用。

The core difference between these two is the semantics we attach to how we use the feature. For the framework, the difference is minimal – and it basically deals with these in exactly the same way.

使用RBAC，你可以控制最终用户在广义和精细级别上可以做什么。您可以指定用户是管理员，专家用户还是普通用户，并使角色和访问权限与组织中员工的身份职位保持一致。仅根据需要为员工完成工作的足够访问权限来分配权限。

OAuth2

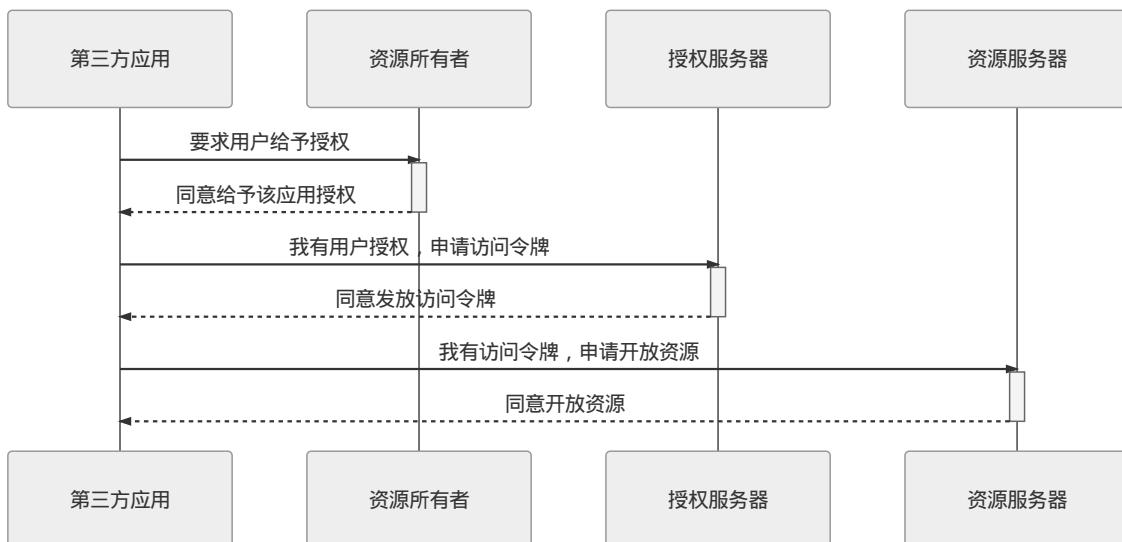
简要介绍过RBAC，下面我们再来看看相对要复杂繁琐一些的OAuth2认证授权协议（顺带一提，更繁琐的OAuth1已经完全被废弃了，勿念）。先明确一件事情，OAuth2是一个多方系统中的认证授权协议，如果你的系统并不涉及到第三方（譬如我们单体架构的Bookstore，即不为第三方提供服务，也不使用第三方的服务），引入OAuth2其实并无多大必要。我们之所以把OAuth2提前引入，主要是为了给微服务架构做铺垫。

OAuth2是在RFC 6749中定义授权协议，在RFC 6749正文的第一句就明确了OAuth2是解决第三方应用（Third-Party Application）的认证授权协议。前面也说到，如果只是单方系统，授权过程是比较容易解决的，至于多方系统授权过程会有什么问题，这里举个现实的例子来说明。

譬如你现在正在阅读的这个网站（<https://icyfenix.cn>），它的建设和更新大致流程是：笔者以Markdown形式写好了某篇文章，上传到由GitHub提供的代码仓库，接着由Travis-CI提供的持续集成服务会检测到该仓库发生了变化，触发一次Vuepress编译活动，生成目录和静态的HTML页面，然后推送给GitHub Pages，再触发腾讯云CDN的缓存刷新。这个过程要能顺利进行，就存在一些必须解决的授权问题，Travis-CI只有得到了我的明确授权，GitHub才能同意它读取我代码仓库中的内容，问题是它该如何获得我的授权呢？一种简单粗暴的方案是我把我的用户账号和密码都告诉Travis-CI，但这显然导致了以下这些问题：

- **密码泄漏**：如果Travis-CI被黑客攻破，将导致我GitHub的密码也同时被泄漏
- **访问范围**：Travis-CI将有能力读取、修改、删除、更新我放在GitHub上的所有代码仓库
- **授权回收**：我只有修改密码才能回收授予给Travis-CI的权力，可是我在GitHub的密码只有一个，授权的应用除了Travis-CI之外却还有许多，修改了意味着所有别的第三方的应用程序会全部失效

以上出现的这些问题，也就是OAuth2所要解决的问题，尤其是没有HTTPS支持传输安全的环境下依然可以解决这些问题。OAuth2提出的解决办法是通过一个令牌（Token）代替用户密码作为授权的凭证，有了令牌之后，哪怕令牌被泄漏，也不会导致密码的泄漏，令牌上可以设定访问资源的范围以及时效性，每个应用都持有独立的令牌，哪个失效都不会波及其他，一下子上面提出的三个问题都解决了，有了一层令牌之后，整个授权的流程如下图所示：



这个时序图里面涉及到了OAuth2中几个关键术语，我们通过前面那个具体的上下文语境来解释其含义，这对理解后续几种认证流程十分重要：

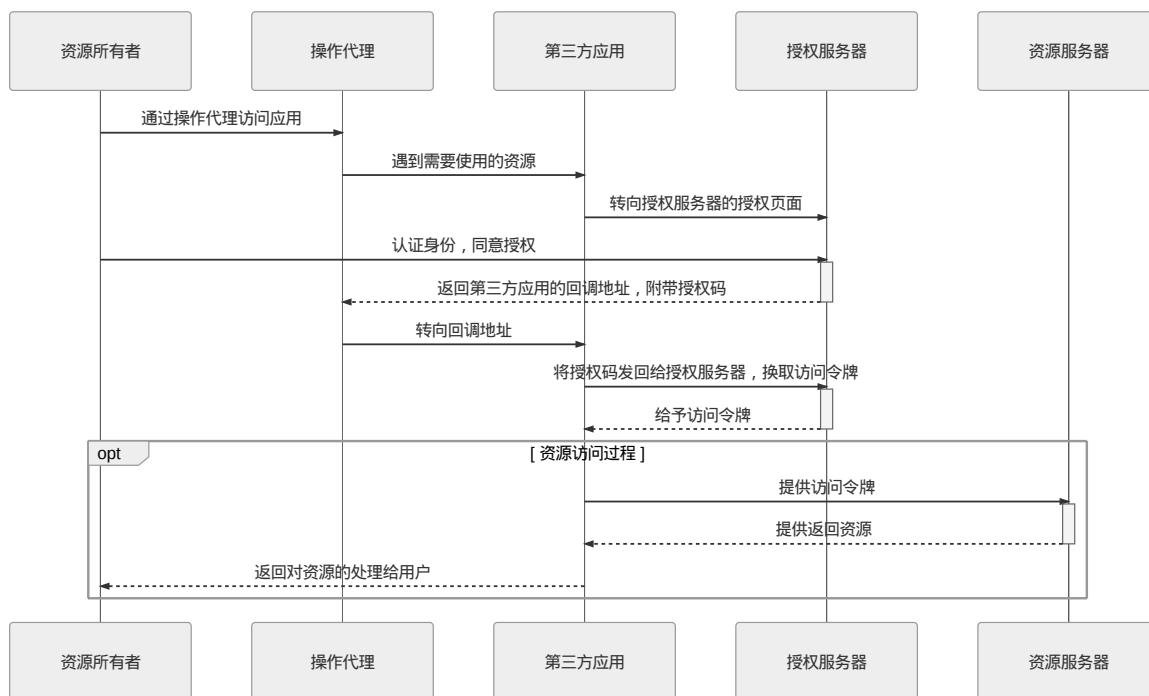
- **第三方应用** (Third-Party Application)：需要得到授权访问我资源的那个应用，即“Travis-CI”
- **授权服务器** (Authorization Server)：能够根据我的意愿提供授权（授权之前肯定已经进行了必要的认证过程，但这在技术上与授权可以没有直接关系）的服务，即“GitHub”
- **资源服务器** (Resource Server)：能够提供第三方应用所需资源的服务（它与认证服务可以是相同的服务器，也可以是不同的服务器），即“代码仓库”
- **资源所有者** (Resource Owner)：拥有授权权限的人，这里即是“我”
- **操作代理** (User Agent)：指用户用来访问服务器的工具，对于指代人类的“用户”来说这个通常就是浏览器，但在微服务中一个服务经常会作为另一个服务的“用户”，此时指的可能就是HttpClient、RPCClient或者其他访问途径。

看来“用令牌代替密码”确实是解决问题的好方法，但这最多只能算个思路，距离执行步骤还是不够具体的，时序图中的“要求/同意授权”、“要求/同意发放令牌”、“要求/同意开放资源”几个服务请求、响应应该如何设计，这就是执行步骤的关键了。对此，OAuth2一共提出了四种不同的授权方式（这就是我说OAuth2复杂繁琐的原因，摊手），分别为：

- 授权码模式 (Authorization Code)
- 简化模式 (Implicit)
- 密码模式 (Resource Owner Password Credentials)
- 客户端模式 (Client Credentials)

授权码模式

授权码模式是四种模式中最严谨（繁琐）的，它考虑到了几乎所有敏感信息泄漏的预防和后果。具体步骤的时序如下：



在开始完成整个授权过程以前，第三方应用先要到授权服务器上进行注册，所谓注册，是指向认证服务器提供一个域名地址，从授权服务器中获取ClientID和ClientSecret，然后便可以开始如下授权过程：

1. 第三方应用将资源所有者（用户）导向授权服务器的授权页面，并向授权服务器提供ClientID及同意授权后的回调URI，这是一次客户端页面转向。
2. 授权服务器根据ClientID确认第三方应用的身份，用户在授权服务器中决定是否同意向该身份的应用进行授权（认证的过程在此之前应该已经完成）。
3. 如果用户同意授权，授权服务器将转向地第三方应用在第1步调用中提供的回调地址URI，并附带上一个授权码和获取令牌的地址作为参数，这也是一次客户端页面转向。
4. 第三方应用通过回调地址收到授权码，然后将授权码与自己的ClientSecret一起作为参数，**通过服务端向授权服务器提供的获取令牌的服务地址发起请求，换取令牌**。该服务端应与注册时提供的域名一致。

5. 授权服务器核对授权码和ClientSecret，确认无误后，向第三方应用授予令牌。令牌可以是一个或者两个，其中必定要有的是访问令牌（Access Token），可选的是刷新令牌（Refresh Token）。访问令牌用于到资源服务器获取资源，有效期较短，刷新令牌用于在访问令牌失效后重新获取，有效期较长。
6. 资源服务器根据访问令牌所允许的权限，向第三方应用提供资源。

这个过程设计，已经考虑到了几乎所有合理的意外情况，举例几个容易想到的：

- 会不会有其他应用冒充第三方应用骗取授权？

ClientID代表一个第三方应用的“用户名”，这个是可以完全公开的。但ClientSecret应当只有应用自己才知道，这个代表了第三方应用的“密码”。在第5步发放令牌时，调用者必须能够提供ClientSecret才能成功完成。只要第三方应用妥善保管好ClientSecret，就没有人能够冒充它。

- 为什么要先发放授权码，再用授权码换令牌？

这是因为客户端转向（通常就是一次HTTP 302重定向）对于用户是可见的，换而言之，授权码完全可能会暴露给用户（以及用户机器上的其他程序），但由于用户并没有ClientSecret，光有授权码也是无法换取到令牌的，所以避免了令牌在传输转向过程中泄漏的风险。

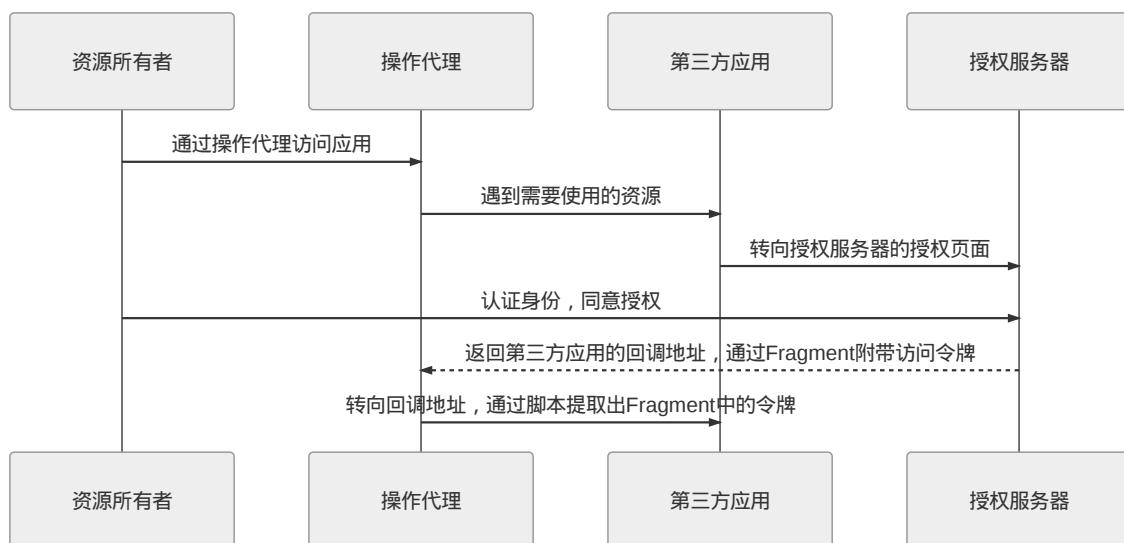
- 为什么要设计一个时限较长的刷新令牌和时限较短的访问令牌？不能直接把访问令牌的时间调长吗？

这是为了缓解OAuth2在**实际应用**中的一个主要缺陷，通常访问令牌一旦发放，除非超过了令牌中的有效期，否则很难（需要付出较大代价）有其他方式让它失效，所以访问令牌的时效性一般设计的比较短（譬如几个小时），如果还需要继续用，那就定期用刷新令牌去更新，授权服务器就可以在更新过程中决定是否还要继续给予授权。至于为什么说很难让它失效，我们将放到下一节“凭证”中解释这一点。

尽管授权码模式是严谨的，但是它并不够好用，这不仅仅体现在它那繁复的调用过程上，还体现在它对第三方应用提出了一个具体的要求：必须有服务端（因为第4步要发起服务端转向，而且服务端的地址必须与注册时提供的回调URI在同一个域内）。不要觉得要求一个系统要有服务端是天经地义理所当然的事情，你现在阅读文章的这个网站就没有任何服务端的支持，里面使用到了GitHub Issue作为每篇文章的留言板，它对GitHub来说照样是第三方应用，需要OAuth2授权来解决。除浏览器外，现在越来越普遍的是移动或桌面端的Client-Side Web Applications，譬如现在大量的基于Cordova、Electron、Node-Webkit.js的PWA应用。所以在此需求里，引出了OAuth2的第二种授权模式：隐式授权。

隐式授权

隐式授权省略掉了通过授权码换取令牌的步骤，整个授权过程都不需要服务端支持，一步到位。其代价是在隐式授权中，授权服务器不会再去验证第三方应用的身份（因为没有服务器了，ClientSecret没有人保管，就没有意义了。但其实还是会限制第三方应用的回调URI地址必须与注册时提供的域名一致，有可能被DNS污染之类的攻击所攻破，但仍算是尽人事努力一下）；也不能避免令牌暴露给资源所有者（以及用户机器上可能意图不轨的其他程序、HTTP的中间人攻击等）了。隐私授权的调用时序如下图（从此之后的授权模式，时序中我就不画资源访问部分的内容了，就是前面opt框中的那一部分，以便更聚焦重点）所示：



在以上过程设计中，与授权码模式模式的显著区别是授权服务器在得到用户授权后，直接返回了访问令牌，这显然降低了安全性，但OAuth2仍然努力尽可能地做到相对安全，譬如在前面提到的隐私授权中，尽管不需要用到服务端，但仍然需要在注册时提供回调域名，此时会要求该域名与接受令牌的域名处于同一个域内。此外，在隐私模式中明确禁止发放刷新令牌。

还有一点，在RFC 6749对隐式授权的描述中，特别强调了令牌是“通过Fragment带回”的。部分对超文本协议没有了解的读者，可能不知道Fragment是个什么东西？

额外知识：Fragment

In computer hypertext, a fragment identifier is a string of characters that refers to a resource that is subordinate to another, primary resource. The primary resource is identified by a Uniform Resource Identifier (URI), and the fragment identifier points to the subordinate resource.

——[URI Fragment](#) , Wikipedia

看了这段英文定义还是觉得概念不好的话，我简单告诉你，Fragment就是地址中“#”号后面的部分，譬如这个地址：

http://bookstore.icyfenix.cn/#/detail/1

后面的“/detail/1”便是Fragment，这个语法是在[RFC 3986](#)中定义的标准，规范中解释了这是用于客户端定位的URI从属资源，譬如HTML中就可以使用Fragment来做文档内的跳转（你现在可以点击一下这篇文章左边菜单中的几个子标题，看看浏览器地址的变化）而不会发起服务端请求。此外，如果浏览器对一个带有Fragment的地址发出Ajax请求，那Fragment是不会跟随请求被发送到服务端的，只能在客户端通过Script脚本来读取。所以隐式授权巧妙地利用这个特性，尽最大努力地避免了令牌从操作代理到第三方服务之间的链路存在被攻击的可能性，而被泄漏出去。而认证服务器到操作代理之间的这一段链路的安全，则可以通过TLS（即HTTPS）来保证没有中间没有受到攻击的，我们可以要求认证服务器都是基于HTTPS的，但无法要求第三方应用都是基于HTTPS。

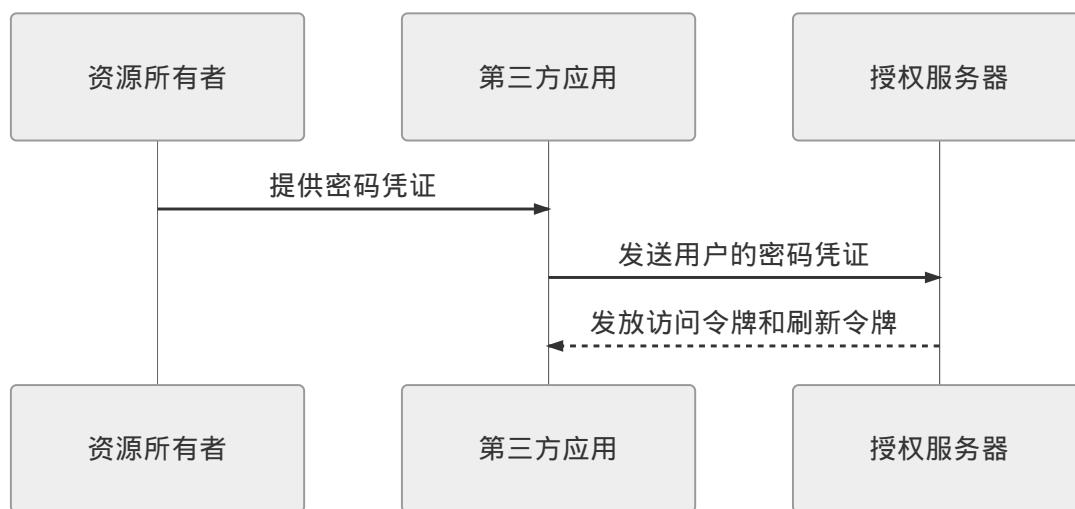
密码模式

前面所说的授权码模式和隐私模式属于纯粹的授权模式，它们与认证没有直接的联系，如何认证用户的真实身份这是与进行授权互相独立的过程。但在密码模式里，认证和授权就被整合成了同一个过程了。

密码模式原本的设计意图是仅限于用户对第三方应用是高度可信任的场景中使用，因为用户需要把密码明文提供给第三方应用，第三方以此向授权服务器获取令牌，一种可能合理的应用场景是操作系统作为第三方应用向授权服务器申请资源，用户把密码提供给操作系统有可能的。除此之外，这种高度可信的第三方本应该是较罕见的，但在分布式系统里，“第三方应用”完全可以看作是逻辑上与授权服务器同属一个系统，物理上独立于授权服务器部署的其他服务节点。譬如微服务集群中，负责用户身份认证的服务与授权服务很

可能都是由同一个服务商所提供的，名义上是第三方，其实是同一个系统，这两者之间自然可以存在可信任的关系。

单体版本、微服务版本的Fenix's Bookstore都直接采用了密码模式将认证和授权统一到一个过程当中。你并不会担心前端的Frontend工程或者后端的Account等工程在接收到用户名、密码后，会通过这些敏感信息，向授权服务器申请权限进行恶意的资源访问，因为这些代码虽然在OAuth2里被视为第三方应用的角色，但它们事实上是Fenix's Bookstore这个系统的一部分。理解了密码模式的用途，它的调用时序就很简单了，如下图所示：



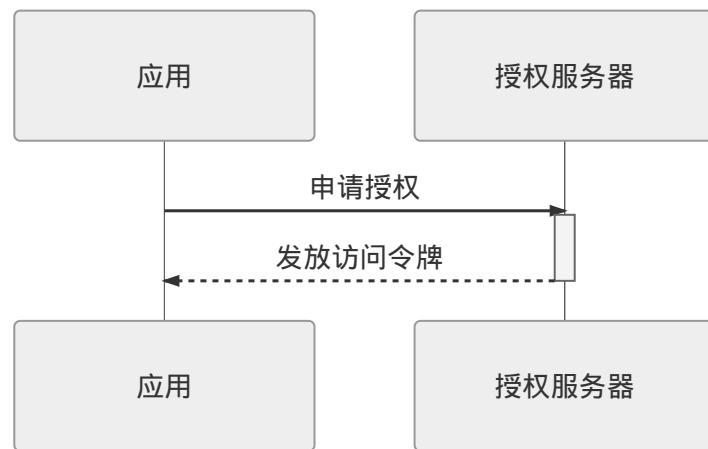
密码模式下“如何保障安全”的职责无法由OAuth2的过程设计来承担，应是由用户和第三方应用来自行保障，尽管OAuth2在规范中强调到“此模式下，第三方应用不得保存用户的密码”，但这并没有任何的约束力。

客户端模式

客户端模式是四种模式中最简单的，它只涉及到两个主体，第三方应用和授权服务器。如果严谨一点，现在说“第三方应用”其实已经不合适了，因为这里已经没有了“第二方”的存在，资源所有者、操作代理都是不存在的。甚至于叫“授权”都不太恰当，资源所有者都没有了，自然也不会有谁授予谁权限的过程。

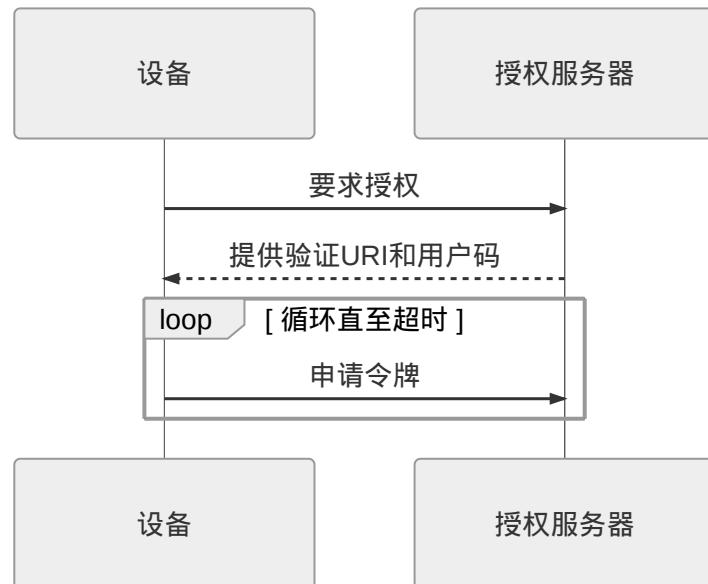
客户端模式是指第三方应用（行文一致考虑，还是继续叫第三方应用）以自己的名义，向授权服务器申请资源许可。这通常用在一些管理或者自动处理形场景之中。举个例子，譬如我开了一家网上书店Fenix's Bookstore，因为小本经营，不像京东那样全国多个仓库可以调货，因此必须保证只要客户成功购买，书店就必须有货可发，不能超卖。但经常有顾客下了订单又拖着不付款，导致部分货物处于冻结状态。所以Fenix's Bookstore中有一个

订单清理的定时服务，自动清理掉超过两分钟的未付款的订单。这件用户里，订单肯定属于用户自己的资源，如果把订单清理服务看作一个独立的第三方应用的话，他就不可能向用户去申请授权，而应该直接以自己的名义向授权服务器申请一个能清理所有用户订单的授权。客户端模式的时序如下图所示：



后面在微服务的“[可靠通讯](#)”中会讲到，并不提倡各个微服务之间有默认的信任关系，所以服务之间调用也需要先认证授权。此时客户端模式也是一种常用的服务间认证授权的解决方案。Spring Cloud版本的Fenix's Bookstore是采用这种方案来保证微服务之间的合法调用的，Istio版本的Fenix's Bookstore则启用了双向mTLS通讯，使用客户端证书保障安全，感兴趣的读者可以对比一下这两种方式差异优劣。

OAuth2中还有一种与客户端模式类似的授权模式，在[RFC 8628](#)中定义为“设备码模式（Device Code）”，这里顺便简单提一下。设备码模式用于在无输入的情况下区分设备是否允许，典型的应用便是手机锁网解锁（锁网在国内较少，但在国外很常见）或者激活（譬如某游戏机注册到某个游戏平台）的过程。时序如下图所示：



进行验证时，设备需要从授权服务器获取一个URI地址和一个用户码，然后需要用户手动或设备自动地到验证URI中输入用户码。在这个过程中，设备会一直循环，尝试去获取令牌，直到拿到令牌或者用户码过期为止。

凭证

凭证 (Credentials)

系统如何保证它与用户之间的承诺是双方当时真实意图的体现，是准确、完整且不可抵赖的？

在前面介绍OAuth2的内容中，每一种授权模式的目的都是拿到访问令牌，但从未涉及过拿回来的令牌应该长什么样子？反而还挖了一些坑（为何说OAuth2的一个主要缺陷是令牌难以主动失效）还没有填。这节我们讨论凭证，此话题中令牌必须得是主角了，此外，我们还要在这节讨论不使用OAuth2、最传统的方式是如何完成前面所讨论的认证、授权的。

Cookie-Session

我们知道，HTTP协议是一种无状态的传输协议，无状态是指协议对事务处理没有上下文的记忆能力，每一个请求都是完全独立的，但是我们中肯定有许多人并没有意识到HTTP协议无状态的重要性。假如你做了一个简单的网页，其中包含了1个HTML、2个Script脚本、3个CSS、还有10张图片，这个网页成功展示在用户屏幕前，需要完成16次与服务端的交互，由于服务器响应的顺序与发送请求的先后没有直接联系，按照可能出现的响应顺序，一共会有 $P(16,16) = 20922789888000$ 种可能性。试想一下，如果HTML协议不是设计成无状态的，这16次请求各个有依赖关联，先调用哪一个、先返回哪一个，都会对结果产生影响的话，那协调工作会有多么复杂。

可是，HTTP协议的无状态特性又有悖于我们最常见的网络应用，譬如认证、授权方面，系统总得要获知用户身份才能提供服务，因此，我们也希望HTTP能有一种手段，让服务器至少有办法能够区分出发送请求的用户是谁。为了实现这个目的，[RFC 6265](#)规范中定义了HTTP的状态管理机制，在HTTP协议中设计了Set-Cookie指令，该指令的含义是以K/V值对的方式向客户端发送信息，此信息将在此后一定时间内的每次HTTP请求中，以名为Cookie的Header中附带着重新发回服务端，一个典型的Set-Cookie指令如下所示：

```
Set-Cookie: id=icyfenix; Expires=Wed, 21 Feb 2020 07:28:00 GMT; Secure;  
HttpOnly
```

从此以后，当客户端对同一个域名（或者Path）的请求中都会带有值对信息“id=icyfenix”，例如以下所示：

```
GET /index.html HTTP/2.0  
Host: icyfenix.cn  
Cookie: id=icyfenix; sessionid=38afes7a8
```

根据每次请求传到服务端的Cookie，服务器就能分辨出请求来自于哪一个用户。由于Cookie是放在请求头上的负载（Payload，这个词后面还要频繁用到），不可能存储太大量的数据，放在Cookie中传输也不安全（被窃取，被篡改），所以通常是不会像例子中“id=icyfenix”这样的直接携带数据的。一般来说，Cookie中一般传输的是一个无意义的不重复的字符串，通常以sessionid或者jsessionid为名，服务器拿这个字符串为Key，再在内存中开辟一块空间，以Key/Entity的结构存储每一个在线用户的上下文状态，并辅以一些超时自动清理的管理措施，这种服务端的状态管理机制就是今天大家耳熟能详的Session，Cookie-Session就是在今天广泛应用于大量系统中的、服务端与客户端联动的状态管理机制。

Cookie-Session的方案在本章的主题“安全”上其实多少是占有一定优势的：信息都存储于服务器，不易遭遇传输中被泄漏、篡改的风险，只要通过域保护机制和传输层安全，保证Cookie中的键值不被窃取（如在“漏洞利用”小节中介绍的CSRF、XSS攻击）导致被冒认身份即可。Cookie-Session方案另一大优点是服务端有主动的管理能力，可根据自己的意愿随时修改、清除任意上下文状态，如实现强制某用户下线的功能就很容易。

Session-Cookie在单节点单体服务环境中是非常合适的方案，但当服务能力需要水平扩展，要部署集群时就开始面临一些麻烦了，由于Session建立在服务器的内存中，当服务器水平拓展成多节点时，我们必须在以下三种方案中选择其一：

- 要么就牺牲集群的一致性（Consistency）能力，让均衡器采用亲和式的负载均衡算法（譬如根据用户IP或者sessionid来分配节点），每一个特定用户发出的所有请求都一直被分配到其中某一个节点来提供服务，每个节点都不重复地保存着一部分用户的状态，如果这个节点崩溃了，里面的用户状态便完全丢失。

- 要么就牺牲集群的可用性（ Availability ）能力，让各个节点之间采用复制式的Session，每一个节点中的Session变动都会发送到组播地址的其他服务器上，这样某个节点崩溃了，不会中断都某个用户的服务，但Session之间组播复制的同步代价高昂，节点越多时越是如此。
- 要么就牺牲集群的分区容错（ Partition Tolerance ）能力，让普通的服务节点中不再保留状态，将上下文集中放在一个所有服务节点都能访问到的数据节点中进行存储。此时的矛盾是数据节点就成为了单点，一旦数据节点损坏，整个集群都不能提供服务。（多说一句，现在数据节点常见以Redis来搭建，本身Redis通常也会做集群，但将大集群的CAP问题放到小集群里，并不会让问题消失，简而言之就是：[禁止套娃](#)）

以后，我们在微服务架构中还会遇到更多分布式的问题，还会经常受到CAP理论（C、A、P必须牺牲一个）的打击，这是一个很值得深入探讨的技术权衡，但毕竟与本章的“安全”关系不大，这里就不再展开了。现在我只想知道一个问题的答案：前面三种方案都有缺陷，那在分布式应用中，就没有能绕过这些问题的解决方案吗？

我的答案是：有，也没有。如果说要解决分布式环境下的共享数据的CAP矛盾，这是被数学严格证明了不可能的，所以分布式环境中的状态管理一定会受到CAP的限制。但如果是在解决分布式下的认证授权问题，那确实还有一些别的法子可想。前面这句话的言外之意是提醒读者，接下来的JWT令牌与Cookie-Session并不是对等的技术方案，它只解决认证授权问题，充其量能携带少量非敏感的信息，只是Cookie-Session在认证授权问题上的替代品，而不会成为Cookie-Session本身的革命者与继承人。

JWT

前面介绍的Cookie-Session机制在分布式环境下遇到一些问题，在多方系统中，就更不可能谈什么Session层面的数据共享了，而且Cookie也没法跨域。看来，服务器多了，确实不好解决，那就换个思路吧，客户端是唯一的，把数据存储在客户端，每次随着请求发回服务器——JWT就是这种思路的典型代表。

JSON Web Token（JWT），定义于[RFC 7519](#)的令牌格式，是目前广泛使用的一种令牌，尤其是与OAuth2配合应用于分布式的、涉及多方的应用系统之中。介绍JWT的具体构成之前，我们先来看一下它是什么样子的，一个JWT的例子如下图所示：

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
1c2VyX25hbWUiOiJpY31mZW5peCIsInNjb3BlIjp
bIkFMTCJdLCJleHAiOjE10DQyNTA3MDQsImF1dGh
vcml0aWVzIjpBI1JPTEVfVVNFUiIsI1JPTEVfQUR
NSU4iXSianRpIjoiMTNmNGN1MWQtNmY20C00NzQ
xLWI5YzYtMzkyNzU10GQ5NzR1IiwiY2xpZW50X21
kIjoiYm9va3N0b3J1X2Zyb250ZW5kIiwidXNlcm5
hbWUiOiJpY31mZW5peCJ9.82awQU4IcLVXr7w6px
cUCWrcEHKq-LRT7ggPT_ZPhE0
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYOUT: DATA

```
{
  "user_name": "icyfenix",
  "scope": [
    "ALL"
  ],
  "exp": 1584250704,
  "authorities": [
    "ROLE_USER",
    "ROLE_ADMIN"
  ],
  "jti": "13f4ce1d-6f68-4741-b9c6-3927558d974e",
  "client_id": "bookstore_frontend",
  "username": "icyfenix"
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
)  secret base64 encoded
```

JWT令牌结构

以上截图来自于网站<https://jwt.io/>，当然，数据是我自己编的。左边的是JWT的本体，它通过名为Authorization的Header发送给服务端，前缀是在[RFC 6750](#)中定义的bearer，这点在之前关于“认证”的小节中提到过，一个完整的HTTP请求实例如下所示：

```
GET /restful/products/1 HTTP/1.1
Host: icyfenix.cn
Connection: keep-alive
Authorization: bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX25hbWUiOiJpY31mZW5peCIsIn
Njb3BlIjpBIkFMTCJdLCJleHAiOjE10DQ5NDg5NDcsImF1dGhvcmI0aWVzIjpBI1JPTEVfV
VNFiIsI1JPTEVfQURNSU4iXSianRpIjoiOWQ3NzU4NmEtM2Y0Zi00Y2JiLTk5MjQtZmUy
Zjc3ZGZhMzNkIiwiY2xpZW50X2lkIjoiYm9va3N0b3J1X2Zyb250ZW5kIiwidXNlcm5hbWU
i0iJpY31mZW5peCJ9.539WMzbjv63wBtx4ytYYw_Fo1ECG_9vsgAn8bheflL8
```

图中右边的内容是经过Base64URL转码之后的令牌明文，是的，明文，JWT令牌默认是不加密的（你自己要加密也行就，接收时自己解密即可）。从明文中可以看到JWT令牌是以JSON结构（毕竟叫JSON Web Token）存储的，结构上可划分为三个部分，每个部分间用点号“.”分隔开。

第一部是令牌头 (Header) , 内容如下所示 :

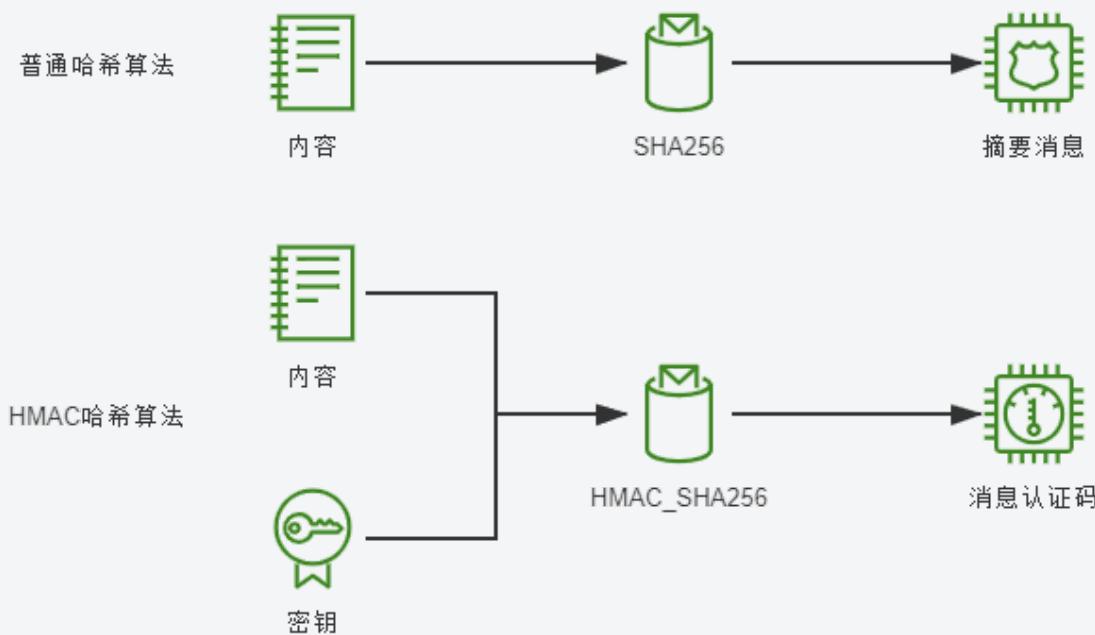
```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

它描述了令牌的类型 (统一为 typ:JWT) 和令牌签名的算法 , 示例中 HS256 为 HMAC SHA256 算法的缩写 , 其他各种系统所支持的签名算法可以参考 <https://jwt.io/> 网站所列。

额外知识 : 散列消息认证码

在本节及后面其他关于安全的内容中 , 经常会在某种算法前出现 “HMAC” 的前缀 , 这是指散列消息认证码 (Hash-based Message Authentication Code , HMAC) 。目标上可以简单将它理解为一种带有密钥的哈希摘要 , 实现形式上可以简单理解为密钥通过加盐方式混入 , 与内容一起做哈希即可。

HMAC 哈希与普通哈希算法的差别是普通的哈希算法通过 Hash 函数结果易变性保证了原有内容未被篡改 , HMAC 不仅保证了内容未被篡改过 , 还保证了该哈希确实是密钥的持有人所生成的。



第二部分是负载 (Payload) , 是令牌真正需要向服务端传递的信息 , 在认证问题中 , 至少应该包括告诉服务端 “ 我是谁 ” 的信息 , 在授权问题中 , 至少应该包括告诉服务端 “ 我属于什

么角色/权限，有哪些许可”。负载部分是可以完全自定义的，根据具体要解决的问题不同，设计自己所需要的信息（但不能太多，毕竟受HTTP Header大小的限制）。一个JWT负载的示例如下所示：

```
{  
    "username": "icyfenix",  
    "authorities": [  
        "ROLE_USER",  
        "ROLE_ADMIN"  
    ],  
    "scope": [  
        "ALL"  
    ],  
    "exp": 1584948947,  
    "jti": "9d77586a-3f4f-4cbb-9924-fe2f77dfa33d",  
    "client_id": "bookstore_frontend"  
}
```

json

而JWT在RF 7519中推荐（无强制约束）了7个声明名称（Claim Name），如有需要用到这些内容，建议字段名与官方的保持一致：

- iss (Issuer) : 签发人
- exp (Expiration Time) : 令牌过期时间
- sub (Subject) : 主题
- aud (Audience) : 令牌受众
- nbf (Not Before) : 令牌生效时间
- iat (Issued At) : 令牌签发时间
- jti (JWT ID) : 令牌编号

此外在RFC 8225、RFC 8417、RFC 8485等规范文档，以及OpenID中都定义有约定公有含义的名称，比较多我就不贴出来了，可以参考[IANA JSON Web Token Registry](#)。

第三部分是**签名**（Signature），签名的意思是，使用特定的签名算法（在对象头中公开），使用特定的密钥（Secret，由服务器进行保密，不能公开）对前面两部分内容进行加密计算，以例子中JWT默认的HMAC SHA256算法为例，将通过以下公式产生签名值：

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload),  
secret)
```

java

签名的意义在于确保负载中的信息是可信的、没有被篡改的，也没有在传输过程中丢失。因为被签名的内容哪怕发生了一个字节的变动，也会导致整个签名发生显著变化。此外，由于这件事情只能由认证/授权服务器完成（只有它知道Secret），任何人都无法在篡改后重新计算出合法的签名值，所以服务端才能够完全信任客户端传上来的JWT中的负载信息。

之前提到了JWT默认采用的签名算法是HMAC SHA256，是一种哈希摘要算法，属于不可逆的“加密”，过程住实质上是不用依赖密钥的，这时候的密钥实际上承担了加盐的作用。由于加密与验证均需要中心化的授权服务器来提供，所以这种方式较适合于单体应用。在多方系统或者授权服务与资源服务分离的分布式应用中，通常会采用非对称加密算法（主要有基于大数分解困难性的RSA SHA256算法和基于椭圆曲线的ECDSA SHA256算法）来进行签名，这时候除了授权服务端持有的可以用于签名的私钥外，还会对其他服务器公开一个公钥（公钥公开格式一般遵循[JSON Web Key规范](#)），公钥不会用来签名，但是能被其他服务用于验证签名是否由私钥所签发的。这样其他服务器也能不依赖授权服务器独立判断JWT令牌中的信息的真伪。

在Fenix's Bookstore的单体服务版本中，将会采用了JWT默认的HMAC SHA256算法来加密签名。在Fenix's Bookstore的Spring Cloud和Istio服务网格版本里，终端用户认证过程将会由服务网格的基础设施参与，此时将采用RSA SHA256算法进行非对称加密来进行签名，希望更深入了解凭证安全的同学，可以通过代码对比，想一下为什么微服务版本要改HMAC为RSA（非对称版本在微服务中交互次数明显小于默认的中心化加密版本）来进一步理解两者的差异。更多关于哈希、对称、非对称加密的讨论，会在“[传输](#)”一节中继续进行。

JWT令牌是多方系统中一种优秀的凭证载体，它不需要任何一个服务节点保留任何一点状态信息，就能够保障认证服务与用户之间的承诺是双方当时真实意图的体现，是准确、完整、不可篡改、且不可抵赖的。同时，由于JWT本身可以携带少量信息，这十分有利于RESTful API的设计，能够较容易地做成无状态服务，在做水平扩展时就不需要像前面Cookie-Session方案那样考虑如何部署的问题。现实中也确实有一些项目（譬如Fenix's Bookstore）直接采用JWT来承载上下文来实现完全无状态的服务端，这能获得很大的好处，譬如，在你调试Fenix's Bookstore的程序时，随时都可以停止、重启服务端程序，服务重启

后客户端仍然是可以毫无感知地继续操作流程；而对于有状态的系统，一般就必须通过再次登录、进行前置业务操作来给服务端重建状态（以上这句话所指的“好处”不是开发时方便重启，而是指不必顾虑状态地增加或者减少服务来进行伸缩）。

目前，在大型系统中完全使用JWT来保存上下文状态，服务端完全不持有状态仍是不太现实的，不过将最热点的服务接口单独抽离出来，做成无状态的、幂等的服务，是一种很有效的提升系统吞吐能力的架构设计。这部分内容将在微服务架构的部分如何划分微服务的章节中进一步探讨。

JWT并不是没有缺点的完美方案，它存在着以下几个明显或者不明显的缺点：

- **令牌难以主动失效**：JWT令牌一旦签发，理论上就和认证服务器再没有什么瓜葛了，在到期之前就会始终有效，除非服务器部署额外的逻辑，这对某些管理功能的实现是很不利的。譬如，有一种颇为常见的需求是：要求一个用户只能在一台设备上登录，在B设备登陆后，之前已经登录过的A设备就应该自动退出。如果采用JWT，就必须设计一个“黑名单”的额外的逻辑，用来把要主动失效的令牌集中存储起来，而无论这个黑名单是实现在Session、Redis或者数据库中，都会让服务退化回有状态，降低了JWT本身的价值（但黑名单还是很常见的做法，需要维护的黑名单一般是很小的状态量，不少场景中是有存在意义的）。
- **更容易遭受重放攻击**：首先说明Cookie-Session也是有重放攻击问题的，只是因为Session中的数据控制在服务端手上，应对重放攻击会相对主动一些。要在JWT层面解决重放攻击需要付出比较大的代价，无论是加入全局序列号（HTTPS协议的思路）、Nonce字符串（HTTP Digest验证的思路）、挑战应答码（当下网银动态令牌的思路）、还是缩短令牌有效期强制频繁刷新令牌，在真正应用起来时其实都是很麻烦的，真要处理重放攻击，启用HTTPS是正道。
- **只能携带相当有限的数据**：HTTP协议并没有强制约束Header的最大长度，但是，各种服务器（甚至是浏览器）都会有约束，譬如Tomcat就要求Header最大不超过8KB，而在Nginx中则默认为4KB，因此在令牌中存储过多的数据不仅浪费带宽，还有额外的出错风险。
- **令牌在客户端如何存储**：严谨地说，这个并不是JWT的问题而是你的问题。如果授权之后，操作完了关掉浏览器这是结束了，那把令牌放到内存里面，压根不考虑持久化那是最理想的。但并不是谁都能忍受一个网站关闭之后下次就一定强制要重新登陆的（大概也就银行的网站可以忍）。那这样的话，客户端该把令牌存放到哪里？Cookie？localStorage？还是其他什么？这个问题的答案可能因人而异，但无论如何，你都需要考虑到令牌的安全性和持久性。

orage ? Indexed DB ? 它们都有泄漏的可能 , 而令牌一旦泄漏 , 别人就可以冒充你的身份做任何事情。

- **无状态也不总是好处** : 这个其实不也是JWT的问题。如果不能想像无状态会有什么不好的话 , 我给你提个需求 : 请基于无状态JWT的方案 , 做一个在线用户统计功能。兄弟 , 难搞哦。

我在写这篇文章的时候 , 在网上搜索资料 , 发现JWT的争议和吹捧都不少。技术只是工具而已 , 无论是迷信它还是抵制它^④ , 都并无必要。

保密

保密 (Confidentiality)

系统如何保证敏感数据无法被包括系统管理员在内的内外部人员所窃取、滥用？

保密是加密和解密的统称，是以某种特殊的算法改变原有的信息数据，使得未授权的用户即使获得了已加密的信息，但因不知解密的方法，仍然无法了解信息的内容。

保密这个话题，按照需要保密信息所处的环节不同，可以划分为“信息在客户端时的保密”、“信息在传输时的保密”和“信息在服务端时的保密”，又或者进一步概括为“端的保密”和“链路的保密”。我们把最复杂、最有效，但却又早就有了标准解决方案的“传输环节”单独提取出来，放到下一个节去讨论。在本小节中，只讨论两个端的环节，即在客户端和服务端中的信息保密问题，谈一下笔者的几个观点。

安全的强度

首先来说一下“安全”的程度问题，保密的安全与否不应该被视为一个离散的二元选项，不是仅有“安全”或者“不安全”的差别，而是随着你的应用所要求的保密程度不同，应该有着不同的安全强度与之对应。这里面说的意思与很多口号中强调的“安全无小事”、“99%安全加1%的漏洞等于零”并不是一码事。我通过以下这些逐步提升的攻击手段和应对措施来解释“安全强度”是意味着什么：

1. 给密码做最简单的MD5，如果你的密码本身比较复杂，那一次简单的MD5至少可以保证密码不会被逆推出明文，密码在一个系统中泄漏了不至于威胁到其他系统的使用，但这不能阻止弱密码被彩虹表攻击所逆推。
2. 给密码加上固定的盐值，如果给密码加上盐值，可以替弱密码建立一道防御屏障，一定程度上防御已有的彩虹表攻击，但不能阻止加密结果被窃取后（譬如在链路上被抓包了），攻击者直接发送加密结果给服务端进行冒认。

3. 给密码加上动态的盐值，如果每次密码向服务端传输时都加入了动态的盐值，让每次加密的结果都不同，那即时传输给服务端的加密密码被窃取了，也无法用来冒认，但这只能保护登录这一个操作，无法阻止对其他功能的重放攻击。
4. 采用动态令牌与服务端的逻辑配合，可以做到防止重放攻击，依然无法抵御传输过程中被嗅探而泄漏信息的问题（如前面说的在链路上被抓包了）。
5. 启用HTTPS（且恰当选择支持的密码学算法、保护好证书），可以防御链路上的恶意嗅探，也在协议层面解决了重放攻击的问题，但它依然存在有被攻击的可能性，譬如受到证书攻击导致握手失败。
6.

到了第5点，只要做法规范，已经可以抵御绝大多数系统性的安全风险了，但也意味着你需要为它付出一些代价（包括加解密的算力，也包括购买证书的费用）。而安全的强度还可以用不同途径继续往上提升，如许多网站会使用手机验证码开辟另一条独立的信息传输渠道来保障安全、如银行会使用有专门物理存储的证书（就是俗称的U盾）来保障安全、如国家电网那样建设遍布全国各地的与公网物理隔离的专用网络来保障安全，等等。显然追求安全强度同时也意味着付出更多代价，肯定不是任何一个网站、系统都需要无限拔高的安全强度。

另一个问题是安全强度有尽头吗？存不存在某种绝对安全的保密方式？答案可能出乎多数人的意料，确实是有的。信息论之父香农严格证明了一次性密码（One Time Password）的绝对安全性。但是使用一次性密码必须有个前提，就是先把安全的把密码（密码列表）传达给对方。譬如，给你的朋友（人肉）送去一本存储了完全随机密码的密码本，然后每次使用其中一条密码来进行加密通讯，用完一条丢弃一条，理论上这是绝对安全的，但显然这对于公众互联网是没有任何的可执行性。

客户端加密

客户端在用户登录、注册一类场景里是否需要对密码进行加密，这个问题一直存有争议。我的观点是，为了保证密码不被黑客窃取而做客户端加密没有太多意义，上HTTPS可以说是唯一的普通系统实际可行的解决方案。但是！为了保证密码不在服务端被滥用，在客户端就开始加密是很有意义的。大网站被拖库的事情层出不穷，密码明文被写入数据库、被输出到日志中之类的事情屡见不鲜，做系统设计时就应该把明文密码这种东西当成是最烫手的山芋来看待，越早消灭掉越好。

关于第一个“没有太多意义”，有人不理解为什么为什么客户端加密对防御黑客会没有意义，我举个例子，在极端情况下，客户端可能被整个架空掉，这样上面无论做了什么防御措施都成“马其诺防线”了。典型的就是之前已经提到的中间人攻击，它可以通过劫持掉了客户端到服务端之间的某个节点，包括但不限于代理（通过HTTP代理返回赝品）、路由器（通过路由导向赝品）、DNS服务（直接将你机器的DNS查询结果替换为赝品地址）等等，把你想要访问的登陆页面整个给替换掉（全替换掉工作量太大，一般不会去做，都是注入一段恶意的JavaScript代码到正版的页面里）。最简单的劫持路由器，在局域网内其他机器释放ARP病毒便有可能做到这一点。这部分内容属于链路安全，我们将在下一节来讲如何防御，这里附带Mozilla[对中间人攻击的一段介绍以供参考](#)。

额外知识：中间人攻击（Man-in-the-Middle Attack，MitM）

在消息发出方和接收方之间拦截双方通讯。用日常生活中的写信来类比的话：你给朋友写了一封信，邮递员可以把每一份你寄出去的信都拆开看，甚至把信的内容改掉，然后重新封起来，再寄出去给你的朋友。朋友收到信之后给你回信，邮递员又可以拆开看，看完随便改，改完封好再送到你手上。你全程都不知道自己的信件和收到的信件都经过邮递员这个“中间人”转手和处理——换句话说，对于你和你朋友来讲，邮递员这个“中间人”角色是不可见的。

关于第二个“很有意义”，居然也有人会抬杠。一种是说涉及到密码等敏感信息的都会由靠谱的人完成，或者就是他本人做的，所以不会出问题，我觉得这个就没什么必要反驳了，开心就好。另一种的观点是保存明文密码（把不含盐的哈希结果也作明文看待）的目的是为了便于客户端做动态盐值，因为这需要服务端存储了明文才能每次用新的盐值重新加密来与客户端传上来的加密结果进行比较。我的观点是每次从服务端请求盐值在客户端动态加盐往往得不偿失，应在真正防御性的密码加密存储应该在服务端进行，因为客户端无论是否动态加盐，都不能代替HTTPS。

密码存储和验证

下面以Fenix's Bookstore的实现为具体样例，介绍从密码如何从客户端传输到服务端，存储进数据库的全过程。在保障一定安全强度的同时，避免消耗过多的运算资源，验证起来也比较便捷。这套过程对于一般的系统，配合一定的约束（如密码要求长度、特殊字符等），再配合HTTPS传输应该是够用的。即使在客户采用了弱密码、客户端盐值泄漏（本

来就不是保密的）、服务端被拖库泄漏了存储的密文和动态盐值这些问题同时发生，也没有用户明文密码被逆推出来的风险。

以下为密码创建的过程，

1. 用户在客户端注册，输入明文密码：123456。

```
password = 123456
```

java

2. 客户端对用户密码进行简单Hash，可选的算法有MD2/4/5、SHA1/256/512、BCrypt、PBKDF1/2，等等。

```
client_hash = MD5(password) // e10adc3949ba59abbe56e057f20f883e
```

java

3. 为了防御彩虹表攻击，应加盐处理，客户端加盐可取固定的字符串，或者伪动态（日期、用户名加上固定字符串，反正就是服务端不需要额外通讯可以得到的值）的盐值。

```
client_hash = MD5(MD5(password) + salt) // SALT =  
$2a$10$o5L.dWYEjZjaej0mN3x4Qu
```

java

4. 我们假设攻击者截获了传输，把哈希值和盐值都拿到了，那他可以枚举遍历所有10位数以内（10位数只是举个例子，反正就是弱密码，你拿1024位随机字符当密码用，加不加盐，彩虹表都跟你没关系）的弱密码，然后对每个密码再加盐计算，得到一个固定盐值的对照彩虹表。为了应对这种暴力破解，我们需要引入慢哈希函数来代替MD5来加强安全性。

慢哈希函数是指这个函数执行时间（准确地说是运算次数）是可以调节的，BCrypt算法就是一种慢哈希函数，在做哈希时接收盐值salt和执行成本cost两个参数（代码层面cost一般是混入在salt中，譬如上面例子中的salt就是混入了10轮运算的盐值，10轮的意思是 2^{10} 次哈希，cost参数是放在指数上的，最大取值就31）。如果我们控制BCrypt的执行时间大概是0.1秒完成一次哈希计算的话，按照1秒生成10个哈希的速度，算完所有的10位大小写字母和数字组成的弱密码大概需要 $P(62,10)/(3600*24*365)/0.1=1,237,204,169$ 年。

```
client_hash = BCrypt(MD5(password) + salt) //  
MFfTW3uNI4eqhwDkG7HP9p2mzEUu/r2
```

java

5. 现在将哈希传输到服务端，链路这段安全在下一节去探讨。

服务端接受到哈希值后，对每一个密码都动态生成一个随机盐值。比较主流的建议是采用“[密码学安全伪随机数生成器](#)”（ Cryptographically Secure Pseudo-Random Number Generator , CSPRNG ）来产生一个长度与哈希值相等的随机字符串。对于Java语言，从Java SE 7起提供了java.security.SecureRandom类，用于支持CSPRNG字符串生成。

```
SecureRandom random = new SecureRandom();  
byte server_salt[] = new byte[36];  
random.nextBytes(server_salt); // tq2pdxb1kbgp8vt8kbdpmzdh1w8bex
```

java

6. 将盐混入客户端传来的哈希值，生成要存入数据库的密文，并将随机生成的盐值一并写入到同一条记录中。在服务端中就不建议采用慢哈希算法，对CPU占用率的影响较大，Spring Security 5的StandardPasswordEncoder提供了SHA256哈希算法的实现，就以此为例。

```
server_hash = SHA256(client_hash + server_salt); //  
55b4b5815c216cf80599990e781cd8974a1e384d49fbde7776d096e1dd436f67  
DB.save(server_hash, server_salt);
```

java

7. (可选) 出于对SHA256安全性的不信任，Spring Security 5中StandardPasswordEncoder已被@Deprecated，介意或者想偷懒简化操作的话，推荐第5、6步采用BCryptPasswordEncoder来替代。尽管使用的是BCrypt算法，但默认构造函数中的cost是-1，即进行 $2^{-1}=1$ 次哈希计算，这并不会造成服务端压力。说可以偷懒是因为用BCryptPasswordEncoder的话就不需要专门传入盐值，它本身就会调用CSPRNG产生盐值，也不需要给数据库添加盐值字段了，在它生成密码的前32位自动存储了盐值。

以下为密码验证的过程：

1. 客户端，用户在登陆页面中输入密码明文：123456，经过与注册相同的加密过程，向服务端传输加密后的结果。

```
authentication_hash = MFFTW3uNI4eqhwDkG7HP9p2mzEUu/r2
```

java

2. 服务端，接受到客户端传输上来的哈希值，从数据库中取出登陆用户对应的密文和盐值，采用服务端的哈希算法，对客户端传来的哈希值、服务端盐值计算出哈希结果。

```
result = SHA256(authentication_hash + server_salt); //  
55b4b5815c216cf80599990e781cd8974a1e384d49fbde7776d096e1dd436f67
```

java

3. 比较上一步的结果和数据库储存的哈希值是否相同，如果相同那么密码正确，反之密码错误。

```
authentication = compare(result, server_hash) // yes
```

java

传输

传输 (Transport Security)

系统如何保证通过网络传输的信息无法被第三方窃听、篡改和冒充？

这节的主角是签名、证书、TLS等，但不会涉及“到哪找免费的CA证书？”、“如何生成数字证书？”、“如何把证书置入Web服务器？”这一类操作性的话题，而更多是对整套传输安全层原理的讲述。尽管这部分内容相对较难，但如果前面你已经阅读过并理解了认证、授权、凭证、保密的内容，而又对SSL/TLS本身没有什么了解的话，那这一节可能会是最容易理解的讲述传输安全层工作原理的方式。笔者将从“假设传输层安全得不到保障，攻击者如何摧毁之前认证、授权、凭证、保密中所提到的种种安全机制”为具体场景来讲解传输层安全所面临的问题和它的解决方案。

摘要、加密与签名

我们从JWT令牌的一小段“题外话”来引出整套现代加密通讯体系，以便于阐述哈希摘要、对称/非对称加密的特点与局限。我们知道，JWT中携带信息的价值来自于它是被签名的、不可篡改的信息。这一点之前介绍到：

签名的意义在于确保负载中的信息是可信的、没有被篡改的，也没有在传输过程中丢失。因为被签名的内容哪怕发生了一个字节的变动，也会导致整个签名发生显著变化。此外，由于这件事情只能由认证/授权服务器完成（只有它知道Secret），任何人都无法在篡改后重新计算出合法的签名值，所以服务端才能够完全信任客户端传上来的JWT中的负载信息。

我们来深入分析一下，“签名”具体是如何让负载中的信息变得“不可篡改”的。以默认的SHA 256哈希算法为例，进行签名，相当于进行如下计算过程：

```
signature = SHA256(base64UrlEncode(header) + "." +  
base64UrlEncode(payload), secret)
```

java

理想的哈希算法通常都会具备两个特性：一是**易变性**，这是指算法的输入发生了任何一点变动，都会导致**雪崩效应**（Avalanche Effect），使得输出结果发生极大的变化。这个特性常被用来做校验，譬如互联网上下载大文件，常会附有一个哈希校验码，以确保下载下来的文件没有因网络或其他原因与原文件产生哪怕一个字节的偏差。二是**不可逆性**，这是指算法根据输入计算输出结果耗费的算力资源极小，但根据输出结果反过来推算原本的输入，耗费的算力就极大。一个经常被人们用来讲解不可逆性的例子是**大数分解**，我们可以轻而易举的地（以O(nlogn)的复杂度）计算出两个大素数的乘积：

```
97667323933 * 128764321253 = 12576066674829627448049
```

java

根据**算术基本定理**，质因数的分解形式是唯一的，且前面笔者所举例的运算因子已经都是素数，所以12576066674829627448049的分解形式只有唯一的上面所示的一种答案，但是如何对大数进行因数分解，迄今没有找到多项式时间的解法（24位十进制数的因数分解完全在现代计算机的暴力处理能力范围内，这里只是举例。但目前很多计算机科学家都相信大数分解问题就是一种P!=NP的证例，尽管也并没有人能证明它一定不存在多项式时间的解法）。不可逆性常被人用来做数字签名，利用的就是如果你知道密钥，很容易通过明文算出签名值，但知道明文和签名值，几乎不可能逆推出密钥，这就实现了签名易于验证，难以破解的特点。

必须注意，签名“易于验证、难以破解”是建立在密钥不会泄漏，也不会被篡改的基础上的。当授权服务器与资源服务器是同一个服务时，JWT运作不会遭遇什么风险。而当授权服务器与资源服务器并不是同一个，他们之间就涉及到资源服务其如何验证的问题，无论是资源服务器对每个收到的令牌都请求授权服务器验证一下，还是资源服务器自己也拿到密钥来自行验证令牌真伪都是不可行的。这种情况的解决方案前面讨论中已提到过：

在多方系统、授权服务与资源服务分离的实际应用中，通常会采用非对称加密算法（典型如RSA）来进行签名，这时候除了授权服务端持有的可以用于签名的私钥外，还会对其他服务器公开一个公钥，公钥不会用来签名，但是能被其他服务用于验证签名是否由私钥所签发的。这样其他服务器也能不依赖授权服务器独立判断JWT令牌中的信息的真伪。

非对称加密就是加密和解密使用的是不同的密钥的算法，那自然对称加密就是指加密是指加密和解密是一样的密钥的算法。不知道上面看这段话的时候，你心中是否会想“这里写JWT通常会采用非对称加密算法，那改用对称加密行不行呢？”之类的疑问。答案是除非有其他传递或者动态协商密钥的途径，否则这个场景中对称加密是不可行的，因为对称加密

只有一个密钥，授权和资源服务不在同一台服务器的话，如何将这个密钥传送给资源服务器？再加密一次传送的话就成了“[蛋鸡悖论](#)”了。

事实上，在分布式环境中，真正能够用来签名的，通常都只有非对称加密算法。在它的密钥对中，其中一个密钥是对外公开的，所有人都可以获取到，称为公钥，另外一个密钥是不公开的称为私钥。这两个密钥谁加密、谁解密构成了两种不同的用途：

1. 公钥加密，私钥解密，这种就是**加密**，用于向公钥所有者发布信息，这个信息可能被他人篡改，但是无法被他人获得。如果甲想给乙发一个安全的保密的数据，那么应该甲乙各自有一个私钥，甲先用乙的公钥加密这段数据，再用自己的私钥加密这段加密后的数据。最后再发给乙，这样确保了内容即不会被读取，也不会被篡改。
2. 私钥加密，公钥解密，这种就是**签名**，用于让所有公钥所有者验证私钥所有者的身份并且用来防止私钥所有者发布的内容被篡改。但是不用来保证内容不被他人获得。

看到这里，可能有人在想只用“非对称加密”行不行？为什么还需要对称加密？答案是非对称加密算法对加密内容的长度有限制，不能超过公钥长度。譬如说现在常用的公钥是长度是2048 bits，意味着明文不能超过256 bytes。此外，由于对称加密的设计难度相对较小，其加密的效率一般远高于非对称加密，这决定了在大数据量的加密数据传输中，通常是两种加密算法结合使用的，用非对称加密来传递密钥，收到密钥后用对称加密来加密内容。

下表把前面涉及到的三种算法放到一块，列举了它们的主要特征、用途和局限性：

类型	特点	常见实现	主要用途	主要局限
哈希摘要	不可逆，即不能解密，所以并不是加密算法，只是一些场景把它当作加密算法使用 易变性，输入发生1Bit变动，就可能导致输出结果50%的内容发生改变。 无论输入长度多少，输出长度固定（2的N次幂）	MD2/4/ 5/6、SH A0/1/25 6/512	摘要	无法解密
对称加密	加密是指加密和解密是一样的密钥 设计难度相对较小，执行速度相对较块 加宽数明文长度不受限制	DES、A ES、RC 4、IDEA	加密	要解决如何把密钥安全地传递给解密者

类型	特点	常见实现	主要用途	主要局限
非对称加密	加密和解密使用的是不同的密钥 明文长度不能超过公钥长度	RSA、B CDSA、 ElGamal	签名、 传递 密钥	加宽数明文长 度受限

现在我们再回到多方系统如何验证令牌的问题中来。有了非对称加密，公钥可以不需要加密地公开了，那问题难道还没有解决了吗？并没有，还存在一个明显的漏洞，公钥虽然是公开的，但如何保证要获取公钥的资源服务，拿到的公钥就是授权服务所希望它拿到的呢？如果公钥在网络传输过程中，获取公钥的这一步被攻击者截获并篡改了，返回了攻击者自己提供的公钥，那以后攻击者就可以用自己的私钥签名，让资源服务器无条件信任自己的所有动作了。这里公钥显然也无法再用加密来传输，否则也是一个蛋鸡问题。

数字证书

当我们无法以“签名”的手段来达成信任时，就只能求助于其他途径。不妨想想真实的世界中，我们是如何达成信任的，其实不外乎以下两种：

- **基于共同私密信息的信任**

譬如某个陌生号码找你，说是你的老同学，生病了要找你借钱。你能够信任他的方式是向对方询问一些你们两个应该知道，且只有你们两个知道的私密信息，如果对方能够回答上来，他有可能真的是你的老同学，否则他十有八九就是个诈骗犯。

- **基于权威公证人的信任**

如果有个陌生人找你，说他是警察，让你把存款转到他们的安全账号上。你能够信任他的方式是找到公安局，如果公安局担保他确实是个警察，那他有可能真的是警察，否则他十有八九就是个诈骗犯。

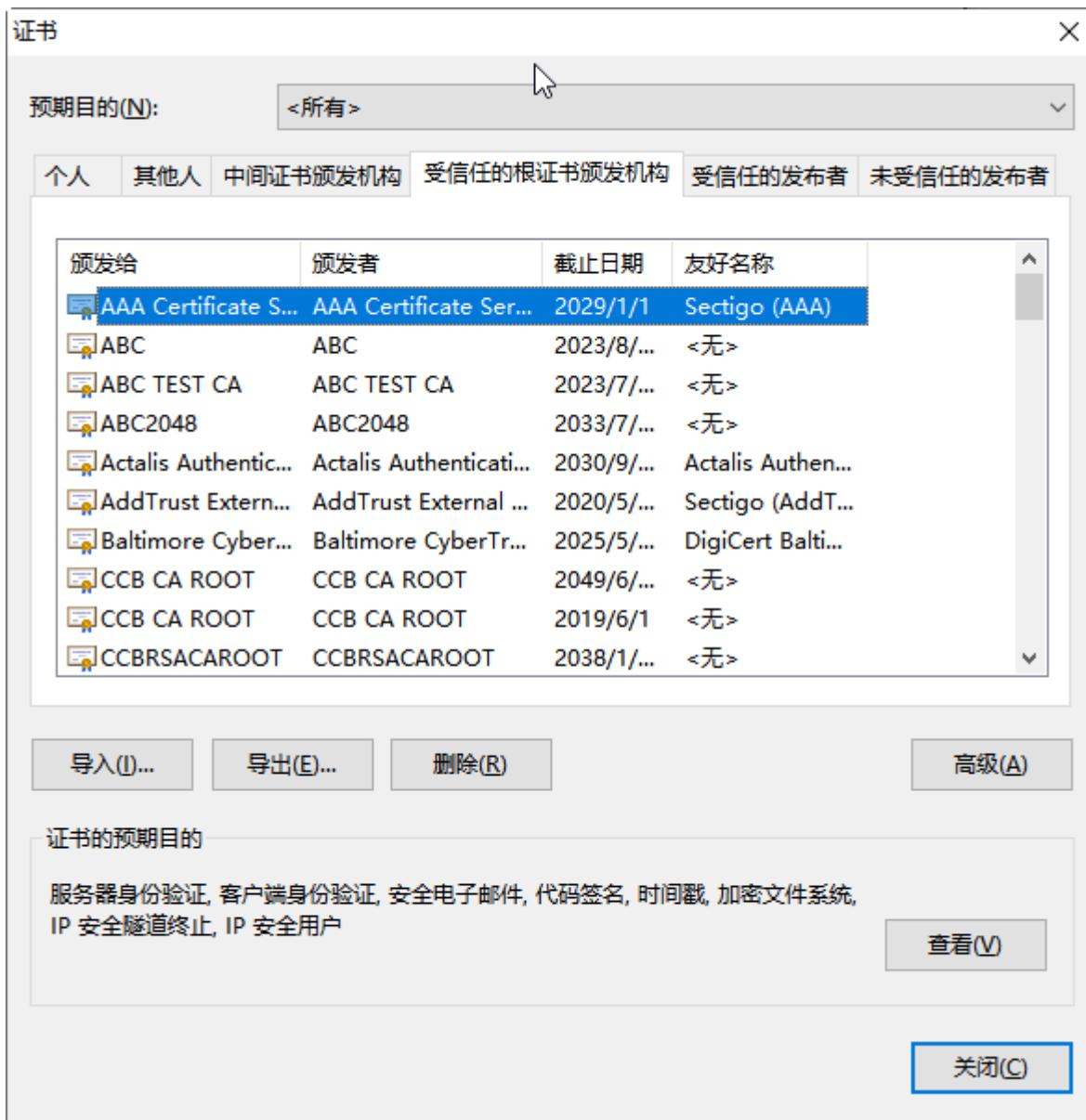
回到网络世界中，我们并不能假设授权服务器和资源服务器是互相认识的，所以通常不太会采用第一种方式，而第二种就是目前标准的保证公钥可信分发的标准，这个标准一个名字：[公开密钥基础设施](#)（ Public Key Infrastructure , PKI ）。

额外知识：公开密钥基础设施（Public Key Infrastructure , PKI）

又称公开密钥基础架构、公钥基础建设、公钥基础设施、公开密码匙基础建设或公钥基础架构，是一组由硬件、软件、参与者、管理政策与流程组成的基础架构，其目的在于创造、管理、分配、使用、存储以及撤销数字证书。

密码学上，公开密钥基础建设借着数字证书认证中心（Certificate Authority，CA）将用户的个人身份跟公开密钥链接在一起。对每个证书中心用户的身份必须是唯一的。链接关系通过注册和发布过程创建，取决于担保级别，链接关系可能由CA的各种软件或在人为监督下完成。PKI的确定链接关系的这一角色称为注册管理中心（Registration Authority，RA）。RA确保公开密钥和个人身份链接，可以防抵赖。

我们不纠缠于PKI概念上的内容，只要知道里面定义了数字证书认证中心便相当于前面例子中“权威公证人”的角色，负责发放和管理数字证书的权威机构（你也可以签发证书，不权威罢了），它作为受信任的第三方，承担公钥体系中公钥的合法性检验的责任。可是，这里和现实世界仍然有一些区别，现实世界你去找的公安局，那大楼不大可能是剧场布景冒认的；而网络世界，在假设所有网络传输都有可能被截获、冒认的前提下，“去CA中心进行认证”本身也是一种网络操作，这与之前的“去获取去公钥”本质上不是没什么差别吗？其实还是有差别的，公钥成千上万不可数，而权威的CA中心则应是可数的，“可数的”意味着可以不通过网络，而在浏览器、操作系统出厂时预置好，或者在专门安装（如银行的证书），下图为我机器上的现存的根证书。



Windows系统的CA证书

到这里终于出现了一个这节的关键词之一：证书（Certificate），证书是权威CA中心对特定公钥信息的一层公证载体，由于客户的机器上已经预置了这些权威CA中心本身的证书（称为根证书），使得我们能够在不依靠网络的前提下，使用里面的公钥信息对其所签发的证书中的签名进行确认。到此终于打破了鸡生蛋、蛋生鸡的循环，使得整套数字签名体系有了逻辑基础。

PKI中采用的证书格式是[X.509标准格式](#)，它定义了证书中应该包含哪些信息，并描述了这些信息是如何编码的，里面最关键的就是认证机构的数字签名和公钥信息两项内容。一个证书具体包含以下内容：

- 版本号（Version）**：指出该证书使用了哪种版本的X.509标准（版本1、版本2或是版本3），版本号会影响证书中的一些特定信息，目前的版本为3。

2. **序列号 (Serial Number)** : 标识证书的唯一整数 , 由证书颁发者分配的本证书的唯一标识符。
3. **签名算法标识符** : 用于签证书的算法标识 , 由对象标识符加上相关的参数组成 , 用于说明本证书所用的数字签名算法。例如 , SHA1和RSA的对象标识符就用来说明该数字签名是利用RSA对SHA1杂凑加密。
4. **认证机构的数字签名** : 这是使用证书发布者私钥生成的签名 , 以确保这个证书在发放之后没有被篡改过。
5. **认证机构** : 证书颁发者的可识别名 , 是签发该证书的实体唯一的CA的X.500名字。使用该证书意味着信任签发证书的实体 (注意 : 在某些情况下 , 比如根或顶级CA证书 , 发布者自己签发证书)。
6. **有效期限 (Validity)** : 证书起始日期和时间以及终止日期和时间 ; 指明证书在这两个时间内有效。
7. **主题信息 (Subject)** : 证书持有人唯一的标识符 (Distinguished Name) 这个名字在互联网上应该是唯一的。
8. **公钥信息 (Public-Key)** : 包括证书持有人的公钥、 算法(指明密钥属于哪种密码系统)的标识符和其他相关的密钥参数。
9. **颁发者唯一标识符 (Issuer)** : 标识符—证书颁发者的唯一标识符 , 仅在版本2和版本3中有要求 , 属于可选项。

传输安全层

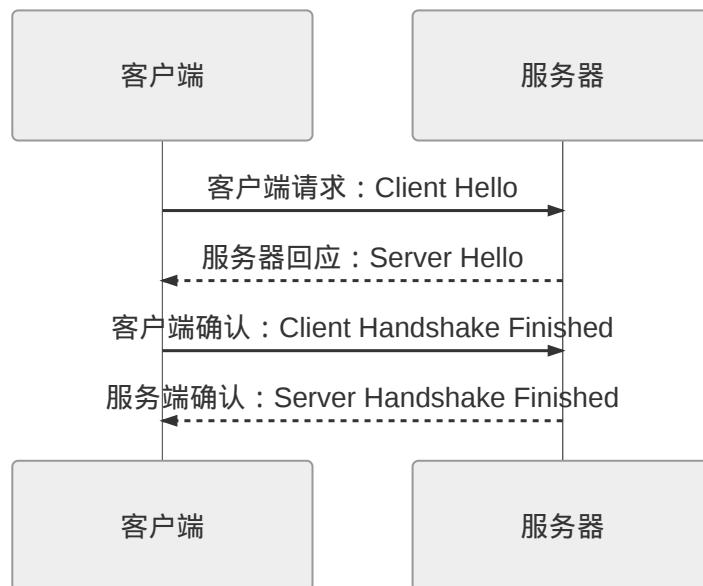
尽管到此为止 , 数字签名的安全性已经可以自洽了 , 但相信你也感受到了这套信任链的繁琐 , 如果从确定加密算法、 生成密钥、 公钥分发、 CA认证、 核验公钥、 签名、 验证签名这些步骤都要由用户来承担的话 , 这样意义的“安全”估计只能一直是存于实验室中的阳春白雪。如何把这一套繁琐的技术体系自动化地应用于无处不在的网络通讯之中 , 是这一节要讨论的话题。

在计算机科学里 , 隔离复杂性的常用手段之一就是分层 , 在网络中更是如此 , OSI模型、 TCP/IP模型将网络从物理特性 (比特流) 开始 , 逐层封装隔离 , 到了HTTP协议这种面向应用的协议里 , 就已经不会去关心网卡/交换机如何处理数据帧、 MAC地址 ; 不会去关心ARP如何做地址转换 ; 不会去关心IP寻址、 TCP传输等等 ; 那要在网络中让用户无感知地、 自动地安全通讯 , 最合理的做法就是在传输层之上、 应用层之下加入一层安全层来实现 , 这样对上层原本基于HTTP的Web应用来说 , 几乎可以是毫无感知的。而构建传输安全层这件

事情，可以说是和万维网的历史一样长，早在1994年，就已经有公司开始着手去尝试了，这里先简单回顾这将近30年来的进展：

- 1994年，网景（Netscape）公司开发了SSL协议（Secure Sockets Layer）的1.0版，这是构建传输安全层的起源，但是SSL 1.0并未正式对外发布。
- 1995年，Netscape把SSL升级到2.0版，正式对外发布，但是刚刚发布不久就被发现有严重漏洞，所以并未大规模使用。
- 1996年，修补好漏洞的SSL 3.0对外发布，这个版本得到了广泛的应用，成为Web网络安全层的事实标准。
- 1999年，互联网标准化组织接替Netscape，将SSL改名TLS（Transport Layer Security）后推进为国际标准，第一个正式的版本是[RFC 2246](#) 定义的TLS 1.0。这个版本的TLS生存时间极长，直至我写下这段文字的2020年3月，主流浏览器（Chrome、Firefox、IE、Safari）才刚刚共同宣布停止TLS 1.0/1.1的支持。而讽刺的是，由于停止后许多政府网站被无法被浏览，此时又正值新冠病毒（COVID-19）爆发期，Firefox紧急发布公告[宣布撤回该改动](#)，TLS 1.0的生命还在顽强延续。
- 2006年，TLS的第一个升级版1.1发布（[RFC 4346](#)），但却沦为了被遗忘的孩子，很少人使用TLS 1.1，甚至到了因此该版本没有已知的协议漏洞被提出的程度。
- 2008年，TLS 1.1发布2年之后，TLS 1.2标准发布（[RFC 5246](#)），迄今超过90%的互联网HTTPS流量是由TLS 1.2所支持的，现在仍在使用的浏览器几乎都完美支持了该协议。
- 2018年，最新的TLS 1.3（[RFC 8446](#)）发布，比起前面版本相对温和的升级，TLS 1.3做了出了一些激烈的改动，包括修改了从1.0起一直没有大变化的2轮4次（2-RTT）握手，首次连接仅需1轮（1-RTT）即可完成，在连接复用时甚至将TLS 1.2原本的1-RTT下降到了0-RTT，显著提升了访问速度。

下面笔者将以TLS 1.2为例，介绍传输安全层是如何保障所有信息都是第三方无法窃听（加密传输）、无法篡改（一旦篡改通讯算法会立刻发现）、无法冒充（证书验证身份）的。TLS 1.2在传输之前的握手过程一共需要进行上下2轮、4次信息发送，时序如下所示：



在介绍这四次握手具体会做什么之前，先推荐一个制作很用心的网站（<https://tls.ulfheim.net/>），上面以网页的方式详细解释了每一次握手过程中所做的事情、发送的数据、收到的响应等内容。然后，让我们开始一段相对要枯燥困难一些的握手过程：

1. 客户端请求：Client Hello

客户端（一般就是浏览器了）向服务器请求进行加密通讯，在这个请求里面，它会以明文的形式，向服务端提供以下信息：

- 支持的协议版本，譬如TLS 1.2。但是要注意，1.0-3.0分别代表SSL1.0-3.0，TLS1.0则是3.1，一直到TLS1.3的3.4。
- 一个客户端生成的32 bytes随机数，这个随机数将稍后用于产生加密的密钥。
- 一个SessionID（可选），不要和前面Cookie-Session那套混淆了，这个SessionID是只链接的SessionID，是为了TLS的链接复用。
- 一系列支持的密码学算法套件，它应该是一组算法的组合，例如TLS_RSA_WITH_AES_128_GCM_SHA256，代表着密钥交换算法是RSA，加密算法是AES128-GCM，消息认证码算法是SHA256
- 一系列支持的压缩算法。
- 其他可扩展的信息，为了保证协议的稳定，后续对协议的功能扩展大多都添加到这个变长结构中。譬如TLS 1.0中由于发送的数据并不包含服务器的域名地址，导致了一台服务器只能安装一张数字证书，这对虚拟主机来说就很不方便，所以TLS 1.1起就增加了名为“Server Name”的扩展，以便一台服务器给不同的站点安装不同的证书。

2. 服务器回应 : Server Hello

服务端收到客户单的通讯请求后，如果客户端支持的协议版本、加密算法组合在服务端中能找到一致的，将向客户端发出回应。如果不行，将会返回一个握手失败的警告提示。这次回应同样是明文的，包括以下信息：

- 服务端确认使用的协议版本。
- 第二个32 bytes的随机数，稍后用于产生加密的密钥。
- 一个SessionID，以后链接复用可以减少一轮握手。
- 服务端选定的密码学算法套件。
- 服务端选定的压缩方法。
- 其他可扩展的信息。
- 如果协商出的加密算法组合是依赖证书认证的，服务端要发送出自己的X.509证书，而证书中的公钥是什么，也必须根据协商的加密算法组合来决定。
- 密钥协商消息，这部对于不同密码学套件有不同的价值，譬如对于ECDH + anon这样得密钥协商算法组合（基于椭圆曲线的ECDH算法可以在双方通讯都公开的情况下协商出一组只有通讯双方知道的密钥）就不依赖证书中的公钥，而是通过Server Key Exchange消息协商出密钥。

3. 客户端确认 : Client Handshake Finished

由于密码学套件的组合复杂多样，这里仅以RSA算法为密钥交换算法为例介绍后续过程。客户端收到服务器应答后，先要验证服务器证书。如果证书不是可信机构颁布、或者证书中信息存在问题（域名与实际域名不一致、或者证书已经过期、或通过[在线证书状态协议](#)得知证书已被吊销，等等），都会向访问者显示一个“证书不可信任”的警告，由其选择是否还要继续通信。如果证书没有问题，客户端就会从证书中取出服务器的公钥，并向服务器发送以下信息：

- 客户端证书（可选）。部分服务端并不是面向全公众，只对特定的客户端提供服务，此时客户端需要发送它自身的证书，如果不发送，或者验证不通过，服务端可执行决定是否要继续握手，或者返回一个握手失败的信息。
- 第三个32 bytes的随机数，这个随机数不再是明文发送，而是以服务端传过来的公钥加密的，它被称为PreMasterSecret，将与前两次发送的随机数一起，根据[特定算法](#)计算出48 bytes的MasterSecret，此即为后续内容传输时的对称加密算法所采用的私钥。
- 编码改变通知，表示随后的信息都将用双方商定的加密方法和密钥发送。

- 客户端握手结束通知，表示客户端的握手阶段已经结束。这一项同时也是前面发送的所有内容的哈希值，用来供服务器校验。

4. 服务端确认：Server Handshake Finished

服务端向客户端回应最后的确认通知，包括以下信息：

- 编码改变通知，表示随后的信息都将用双方商定的加密方法和密钥发送。
- 服务器握手结束通知，表示服务器的握手阶段已经结束。这一项同时也是前面发送的所有内容的哈希值，用来供客户端校验。

至此，整个握手阶段全部结束，一个安全的链接已经建立，每一个链接建立时，客户端和服务端均通过上面的握手过程协商出了一个只有双方才知道的随机产生的密钥，以及后面传输过程中要采用的对称加密算法（如例子中的AES128），此后该链接的通讯将使用此密钥和加密算法进行加、解密。这种处理方式对上层协议的功能上完全没有影响（性能上当然有影响），建立在这层安全传输层之上的HTTP协议，就被称为“HTTP Over SSL/TLS”，即大家熟知的HTTPS。

从上面握手协商的过程中我们还可以得知，HTTPS同样并非是离散的二元选项，不是只有“启用了HTTPS”和“未启用HTTPS”的差别，采用不同的协议版本、不同的密码学套件、证书是否有效、服务端/客户端对面对无效证书时的处理策略如何都导致了不同HTTPS站点的安全强度的不同。你可以使用[亚洲诚信](#)的诊断服务查看以下几个网站的安全评分，以对安全强度有更加量化直观的理解：

- 亚洲诚信：<https://myssl.com/myssl.com>
- 腾讯网：<https://myssl.com/www.qq.com>
- 本站：<https://myssl.com/icyfenix.cn>
- 趣店：<https://myssl.com/www.quqianbao.com>

验证

验证 (Verification)

系统如何确保提交到每项服务中的数据是合乎规则的，不会对系统稳定性、数据一致性、正确性产生风险？

一般认为，数据验证不归属在安全这个话题中，但请相信我，从数量来讲，数据验证不严谨而导致的安全问题比其他安全攻击导致的要多得多；而风险上讲，由数据质量导致的问题，风险有高有低，真遇到高风险的数据问题，导致的损失不一定比被拖库什么来的小。

不过，相比起其他富有挑战性的安全措施，防御与攻击两者缠斗的精彩，数学、心理、社会工程和计算机等跨学科知识的结合运用，数据验证倒确实是不可否认地有些无聊枯燥的，这是一项非常常见的工作，在日常的开发中贯穿于代码的各个层次，每个程序员都肯定写过。以架构者的视角，这种常见的代码反而是迫切需要被架构约束的，缺失的校验影响数据质量，过度的校验不会使得系统更加健壮，某种意义上反而是垃圾代码，甚至有副作用。来看看下面这个段子：

前 端： 提交一份用户数据（姓名：某， 性别：男， 爱好：女， 签名：xxx， 手机：xxx， 邮箱：null）

控制器： 发现邮箱是空的，抛ValidationException("邮箱没填")

前 端： 已修改，重新提交

安 全： 发送验证码时发现手机号少一位，抛RemoteInvokeException("无法发送验证码")

前 端： 已修改，重新提交

服务层： 邮箱怎么有重复啊，抛BusinessRuntimeException("不允许开小号")

前 端： 已修改，重新提交

持久层： 签名字段超长了插不进去，抛SQLException("插入数据库失败，SQL : xxx")

.....

前 端： 你们这些坑爹挖不管埋的后端，各种异常都往前抛！

用 户： 这系统牙膏厂生产的？

最基础的数据问题可以在前端做表单校验来处理，但后端验证肯定是要做的，上面的段子看完了想一想，服务端应该在哪一层去做校验？可能会有这样的答案：

- 在Controller层做，在Service层不做。理由是从Service开始会有同级重用，出现Service A.foo(params)调用ServiceB.bar(params)时，相当于对params重复校验了两次。
- 在Service层做，在Controller层不做。理由是无业务含义的格式校验已在前端表单验证处理过，有业务含义的校验，放在Controller层无论如何不合适。
- 在Controller、Service层各做各的。Controller做格式校验，Service层做业务校验，就是上面那段子中的行为。
- 还有其他一些意见，譬如还有提在持久层做校验，理由是这是最终入口，把守好写入数据库的质量最重要。

上述的讨论大概是没有统一的正确结论，但是在Java里确实是有验证的标准做法，提倡的是把校验行为从分层中剥离出来，不是在哪一层做，而是在Bean上做。即Java Bean Validation。从2009年的[JSR 303](#)的1.0，到2013年的[JSR 349](#)更新的1.1，到目前最新的2017年发布的[JSR 380](#)，定义了Bean验证的全套规范。单独将验证提取、封装，可以获得不少好处：

- 对于无业务含义的格式验证，可以做到预置。
- 对于有业务含义的业务验证，可以做到重用。一个Bean适用于多个方法是非常常见的。
- 利于集中管理，譬如统一认证的异常体系，统一做国际化、统一给客户端的返回格式等等。
- 避免对输入数据的防御污染到业务代码，如果你的代码里面如果很多下面这样的条件判断，应该考虑重构

```
// 一些已执行的逻辑
if (someParam == null) {
    throw new RuntimeException("客官不可以！")
}
```

java

- 利于多个校验器统一执行，统一返回校验结果，避免用户踩地雷、挤牙膏式的试错体验。

其实，据我了解，国内的项目使用Bean Validation的还是不少的，但多数都只使用到它的Built-In Constraint，即下面这堆注解（含义我就不写了，用处基本上看类名就能明白）：

```
java  
@Null、@NotNull、@AssertTrue、@AssertFalse、@Min、@Max、@DecimalMin、  
@DecimalMax、@Negative、@NegativeOrZero、@Positive、@PositiveOrZero、  
@Szie、@Digits、@Pass、@PassOrPresent、@Future、@FutureOrPresent、  
@Pattern、@NotEmpty、@NotBlank、@Email
```

一般实现会采用Hibernate Validator，另外一个非主流选择是Apache BVal，它们都扩展了自己的私有注解。其中有一些注解，像@Email、@NotEmpty、@NotBlank，从以前Hibernate Validator私有注解，随着版本升级转正成为标准。

但是其中多数项目对Bean Validation的使用就到此为止了，带业务含义的代码都还是习惯写到方法体内，导致完全没法管理。其实这部分带有复杂逻辑的校验，才是最需要约束的，更加应该借助Bean Validation来完成。以Fenix's Bookstore的在用户资源上的两个方法为例：

```
java  
/**  
 * 创建新的用户  
 */  
@POST  
public Response createUser(@Valid @UniqueAccount Account user) {  
    return CommonResponse.op(() -> service.createAccount(user));  
}  
  
/**  
 * 更新用户信息  
 */  
@PUT  
@CacheEvict(key = "#user.username")  
public Response updateUser(@Valid @AuthenticatedAccount  
@NotConflictAccount Account user) {  
    return CommonResponse.op(() -> service.updateAccount(user));  
}
```

注意其中的三个自定义校验注解，它们的含义分别是：

- @UniqueAccount：传入的用户对象必须是唯一的，不与数据库中任何已有用户的名称、手机、邮箱产生重复。
- @AuthenticatedAccount：传入的用户对象必须与当前登陆的用户一致。
- @NotConflictAccount：传入的用户对象中的信息与其他用户是无冲突的，譬如将一个注册用户的邮箱，修改成与另外一个已存在的注册用户一致的值，这便是冲突。

这里的需求很容易想明白，注册新用户时，应约束不与任何已有用户的关键信息重复；而修改自己的信息时，只能与自己的信息重复，而且只能修改当前登陆用户的信息。这些约束规则不仅仅为这两个方法服务，它们可能会在用户资源中的其他入口被使用到，甚至在其他分层的代码中被使用到。下面是这三个自定义注解对应校验器的实现类：

```
java
public static class AuthenticatedAccountValidator extends
AccountValidation<AuthenticatedAccount> {
    public void initialize(AuthenticatedAccount constraintAnnotation) {
        predicate = c -> {
            AuthenticAccount loginUser = (AuthenticAccount)
SecurityContextHolder.getContext().getAuthentication().getPrincipal();
            return c.getId().equals(loginUser.getId());
        };
    }
}

public static class UniqueAccountValidator extends
AccountValidation<UniqueAccount> {
    public void initialize(UniqueAccount constraintAnnotation) {
        predicate = c ->
!repository.existsByUsernameOrEmailOrTelephone(c.getUsername(),
c.getEmail(), c.getTelephone());
    }
}

public static class NotConflictAccountValidator extends
AccountValidation<NotConflictAccount> {
    public void initialize(NotConflictAccount constraintAnnotation) {
        predicate = c -> {
            Collection<Account> collection =
repository.findByUsernameOrEmailOrTelephone(c.getUsername(),
c.getEmail(), c.getTelephone());
            // 将用户名、邮件、电话改成与现有完全不重复的，或者只与自己重复的，就
不算冲突
            return collection.isEmpty() || (collection.size() == 1 &&

```

```
collection.iterator().next().getId().equals(c.getId()));  
    };  
}  
}
```

这样业务校验便和业务逻辑分离开来，在需要使用时用@Valid注解自动或者通过代码手动触发执行，可根据你们公司的要求，使用于控制器、服务层、持久层等任何层次之中。此外，校验结果不满足时的提示信息，也便于统一处理，如提供默认值、提供国际化支持（这里没做）、提供统一的客户端返回格式（创建一个用于ConstraintViolationException的异常处理器），以及批量执行全部校验避免挤牙膏等诸多好处。下面是预置默认提示信息的例子：

```
java
/**
 * 表示一个用户的信息是无冲突的
 *
 * “无冲突”是指该用户的敏感信息与其他用户不重合，譬如将一个注册用户的邮箱，修改成
与另外一个已存在的注册用户一致的值，这便是冲突
 */
@Documented
@Retention(RUNTIME)
@Target({FIELD, METHOD, PARAMETER, TYPE})
@Constraint(validatedBy =
AccountValidation.NotConflictAccountValidator.class)
public @interface NotConflictAccount {
    String message() default "用户名、邮箱、手机号码与现存用户产生重复";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

另外一条建议是将不带业务含义的格式校验注放到类上，将带业务含义的注放到外面。譬如用户账号实体中的部分代码为：

```
private String avatar;

@Pattern(regexp = "1\\d{10}", message = "手机号格式不正确")
private String telephone;

@email(message = "邮箱格式不正确")
private String email;
}
```

把校验注解放在类定义中，意味着所有执行校验的时候它们都会被运行（譬如Insert、Update的时候，Hibernate都会自动执行DO上的校验注解）。而不带业务含义的注解运行是不需要其他外部资源（譬如数据库）参与的，这种重复执行通常并无坏处（系统的压力往往不在CPU，闲着也是闲着）。

如果真的遇到一些非典型情况，譬如“新增”操作A需要执行全部校验规则，“修改”操作B中希望不校验某个字段，“删除”操作C中希望改变某一条校验规则，这时候要就要启用分组校验来处理，设计一套“新增”、“修改”、“删除”这样的标识类，置入到校验注解的groups参数中。

分布式共识算法

前置知识

关于分布式中CAP问题，请先阅读“[分布式事务](#)”中的介绍，后文中提及的一致性、可用性、网络分区等概念，均在此文中有过介绍。

在本章正式开始探讨各种分布式环境中面临的技术问题和解决方案之前，笔者先安排一篇“纯理论”的文章，来分析分布式环境中对共享数据操作的本质。分布式系统里，如果准备在各个分布式节点中进行一致的操作，并且期望获得一致的结果，均可以理解为是一种“[状态机复制](#)”（State Machine Replication）过程，无论这个操作是新增、修改、删除抑或是其他可能的程序行为，都可以理解为要将一连串的操作日志正确地复制到各个分布式节点上，如果分布式系统各个节点的初始状态一致，接受到的操作序列都相同，那各个节点最终都能得到一致的状态。这句话听起来颇为抽象，如果你现在暂时不能理解的话，不妨先在心中回想一下经典数据库中的重做和回滚日志的做法，然后跟后续的讲解进行类比。

额外知识：状态机复制

[状态机](#)（State Machine）有一个特性：任何初始状态一样的状态机，如果执行的命令序列一样，则最终达到的状态也一样。如果将此特性应用在多参与者进行协商共识上，可以理解为系统中存在多个具有完全相同的状态机（参与者），这些状态机能最终保持一致的关键就是起始状态完全一致和执行命令序列完全一致。

为了解释清楚分布式环境中共享数据所面临的问题，笔者先从一个最浅显的场景开始说起：

如果你有一份很重要的数据，要确保它长期存储在电脑上不会丢失，你会怎么做？

这不是什么脑筋急转弯的古怪问题，答案就是去买几块硬盘，把数据在不同磁盘上多备份几个副本。假设一块硬盘每年损坏的概率是5%，那把文件复制到另一块备份盘上，由于两

块磁盘同时损坏而丢失数据的概率就只有0.25%，如果使用三块硬盘存储则是0.00125%，四块是0.0000625%，换而言之，这已经保证了数据超过99.9999%的概率是不会丢失的。

在软件系统里，要保障系统的可靠性，采用的办法也和那几个备份磁盘大体上并无区别，单个节点的系统宕机无法提供服务的原因可能有很多，譬如程序出错、硬件损坏、网络分区、电源断电，等等，往往一年中出现系统宕机的概率要远高于5%，这更加决定了软件系统也必须有多台机器能够拥有一致的数据，才能对外提供一致的服务。但分布式的软件系统与备份磁盘又有着本质的区别，磁盘之间是孤立的，不需要互相通讯，备份数据初始化后状态就是不变的，由人工完成的文件复制操作保障了数据各个副本的一致；而分布式系统里面，我们必须考虑数据如何在可能出现分区的网络环境下在各个节点之间正确复制的问题：

如果你有一份很重要的数据，要确保它正确地存储于网络中的几台不同机器之上，你会怎么做？

一个最容易想到的答案是“同步”（Synchronous）：每当数据有变化，把变化情况在各个节点间的复制视作一种原子性的操作，只有系统里每一台机器都反馈成功地完成磁盘写入后，数据的变化才宣告成功，我们在[前文中](#)曾经讲解过，可以使用2PC/3PC来实现这种同步操作。同步的其中一种真实应用场景是数据库中的主从全同步复制（Fully Synchronous Replication），譬如MySQL Cluster，进行全同步复制时，会等待所有Slave的Binlog都完成写入后，Master的事务才进行提交。这里有一个显而易见的问题，尽管可以确保Master和Slave中的数据是绝对一致的，但任何一个Slave节点因为任何原因未响应均都会阻塞整个事务，每增加一个Slave，都导致造成整个系统可用性风险增加一分。显然这种简单的一致性保障手段，是以完全牺牲可用性为代价的，我们在建设分布式系统的时候，往往不能承受这种代价，一些关键系统，在要求数据正确的前提下，对可用性的要求也非常苛刻，譬如要达到99.99999%可用的程度，这就引出了我们的第三个问题：

如果你有一份很重要的数据，要确保它正确地存储于网络中的几台不同机器之上，并且要尽可能保证及时地应用到正确的数据，你会怎么做？

在网络分区不可能消除的前提下，一致性与可用性的矛盾造成了增加机器数量反而带来可用性的降低，为缓解这个矛盾，我们不再追求系统内所有节点在任何情况下的数据状态都一致，改为采用“少数服从多数”的原则，一旦数据变化在系统中过半数的节点中完成了复制，就认为数据的变化已经正确地存储在系统当中，这样就可以容忍少数（不超过半数）的机器形成网络分区，使得增加机器数量对系统整体的可用性变成是有益的。此模式在分

布式中被称为“Quorum机制”[□](#)，或者可以直接形象地称其为“多数派机制”（ Majority ）。在这个前提下，我们需要设计一种算法，能够让分布式系统内部可以暂时容忍存在不同的状态，但最终全部节点的状态均能够达成一致；同时能够让分布式系统外部看来，始终表现出整体一致的结果，这个过程，我们称其为“协商共识”（ Consensus ）。

请注意共识与一致性（ Consistency ）的区别，一致性指的是数据不同副本之间的差异，而共识是指达成一致性的方法与过程。由于翻译的关系，很多中文资料把 Consensus 同样翻译为“一致性”，导致网络上大量的二手中文资料把这两个概念混淆起来，如果你在网上看到“分布式一致性算法”，应明白其所指其实是“Distributed Consensus Algorithm”。

Paxos

Distributed Consensus Algorithm

There is only one consensus protocol, and that's "Paxos" — all other approaches are just broken versions of Paxos

世界上只有一种共识协议，就是Paxos，其他所有共识算法都是Paxos的退化版本。

—— Mike Burrows [↗](#) , Inventor of Google Chubby

Paxos是由[Leslie Lamport](#) [↗](#)（就是大名鼎鼎的[LaTeX](#) [↗](#)中的“La”）提出的一种基于消息传递的协商共识算法，现已是当今分布式系统最重要的理论基础，几乎就是“共识”二字的代名词（这句话是Raft作者在论文中说的）。尽管不像Mike Burrows说的“世界上只有Paxos一种分布式共识算法”那么夸张，但是如果沒有Paxos，那后续的Raft、ZAB算法，ZooKeeper、Etcd这些分布式协调框架、Hadoop、Consul等在此基础上的各类分布式应用都很可能会延后几年面世。

Lamport虚构了一个名为“Paxos”的希腊城邦，这个城邦按照民主制度制定法律，却又不存在一个中心化的专职立法机构，立法靠着“兼职议会”（Part-Time Parliament）来完成，无法保证所有城邦居民都能够及时地了解新的法律提案、也无法保证居民会及时为提案投票。Paxos算法的目标就是让城邦能够在每一位居民都无法承诺一定会及时参与的情况下，依然可以按照少数服从多数的原则，最终达成一致意见（但是并不考虑[拜占庭将军问题](#) [↗](#)，即假设信息可能丢失也可能延迟，但不会被错误传递）。

Lamport最初在1990年首次发表了Paxos算法，选的题目就是“The Part-Time Parliament”[↗](#)。由于算法本身较为复杂，用希腊城邦作为比喻反而使得描述更为晦涩，论文的三个审稿人一致要求他把希腊城邦的故事删除掉，这令Lamport感觉颇为不爽，然后干脆就撤稿不发了，所以Paxos刚刚被提出的时候并没有引起什么反响。八年之后（1998年），Lamport再次将此文章重新整理后投到《ACM Transactions on Computer Systems》[↗](#)，这次论文

成功发表，吸引了一些人去研究，结果是并没有什么人能弄懂。时间又过去了三年（2001年），Lamport认为前两次是同行们无法理解他以“希腊城邦”来讲故事的幽默感，第三次以“[Paxos Made Simple](#)”为题，在《[SIGACT News](#)》杂志上发表文章，终于放弃了“希腊城邦”的比喻，尽可能用（他认为）简单直接、（他认为）可读性较强的方式去介绍Paxos算法，情况虽然比前两次要好，但以Paxos本应获得的重视程度来说，这次依然只能算是应者寥寥。这段跟网络段子一般的经历被Lamport以自嘲的形式放到了[他自己的个人网站](#)上。尽管我们作为后辈应该尊重Lamport老爷子，但当笔者翻开“Paxos Made Simple”读到只有一句话的“摘要”时，心里实在是不得不怀疑Lamport这样写论文是不是在恶搞审稿人和读者，在嘲讽“你们这些愚蠢的人类！”。

Abstract

The Paxos algorithm, when presented in plain English, is very simple.

Paxos Made Simple ↗

虽然Lamport本人连发三篇文章都没能让大多数同行理解Paxos，但是到了2006年，Google的Chubby、Megastore以及Spanner等分布式系统都使用Paxos解决了分布式共识的问题，并将其整理成正式的论文发表之后，得益于Google的行业影响力，辅以Chubby作者Mike Burrows那略显夸张但足够吸引眼球的评价推波助澜，致使Paxos一夜间成为计算机科学分布式这条分支中最炙手可热网红概念，从这时起被学术界众人争相研究。2013年，La

import本人因其对分布式系统的杰出理论贡献获得了2013年的图灵奖，足可见技术圈里即使再有本事，也还是需要好好包装一下的。

讲完段子吃过西瓜，希望你没有被这些对Paxos的“困难”做的铺垫所吓倒，反正又不让你去实现它，假如放弃些许严谨性，并简化分支细节和特殊情况的话，Paxos是完全可能去通俗地理解的，Lamport在论文中也只用两段话就描述“清楚”了它的工作流程，下面我们正式来学习Paxos算法（在本节中均特指Basic Paxos算法）。Paxos算法将分布式系统中的节点分为三类：

- **提案节点**（称为Proposer）：提出对某个值进行设置操作的节点，设置值这个行为就被称为“提案”（Proposal）。请注意，这里的“设置值”不要类比成程序中变量赋值操作，应该类比成日志记录操作，值一旦设置成功，就是不丢失、不可变的，在后面介绍的Raft算法中就索性直接把“提案”叫做“日志”了。
- **决策节点**（称为Acceptor）：应答提案节点该提案是否可被投票、是否可被接受。提案一旦得到过半数决策节点的接受，即称该提案被批准，提案被批准即意味着该值不能再被更改，也不会丢失，且最终所有节点都会接受该它。
- **记录节点**（被称为Learner）：不参与提案，也不参与决策，只是单纯地从提案、决策节点中学习已经达成一致的提案，譬如少数派节点从网络分区中恢复时，将会进入这种状态。

使用Paxos的分布式系统里的，所有的节点都是平等的，它们都可以承担以上某一种或者多种的角色，不过为了便于确保有明确的多数派，决策节点的数量应该被设定为奇数个，且在初始化时，网络中每个节点都知道整个网络所有决策节点的数量、地址等信息。

分布式环境下，我们如果说各个节点“就某个值（提案）达成一致”，所指的意思是“不存在某个时刻有一个值为A，另一个时刻该值又为B的情景”。解决这个问题的复杂度主要来源于以下两个方面因素的共同作用：

- 系统内部各个节点通讯是不可靠的，不论对于系统中企图设置数据的提案节点抑或是决定是否批准设置行为的决策节点，其发出、收到的信息可能延迟送达、也可能丢失，但不去考虑消息传递错误的情况。
- 系统外部各个用户访问是可并发的，如果系统只会有一个用户，如果每次只对系统进行串行访问，那单纯的应用Quorum机制已经足以保证值被正确地读写。

第一点是网络通讯中客观存在的现象，也是我们要重点解决的问题，而第二点其实也很好理解：现在我们讨论的是“分布式环境下操作的共享数据”的问题，而即使是在非分布式的环境下，有一个变量i当前在系统中存储的数值为2，如果同时有外部请求A、B分别对系统发送指令“把i的值加1”和“把i的值乘3”，如果不加任何并发控制的话，将可能得到“(2+1)*3=9”与“2*3+1=7”两种可能的结果。为此，对同一个变量的修改请求必须先“加锁”（实际操作上并不是并发控制中互斥量的加锁），不能让A、B的请求被交替处理。在分布式的环境下，由于还要同时考虑到分布式系统内可能在任何时刻出现的通讯问题，如果一个节点在取得同步锁之后，在释放锁之前发生失联，这将导致无限期的阻塞等待，因此还必须提供一个其他节点能抢占锁的机制，以避免死锁。

Paxos算法包括两个阶段，其中第一阶段“准备”（Prepare）就相当于上面抢占锁的过程。如果某个提案节点准备发起提案，必须先向所有的决策节点广播一个许可申请（称为Prepare请求）。提案节点的Prepare请求中会附带一个全局唯一的数字n作为提案ID，决策节点收到后，将会给予提案节点两个承诺和一个应答。

两个承诺是：

- 承诺不会再接受提案ID小于或等于n的Prepare请求。
- 承诺不会再接受提案ID小于n的Accept请求。

一个应答是：

- 不违背以前作出的承诺的前提下，回复已经批准过的提案中提案ID最大的那个提案所设定的值和提案ID，如果该值从来没有被任何提案设定过，则返回空值。如果违反此前做出的承诺，即收到的提案ID并不是决策节点收到过的最大的，那可以直接对此Prepare请求不予理会。

当提案节点收到了多数派决策节点的应答（称为Promise应答）后，可以开始第二阶段“批准”（Accept）过程，这时有如下两种可能：

- 如果提案节点发现所有响应的决策节点此前都没有批准过该值（即为空），那说明它是第一个设置值的节点，可以随意地决定要设定的值，将自己选定的值与提案ID，构成一个二元组“(n, value)”，再次广播给全部的决策节点（称为Accept请求）。
- 如果提案节点发现响应的决策节点中，已经有至少一个节点的应答中包含有值了，那它就不能够随意取值了，必须无条件地从应答中找出提案ID最大的那个值并接受，构成一个二元组“(n, maxAcceptValue)”，再次广播给全部的决策节点（称为Accept请求）。

当每一个决策节点收到Accept请求时，都会在不违背以前作出的承诺的前提下，接收并持久化对当前提案ID和提案附带的值。如果违反此前做出的承诺，即收到的提案ID并不是决策节点收到过的最大的，那可以直接对此Accept请求不予理会。

当提案节点收到了多数派决策节点的应答（称为Accepted应答）后，协商结束，共识决议形成，将形成的决议发送给所有记录节点进行学习。整个过程的时序如下图所示：

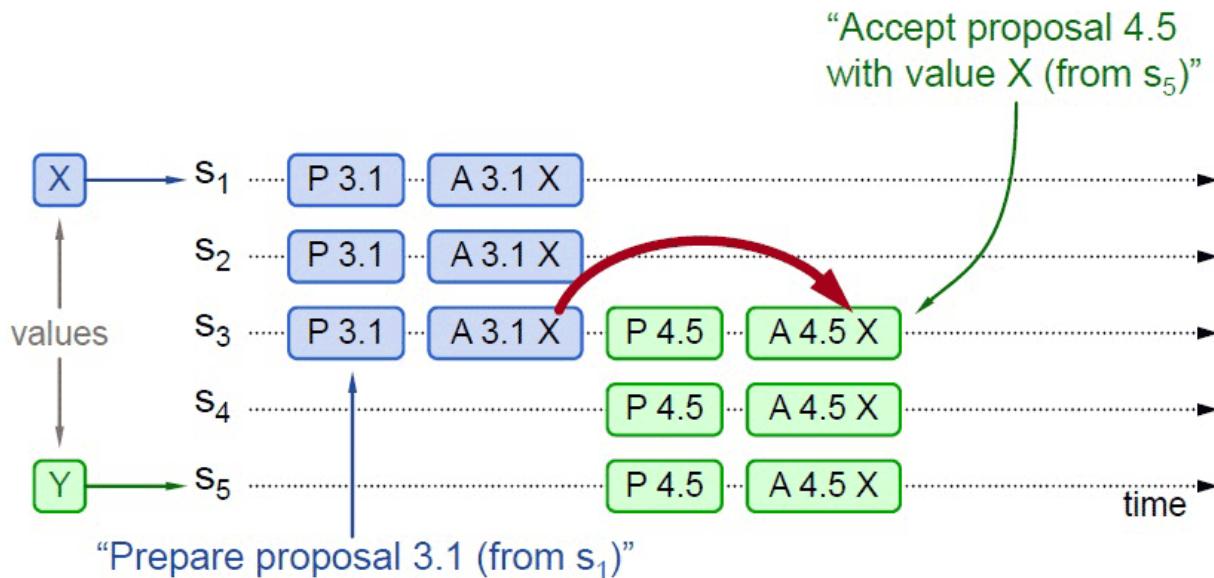


整个Paxos算法的工作流程确实并不复杂，如果你此前并未专门学习过分布式的知识，相信阅读到这里，不一定会对操作过程有疑惑，但估计还是不能对Paxos究竟是如何解决协商共识的形成具体的概念的，下面笔者将举一个具体的例子来讲解，例子与截图来源于《Implementing Replicated Logs with Paxos》[\[1\]](#)，在此统一注明，后面就不单独列出了。

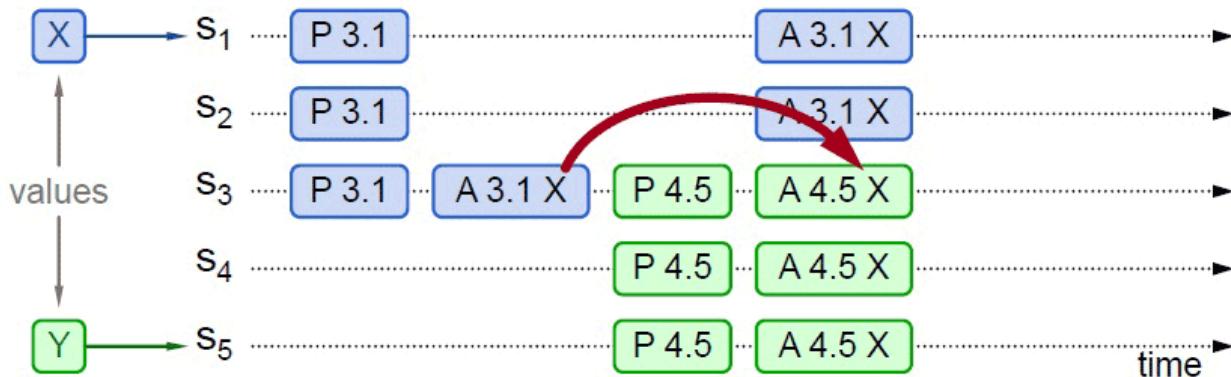
假设一个分布式系统有五个节点，分别命名为S₁、S₂、S₃、S₄、S₅，只讨论正常场景，不会涉及到网络分区。全部节点都同时扮演着提案节点和决策节点的身份。此时，有两个并发的请求分别希望将同一个值分别设定为X（由S₁作为提案节点提出）和Y（由S₅作为提案节点提出），以P代表准备阶段，以A代表批准阶段，这时候可能发生以下情况：

- 情况一：譬如，S₁选定的提案ID是3.1（全局唯一ID加上节点编号），先取得了多数派决策节点的Promise和Accepted应答，此时S₅选定提案ID是4.5，发起Prepare请求，收到的多数派应答中至少会包含1个此前应答过S₁的决策节点，假设是S₃，那么S₃提供的

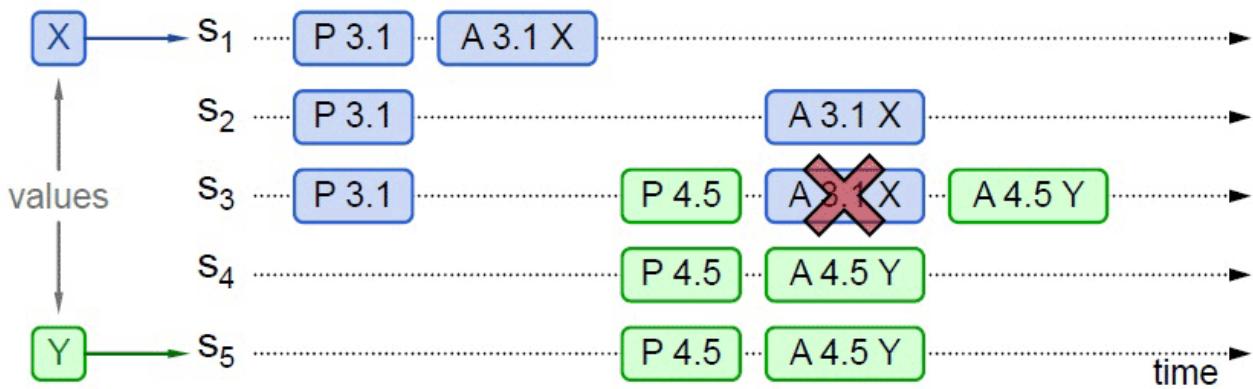
Promise中必将包含S₁已设定好的值X，S₅就必须无条件地用X代替Y作为自己提案的值，由此整个系统对“取值为X”这个事实达成了一致。如下图所示：



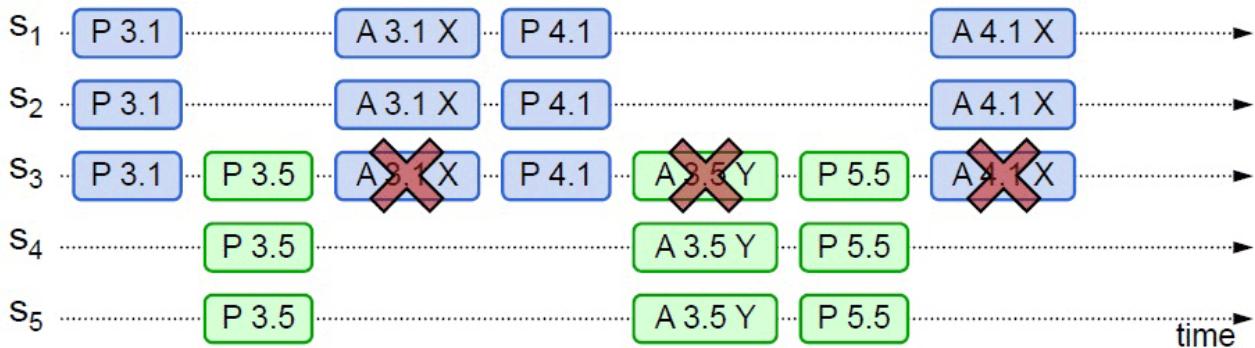
- 情况二：事实上，对于情况一，X被选定为最终结果是必然结果，但从图中可以看出，X被选定为最终值并不是必定需要多数派的共同批准，只取决于S₅提案时Promise应答中是否已包含了批准过X的决策节点，譬如下图所示，S₅发起提案的Prepare请求时，X并未获得多数派批准，但由于S₃已经批准的关系，最终共识的结果仍然是X。



- 情况三：当然，另外一种可能的结果是S₅提案时Promise应答中并未包含批准过X的决策节点，譬如应答S₅提案时，节点S₁已经批准了X，节点S₂、S₃未批准但返回了Promise应答，此时S₅以更大的提案ID获得了S₃、S₄、S₅的Promise，这三个节点均未批准过任何值，那么S₃将不会再接受来自S₁的Accept请求，因为它的提案ID已经不是最大的了，这三个节点将批准Y的取值，整个系统最终会对“取值为Y”达成一致。



- 情况四：从以上情况三可以推导出另一种极端的情况，如果两个提案节点交替使用更大的提案ID使得准备阶段成功，但是批准阶段失败的话，这个过程理论上可以无限持续下去，形成活锁（Livelock）。在算法实现中会引入随机超时时间来避免活锁的产生。



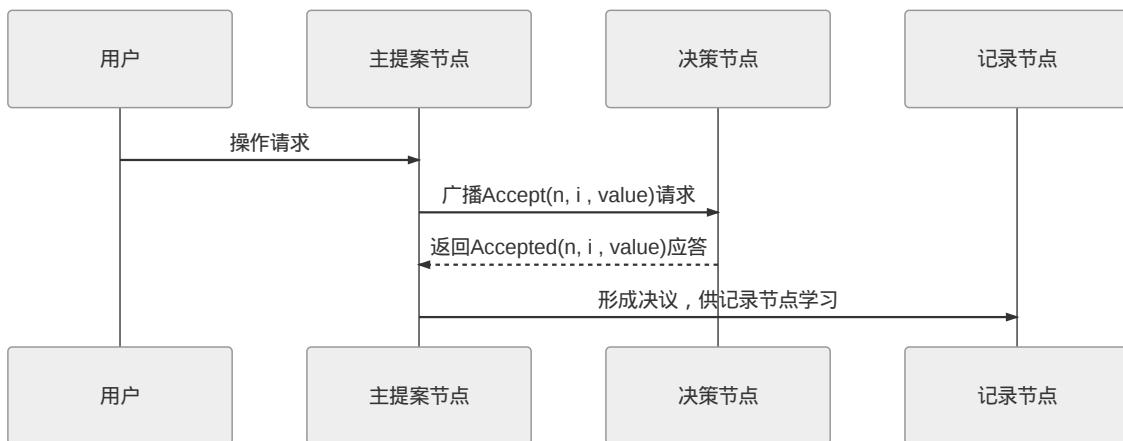
以上介绍是基于Basic Paxos、以正常流程（未出现网络分区等异常）、以通俗的方式介绍的Paxos算法，并未涉及到严谨的逻辑和数学原理，也未讨论Paxos的推导证明过程，对于普通的技术人员，理解起来应该并不算困难的。

本节介绍的Basic Paxos只能对单个值形成决议，并且决议的形成至少需要两次网络请求和应答（准备和批准阶段各一次），高并发情况下将产生较大的网络开销，极端情况下甚至可能形成活锁。总之，Basic Paxos是一种很学术化但对工程并不友好的算法，现在几乎只用来做理论研究，并不直接应用在实际软件研发当中。实际的应用都是基于Multi Paxos和Fast Paxos算法的，接下来我们将会了解Multi Paxos与它的理论等价算法Raft和ZAB算法。

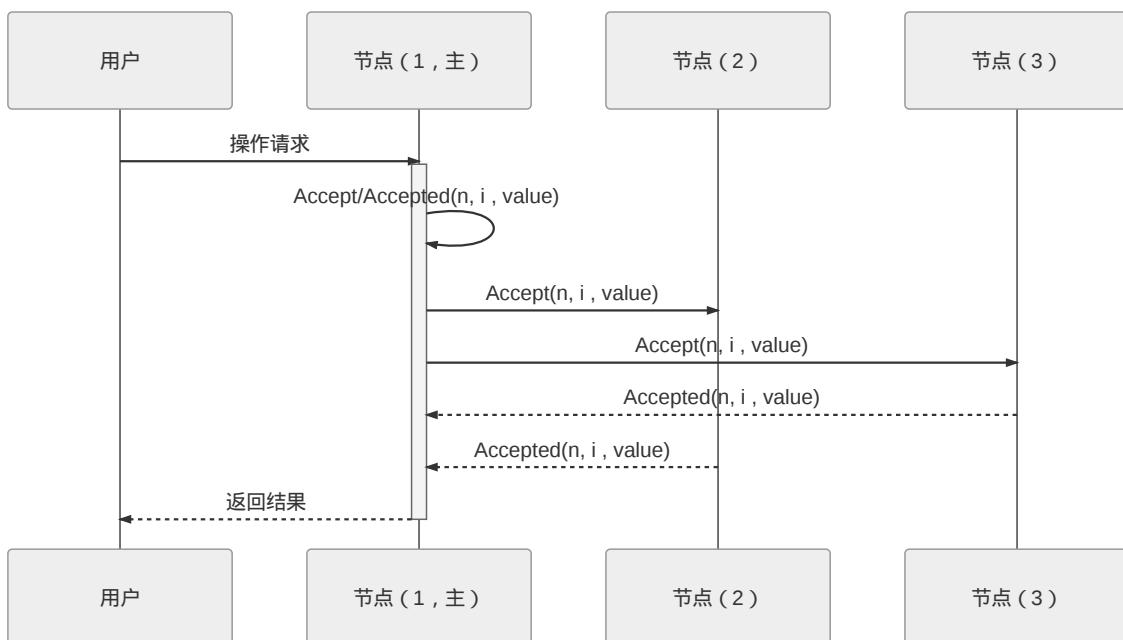
Multi Paxos与Raft

上一节的最后，笔者举例介绍了Basic Paxos的活锁问题，两个提案节点互不相让地争相提出自己的提案，抢占同一个值的修改权限，导致整个系统在持续性地“反复横跳”，外部看起来就像被锁住了一样。而在上一节的开头，笔者还陈述过一个观点，分布式共识的复杂性，主要来源于网络的不可靠与请求的可并发两大因素，活锁问题与许多Basic Paxos异常场景中所遭遇的麻烦，都可以看作是由于任何一个提案节点都能够完全平等地、与其他节点并发地提出提案而带来的复杂问题。为此，专门有一种Paxos的主要改进版本“Multi Paxos”算法被设计（设计的意思：在Lamport的论文中随意提了几句可以这么做）出来，希望能够找到一种两全其美的办法，既不破坏Paxos中“众节点平等”的原则，又能在提案节点中实现主次之分，限制每个节点都有不受控的提案权利，这两个目标听起来似乎是矛盾的，但现实世界中的选举，就很符合这种在平等节点中挑选意见领袖的场景。

Multi Paxos对Basic Paxos的关键改进是有了“选主”的过程，提案节点会通过定时轮询（心跳），确定当前网络中的所有节点里是否存在有一个主提案节点，一旦没有发现主节点存在，节点就会在心跳超时后使用Basic Paxos中定义的准备、批准的两轮网络交互过程，向所有其他节点广播自己希望竞选主节点的请求，希望整个分布式系统对“由我作为主节点”这件事情协商达成一致共识，如果得到了决策节点中多数派的批准，便宣告竞选成功。当选主完成之后，除非主节点失联之后发起重新竞选，否则从此往后，就只有主节点本身才能够提出提案。此时，无论哪个提案节点接收到客户端的操作请求，都会将请求转发给主节点来完成提案，而主节点提案的时候，也就无需再次经过准备过程，因为可以视作是经过选举时的那一次准备之后，后续的提案都是对相同提案ID的一连串的批准过程。也可以通俗理解为选主过后，就不会再有其他节点与它竞争，相当于是处于无并发的环境当中进行的有序操作，所以此时系统中要对某个值达成一致，只需要进行一次批准的交互即可，具体如下序列所示：



可能有人注意到这时候的二元组(n, value)已经变成了三元组(n, i, value)，这是因为需要为主节点增加一个“任期编号”，这个编号必须是严格单调递增的，以应付主节点陷入网络分区后重新恢复，但另外一部分节点仍然有多数派，且已经完成了重新选主的情况，此时必须以任期编号大的主节点为准。当节点有了选主机制的支持，在整体来看，就可以进一步简化节点角色，不去区分提案、决策和记录节点了，统统以“节点”来代替，节点只有主 (Leader) 和从 (Follower) 的区别，此时协商共识的时序如下：



在这个理解的基础上，我们换一个角度来重新思考“分布式系统中如何对某个值达成一致”这个问题，可以把该问题划分做三个子问题来考虑，可以证明（具体证明就不写了，参考文末的论文）当以下三个问题同时被解决时，即等价于达成共识：

- 如何选主 (Leader Election)
- 如何把数据复制到各个节点上 (Entity Replication)
- 如何保证过程是安全的 (Safety)

选主问题尽管还涉及到许多工程上的细节，譬如心跳、随机超时、并行竞选，等等，但要只论原理的话，如果你已经理解了Paxos算法的介绍，相信对选主并不会有什么疑惑，因为这本质上仅仅是分布式系统对“谁来当主节点”这件事情的达成共识而已，我们在前一节已经花了几千字来讲述分布式系统该如何对一件事情达成共识，这里就不继续展开了，下面直接介绍数据（Paxos中的提案、Raft中的日志）在网络各节点间的复制问题。

在正常情况下，当客户端向主节点发起一个操作，譬如提出“将某个值设置为X”，此时主节点将X写入自己的变更日志，但先不提交，接着把变更X的信息在下一次心跳包中广播给所有的从节点，并要求从节点回复确认收到的消息，从节点收到信息后，将操作写入自己的变更日志，然后给主节点发送确认签收的消息，主节点收到过半数的签收消息后，提交自己的变更、应答客户端并且给从节点广播可以提交的消息，从节点收到提交消息后提交自己得变更，数据在节点间的复制宣告完成。

在异常情况下，网络出现了分区，部分节点失联，但只要仍能正常工作的节点的数量能够满足多数派，分布式系统就仍然可以正常工作，这时候数据复制过程如下：

- 假设有 S_1 、 S_2 、 S_3 、 S_4 、 S_5 五个节点， S_1 是主节点，由于网络故障，导致 S_1 、 S_2 和 S_3 、 S_4 、 S_5 之间彼此无法通讯，形成网络分区。
- 一段时间后， S_3 、 S_4 、 S_5 三个节点中的某一个（譬如是 S_3 ）最先达到心跳超时的阈值，获知当前分区中已经不存在主节点了，它向所有节点发出自己要竞选的广播，并收到了 S_4 、 S_5 节点的批准响应，加上自己一共三票，即得到了多数派的批准，竞选成功，此时系统中同时存在 S_1 和 S_3 两个主节点，但由于网络分区，它们不会知道对方的存在。
- 这种情况下，客户端发起操作请求：
 - 如果客户端连接到了 S_1 、 S_2 之一，都将由 S_1 处理，但由于操作只能获得最多两个节点的响应，不构成多数派的批准，所以任何变更都无法成功提交。
 - 如果客户端连接到了 S_3 、 S_4 、 S_5 之一，都将由 S_3 处理，此时操作可以获得最多三个节点的响应，构成多数派的批准，是有效的，变更可以被提交，即系统可以继续提供服务。
 - 事实上，以上两种“如果”情景很少机会能够并存。网络分区是由于软、硬件或者网络故障而导致的，内部网络出现了分区，但两个分区仍然能分别与外部网络的客户端正常通讯的情况甚为少见。通常网络中下线了一部分节点，按照这个例子来说，如果下

线了两个节点，系统正常工作，下线了三个节点，那剩余的两个节点也不可能继续提供服务了。

- 假设现在故障恢复，分区解除，五个节点重新可以通讯了：

- S_1 和 S_3 都向所有节点发送心跳包，从各自的心跳中可以得知两个主节点里 S_3 的任期编号更大，它是最新的，此时五个节点均只承认 S_3 是唯一的主节点。
- S_1 、 S_2 回滚它们所有未被提交的变更。
- S_1 、 S_2 从主节点发送的心跳包中获得它们失联期间发生的所有变更，将变更提交写入本地磁盘。
- 此时分布式系统各节点的状态达成最终一致。

下面我们来看第三个问题：“如何保证过程是安全的”，你是否感受到这个问题与前两点的存在一点差异？选主、数据复制都是很具体的行为，但是“安全”就很模糊，什么算是安全或者不安全？

在分布式理论中，Safety 和 Liveness 两种属性是有预定义的，在专业的书籍中一般翻译成“协定性”和“终止性”，它们的概念也是由Lamport最先提出，当时给出的定义是：

- 协定性（Safety）：所有的坏事都不会发生（something “bad” will **never** happen）
- 终止性（Liveness）：所有的好事都终将发生，但不知道是啥时候（something “good” will **must** happen, but we don't know when）

这种就算解释了你也看不明白的定义，是不是很符合Lamport老爷子一贯的写作风格？

（笔者无奈地摊摊手），我们不纠结严谨的定义，仍通过举例来说明，譬如以选主问题为例，Safety 保证了选主的结果一定是有且只有一个主节点，不可能同时出现两个主节点；而 Liveness 则要保证选主过程是一定可以在某个时刻能够结束的。由前面对活锁的介绍可以得知，在 Liveness 这个属性上选主问题是存在理论上的瑕疵的，可能会由于活锁而导致一直无法选出明确的主节点，所以Raft论文中只写了对 Safety 的保证，但由于工程实现上的处理，现实中是几乎不可能会出现终止性的问题。

最后，以上这种把共识问题分解为“Leader Election”、“Entity Replication”和“Safety”三个问题来思考、解决的解题思路，即是本节主题中的“Raft算法”，这篇以“一种可以让人理解的共识算法”（In Search of an Understandable Consensus Algorithm，Lamport：好像有人在论文标题中对我有意见？）为题的论文提出了Raft算法，获得了USENIX ATC 2014的Best Paper，后来更是成为了日后Etcd、LogCabin、Consul等重要分布式程序的实现基

础，ZooKeeper的ZAB算法与Raft的思路也非常类似，这些算法都被认为是与Multi Paxos的等价派生实现。

Gossip协议

Gossip

Trying to squash a rumor is like trying to unring a bell.

—— Shana Alexander↗，American Journalist

Paxos、Raft、ZAB这些分布式算法经常会被称作是“强一致性”的分布式共识协议，这样的描述扣细节概念的话是很别扭的，有语病嫌疑，但我们都明白它的意思其实是在说“尽管系统内部节点可以存在不一致的状态，内部是最终一致的，但从系统外部看来，不一致的情况并不会被观察到，所以整体上看系统是强一致性的”。与它们相对的，还有一类被冠以“最终一致性”的分布式共识协议，这表明系统中不一致的状态有可能能够在一定时间内被外部观察到。一种典型而又极为常见的最终一致的分布式系统就是DNS系统，在各节点缓存的TTL到期之前，都有可能与真实的域名翻译结果存在不一致，在本节中，我们将介绍在比特币网络和许多重要分布式框架中都有应用的另一种具有代表性的“最终一致性”的分布式共识协议：Gossip协议。

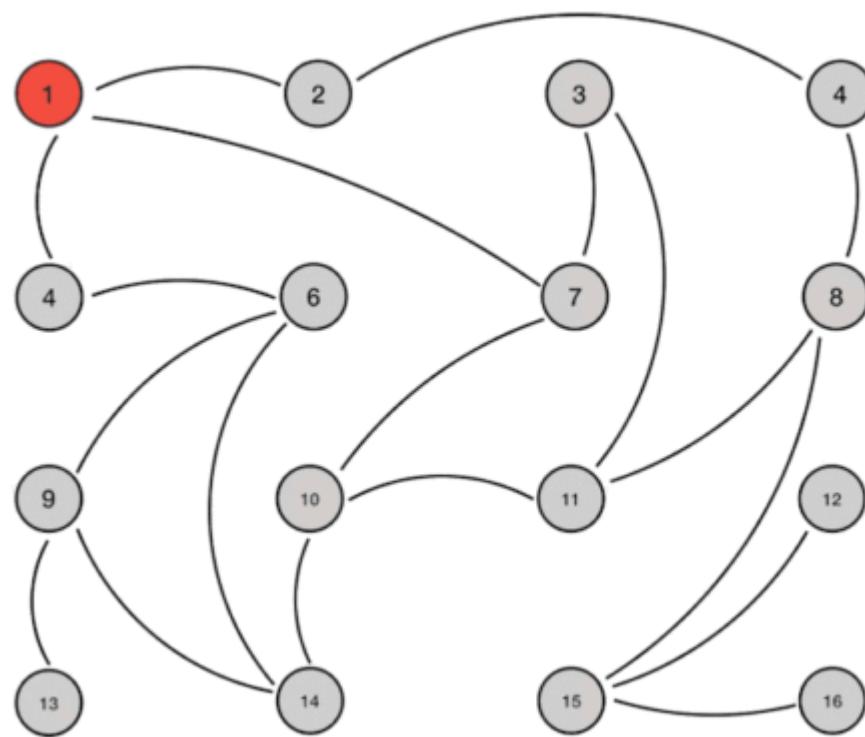
Gossip最早由施乐公司↗（Xerox，现在可能很多人不了解施乐了，或只把施乐当一家复印产品公司看待，这家公司是计算机许多关键技术的鼻祖，图形界面的发明者、以太网的发明者、激光打印机的发明者、MVC架构的提出者、RPC的提出者、BMP格式的提出者……）Palo Alto研究中心在论文《Epidemic Algorithms for Replicated Database Maintenance↗》中提出的一种用于分布式数据库在多节点间复制同步数据的算法。从论文题目中可以看出，最初它是被称作“流行病算法”（Epidemic Algorithm）的，而今天Gossip这个名字使用得更为普遍。除此以外，它还有“流言算法”、“八卦算法”、“瘟疫算法”等别名，这些名字都是很形象化的描述，反应了Gossip的特点：要同步的信息如同流言一般传播、病毒一般扩散。

尽管笔者按照习惯也把Gossip也称作是“共识协议”，但首先必须强调它所解决的问题并不是直接与Paxos、Raft这些共识协议等价的，只是基于Gossip之上可以通过某些方法去实现与Paxos、Raft一致的目标。一个最典型的例子是比特币网络中使用到了Gossip协议，

用它来在各个分布式节点中互相同步区块头和区块体的信息，这是整个网络能够正常交换信息的基础，但并不能称作共识；比特币使用[工作量证明](#)（Proof of Work，PoW）来对“这个区块由谁来记账”这一件事情在全网达成共识，这个目标才可以认为与Paxos、Raft的目标是一致的。

下面，我们来了解Gossip的具体算法过程。相比起Paxos、Raft，Gossip的算法过程可算是十分简单了，它可以看作是以下两个步骤的简单循环：

- 如果有某一项信息需要在整个网络中所有节点中传播，那从信息源开始，选择一个固定的传播周期（譬如1秒），随机选择它相连接的k个节点（称为Fan-Out）来传播消息。
- 每一个节点收到消息后，如果这个消息是它之前没有收到过的，将在下一个周期内，选择除了发送消息给它的那个节点外的其他相邻k个节点发送相同的消息，直到最终网络中所有节点都收到了消息，尽管这个过程需要一定时间，但是理论上最终网络的所有节点都会拥有相同的消息。



Gossip传播示意图（[图片来源](#)）

上图是Gossip传播过程的示意图，根据示意图和Gossip的过程描述，我们很容易发现Gossip对网络节点的连通性和稳定性几乎没有任何要求，它将网络某些节点只能与一部分节点部分连通（Partially Connected Network）而不是以全连通网络（Fully Connected Net

work) 作为前提；能够容忍网络上节点的随意地增加或者减少，随意地宕机或者重启，新增加或者重启的节点的状态最终会与其他节点同步达成一致。Gossip把网络上所有节点都视为平等的、普通的，没有任何中心化节点或者主节点的概念，这些特点使得Gossip具有很强的鲁棒性，而且极为适合在公众互联网中应用。

同时我们也很容易找到Gossip的缺点，消息最终是通过多个轮次的散播而到达全网的，因此它必然会产生全网各节点状态不一致的情况，而且由于是随机选取发送消息的节点，所以尽管可以在整体上测算出传播速率，但对于个体消息来说，无法准确地预计到需要多长时间才能达成全网一致。另外一个缺点是消息的冗余，同样是由于随机选取发送消息的节点，也就不可避免的存在消息重复发送给同一节点的情况，增加了网络的传输的压力，也给消息节点带来额外的处理负载。

Gossip传播消息时，有两种可能的传播方式：反熵（Anti-Entropy）和传谣（Rumor-Mongering），这两个名字都挺文艺的。熵（Entropy）是生活中少见但科学中很常用的概念，它代表着事物的混乱程度。反熵的意思就是反混乱，以提升网络各个节点之间的相似度为目标，所以在反熵模式下，会同步节点的全部数据，以消除各节点之间的差异，目标是整个网络各节点完全的一致。但是，在节点本身就会发生变动的前提下，这个目标将使得整个网络中消息的数量会非常庞大，给网络带来巨大的传输开销。而传谣模式是以传播消息为目标，仅仅发送新到达节点的数据，即只对外发送变更信息，这样消息数据量将显著缩减，网络开销也较小。

从类库到服务

通过服务来实现组件

Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services.

微服务架构也会使用到类库，但构成软件系统组件的主要方式是将其拆分为一个个服务。

—— Martin Fowler [↗](#) / James Lewis [↗](#), Microservices [↗](#), 2014

微服务架构其中一个重要设计原则是“通过服务来实现独立自治的组件” (Componentization via Services)，微服务强调通过“服务” (Service) 而不是“类库” (Library) 来构建组件，两者的差别是类库是在编译期静态链接到程序中的，通过本地调用来提供功能，而服务是进程外组件，通过远程调用来提供功能。基于服务来构建程序，迫使微服务在复杂性与执行性能方面作出了极大的让步，换来的是软件系统“整体”与“部分”的物理层面的真正的隔离。

服务发现

类库（ Library ）概念的普及，令计算机实现了通过位于不同模块的方法调用来组装复用指令序列，打开了软件达到更大规模一扇大门。无论是编译期链接的C/CPP，抑或是运行期链接的Java，都要通过[链接器](#)（ Linker ）将代码里的[符号引用](#)转换为模块入口或进程内存地址的直接引用。服务（ Service ）概念的普及，使得通过分布于网络中不同机器能互相协作来复用功能，这是软件发展规模的第二次飞跃，此时如何确定目标方法的确切位置，便是与编译链接有着等同意义的问题，解决该问题的过程就被称作“[服务发现](#)”（ Service Discovery ）。

所有的远程服务调用都是使用“[全限定名](#)（ Fully Qualified Domain Name , FQDN ）、端口号、服务标识”构成的三元组来确定一个远程服务的精确坐标的。全限定名代表了网络中某台主机的精确位置，端口代表了主机上某一个提供服务的程序，服务标识则代表了该程序所提供的一个方法接口。其中“全限定名、端口号”的含义在各种远程服务中都一致，而“服务标识”则与具体的应用层协议相关，可以是多样的，譬如HTTP的远程服务，标识是URL地址；RMI的远程服务，标识是Stub类中的方法；SOAP的远程服务，标识是WSDL中的定义，等等。远程服务的多样性导致了“服务发现”也会有两种不同的理解，一种是以UDI为代表的“百科全书式”的服务发现，上至提供服务的企业信息（企业实体、联系地址、分类目录等等），下至服务的程序接口细节（方法名称、参数、返回值、技术规范等等）都在服务发现的管辖范围之内；另一种是类似于DNS这样“门牌号码式”的服务发现，只满足从某个代表服务提供者的全限定名到服务实际主机IP地址的翻译转换，并不关心服务具体是哪个厂家提供的，也不关心服务由几个方法，各自有什么参数所构成，默认这些细节信息是服务消费者本身所了解的，此时服务坐标就可以退化为简单的“全限定名+端口号”。当今，后一种服务发现占主流地位，本文后续所说的服务发现，如无说明，均是特指的是后者。

原本服务发现只依赖DNS将一个全限定名翻译为一个或者多个IP地址（或者SRV等其他记录）便可实现，后来的负载均衡器也实质上承担了一部分服务发现的职责（指外部IP地址到各个服务内部实际IP的转换），这些内容我们在“[透明多级分流系统](#)”一节中曾经详细解析过，这种方式在软件追求不间断长时间运行的时代是合适的。但随着微服务的逐渐流行，

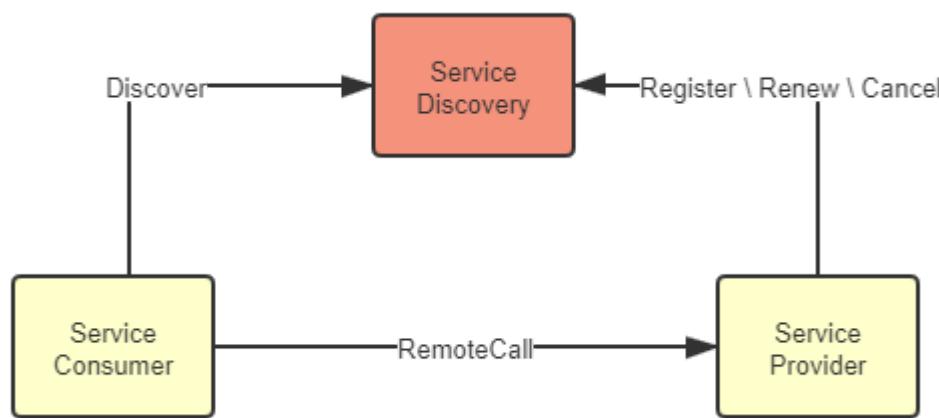
服务的非正常宕机重启和正常的上线下线变得更加频繁，仅靠着DNS服务器和负载均衡器等基础设施就显得有些逐渐疲于应对，无法跟上服务变动的步伐了。人们开始尝试使用ZooKeeper这样的分布式K/V框架，通过软件自身来完成服务注册与发现，ZooKeeper曾短暂统治过远程服务发现，是微服务早期对服务发现的主流选择，但毕竟ZooKeeper是很底层的分布式工具，用户自己还需要做相当多的工作才能满足服务发现的需求。到了2014年，在Netflix内部经受过长时间实际考验的、专门用于服务发现的Eureka宣布开源，并很快被纳入Spring Cloud，成为Spring默认的远程服务发现的解决方案。从此Java程序员再无需再在服务注册这件事情上花费太多的力气。到2018年，Spring Cloud Eureka进入维护模式以后，HashiCorp的Consul和阿里巴巴的Nacos很快就从Eureka手上接过传承的衣钵。此时的服务发现框架已经发展得相当成熟，考虑到几乎方方面面的问题，譬如支持通过DNS或者HTTP请求进行符号与实际地址的转换，支持各种各样的服务健康检查方式，支持集中配置、K/V存储、跨数据中心的数据交换等多种功能，可算是应用自身去解决服务发现的一个顶峰。如今，云原生时代来临，基础设施的灵活性得到大幅度的增强，最初的使用基础设施来透明化地做服务发现的方式又重新被人们所重视，如何在基础设施和网络协议层面，对应用尽可能无感知、尽可能方便地实现服务发现是目前一个主要的发展方向。

本文中，我们将会分析服务发现的几个关键的子问题，并且探讨、对比时下最常见的用作服务发现的几种形式。首先，第一个问题是“服务发现”具体是指进行过什么操作？这里面其实包含了三个必须的过程：

- **服务的注册**（Service Registration）：当服务启动的时候，它应该通过某些形式（譬如调用API、产生事件消息、在ZooKeeper/Etcd的指定位置记录、存入数据库，等等）将自己的坐标信息通知到服务注册中心，这个过程可能由应用程序来完成（譬如Spring Cloud的@EnableDiscoveryClient注解），也可能有容器框架（譬如Kubernetes）来完成。
- **服务的维护**（Service Maintaining）：尽管服务发现框架通常都有提供下线机制，但并没有什么办法保证每次服务都能优雅地下线（Graceful Shutdown）而不是由于宕机、断网等原因突然失联。所以服务发现框架必须要自己去保证所维护的服务列表的正确性，以避免告知消费者服务的坐标后，得到的服务却不能使用的尴尬情况。现在的服务发现框架，往往都能支持多种协议（HTTP、TCP等）、多种方式（长连接、心跳、探针、进程状态等）去监控服务是否健康存活，将不健康的服务自动下线。
- **服务的发现**（Service Discovery）：这里的发现是狭义特指消费者从服务发现框架中，把一个符号（譬如Eureka中的ServiceID、Nacos中的服务名、或者通用的FDQN）转

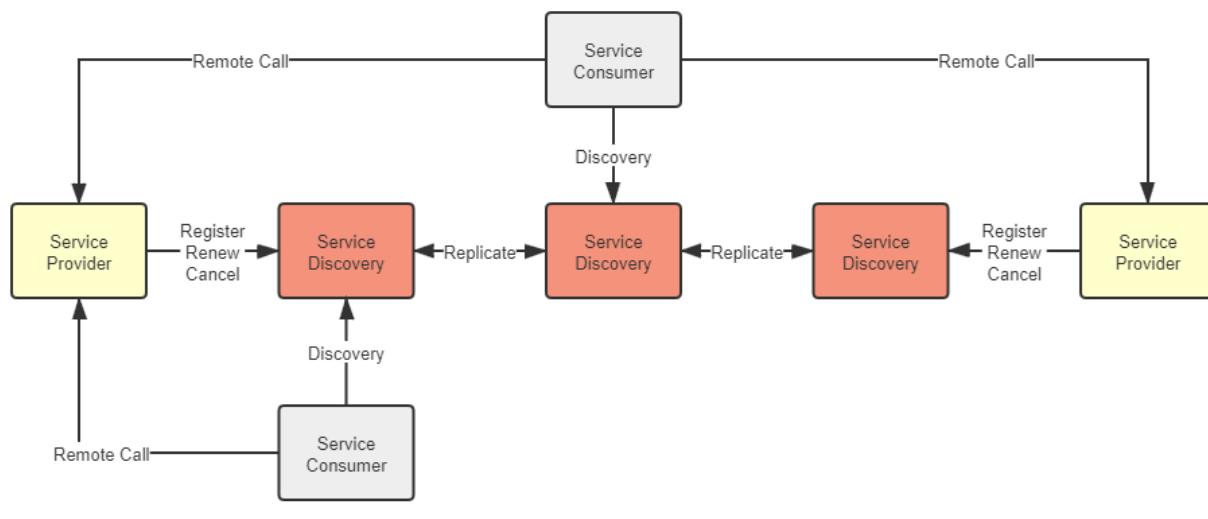
换为服务实际坐标的过程，这个过程现在一般是通过HTTP API请求或者通过DNS Look up操作来完成（还有一些相对少用的方式，如Kubernetes也支持注入环境变量）。

以上三点只列举了必须的过程，在此之余还会有一些可选的功能，譬如在服务发现时进行的负载均衡、流量管控、K/V存储、元数据管理、业务分组，等等，这部分后续会有专门介绍，就不再展开。我们来讨论另一个很常见的问题，说起服务发现的文章，总是无可避免地会先扯到“CP”还是“AP”的问题上。为什么服务发现对CAP如此关注、如此敏感呢？我们可以从服务发现在整个系统中所处的角色来着手分析这个问题，在概念模型中，服务中心所处的地位是如下图所示这样的：提供者在服务发现中注册、续约和下线自己的真实坐标，消费者根据某种符号从服务发现中获取到真实坐标，它们都可以视为系统中平等的微服务，如下图所示：



概念模型

但在真实的系统中，服务发现的地位还是有一些特殊，还不能为完全视其为一个普通的服。务。服务发现是整个系统中所有其他服务都直接依赖的最基础服务（类似相同待遇的大概就数配置中心了，现在服务发现框架也开始同时提供配置中心的功能，以避免配置中心又去专门搞出一集群的节点来），几乎没有办法在业务层面进行容错处理。服务注册中心一旦崩溃，整个系统都受波及，因此必须尽最大可能在技术层面保证可用性。所以，分布式系统中，服务注册中心一般会以内部小集群的方式部署，提供三个或者五个节点（通常最多七个，一般也不会更多了，否则日志复制的开销太高）来保证高可用性，如下图所示：



真实系统

同时，也请注意到上图中各服务发现节点之间的“Replicate”字样，作为用户，我们当然期望服务注册一直可用永远健康的同时，也能够在访问每一个节点中都能取到一致的数据，这两个需求就构成了CAP矛盾。以AP、CP两种取舍作为选择维度，以最有代表性的Eureka和Consul为例，Consul采用Raft协议，要求多数派节点写入成功后服务的注册或变动才算完成，严格地保证了在集群外部读取到的服务发现结果一定是一致的；Eureka的各个节点间采用异步复制来交换服务注册信息，服务注册或变动时，并不需要等待信息在其他节点复制完成，而是马上在该服务发现节点就宣告可见（但其他节点是否可见并不保证）。这两点差异带来的影响并不在于服务注册的快慢（当然，快慢确实是有差别），而在于你如何看待以下这件事情：

假设系统形成了A、B两个网络分区后，A区的服务只能从区域内的服务发现节点获取到A区的服务坐标，B区的服务只能取到在B区的服务坐标，这对你的系统会有什么影响？

- 如果这件事情对你并没有什么影响，甚至有可能还是有益的，就应该倾向于选择AP的服务发现。譬如假设A、B就是不同的机房，是机房间的网络交换机导致服务发现集群出现的分区问题，但每个分区中的服务仍然能独立提供完整且正确的服务能力，此时尽管不是有意而为，但网络分区在事实上避免了跨机房的服务请求，反而还带来了服务调用链路优化的效果。
- 如果这件事情也可能对你影响非常大，甚至可能带来比整个系统宕机更坏的结果，就应该倾向于选择CP的服务发现。譬如系统中大量依赖了集中式缓存、消息总线、或者其他有状态的服务，一旦这些服务全部或者部分被分隔到某一个分区中，会对整个系统的操作的正确性产生直接影响的话，那与其搞出一堆数据错误，还不如停机来得痛快。

数据一致性是分布式系统永恒的话题，在服务发现这个场景里，权衡的主要关注点是一旦出现分区所带来的后果，其他在正常运行过程中的速度问题都是次要的。最后，我们再来讨论一个很“务实”的话题，现在那么多的服务发现框架，哪一款最好？或者说应该如何挑选适合的？

现在直接以服务发现、服务注册中心为目标，或者间接用来实现这个目标的方式主要有以下三类：

- 在分布式K/V存储框架上自己实现的服务发现，这类的代表是ZooKeeper、Doozerd、Etcd

这些K/V框架提供了分布式环境下读写操作的共识保证，Etcd采用的是我们学习过的Raft算法，ZooKeeper采用的是ZAB算法（一种Multi Paxos的派生算法），所以采用这种方案，就不必纠结CP还是AP的问题，它们都是CP的。这类框架的宣传语中往往会主动提及“高可用性”，潜台词其实是“在保证一致性和分区容错性的前提下，尽最大努力实现最高的可用性”，譬如Etcd的宣传语就是“高可用的集中配置和服务发现”（Highly-Available Key Value Store for Shared Configuration and Service Discovery）。这些K/V框架的另一个共同特点是在整体较高复杂度的架构和算法的外部，维持着极为简单的应用接口，只有基本的CRUD和Watch等少量API，所以要在上面完成功能齐全的服务发现，很多基础的能力，譬如服务如何注册、如何做健康检查，等等都必须自己实现，如今一般也只有“大厂”才会直接这些框架去做服务发现了。

- 以基础设施（主要是指DNS服务器）来实现服务发现，这类的代表是SkyDNS、CoreDNS

在Kubernetes 1.3之前的版本使用SkyDNS作为默认的DNS服务，其工作原理是从API Server中监听集群服务的变化，然后根据服务生成NS、SRV等DNS记录存放到Etcd中，kubelet会在每个Pod内部设置DNS服务的地址为SkyDNS的地址，需要调用服务时，只需查询DNS把域名转换成IP列表便可实现分布式的服务发现。在Kubernetes 1.3之后，SkyDNS不再是默认的DNS服务器，由不使用Etcd而是只将DNS记录存储在内存中的Kube DNS代替，到了1.11版，就更推荐采用扩展性很强的CoreDNS，此时可以通过各种插件来决定是否要采用Etcd存储、重定向、定制DNS记录、记录日志，等等。

采用这种方案，是CP还是AP就取决于后端采用何种存储，如果是基于Etcd实现的，那自然是CP的，如果是基于内存异步复制的方案实现的，那就是AP的。以基础设施来做服务发现，好处是对应用透明，任何语言、框架、工具都肯定是支持HTTP、DNS的，所以完全不受程序技术选型的约束，但坏处是透明的并不一定是简单的，你必须自己考

虑如何去做客户端负载均衡、如何调用远程方法等这些问题，而且必须遵循或者说受限于这些基础设施本身所采用的实现机制，譬如服务健康检查里，服务的缓存期限就必须采用TTL（Time to Live）来决定，这是DNS协议所规定的，如果想改用KeepAlive长连接来实时判断服务是否存活就很麻烦。

- 专门用于服务发现的框架和工具，这类的代表是Eureka、Consul和Nacos
这一类框架中，你可以自己决定是CP还是AP的问题，譬如CP的Consul、AP的Eureka，还有同时支持CP和AP的Nacos（Nacos采用类Raft协议做的CP，采用自研的Distro协议做的AP，这里“同时是”都支持的意思，它们必须二取其一，不是说CAP全能满足）。另外，还有很重要一点是它们对应用并不是透明的，尽管Consul、Nacos也支持基于DNS的服务发现，尽管这些框架都基本上做到了以声明代替编码（譬如在Spring Cloud中只改动pom.xml、配置文件和注解即可实现），但它们依然是应用程序有感知的。所以或多或少还需要考虑你所用的程序语言、技术框架的集成问题。但这一点其实并不见得就是坏处，譬如采用Eureka做服务注册，那在远程调用服务时你就可以用OpenFeign做客户端，写个声明式接口就能跑，相当能偷懒；在做负载均衡时你就可以采用Ribbon做客户端，要换均衡算法改个配置就成，这些“不透明”实际上都为编码开发带来了一定便捷，而前提是你选用的语言和框架支持。如果老板提出要在Rust上用Eureka，你就只能无奈叹息了（原本这里我写的是Node、Go、Python等，查了一下这些居然都有非官方的Eureka客户端，用的人多就是有好处啊）。

网关路由

网关（Gateway）这个词在计算机科学中，尤其是计算机网络中很常见，它用来表示位于内部区域边缘，与外界进行交互的某个物理或逻辑设备，譬如你家里的路由器就属于家庭内网与互联网之间的网关。

在单体架构下，我们一般不太强调“网关”这个概念，为各个单体系统的副本分发流量的负载均衡器实质上承担着内部服务与外部调用之间的网关角色。不过在微服务环境中，网关的存在感就极大地增强了，甚至成为微服务集群中必不可少的设施之一。其中原因并不难理解，试想在微服务架构下，每个服务节点都由不同团队负责，有自己独立的、各不相同的能力，如果服务集群没有一个统一对外交互的代理人角色，那外部的服务消费者就必须知道所有微服务在集群中的精确坐标（在[服务发现](#)中介绍过“坐标”的概念），这样，消费者不仅会受到服务集群的网络限制（不能确保集群中每个节点都有外网连接）、安全限制（不仅是服务节点的安全，外部自身也会受到如浏览器[同源策略](#)的约束）、依赖限制（服务坐标这类信息不属于对外接口承诺的内容，随时可能变动，不应该依赖它），就算是程序员自己也不可能愿意记住每一个服务的坐标位置来编写代码。所以，微服务中网关的首要职责就是以统一的地址对外提供服务，将外部访问这个地址的流量，根据适当的规则路由到内部集群中正确的服务节点之上，因此，微服务中的网关，也常被称为“服务网关”或者“API网关”。可见，微服务的网关首先应该是个路由器，在满足此前提的基础上，网关还可以根据需要作为流量过滤器来使用，以提供某些额外的可选的功能，譬如安全、认证、授权、限流、监控、缓存，等等（这部分内容在文档的其他文章中有专门讲解，这里不会涉及）。简而言之：

网关 = 路由器（基础职能） + 过滤器（可选职能）

在“路由”这个基础职能里，服务网关主要考虑是能够支持路由的“网络层次与协议”和“性能与可用性”两方面的因素。网络层次是指[负载均衡](#)中介绍过的四层流量转发与七层流量代理，仅从技术实现角度来看，对于路由流量这项工作，负载均衡器与服务网关的实现是没有什么差别的，很多服务网关本身就是基于老牌的负载均衡器来实现的，譬如Nginx、HAProxy对应的Ingress Controller；而从目的角度看，负载均衡器与服务网关的区别在于前者是为了根据均衡算法对流量进行平均地路由，后者是为了根据流量中的某种特征进行正确

地路由。网关必须能够识别流量中的特征，这意味着网关能够支持的网络层次、通讯协议的数量，将会直接限制后端服务节点能够选择的服务通讯方式。如果服务集群只提供如Etc d这类直接基于TCP的访问的服务，那可以只部署四层网关，以TCP报文中源地址、目标地址为特征进行路由；如果服务集群要提供HTTP服务的话，就必须部署一个七层网关，根据HTTP的URL、Header等信息为特征进行路由；如果服务集群要提供更上层的WebSocke t、SOAP等服务，那就必须要求网关同样能够支持这些上层协议，才能从中提取到特征。以下是一段基于SpringCloud实现的Fenix's Bootstore中用到的Netflix Zuul网关的配置，Zu ul是HTTP网关，/restful/accounts/** 和 /restful/pay/** 是HTTP中URL的特征，而配置中的 serviceId 就是路由的目标服务。

```
routes:  
  account:  
    path: /restful/accounts/**  
    serviceId: account  
    stripPrefix: false  
    sensitiveHeaders: "*"  
  
  payment:  
    path: /restful/pay/**  
    serviceId: payment  
    stripPrefix: false  
    sensitiveHeaders: "*"
```

yaml

现在围绕微服务的各种技术均处于快速发展期，笔者并不提倡针对每一种框架本身去记忆配置细节，就是无需纠结上面这些配置的确切写法、每个指令的含义。如果你从根本上理解了网关的原理，参考一下技术手册，很容易就能够将上面的信息改写成Kubernetes Ingress Controller、Istio VirtualServer或者其他服务网关所需的配置形式。

网关的另一个关注点是它的性能与可用性。由于网关是所有服务对外的总出口，流量必经之地，所以网关的路由性能是全局的、系统性的，如果经过网关路由会有10毫秒的性能损失，就意味着整个系统所有服务的性能都会降低10毫秒。网关的性能与它的工作模式和自身实现都有关系，但毫无疑问工作模式是最主要的，如果能够采用DSR三角传输模式，原理上就决定了性能一定会比代理模式来的强（DSR、IP Tunnel、NAT、代理等这些都是负载均衡的基础知识，笔者曾在[负载均衡器](#)中详细讲解过）。不过，因为今天REST和JSON -RPC等基于HTTP协议的接口形式在对外部提供的服务中占绝对主流的地位，所以我们所讨论的服务网关默认都必须支持七层路由，通常就默认无法转发只能采用代理模式。在这

个前提约束下，网关的性能就主要取决于它们如何代理网络请求的，也即是它们的网络I/O模型了，笔者在这里先简要复习一下网络I/O的基础知识。

在套接字接口抽象下，网络I/O的本质是Socket的读取，Socket在操作系统接口中被抽象为数据流，网络I/O可以理解为就是对流的操作。对于每一次网络访问，从远程主机返回的数据会先存放到操作系统内核的缓冲区中，然后再从内核的缓冲区复制到应用程序的地址空间，所以当发生一次网络请求发生后，将会按顺序经历“等待数据从远程主机到达缓冲区”和“将数据从缓冲区拷贝到应用程序地址空间”两个阶段，根据完成这两个阶段的不同方法，可以把网络I/O模型总结为两类、五种模型，两类是指 同步I/O 与 异步I/O，五种是指在 同步I/O 中又分有划分出 阻塞I/O 、 非阻塞I/O 、 多路复用I/O 和 信号驱动I/O 四种细分模型。同步就是调用端发出请求之后，得到结果之前必须一直等待，与之相对的就是异步，发出调用请求之后将立即返回，不会马上得到处理结果，结果将通过状态变化和回调来通知调用者。而阻塞和非阻塞针对请求处理过程，指收到调用请求，返回结果之前，当前处理线程是否会被挂起。这种概念上的叙述估计是不好理解的，笔者以“你如何领到盒饭”为情景，将之类比解释如下：

- **异步I/O (Asynchronous I/O)**：好比你在美团外卖订了个盒饭，付款之后你自己该干嘛还干嘛去，饭做好了骑手自然会到门口打电话通知你。异步I/O中数据到达缓冲区后，不需要由调用进程主动进行从缓冲区复制数据的操作，而是复制完成后由操作系统向线程发送信号，所以它一定是非阻塞的。
- **同步I/O (Synchronous I/O)**：好比你自己去饭堂打饭，这时可能有如下情形发生：
 - **阻塞I/O (Blocking I/O)**：你去到饭堂，发现饭还没做好，你也干不了别的，只能打个瞌睡（线程休眠），直到饭做好。阻塞I/O是最直观的I/O模型，逻辑清晰，也比较节省CPU资源，但缺点就是线程休眠所带来的上下文切换，这是一种需要切换到内核态的重负载操作，不应当频繁进行。
 - **非阻塞I/O (Non-Blocking I/O)**：你去到饭堂，发现饭还没做好，你就回去了，然后每隔3分钟来一次饭堂看饭做好了没，直到饭做好。非阻塞I/O能够避免线程休眠，对于一些很快就能返回结果的请求，非阻塞I/O可以节省切换上下文切换的消耗，但是对于较长时间才能返回的请求，非阻塞I/O反而白白浪费了CPU资源，所以目前并不常用。
 - **多路复用I/O (Multiplexing I/O)**：多路复用I/O本质上是阻塞I/O的一种，但是它的好处是可以在同一条阻塞线程上处理多个不同端口的监听。类比的情景是你叫雷锋，代

表整个宿舍去饭堂打饭，去到饭堂，发现饭还没做好，还是继续打瞌睡，但哪个舍友的饭好了，你就马上把那份饭送回去，然后继续打着瞌睡哼着歌等待其他的饭做好。多路复用I/O是目前的高并发网络应用的主流，它下面还可以细分select、epoll、kqueue等不同实现，这里就不作展开了。

- **信号驱动I/O（Signal-Driven I/O）**：你去到饭堂，发现饭还没做好，但你跟厨师熟，跟他说饭做好了叫你，然后回去该干嘛干嘛，等收到厨师通知后，你把饭从饭堂拿回宿舍。这里厨师的通知就是那个“信号”，信号驱动I/O与异步I/O的区别是“从缓冲区获取数据”这个步骤的处理，前者收到的通知是可以开始进行复制操作了，即要你自己从饭堂拿回宿舍，在复制完成之前线程处于阻塞状态，所以它仍属于同步I/O操作，而后者收到的通知是复制操作已经完成，即外卖小哥已经把饭送到了。

显而易见，异步I/O模型是最方便的，毕竟能叫外卖谁愿意跑饭堂啊，但前提是学校里有美团外卖。同样，异步I/O受限于操作系统，Windows NT内核早在3.5以后，就通过IOCP实现了真正的异步I/O模型。而Linux系统下，是在Linux Kernel 2.6才首次引入，目前也还并不完善，因此在Linux下实现高并发网络编程时仍是以多路复用I/O模型模式为主。

回到服务网关的话题上，有了网络I/O模型的知识，我们就可以在理论上定性分析不同网关的性能差异了。服务网关处理一次请求代理时，包含了两组网络操作，分别是作为服务端对外部请求的应答，和作为客户端对内部服务的调用，理论上这两组网络操作可以采用不同的模型去完成，但一般来说并没有必要这样做。以Zuul网关为例，在Zuul 1.0时，它采用的是阻塞I/O模型来进行最经典的“一条线程对应一个连接”（Thread-per-Connection）的方式来代理流量，采用阻塞I/O意味着它会有线程休眠，就有上下文切换的成本，所以如果后端服务普遍属于计算密集型（CPU Bound，可以通俗理解为服务耗时比较长，主要消耗在CPU上）时，这种模式能够节省网关的CPU资源，但如果后端服务普遍都是I/O密集型（I/O Bound，可以理解服务都很快返回，主要消耗在I/O上），它就会由于频繁的上下文切换而降低性能。在Zuul的2.0版本，最大的改进就是基于Netty Server实现了异步I/O模型来处理请求，大幅度减少了线程数，获得了更高的性能和更低的延迟。根据Netflix官方自己给出的数据，Zuul 2.0大约要比Zuul 1.0快20%左右。还有一些网关，可以自行配置，或者根据环境选择不同的网络I/O模型，典型的就是Nginx，可以支持在配置文件中指定select、poll、epoll、kqueue等并发模型。

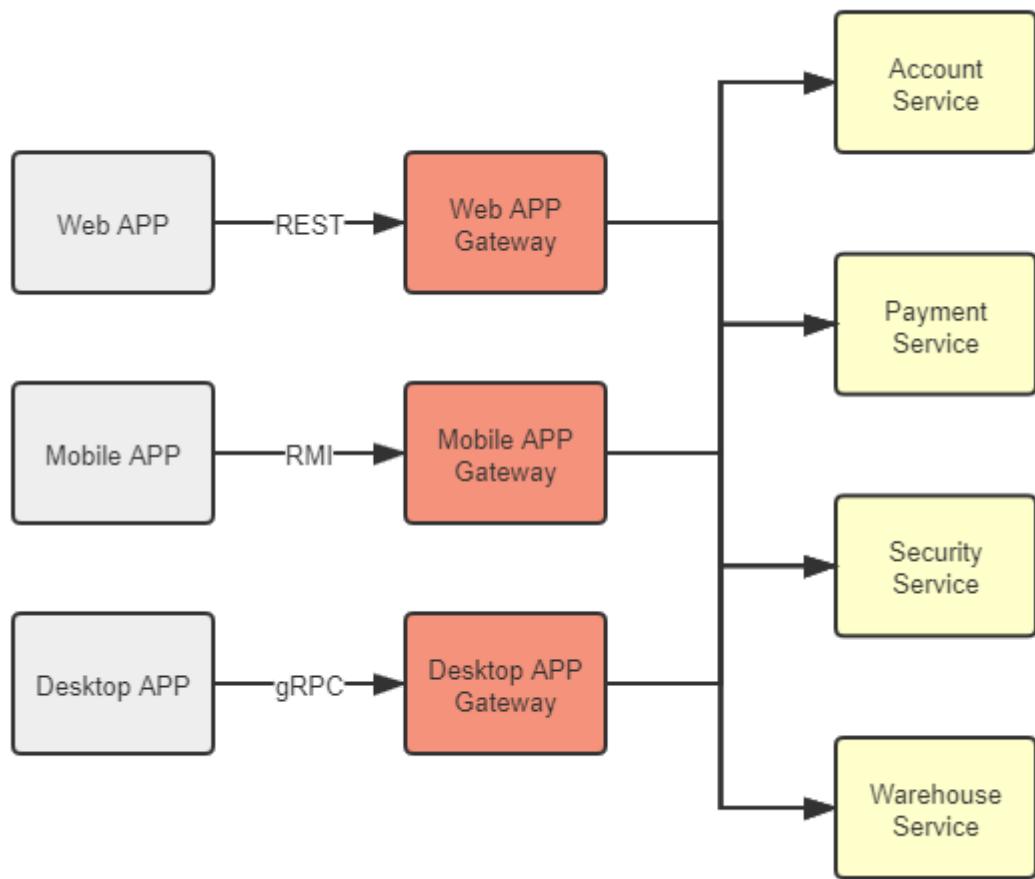
网关的性能高低一般只去定性分析，要定量地说哪一种网关性能最高、高多少是很难的，就像我们都认可Chrome要比IE快，但具体要快上多少很难说的清楚。尽管笔者上面引用了Netflix官方对Zuul两个版本的量化对比，网络上也有不少关于各种网关的性能对比数据，

但要是脱离具体应用场景去定量地比较不同网关的性能差异还是难以令人信服，不同的测试环境、后端服务都会直接影响结果。

网关还有一点要关注的是可用性问题。任何系统的网络调用过程中都至少会有一个单点存在，这是由用户只通过唯一的一个地址去访问系统决定的。即使是淘宝、亚马逊这样全球多数据中心部署的大型系统，为多数据中心翻译地址的权威DNS服务器也可以认为是它的单点。对于更普遍的小型系统（小型是相对淘宝这些而言）来说，作为后端对外服务代理人角色的网关经常被视为系统的入口，往往很容易成为网络访问中的单点，这时候它的可用性就尤为重要。由于网关具有唯一性，就不像之前[服务发现](#)那些注册中心那样直接做个集群，随便访问哪一台都可以解决问题。为此，对网关的可用性方面，我们应该考虑到以下几点：

- 网关应尽可能轻量，尽管网关作为服务集群统一的出入口，可以很方便地做安全、认证、授权、限流、监控，等等的功能，但给网关附加这些能力时还是要仔细权衡，取得功能性与可用性之间的平衡，过度增加网关的职责是危险的。
- 网关选型时，应该尽可能选择较成熟的产品实现，譬如Nginx Ingress Controller、KONG、Zuul这些经受过长期考验的产品，而不能一味只考虑性能选择最新的产品，性能与可用性之间的平衡也需要权衡。
- 在需要高可用的生产环境中，应当考虑在网关之前部署负载均衡器或者[等价路由器](#)（ECMP），让那些更成熟健壮的（往往是硬件物理设备）的设施去充当整个系统的入口地址，这样网关就可以很方便地设置多路扩展了。

提到网关的唯一性、高可用与扩展，笔者顺带也说一下近年来随着微服务一起火起来的概念“BFF”（Backends for Frontends）。这个概念目前还没有权威的中文翻译，在我们讨论的上下文里，它的意思是，网关不必为所有的前端提供无差别的服务，而是应该针对不同的前端，聚合不同的服务，提供不同的接口和网络访问协议支持。譬如，运行于浏览器的Web程序，由于浏览器一般只支持HTTP协议，服务网关就应提供REST等基于HTTP协议的服务，但同时我们亦可以针对运行于桌面系统的程序部署另外一套网关，它能与Web网关有完全不同的技术选型，能提供出基于更高性能协议（如gRPC）的接口来获得更好的体验。在网关这种边缘节点上，针对同一样的后端集群，裁剪、适配、聚合出适应不一样的前端的服务，有助于后端的稳定，也有助于前端的赋能。



Backends for Frontends 网关

客户端负载均衡

前置知识

关于经典的集中式负载均衡的工作原理，笔者已在“[负载均衡](#)”一节中介绍过，其中许多知识是相通的，笔者在本篇中将不再重复，建议读者先行阅读。

在正式开始讨论之前，我们先来明确区分清楚几个容易混淆的概念，分别是本章节中介绍到的 服务发现 、 网关路由 、 负载均衡 以及在服务流量治理章节中将会介绍的 调用容错 。这几个技术名词都带有着“从服务集群中寻找到一个合适的服务来调用”的含义，笔者通过一个具体场景来说明它们之间的差别：

案例场景：

你身处广东，要上Fenix's Boosstore上购买一本书，在程序业务逻辑里，购书其中一个关键步骤是调用商品出库服务来完成货物准备，在代码中该服务的调用请求为：

```
PATCH https://warehouse:8080/restful/stockpile/3  
{amount: -1}
```

假设Fenix's Boosstore是个大书店，在北京、武汉、广州的机房均部署有服务集群，此时按顺序发生了以下事件：

- 首先是将warehouse这个服务名称转换为恰当的服务地址，“恰当”是个宽泛的描述，一种典型的“恰当”便是因调用请求来自广东，优先分配给传输距离最短的广州机房来应答。其实按常理来说这次出库服务的调用应该是集群内的流量，而不是用户浏览器直接发出的请求，所以尽管结果没有不同，但更接近实际的情况是用户访问首页时已经被DNS服务器分配到了广州机房，请求出库服务时，应优先选择同机房的服务进行调用，此时请求变为：

```
PATCH https://guangzhou-ip-wan:8080/restful/stockpile/3
```

2. 广州机房的服务网关将该请求与配置中的特征进行比对，由URL中的 `/restful/stockpile/**` 得知该请求访问的是商品出库服务，因此，将请求的IP地址转换为内网中warehouse服务集群的入口地址：

```
PATCH https://warehouse-gz-lan:8080/restful/stockpile/3
```

3. 集群中部署有多个warehouse服务，收到调用请求后，负载均衡器要在多个服务中根据某种标准——可能是随机挑选，也可能是按顺序轮询，抑或是选择此前调用次数最少那个，等等。根据均衡策略找出要响应本次调用的服务，称其为warehouse-gz-lan-node1。

```
PATCH https://warehouse-gz-lan-node1:8080/restful/stockpile/3
```

4. 访问warehouse-gz-lan-node1服务，没有返回需要的结果，而是抛出500错。

```
HTTP/1.1 500 Internal Server Error
```

5. 根据预置的[故障转移](#)（Failover）策略，重试将调用分配给能够提供该服务的其他节点，称其为warehouse-gz-lan-node2。

```
PATCH https://warehouse-gz-lan-node2:8080/restful/stockpile/3
```

6. warehouse-gz-lan-node2服务返回商品出库成功。

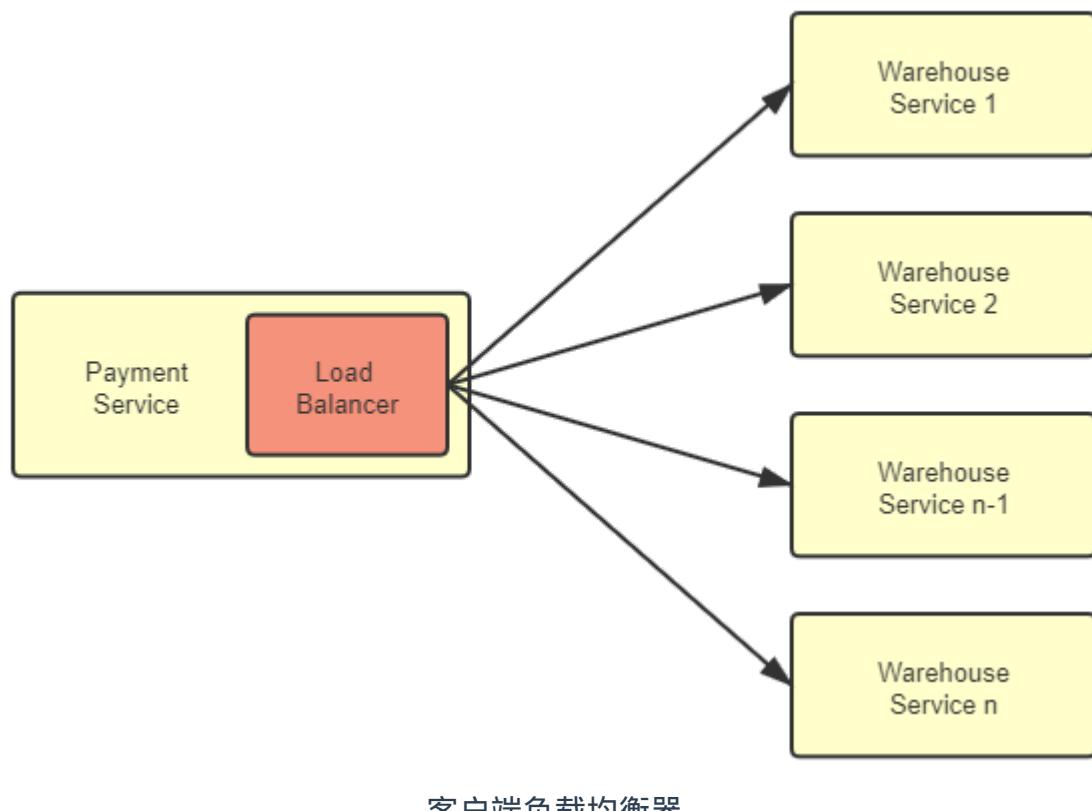
```
HTTP/1.1 200 OK
```

以上过程从整体上看，步骤1、2、3、5，分别对应了 服务发现 、 网关路由 、 负载均衡 和 调用容错，在细节上看，其中部分职责又是有交叉的，并不是注册中心就只关心服务发现，网关只关心路由，均衡器只关心负载均衡。譬如步骤1服务发现的过程中，“根据请求来源的物理位置来分配机房”这个操作本质上是根据请求中的特征（地理位置）进行流量分发，这实际是一种路由行为。实际系统中，在DNS服务器（DNS智能线路）、服务注册中心（如Eureka等框架中的Region、Zone概念）或者负载均衡器（可用区负载均衡，如AWS的NLB，或Envoy的Region、Zone、Sub-zone）中都有可能实现。此外，你是否感觉到

以上网络调用过程似乎过于繁琐了，一个从广州机房内网发出的服务请求，绕到了网络边缘的网关、负载均衡器这些设施上，然后再被分配回内网中另外一个服务去响应。不仅消耗了带宽，降低了性能，也增加了链路上的风险和运维的复杂度。可是，如果流量不经过这些设施，它们相应的职责就无法发挥作用——不经过负载均衡器的话，甚至连请求应该具体交给哪一个服务去处理都无法确定。

客户端负载均衡器

对于任何一个大型系统，负载均衡器都是必不可少的设施。以前，负载均衡器大多只部署在整个服务集群的前端，将用户的请求分流到各个服务进行处理，这种经典的部署形式现在被称为集中式的负载均衡。随着微服务日渐流行，服务集群的收到的请求来源不再局限于外部，越来越多的访问请求是由集群内部的某个服务发起，由集群内部的另一个服务进行响应的，对于这类流量的负载均衡，既有的方案依然是可行的，但针内部流量的特点，直接在服务集群内部消化掉，肯定是更合理更受开发者青睐的办法。由此一种全新的、独立位于每个服务前端的、分散式的负载均衡方式正逐渐变得流行起来，这就是本节我们要讨论的主角：客户端负载均衡器（Client-Side Load Balancer）。



客户端负载均衡器的理念提出以后，此前的集中式负载均衡器也有了一个方便与它对比的名字“服务端负载均衡器”（ Server-Side Load Balancer ）。从上图中能够清晰地看到这两种均衡器的关键差别所在：客户端均衡器是和服务实例一一对应的，而且与服务实例并存于同一个进程之内。这能为它带来很多好处，如：

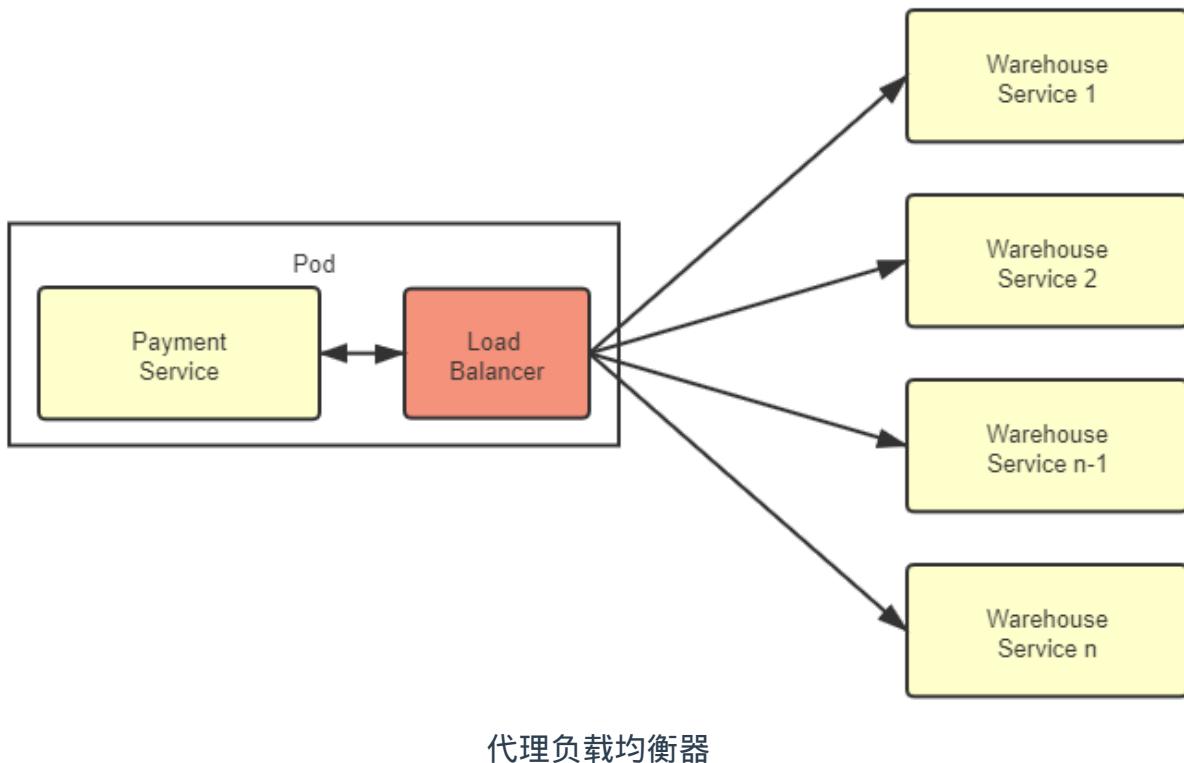
- 均衡器与服务之间信息交换是进程内的方法调用，不存在任何额外的网络开销。
- 不依赖集群边缘的设施，所有内部流量都仅在服务集群的内部循环，避免了出现前文那样，集群内部流量要“绕场一周”的尴尬局面。
- 分散式的均衡器意味着天然避免了集中式的单点问题，它的带宽资源将不会像集中式均衡器那样敏感，这在以七层均衡器为绝对主流、不能通过IP隧道和三角传输这样方式节省带宽的微服务环境中显得更具优势。
- 客户端均衡器要更加灵活，能够针对每一个服务实例单独设置均衡策略等参数，访问某个服务，是不是需要具备亲和性，选择服务的策略是随机、轮询、加权还是最小连接等等，都可以单独设置而不影响其它服务。
-

但是，客户端均衡器并不是银弹，它的缺点同样是不少的：

- 它与服务运行于同一个进程之内，意味着它的选型受到服务所使用的编程语言的限制，譬如用Golang开发的微服务就不太可能搭配Spring Cloud Load Balancer来使用，要为每种语言都实现对应的能够支持复杂网络情况的均衡器是非常难的。客户端均衡器的这个缺陷有违于微服务中技术异构不应受到限制的原则。
- 从个体服务来看，由于是共用一个进程，均衡器的稳定性会直接影响整个服务进程的稳定性，消耗的CPU、内存等资源也同样影响到服务的可用资源。从集群整体来看，在服务数量达成千乃至上万规模时，客户端均衡器消耗的资源总量是相当可观的。
- 由于请求的来源可能是来自集群中任意一个服务节点，而不再是统一来自集中式均衡器，这就使得内部网络安全和信任关系变得复杂，当攻破任何一个服务时，更容易通过该服务突破集群中的其他部分。
- 服务集群的拓扑关系是动态的，每一个客户端均衡器必须持续跟踪其他服务的健康状况，以实现上线新服务、下线旧服务、自动剔除失败的服务、自动重连恢复的服务等均衡器必须具备的功能。由于这些操作都需要通过访问服务注册中心来完成，数量庞大的客户端均衡器一直持续轮询服务注册中心，也会为它带来不小的负担。
-

代理负载均衡器

在Java领域，客户端均衡器中最具代表性的产品是Netflix Ribbon和Spring Cloud Load Balancer，随着微服务的流行，它们在Java微服务中已积聚了相当可观的使用者。到了最近两三年，服务网格（Service Mesh）开始盛行，另外一种被称为“代理客户端负载均衡器”（Proxy Client-Side Load Balancer，后文简称“代理均衡器”）的客户端均衡器变体形式开始引起不同编程语言的微服务开发者共同关注，它解决了此前客户端均衡器的大部分缺陷。代理均衡器对此前的客户端负载均衡器的改进是将原本嵌入在服务进程中的均衡器提取出来，放到边车代理中去实现，它的流量关系如下图所示。



代理负载均衡器

虽然代理均衡器与服务实例不再是进程内通讯，而是通过虚拟化网络进行数据交换的，数据要经过操作系统的协议栈，要进行打包拆包、计算校验和、维护序列号等网络数据的收发等步骤（Envoy中支持使用[Unix Domain Socket](#)来进一步避免这种消耗），流量比起之前的客户端均衡器确实多经历了一次代理过程。不过，Kubernetes严格保证了同一个Pod中的容器不会跨越不同的节点，相同Pod中的容器共享同一个网络和[Linux UTS名称空间](#)，因此代理均衡器与服务实例的交互仍然要比真正的网络交互高效且稳定得多，代价很小，但从服务进程中分离出来的收益则是显著的：

- 代理均衡器不再受编程语言的限制。发展一个支持Java、Golang、Python等所有微服务应用服务的通用的代理均衡器具有很高的性价比。集中不同编程语言的使用者的力量，更容易打造出能面对复杂网络情况的、高效健壮的均衡器。即使退一步说，独立于服务进程的均衡器也不会由于自身的稳定性影响到服务进程的稳定。
- 在服务拓扑感知方面代理均衡器也要更有优势。由于边车代理接受控制平面的统一管理，当服务节点拓扑关系发生变化时，控制平面就会主动向边车代理发送更新服务清单的控制指令，这避免了此前客户端均衡器必须长期主动轮询服务注册中心所造成的浪费。
- 在安全性、可观测性上，由于边车代理都是一致的实现，有利于在服务间建立双向TLS通讯，也有利于对整个调用链路给出更详细的统计信息。
-

总体而言，边车代理这种通过同一个Pod的独立容器实现的负载均衡器是目前处理微服务集群内部流量最理想的方式，只是服务网格本身仍是初生事物，还不足够成熟，对操作系统、网络和运维方面的知识要求也较高，但有理由相信随着时间的推移，未来这将会是微服务的主流通讯方式。

地域与区域

最后，借助前文已经铺设好的上下文场景，笔者想再谈一个与负载均衡相关，但又不仅仅只涉及到负载均衡的概念：地域与区域。你是否有注意到在微服务相关的许多设施中，都带有着Region、Zone参数，如前文中提到过的服务注册中心Eureka的Region、Zone、边车代理Envoy中的Region、Zone、Sub-zone，如果你有云计算IaaS的使用经历，也会发现几乎所有云计算设备都有类似的概念。Region和Zone是公有云计算先驱亚马逊AWS提出的概念，它们的含义是指：

- Region是**地域**的意思，譬如华北、东北、华东、华南，这些都是地域范围。面向全球或全国的大型系统的服务集群往往会被部署在多个不同地域，譬如本文最初设计的案例场景，就是通过不同地域的机房来缩短用户与服务器之间的物理距离，提升响应速度，对于小型系统，地域一般就只在异地容灾时才会涉及到。需要注意，不同地域之间是没有内网连接的，所有流量都只能经过公众互联网相连，如果微服务的流量跨越了地域，实

际就跟调用外部服务商提供的互联网服务没有任何差别了。所以集群内部流量是不会跨地域的，服务发现、负载均衡器默认也是不会支持跨地域的服务发现和负载均衡。

- Zone是**区域**的意思，它是**可用区域**（Availability Zones）的简称，区域指在地理上位于同一地域内，但电力和网络是互相独立的物理区域，譬如在华东的上海、杭州、苏州的不同机房就是同一个地域的几个可用区域。同一个地域的区域之间具有内网连接，流量不占用公网带宽，因此区域是微服务集群内流量能够触及的最大范围。但你的应用是只部署在同一区域内，还是部署到几个可用区域中，主要取决于你是否有做异地双活的需求，以及对网络延时的容忍程度。
 - 如果你追求高可用，譬如希望系统即使在某个地区发生电力或者骨干网络中断时仍然可用，那可以考虑将系统部署在多个区域中。注意异地容灾和异地双活的区别：容灾是非实时的同步，而双活是实时或者准实时的，跨地域或者跨区域做容灾都可以，但只能一般只能跨区域做双活，当然也可以将它们结合起来同时使用，即“两地三中心”模式。
 - 如果你追求低延迟，譬如对时间有高要求的[SLA应用](#)，或者网络游戏服务器等，那就应该考虑将系统部署在同一个区域中，因为尽管内网连接不受限于公网带宽，但毕竟机房之间的专线容量也是有限的，难以跟机房内部的交换机相比，延时也受物理距离、[网络跃点](#)等因素的影响。
- 可用区域对应于城市级别的区域的范围，一些场景中仍是过大了一些，即时是同一个区域中的机房，也可能存在具有差异的不同子网络，所以在部分微服务框架也提供了Group、Sub-zone等做进一步的细分控制，这些参数的意思通常是加权或优先访问同一个子区域的服务，但如果子区域中没有合适的，仍然会访问到可用区域中的其他服务。
- 地域和区域原本是云计算中的概念，对于一些中小型的微服务系统，尤其是非互联网的企业信息系统，很多仍然没有使用云计算设施，只部署在某个专有机房内部，只为特定人群提供服务，这就不需要涉及地理上地域、区域的概念了。此时完全可以自己灵活拓展Region、Zone参数的含义，达到优化虚拟化基础设施流量的目的。譬如，将框架的这类设置与Kubernetes的标签、选择器配合，实现内部服务请求其他服务时，优先使用同一个Node中的提供的服务进行应答，以降低真实的网络消耗。

服务与流量治理

容错性设计

Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore service

由于服务随时都有可能崩溃，因此快速的失败检测和自动恢复就显得至关重要。

—— Martin Fowler [↗](#) / James Lewis [↗](#), Microservices [↗](#), 2014

“容错性设计” (Design for Failure) 是微服务的核心原则之一，也是笔者在此文档中多次反复强调的开发观念转变。不过即使已经有一定的心理准备，大多数首次将微服务架构引入实际生产系统的开发者，在服务发现、网关路由等支持下，踏出了微服务化的第一步以后，很可能仍会经历一段阵痛期，随着拆分出的服务越来越多，往往会随之而来面临以下两个问题的困扰：

- 由于某一个服务的崩溃，导致所有用到这个服务的其他服务都无法工作，进而再层层传递，波及到调用链上与此有关的所有服务。防止雪崩效应是微服务中容错性设计原则的体现，是服务化架构必须解决的问题，否则服务化程度越高，整个系统反而越不稳定。
- 服务虽然没有崩溃，但由于处理能力有限，面临超过预期的突发请求时，大部分请求直至超时都无法完成响应。这种现象产生的后果跟交通堵塞是类似的，如果一开始没有得到及时处理，后面就需要长时间才能使得全部服务都恢复正常。

“服务流量治理”这一部分，我们将围绕这如何解决以上两个问题，提出服务容错、流量控制、服务质量管理等一系列解决方案。这些措施并非孤立的，它们相互之间存在很多联系，其中许多功能必须与此前介绍过的服务注册中心、服务网关、负载均衡器配合才能实现。理清楚这些技术措施背后的逻辑链条，是了解它们工作原理的捷径。

服务容错

Martin Fowler与James Lewis提出的“[微服务的九个核心特征](#)”是构建微服务系统的指导性原则，但不是技术规范，并没有严格的约束力。在实际构建系统时候，其中多数特征可能会有或多或少的妥协，譬如分散治理、数据去中心化、轻量级通讯机制、演进式设计，等等。但也有一些特征是无法做出妥协的，其中的典型就是今天我们讨论的主题：容错性设计（Design for Failure）。

容错性设计不能妥协来源于分布式系统的本质是不可靠的，一个大的服务集群中，程序可能崩溃、节点可能宕机、网络可能中断，这些“意外情况”其实全部都在“意料之中”。原本一个信息系统进行分布式的主要动力之一就是为了提升系统的可用性——最低限度也必须保证将原有系统重构为分布式架构之后，可用性不会倒退下降才行。如果服务集群中出现任何一点差错都能让系统发生“千里之堤溃于蚁穴”的情况，那分布式系统恐怕就根本不会成为一种可用的系统架构形式。

容错策略

要落实容错性设计这条原则，除了思想观念上转变过来，正视服务必然是会出错的，对它进行有计划的防御之外，还必须了解一些常用的 容错策略 和 容错设计模式，作为具体设计与编码实践的指导。 容错策略 指的是“面对故障，我们该做些什么”，稍后将讲解的 容错设计模式 指的是“要实现某种容错策略，我们该如何去做”。常见的容错策略有以下几种：

- **故障转移（Failover）**：高可用的服务集群中，多数的服务——尤其是那些经常被其他服务所依赖的关键路径上的服务，均会部署多有个副本。这些副本可能部署在不同的节点（避免节点宕机）、不同的网络交换机（避免网络分区）甚至是不同的可用区（避免整个地区发生灾害或电力、骨干网故障）中。故障转移是指如果调用的服务器出现故障，系统不会立即向调用者返回失败结果，而是自动切换到其他服务副本，尝试其他副本能否返回成功调用的结果，从而保证了整体的高可用性。

故障转移的容错策略应该有一定的调用次数限制，譬如允许最多重试3个服务，如果都

发生报错，那还是会返回调用失败。原因不仅是因为重试是有执行成本的，更是因为过度的重试反而可能让系统处于更加不利的状况。譬如有以下调用链：

Service A → Service B → Service C

假设A的超时阈值为100ms，而B调用C花费60ms，然后不幸失败了，这时候做故障转移其实已经没有太大意义了，因为即时下一次调用能够返回正确结果，也很可能同样需要耗费60ms时间，时间总和就已经触及A服务的超时阈值，所以在这种情况下故障转移反而对系统是不利的。

- **快速失败 (Failfast)**：还有另外一些业务场景是不允许做故障转移的，故障转移策略能够实施的前提是要求服务具备幂等性，对于非幂等的服务，重复调用就可能产生脏数据，引起的麻烦远大于单纯的某次服务调用失败，此时就应该以快速失败作为首选的容错策略。譬如，在支付场景中，需要调用银行的扣款接口，如果该接口返回的结果是网络异常，程序是很难判断到底是扣款指令发送给银行时出现的网络异常，还是银行扣款后返回结果给服务时出现的网络异常的。为了避免重复扣款，此时最恰当可行的方案就是尽快让服务报错，坚决避免重试，尽快抛出异常，由调用者自行处理。
- **安全失败 (Failsafe)**：在一个调用链路中的服务通常也有主路和旁路之分，并不是其中每个服务都是不可或缺的，有部分服务失败了也并不影响核心业务的正确性。传统上，开发基于Spring管理的应用程序时，通过扩展点、事件或者AOP注入的逻辑往往就属于旁路逻辑，典型的有审计、日志、调试信息，等等。对属于旁路逻辑，通常后续处理不会依赖其返回值，或者它的返回值是什么都不会影响后续处理的结果（譬如只是将返回值记录到数据库，并不使用它参与最终结果的运算）。此时，一种理想的容错策略是即使旁路逻辑调用实际失败了，也当作正确来返回，如果需要返回值的话，系统就自动返回一个符合要求的数据类型的对应零值，然后再自动记录一条服务调用出错的日志备查即可，这种策略被称为安全失败。
- **沉默失败 (Failsilent)**：如果大量的请求需要等到超时（或者长时间处理后）才宣告失败，很容易由于某个远程服务的请求堆积而消耗大量的线程、内存、网络等资源，进而影响到整个系统的稳定。面对这种情况，一种合理的失败策略是当请求失败后，就默认服务提供者一定时间内无法再对外提供服务，将错误隔离开来，避免对系统其他部分产生影响，此即为沉默失败策略。
- **故障恢复 (Fallback)**：故障恢复一般不单独存在，而是作为其他容错策略的补充措施，一般在微服务管理框架中，如果设置容错策略为故障恢复的话，通常默认会采用 快速失败 加上 故障恢复 的策略组合。它是指当服务调用出错了以后，将该次调用失败的

信息存入一个消息队列中，然后由系统自动开始异步重试调用。

故障恢复策略一方面是尽力促使失败的调用最终能够被正常执行，另一方面也可以为服务注册中心和负载均衡器及时提供服务恢复的通知信息。故障恢复显然也是要求服务必须具备幂等性的，由于它的重试是后台异步进行，所以即使最后调用成功了，原来的请求也早已经响应完毕。故障恢复策略可以用于对实时性要求不高的主路逻辑，同样适合处理那些不需要返回值的旁路逻辑。为了避免在内存中异步调用任务堆积，故障恢复同样应该有最大重试次数的限制。

- 并行调用 (Forking)**：上面五种以“Fail”开头的策略是针对调用失败时如何进行弥补的，以下这两种则是在调用之前就开始考虑如何获得最大的成功概率。并行调用的策略很符合人们日常对一些重要环节进行的“双重保险”或者“多重保险”的处理思路，它是指一开始就同时向多个服务副本发起调用，只要有其中任何一个返回成功，那调用便宣告成功，这是一种在关键场景中使用更高的执行成本换取执行时间和成功概率的策略。
- 广播调用 (Broadcast)**：广播调用与并行调用是相对应的，都是同时发起多个调用，但并行调用是任何一个调用结果返回成功便宣告成功，广播调用则是要求所有的请求全部都成功，这次调用才算是成功，任何一个服务提供者出现异常都算调用失败，广播调用通常会被用于实现“刷新缓存”这类的操作。

容错策略并非计算机科学独有的，在交通、能源、航天等很多领域都有容错性设计，也会使用到上面这些策略，并在自己的行业领域中进行解读与延伸。这里介绍的容错策略并非全部，只是最常见的几种，笔者将它们各自的优缺点、应用场景总结为以下表格，供大家使用时参考：

容错策略	优点	缺点	应用场景
故障转移	系统自动处理，调用者对失败的信息不可见	增加调用时间，额外的资源开销	调用幂等服务对调用时间不敏感的场景
快速失败	调用者有对失败的处理完全控制权 不依赖服务的幂等性	调用者必须正确处理失败逻辑，如果一味只是对外抛异常，容易引起雪崩	调用非幂等的服务 超时阈值较低的场景
安全失败	不影响主路逻辑	只适用于旁路调用	调用链中的旁路服务

容错策略	优点	缺点	应用场景
沉默失败	控制错误不影响全局	出错的地方将在一段时间内不可用	频繁超时的服务
故障恢复	调用失败后自动重试，也不影响主路逻辑	推荐用于旁路服务调用，或者对实时性要求不高的主路逻辑 重试任务可能产生堆积，重试仍然可能失败	调用链中的旁路服务 对实时性要求不高的主路逻辑也可以使用
并行调用	尽可能在最短时间内获得最高的成功率	额外消耗机器资源，大部分调用可能都是无用功	资源充足且对失败容忍度低的场景
广播调用	支持同时对批量的服务提供者发起调用	资源消耗大，失败概率高	只适用于批量操作的场景

容错设计模式

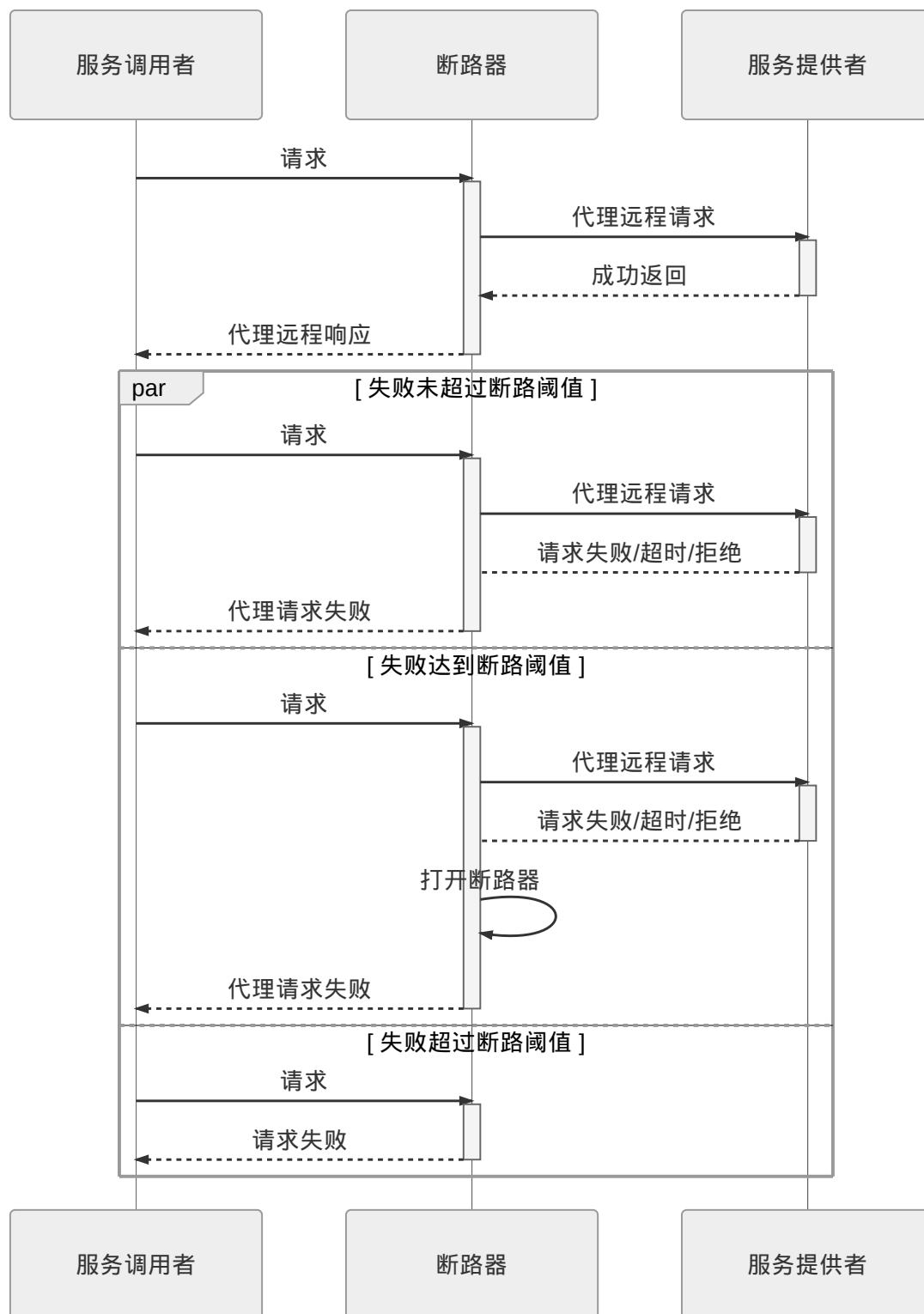
为了实现各种容错策略，开发人员总结出了一些被实践证明有效的服务容错的设计模式，譬如微服务中常见的断路器模式、舱壁隔离模式，超时重试模式等。还要将在下节介绍的流量控制模式，如滑动时间窗模式、漏桶模式、令牌桶模式，等等。

断路器模式

断路器模式是微服务架构中最常见的服务容错设计模式，以至于像Hystrix这种服务治理工具往往被人们忽略了它的服务隔离、请求合并、请求缓存等其他服务治理职能，直接将它称之为微服务断路器或者熔断器。这个设计模式最早由技术作家Michael Nygard在《Release It!》一书中提出的，后又因Martin Fowler的《Circuit Breaker》一文而广为人知。

断路器的基本思路很简单的，就是通过代理（断路器对象）来一对多地（一个远程服务对应一个断路器对象）接管服务调用者的远程请求。断路器会持续监控并统计服务返回的结果（成功、失败、超时、拒绝），当出现故障（失败、超时、拒绝）的次数达到断路器的阈值时，它状态就自动变为“OPEN”，后续此断路器代理的远程访问都将直接返回调用失败，而不会发生真正的远程服务访问。通过断路器对远程服务的熔断，避免因持续的失败或拒绝而消耗资源，因持续的超时而堆积请求，最终的效果就是避免雪崩效应的出现。由

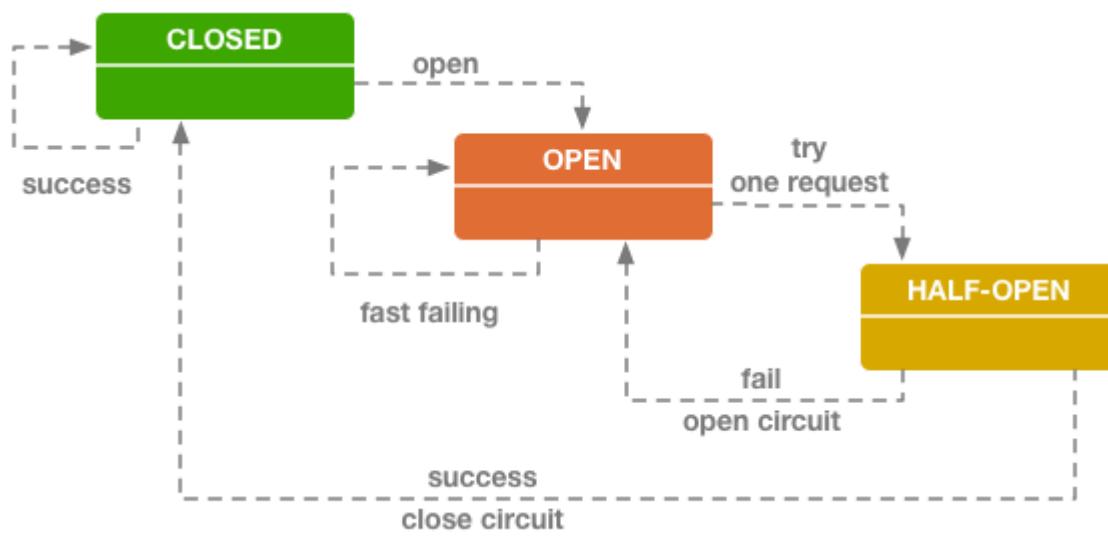
由此可见，断路器本质是一种 快速失败 策略的实现方式，它的工作过程可以通过下面序列图来表示：



从调用序列来看，断路器模式是一种根据状态机变化，自动调整代理请求策略的过程。断路器的状态一般要设置三种：

- **CLOSED**：表示断路器关闭，此时的远程请求会真正发送给服务提供者。断路器刚刚建立时默认处于这种状态，此后将持续监视远程请求的数量和执行结果，决定是否要进入OPEN状态。
- **OPEN**：表示断路器开启，此时不会进行远程请求，直接给服务调用者返回调用失败的信息，以实现 快速失败 策略。
- **HALF OPEN**：这是一种中间状态。断路器必须带有自动的故障恢复能力，当进入OPEN状态一段时间以后，将“自动”（一般是由下一次请求而不是计时器触发的，所以这里自动带引号）切换到HALF OPEN状态。该状态下，会放行一次远程调用，然后根据这次调用的结果成功与否，转换为CLOSED或者OPEN状态，以实现断路器的弹性恢复。

这些状态的转换逻辑与条件如下图所示：



断路器的状态转换逻辑 (图片来源[□](#))

OPEN和CLOSED状态的含义是清晰的，与我们日常生活中电路的断路器并没有什么差别，值得讨论的是这两者的转换条件是什么？一种最简单直接的方案是只要遇到一次调用失败，那就默认以后所有的调用都会接着失败，断路器直接进入OPEN状态，但这样做的效果是很差的，虽然避免了故障扩散和请求堆积，却使得外部看来系统将表现极其不稳定。现实中可行的办法是在以下两个条件同时满足时，断路器状态转变为OPEN：

- 条件1：一段时间（譬如10秒以内）内请求数量达到一定阈值（譬如20个请求）。这个条件的意思是如果请求本身就很少，那就用不着断路器介入。
- 条件2：这些请求的故障率（发生失败、超时、拒绝的统计比例）到达一定阈值（譬如5%）。这个条件的意思是如果请求本身都能正确返回，也用不着断路器介入。

以上两个条件同时满足时，断路器就会转变为OPEN状态。括号中举例的数值是Netflix Hystrix的默认值，其他服务治理的工具，譬如Resilience4j、Envoy等也同样会包含有类似的设计。

借着断路器的上下文，笔者顺带讲一下微服务中两个常见易混淆概念 服务熔断 和 服务降级 之间的联系与差别。断路器做的事情是自动进行服务熔断，这是一种快速失败的容错策略。而在快速失败策略（还有其他策略，如故障转移超过阈值等）明确反馈了故障给上游服务后，上游服务必须能够主动处理调用失败的后果，而不是坐视故障扩散，这里的“处理”指的是一种典型的服务降级逻辑，降级逻辑可以是但不应该全部都是把异常信息抛到用户界面去，而是应该尽力想办法通过其他路径处理问题，把原本要处理的业务记录下来留待以后重新处理是最低限度的通用降级逻辑。举个不太吉利的例子，你女朋友有事召唤你，不仅打你手机没人接，打了你另外三个不同朋友的手机号，都还是没能找到你（故障转移并超过阈值）。这时候她只能在微信上给你留言“不回电话就分手”，以此来与你取得联系（服务降级逻辑）。

服务降级不一定是在出现错误后被动执行的，多数情况下，人们所谈论的降级更可能是指需要主动迫使服务进入降级逻辑的场景。譬如，出于应对可预见的峰值流量，或者是系统检修等原因，要关闭系统部分功能或关闭部分旁路服务，这时候就有可能会主动迫使这些服务降级。当然，此时服务降级就不一定是出于服务容错的目的了，更可能属于下一节要将的讲解的流量控制的范畴。

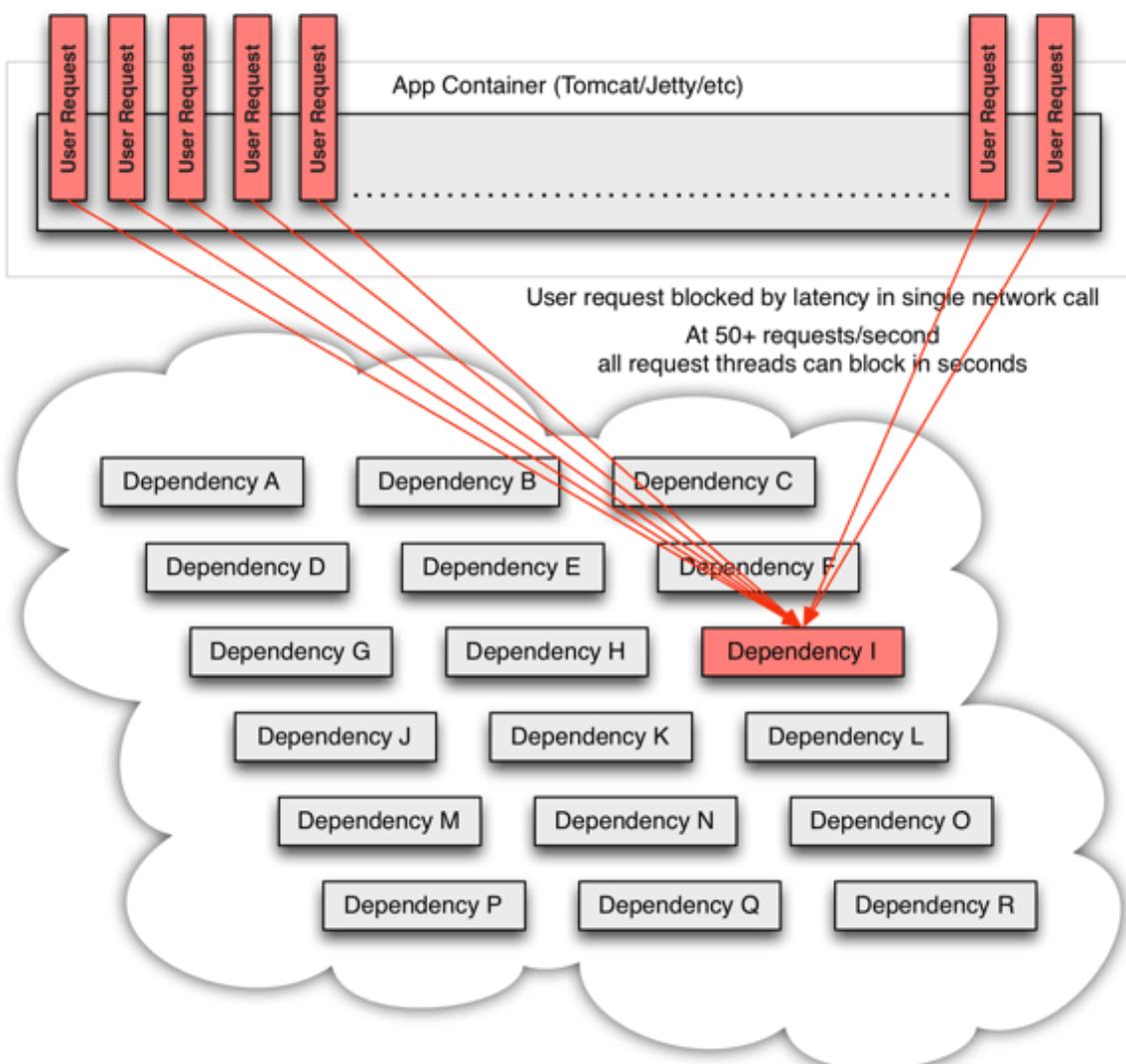
舱壁隔离模式

介绍过服务熔断和服务降级，我们再来看看另一个微服务治理中常听见的概念：服务隔离。舱壁隔离模式是常用的实现服务隔离的设计模式，舱壁这个词是来自造船业的舶来品，它原本的意思是设计舰船时，要在每个区域设计独立的水密舱室，一旦某个舱室进水，也只是影响这个舱室中的货物，而不至于让整艘舰艇沉没。这种思想就很符合容错策略中 失败静默 策略。

前面断路器中已经多次提到，调用外部服务的故障大致可以分为“失败”（如400 Bad Request、500 Internal Server Error等错误），“拒绝”（如401 Unauthorized、403 Forbidden等错误）以及“超时”（如408 Request Timeout、504 Gateway Timeout等错误）三大类，其中“超时”引起的故障尤其容易给调用者带来全局性的风险。这是由于目前主流的网络访问大多是基于TPR并发模型（Thread per Request）来实现的，只要请求一直不结束（无论

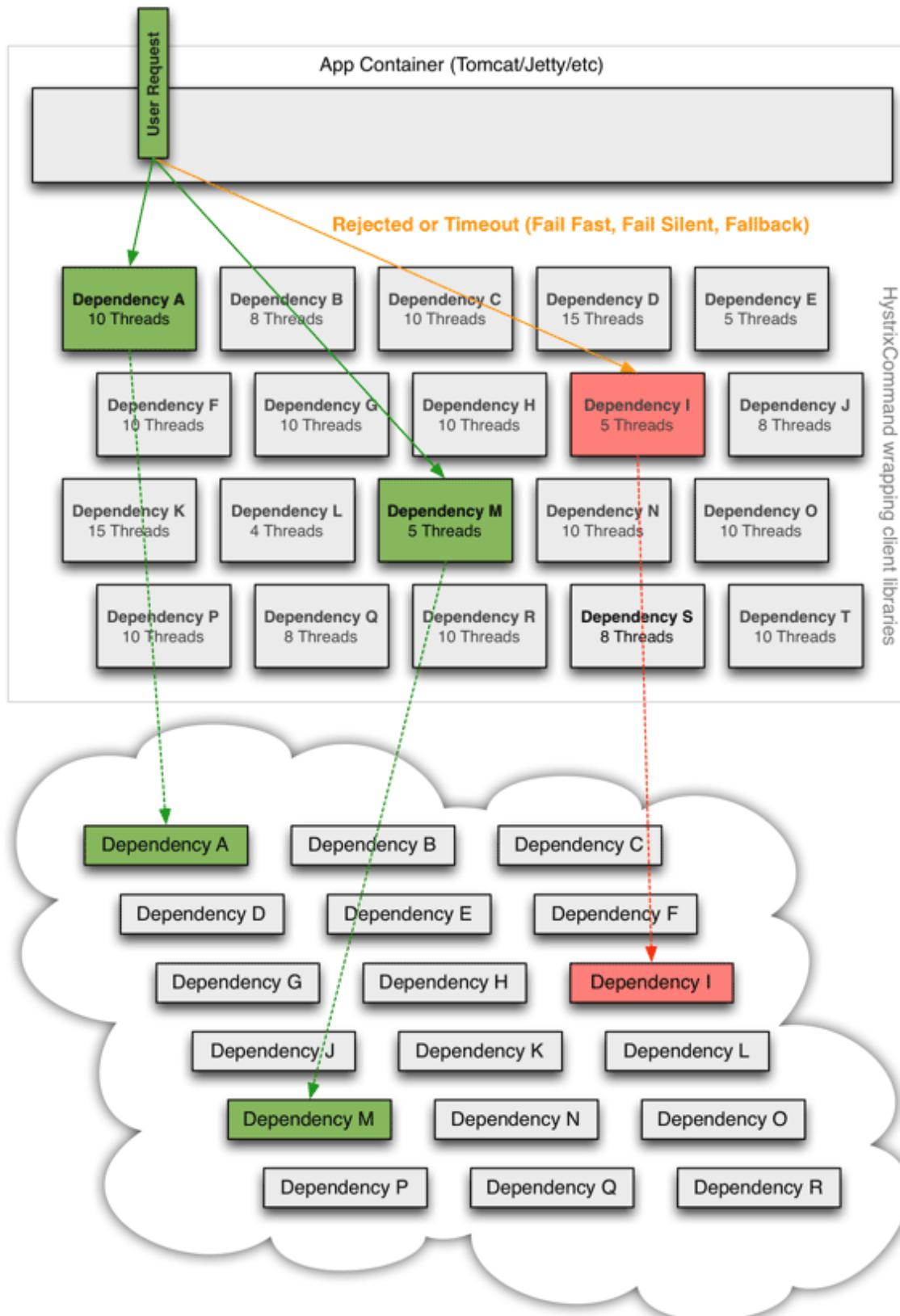
是以成功结束还是以失败结束），就要一直占用着某个线程不能释放。而线程是典型的整个系统的全局性资源，尤其是Java这类将线程映射为操作系统内核线程来实现的语言环境中，为了不让一个远程服务的局部失败演变成全局性的影响，就必须设置某种止损方案，这便是 **服务隔离** 的意义。

我们来看一个具体的场景，当分布式系统所依赖的某个服务，譬如下图中的“服务I”发生了超时，那在高流量的访问下——可以具体点，假设平均1秒钟内对该服务的调用会发生50次，这就意味着该服务不结束的话，每秒会有50条用户线程被阻塞。如果这样的访问量一直持续，我们按Tomcat默认的HTTP超时时间20秒来计算，20秒内将会阻塞掉1000条用户线程，此后才陆续会有用户线程因超时被释放出来，回归Tomcat的全局线程池中。一般Java应用的线程池最大只会设置到200至400之间，这意味着此时系统在外部将表现为全面瘫痪，而不是只有涉及到“服务I”的功能不可用，因为Tomcat已经没有任何空余的线程来为其他请求提供服务了。



由于某个外部服务导致的阻塞（[图片来自Hystrix使用文档](#)）

对于这类情况，一种可行的解决办法是为每个服务单独设立线程池，这些线程池默认不预置活动线程，只用来控制单个服务的最大连接数。譬如，对出问题的“服务I”设置了一个最大线程数为5的线程池，这时候它的超时故障就只会最多阻塞5条用户线程，而不至于影响全局。此时，其他不依赖“服务I”的用户线程依然能够正常对外提供服务。



通过线程池将阻塞限制在一定范围内（图片来自Hystrix使用文档[↗](#)）

使用局部的线程池来控制服务的最大连接数有许多好处，当服务出问题时能够隔离影响，当服务恢复后，可以通过清理掉局部线程池，瞬间恢复该服务的调用，而如果是Tomcat的

全局线程池被占满，再恢复就会十分麻烦。但是，局部线程池有一个显著的弱点，它额外增加了CPU的开销，每个独立的线程池都要进行排队、调度和下文切换工作。根据Netflix官方给出的数据，一旦启用Hystrix线程池来进行服务隔离，大概会为每次服务调用增加约3ms至10ms的延时，如果调用链中有20个外部服务，那每次请求就要多付出60ms至200ms的代价来换取服务隔离的安全保障。

另外，还有一种更轻量的可以用来控制服务最大连接数的办法：信号量机制（Semaphore）。如果不考虑清理线程池、客户端主动中断线程这些额外的功能，仅仅是为了控制一个服务并发调用的最大次数，那可以只为每个远程服务维护一个线程安全的计数器即可，并不需要建立局部线程池。具体做法是当服务开始调用时计数器加1，服务返回结果后计数器减1，一旦计数器超过设置的阈值就立即开始限流，在回落到阈值范围之前都不再允许请求了。由于不需要承担线程的排队、调度、切换工作，所以单纯维护一个作为计数器的信号量的性能损耗相对就要小很多。

以上介绍的是从微观的、服务调用的角度应用的舱壁隔离设计模式，舱壁隔离模式还可以在更高层、更宏观的角度来使用，按功能、按子系统、按用户类型等条件来隔离都是可能的，譬如，根据用户等级、用户是否VIP、用户来访的地域等各种因素，将请求分流到独立的服务实例去，这样即使某一个实例完全崩溃了，也只是影响到其中某一部分的用户，把影响尽可能降低。一般来说，我们会选择将服务层面的隔离实现在服务调用端或者边车代理上，将系统层面的隔离实现在DNS或者网关处。

重试模式

我们已经介绍了使用断路器模式实现 快速失败 策略，使用舱壁隔离模式实现 静默失败 策略。在断路器中举例的主动对非关键的旁路服务进行降级，亦可算作是对 安全失败 策略的一种体现。那还剩下 故障转移 和 故障恢复 两种策略的实现尚未涉及。接下来，笔者以重试模式来介绍这两种容错策略的主流实现方案。

故障转移 和 故障恢复 策略都需要对服务进行重复调用，差别是这些重复调用有可能是同步的，也可能是后台异步进行；有可能会重复调用同一个服务，也可能会调用到服务的其他副本。无论具体是通过怎样的方式调用、调用的服务实例是否相同，都可以归结为重试设计模式的应用范畴。

重试模式适合解决系统中的瞬时故障，简单的说就是有可能自己恢复（Resilient，自愈，也称为回弹性）的临时性失灵，网络抖动、服务的临时过载（如返回了503 Bad Gateway

错误) 这些都属于瞬时故障。重试模式实现并不困难 , 即使不考虑框架的支持 , 自己编写十几行代码也足以完成。在实践中 , 重试模式面临的风险反而大多来源于太过简单而导致的滥用。我们判断是否应该且是否能够对一个服务进行重试时 , 要同时满足以下几个前提条件 :

- 仅在主路逻辑的关键服务上进行同步的重试 , 不是关键的服务 , 一般不把重试作为首选容错方案 , 尤其不该进行同步重试。
- 仅对由瞬时故障导致的失败进行重试。尽管一个故障是否属于可自愈的瞬时故障不容易精确判定 , 但从HTTP的状态码上至少可以获得一些初步的结论 , 譬如 , 当发出的请求收到了401 Unauthorized响应 , 说明服务本身是可用的 , 只是你没有权限调用 , 这时候再去重试就没有什么意义。功能完善的服务治理工具会提供具体的重试策略配置 (如 Envoy 的 [Retry Policy](#)) , 可以根据包括HTTP响应码在内的各种具体条件来设置不同的重试参数。
- 仅对具备幂等性的服务进行重试。如果服务调用者和提供者不属于同一个团队 , 那服务是否幂等其实也是一个难以精确判断的问题 , 但仍可以找到一些总体上通用的原则。譬如 , RESTful服务中的POST请求是非幂等的 , 而GET、HEAD、OPTIONS、TRACE由于不会改变资源状态 , 这些请求应该被设计成幂等的 ; PUT请求是幂等的 , 因为n个PUT请求会覆盖相同的资源n-1次 ; DELETE也可看作是幂等的 , 同一个资源首次删除会得到200 OK响应 , 此后应该得到204 No Content响应。这些都是HTTP协议中定义的通用的指导原则 , 虽然对于具体服务如何实现并无强制约束力 , 但当我们自己建设系统时 , 遵循业界惯例也是一种良好的习惯。
- 重试必须有明确的终止条件 , 常用的终止条件有两种 :
 - 超时终止 : 无论是否进行重试 , 所有调用远程服务都应该要有超时机制避免无限期的等待。这里只是强调重试模式更加应该配合上超时机制来使用 , 否则重试对系统很可能反而是有害的 , 笔者已经在前面介绍 故障转移 策略时举过具体的例子 , 这里就不重复了。
 - 次数终止 : 重试必须要有一定限度 , 不能无限制地做下去 (通常就只试2、3次) 。它不仅会给调用者带来负担 , 对于服务提供者也是同样是负担。所以不应该将重试次数设的太大。此外 , 如果服务提供者返回的响应头中带有 [Retry-After](#) 的话 , 尽管它没有强制约束力 , 我们也应该充分尊重服务端的要求 , 做个“有礼貌”的调用者。

由于重试模式可以在网络链路的多个环节中去实现 , 譬如客户端发起调用时自动重试 , 网关中自动重试、负载均衡器中自动重试 , 等等 , 而且现在的微服务框架都足够便捷 , 往往

只许设置一两个开关参数就可以开启对某个服务甚至全部服务的重试机制。所以，对于没有太多经验的程序员，有可能根本意识不到其中会带来多大的负担。譬如，一套基于Netflix OSS建设的微服务系统，如果同时在Zuul、Feign和Ribbon上都打开了重试功能，且不考虑重试被超时终止的话，那总重试次数就相当于它们的重试次数的乘积。假设按它们都重试3次，且Ribbon可以转移3个服务副本来计算，理论上最多会产生 $3 \times 3 \times 3 = 81$ 次请求。

熔断、隔离、重试、降级、超时等概念都是建立具有韧性的微服务系统的必须的保障措施。目前，这些措施的正确运作，主要是依靠开发人员对服务逻辑的了解，以及运维人员的经验去静态调整配置参数和阈值，但是面对能够自动伸缩（Auto Scale）的大型分布式系统，静态的配置往往起不到良好的效果，这就需要系统不仅要有能力自动根据服务负载来调整服务器的数量规模，同时还要有能力根据服务调用的统计的结果，或者启发式搜索的结果来自动变更容错策略和参数，这方面研究现在还处于各大厂在内部分头摸索的初级阶段，是服务治理的未来重要发展方向之一。

本节介绍的容错策略和容错设计模式，目的均是为了避免服务集群中某个节点的故障导致整个系统发生雪崩效应，但仅仅做到容错，只让故障不扩散是远远不够的，我们还希望系统或者至少系统的核心功能能够表现出快速的响应的能力，而不受或少受硬件资源、网络带宽和系统中一两个缓慢服务的拖累。下一节，我们将面向如何解决集群中的短板效应，去讨论服务质量、流量管控等话题。

流量控制

任何一个系统的运算、存储、网络资源都不是无限的，当系统资源不足以支撑外部超过预期的突发流量时，便应该要有取舍和自我保护机制，这个机制就是微服务中常说的“限流”。在介绍限流具体细节前，我们先一起来做一道小学三年级难度的算术四则运算场景应用题：

场景应用题

已知条件：

1. 系统中一个业务操作需要调用10个服务协作来完成
2. 该业务操作的总超时时间是10秒
3. 每个服务的处理时间平均是0.5秒
4. 集群中每个服务均部署了20个实例 副本

求解以下问题：

- 单个用户访问，完成一次业务操作，需要耗费系统多少处理器时间？

答： $0.5 \times 10 = 5$ Sec **CPU Time** ↗

- 集群中每个服务每秒最大能处理多少个请求？

答： $(1 \div 0.5) \times 20 = 40$ **QPS** ↗

- 假设不考虑顺序且请求分发是均衡的，在保证不超时的前提下，整个集群能持续承受最多每秒多少笔业务操作？

答： $40 \times 10 \div 5 = 80$ **TPS** ↗

- 如果集群在一段时间内持续收到100 TPS的业务请求，会出现什么情况？

答：这就超纲了小学水平，得看你们家架构师的本事了。

对于最后这个问题，如果仍然按照小学生的解题思路，最大处理能力为80 TPS的系统遇到100 TPS的请求，应该能完成其中的80 TPS，也即是只有20 TPS的请求失败或被拒绝才对，这其实是最理想的情况，也是我们追求的目标。事实上，如果不做任何处理的话，更可能出现的结果是这100个请求中的每一个都开始了处理，但是大部分请求完成了其中10次服务调用中的8次或者9次，然后就超时没有然后了。即多数服务调用都白白浪费掉，没

有几个请求能够走完整笔业务操作。譬如早期的12306系统就明显存在这样的问题，全国人民都上去抢票的结果是全国人民谁都买不上票。为了避免这种状况出现，一个健壮的系统需要做到恰当的流量控制，更具体地说，需要妥善解决以下三个问题：

- **依据什么限流？**：要不要控制流量，要控制哪些流量，控制力度要有多大，等等这些操作都没法在系统设计阶段给出确定的结论，必须根据系统此前一段时间的运行状况来动态决定。
- **具体如何限流？**：解决系统具体是如何做到允许一部分请求能够通行，另外一部分流量需要进行控制的问题，这必须了解掌握常用的服务限流算法和设计模式。
- **超额流量如何处理？**：超额流量可能会直接返回失败（如429 Too Many Requests），或者被迫使它们进入降级逻辑，这种被称为否决式限流。也可能让请求排队等待，暂时阻塞一段时间后继续处理，这种被称为阻塞式限流。

流量统计指标

要做流量控制，首先要弄清楚到底哪些指标能反映系统的流量压力大小。相较而言，容错的统计指标是明确的，容错的触发条件基本上只取决于请求的故障率，发生失败、拒绝与超时都算作故障；但限流的统计指标就不那么明确了，限流中的“流”到底指什么呢？要解答这个问题，我们先来理清经常用于衡量服务流量压力，但又较容易混淆的三个指标的定义：

- **每秒事务数**（Transactions per Second，TPS）：TPS是衡量信息系统吞吐量的最终标准。“事务”可以理解为一个逻辑上具备原子性的业务操作。譬如你在Fenix's Bookstore买了一本书要进行支付，“支付”就是一笔业务操作，支付无论成功还是不成功，这个操作在逻辑上是原子的，即逻辑上不可能让你买本书还成功支付了前面200页，又失败了后面300页。
- **每秒请求数**（Hits per Second，HPS）：HPS是指每秒从客户端发向服务端的请求数（请将Hits理解为Requests而不是Clicks，国内某些翻译把它理解为“每秒点击数”就有点望文生义的嫌疑了）。如果只要一个请求就能完成一笔业务，那HPS与TPS是等价的，但在一些场景（尤其常见于网页中）里，一笔业务可能需要多次请求才能完成。譬如你在Fenix's Bookstore买了一本书要进行支付，尽管逻辑上它是原子的，但技术实现上，除非你是直接在银行开的商城中购物能够直接扣款，否则这个操作就很难在一次请求里

完成，总要经过显示支付二维码、扫码付款、校验支付是否成功等过程，中间不可避免地会发生多次请求。

- **每秒查询数** (Queries per Second, QPS)：QPS是指一台服务器能够响应的查询次数。如果只有一台服务器来应答请求，那QPS和HPS是等价的，但在分布式系统中，一个请求的响应往往要由后台多个服务节点共同协作来完成。譬如你在Fenix's Bookstore买了一本书要进行支付，扫描支付二维码时尽管客户端只发送了一个请求，但这背后服务端很可能需要向仓储服务确认库存信息避免超卖、向支付服务发送指令划转货款、向用户服务修改用户的购物积分，等等，这里面每次内部访问都要消耗掉一次或多次查询数。

以上这三点都是基于调用计数的指标，在整体目标上我们当然最希望能够基于TPS来限流，因为信息系统最终是为人类用户来提供服务的，用户不关心业务到底是由多少个请求、多少个后台查询来实现。但是，系统的业务五花八门，不同的业务操作对系统的压力往往差异巨大，不具备可比性；而更关键的是，流量控制是针对用户实际操作场景来限流的，这不同于压力测试场景中无间隙（最多有些集合点）的全自动化操作，真实业务操作的耗时无可避免地受限于用户交互带来的不确定性，譬如前面例子中的“扫描支付二维码”这个步骤，如果用户掏出手机扫描二维码前先顺便回了两条短信息，那整个付款操作就要持续更长时间。此时，如果按照业务开始时计数器加1，业务结束时计数器减1，通过限制最大TPS来限流的话，就不能准确地反应出系统所承受的压力，所以直接针对TPS来限流实际上是很困难操作的。

目前，主流系统大多倾向使用HPS作为首选的限流指标，它是相对容易观察统计的，而且能够在一定程度上反应系统当前以及接下来一段时间的压力。但限流指标并不存在任何必须遵循权威法则，根据系统的实际需要，哪怕完全不选择基于调用计数的指标都是有可能的。譬如下载、视频、直播等I/O密集型系统，往往会把每次请求和响应报文的大小而不是调用次数作为限流指标，如只允许单位时间通过100MB的流量。又譬如网络游戏等基于长连接的应用，可能会把登陆用户数作为限流指标，如热门的网游往往超过一定用户数就会让你在登陆前排队等候。

限流设计模式

与容错模式类似，对于具体如何进行限流，也有一些常见常用的设计模式可以参考使用，本文将介绍 流量计数器、滑动时间窗、漏桶 和 令牌桶 四种限流设计模式。

流量计数器模式

做限流最容易想到的一种方法就是设置一个计算器，根据当前时刻的流量计数结果是否超过阈值来决定是否限流。譬如前面场景应用题中，我们计算得出了该系统能承受的最大持续流量是80 TPS，那就控制任何一秒内，超过80 TPS的请求就直接被拒绝掉。这种做法很直观，也确实有一些简单的限流就是这么实现的，但它并不严谨，以下两个结论就很可能出乎对限流算法没有了解的同学意料之外：

- 即使每一秒的统计流量都没有超过80 TPS，也不能说明系统没有遇到过大于80 TPS的流量压力。

你可以想像如下场景，如果系统连续两秒都收到60 TPS的访问请求，但这两个60 TPS请求分别是前1秒里面的后0.5秒，以及后1秒中的前面0.5秒所发生的。这样虽然每个周期的流量都不超过80 TPS请求的阈值，但是系统确实曾经在1秒内实实在在发生了超过阈值的120 TPS请求。

- 即使连续若干秒的统计流量都超过了80 TPS，也不能说明流量压力就一定超过了系统的承受能力。

你可以想像如下场景，如果10秒的时间片段中，前3秒TPS到了100，而后7秒的平均值是30左右，此时系统是否能够处理完这些请求而不产生超时失败？答案是可以的，因为条件中给出的超时时间是10秒，而最慢的请求也能在8秒左右处理完毕。如果只基于固定时间周期来控制请求阈值为80TPS，反而会误杀一部分请求，造成部分请求出现原本不必要的失败。

流量计数器的缺陷根源在于它只是针对时间点进行离散的统计，为了弥补该缺陷，一种名为“滑动时间窗”、可以实现平滑的基于时间片段统计的限流模式被设计出来。

滑动时间窗模式

[滑动窗口算法](#) (Sliding Window Algorithm) 在计算机科学的很多领域中都有成功的应用，譬如编译原理中的[窥孔优化](#) (Peephole Optimization)、TCP协议的[阻塞控制](#) (Congestion Control) 等都使用到滑动窗口算法。在分布式服务中，无论是服务容错中对服务响应结果的统计，还是流量控制中对服务请求数量的统计，都经常要用到滑动窗口算法。关于这个算法的运作过程，不妨先想像以下场景：在不断向前流淌的时间轴上，漂浮

着一个固定大小的窗口，窗口与时间一起平滑地向前滚动。任何时刻静态地通过窗口内观察到的信息，都等价于一段长度与窗口大小相等、动态流动中时间片段的信息。由于窗口观察的目标都是时间轴，所以它被称为形象地称为“滑动时间窗模式”。

举个具体的例子，假如我们准备观察时间片段为10秒，并以1秒为统计精度的话，那可以设定一个长度为10的数组（一般是以双头队列去实现，这里简化一下）和一个每秒触发1次的定时器。假如我们准备通过统计结果进行限流和容错，并定下限流阈值是最近10秒内收到的外部请求不要超过500个，服务熔断的阈值是最近10秒内故障率不超过50%，那每个数组元素（图中称为Buckets）中就应该存储请求的总数（实际是通过明细相加得到）及其中成功、失败、超时、拒绝的明细数，具体如下图所示。

Success	23	47	26	48	38	42	59	46	39	12
Failure	5	8	4	9	4	6	11	5	3	1
Timeout	2	1	0	4	2	7	5	2	5	0
Rejection	0	0	0	0	0	0	1	0	0	0

23	47	26	48	38	42	59	46	39	12	1
5	8	4	9	4	6	11	5	3	1	0
2	1	0	4	2	7	5	2	5	0	0
0	0	0	0	0	0	1	0	0	0	0

滑动窗口模式示意（[图片来自Hystrix使用文档](#)）

文中虽然引用了Hystrix文档的图片，但Hystrix实际上是基于RxJava实现的，RxJava的响应式编程思路与下面描述差异颇大。笔者的本意并不是去讨论某一款流量治理工具的具体实现细节，以下描述的步骤作为原理来理解是合适的。

当频率固定每秒一次的定时器被唤醒时，它应该完成以下几项工作，也即是滑动时间窗的工作过程：

1. 将数组最后一位的元素丢弃掉，并把所有元素都后移一位，然后在数组第一个插入一个新的空元素。这个步骤即为“滑动窗口”。
2. 将计数器中所有统计信息写入到第一位的空元素中。
3. 对数组中所有元素进行统计，并复位清空计数器数据供下一个统计周期使用。

滑动时间窗口模式的限流完全解决了流量计数器的缺陷，可以保证任意时间片段内，只需经过简单的调用计数比较，就能控制住请求次数一定不会超过限流的阈值，在单机限流或者分布式服务单点网关中的限流中很常用。不过，这种限流也有其缺点，它通常只适用于

否决式限流，超过阈值的流量就必须强制失败或降级，很难进行阻塞等待处理，也就很难在细粒度上对流量曲线进行整形，起不到削峰填谷的作用。下面笔者介绍两种适用于阻塞式限流的限流模式。

漏桶模式

在计算机网络中，专门有一个术语[流量整形](#)（Traffic Shaping）用来描述如何限制网络连接的流量突变，使得网络报文以比较均匀的速度向外发送。流量整形通常都需要用到缓冲区来实现，当报文的发送速度过快时，首先在缓冲区中暂存，然后再在控制算法的调节下均匀地发送这些被缓冲的报文。常用的控制算法有[漏桶算法](#)（Leaky Bucket Algorithm）和[令牌桶算法](#)（Token Bucket Algorithm）两种，这两种算法的思路截然相反，但达到的效果又是相似的。

所谓漏桶，就是大家小学做应用题时一定遇到过的“一个水池，每秒以X升速度注水，同时又以Y升速度出水，问水池啥时候装满”的那个奇怪的水池。可以把请求想像是水，水来了都先放进池子里，水池同时又以额定的速度出水，让请求进入系统中。这样，如果一段时间内注水过快时，水池还能充当缓冲区，让出水口的速度不至于过快。不过由于请求总是有超时时间，所以缓冲区大小也必须是有限度的，当注水速度持续超过出水速度一段时间后，水池终究会被灌满，此时，从网络的流量整形的角度看是体现为部分数据包被丢弃，而在信息系统的角度看就体现为有部分请求会遭遇失败和降级。

漏桶在代码实现上非常简单，它其实就是一个以请求对象作为元素的先入先出队列（FIFO Queue），队列长度就相当于漏桶的大小，当队列已满时便拒绝新的请求进入。漏桶实现起来很容易，困难在于如何确定漏桶的两个参数：桶的大小和水的流出速率。如果桶设置得太大，那服务依然可能遭遇到流量过大的冲击，起不到限流的作用；如果设置得太小，那很可能就会误杀掉一部分正常的请求（情况如流量计数器模式中举的例子）。流出速率在漏桶算法中一般是个固定值，这对如本文场景应用题中那样的固定拓扑结构的服务是合适的，但同时也应该明白那是经过最大限度简化的场景，现实中系统的处理速度往往受到其内部拓扑结构变化和动态伸缩的影响，所以能够支持变动请求处理速率的令牌桶算法往往可能会是更受青睐的选择。

令牌桶模式

如果说漏桶是小学应用题中的奇怪水池，那令牌桶就是你去银行办事时摆在门口的那台排队机。它与漏桶一样都是基于缓冲区的限流算法，只是方向刚好相反，漏桶是从水池里往系统出水，令牌桶则是系统往排队机中放入令牌。

假设我们要限制系统在X秒内最大请求次数不超过Y，那就每间隔 X/Y 时间就往桶中放一个令牌，当有请求进来时，首先从桶中拿走一个令牌，然后再进入系统处理。任何时候，一旦请求进入桶中却发现没有令牌可取了，就应该马上失败或进入服务降级逻辑。与漏桶类似，令牌桶同样有最大容量，这意味着当系统比较空闲时，桶中令牌累积到一定程度就不再无限增加，预存在桶中的令牌便是请求最大缓冲的余量。上面这段话，可以转化为以下步骤来指导程序编码：

1. 让系统以由限流目标决定的速率（如要控制系统的访问不超过100次，即 $1/100=10$ 毫秒）向桶中注入令牌。
2. 桶中最多可以存放N个令牌，N的具体数量是由超时时间和服务能力共同决定的。如果桶已满，第N+1个令牌进入的令牌会被丢弃。
3. 请求到时先从桶中取走1个令牌，如果桶已空就进入降级逻辑。

令牌桶模式的实现看似比较复杂，每间隔固定时间就要放新的令牌到桶中，但其实并不需要真的用一个专用线程或者定时器来做这件事情，只要在令牌中增加一个时间戳记录，每次获取令牌前，比较一下时间戳与当前时间，就可以轻易计算出这段时间需要放多少令牌进去，然后一次过放完即可，所以真正编码也并不复杂。

分布式限流

现在，我们再向实际的信息系统前进一步，讨论分布式系统中的限流问题。此前，我们讨论的限流算法和模式全部是针对整个系统的限流，总是有意无意地假设或默认系统只提供一种业务操作，或者所有业务操作的消耗都是等价的，并不涉及到不同业务请求进入系统的服务集群后，分别会调用哪些服务、每个服务节点处理能力有何差别等问题。前面讨论过的那些限流算法，直接使用在单体架构的集群上是完全可行的，但到了微服务架构下，它就最多只能应用于集群最入口处的网关上，对整个服务集群进行流量控制，而无法细粒度地管理流量在内部微服务节点中的流转情况。所以，我们把前面介绍的限流模式都统称为单机限流，把能够精细控制分布式集群中每个服务消耗量的限流算法称为分布式限流。

这两种限流算法实现上的核心差别在于如何处理限流的统计指标，单机限流很好办，指标都是存储在服务的内存当中，而分布式限流目的就是要让各个服务节点的协同限流，无论是将限流功能封装为专门的远程服务，抑或是在系统采用的分布式框架中有专门的限流支持，都需要将原本在每个服务节点自己内存当中的统计数据给开放出来，让全局的限流服务可以访问到。

一种常见的简单分布式限流办法是将所有服务的统计结果都存入集中式缓存（如Redis）中，以实现在集群内的共享，并通过分布式锁、信号量等机制，解决这些数据的读写访问时并发控制的问题。在可以共享统计数据的前提下，原本本地的限流模式理论上也是可以应用于分布式环境中的，可是其代价显而易见：每次服务调用都必须要额外增加一次网络开销，所以这种方法的效率肯定是不高的，流量压力大时，限流本身反倒会显著降低系统的处理能力。

只要集中式存储统计信息，就不可避免地会产生网络开销，为了缓解这里产生的性能损耗，一种可以考虑的办法是在令牌桶限流模式基础上进行“货币化改造”改造，即不把令牌看作是布尔形式的“通行证”，而看作是数值形式的“货币额度”。当请求进入集群时，首先在API网关处领取到一定数额的“货币”，为了体现不同等级用户重要性的差别，这里额度可以有所差异，譬如让VIP用户的额度是更高甚至是无限。我们将用户A的额度表示为 $Quanity_A$ 。由于任何一个服务访问时都需要消耗集群一定量的处理资源，所以访问每个服务时都要求消耗一定量的“货币”，假设服务X要消耗的额度表示为 $Cost_X$ ，那当用户A访问了N个服务以后，他剩余的额度 $Limit_N$ 即表示为：

$$Limit_N = Quanity_A - \sum^N Cost_X$$

此时，我们可以把剩余额度 $Limit_N$ 作为内部限流的指标，规定在任何时候，当剩余额度 $Limit_N$ 小于等于0时，就不再允许访问其他服务了。此时必须先发生一次网络请求，重新向令牌桶申请一次额度，成功后才能继续访问。除此之外的任何时刻，即 $Limit_N$ 不为零时，都无需额外的网络访问，因为计算 $Limit_N$ 是完全可以在本地完成的。基于额度的限流方案对限流的精确度有一定的影响，可能存在业务操作已经进行了一部分服务调用，却无法从令牌桶中再获取到新额度，因“资金链断裂”而导致业务操作失败。这种失败的代价是比较高昂的，它白白浪费了部分已经完成了的服务资源，但总体来说，它仍是一种并发性能和限流效果上都相对折衷可行的分布式限流方案。上一节提到过，对于分布式系统容错是必须要有、无法妥协的措施。但限流与容错不一样，做分布式限流从不追求“越彻底越好”，往往需要权衡方案的代价与收益。

可靠通讯

微服务提倡分散治理，不提倡追求统一的技术平台，提倡让团队有自由选择的权利。在开发阶段构建服务时，分散治理打破了由技术栈带来的约束界限，好处是不言自明的。但在运维阶段考虑安全问题时，由Java、Golang、Python、Node等多种语言和框架组成的微服务系统，出现安全漏洞的概率必然要比只采用其中某种语言、某种框架的构建的单体系统来的更高。为了避免由于单个微服务节点出现漏洞被攻击者突破，进而导致整个系统和内网都遭到入侵，我们必须改变一部分在此前的安全观念，构筑更加可靠的服务间通讯机制。

零信任网络

长期以来，主流的网络安全观念是根据某类特征（如机器所处的网络位置、IP/MAC地址等）把网络划分为不同的区域，不同的区域对应着不同风险级别和允许访问的网络资源，将安全防护措施集中部署在各个区域的边界之上，我们熟知的VPN、DMZ、防火墙、内外网等概念都是由此而生的，这种安全模型今天被称为是 基于边界的安全模型（Perimeter-Based Security Model）。

“边界安全”是很合理的想法，因为安全不可能是绝对的，我们必须在可用性和安全性之间权衡取舍，否则一台关掉电源不能对外提供服务的“服务器”无疑就是最为安全的。边界安全着重对经过网络区域边界的流量进行检查，对可信任区域（内网）内部机器之间的流量则给予直接信任或者至少是较为宽松的处理，以减小安全设施对整个应用系统复杂度的影响，以及网络传输性能的额外损耗，这当然是很合理的。不过，今天单纯的边界安全已不足以满足大规模微服务系统的技术异构和节点膨胀的发展需要。边界安全的核心问题在于边界上的防御措施即使自身能做到永远滴水不漏牢不可破，也很难保证内网中它所尽力保护的某一台服务器不会成为“猪队友”，一旦“可信的”网络区域中的某台服务器被攻陷，那边界安全措施就成了马其诺防线，攻击者很快就能以一台机器为跳板，侵入到整个内网，这是边界安全基因决定的固有的缺陷，从边界安全提出的第一天这就是已预料到的问题。在微服务时代，我们已经转变开发观念，承认服务了总会出错，现在我们也必须转变安全观念，承认一定会有被攻陷的服务，为此我们需要寻找到与之符合的新的网络安全模型。

2010年，Forrester Research¹的首席分析师John Kindervag提出了 零信任安全模型 的概念（Zero-Trust Security Model，最初提出时被称为“Zero-Trust Architecture”，即零信任架构），这个概念当时并没有引发太大的关注，但随着微服务架构的兴起，越来越多的开发、运维人员注意到零信任安全模型与微服务所追求的安全目标完全吻合。

“零信任安全”的中心思想是不应当以某种特征来自动信任任何流量，除非明确知道请求者（请求者不一定是人，更可能是另一台服务）的身份和授权情况，否则一律不会有默认的信任关系。在2019年，谷歌发表了一篇在安全与研发领域里都备受关注的论文《BeyondProd: A New Approach to Cloud-Native Security²》（BeyondProd是谷歌安全框架的名字，从2014年起已连续发表了6篇关于BeyondProd的论文），此文中列举了传统的基于边

界的网络模型与云原生时代下基于零信任网络的安全模型之间的差异，并描述了要完成边界安全模型到零信任安全模型的迁移所需实现的具体需求点，具体如下表所示。

传统、边界安全模型	云原生、零信任安全模型	具体需求
基于防火墙等设施，认为边界内可信	服务到服务通信需认证，环境内的服务之间默认没有信任	保护网络边界（仍然有效）；服务之间默认没有互信
用于特定的IP和硬件（机器）	资源利用率、重用、共享更好，包括IP和硬件	受信任的机器运行来源已知的代码
基于IP的身份	基于服务的身份	同上
服务运行在已知的、可预期的服务器上	服务可运行在环境中的任何地方，包括私有云/公有云混合部署	同上
安全相关的需求由应用来实现，每个应用单独实现	由基础设施来实现，基础设施中集成了共享的安全性要求。	集中策略实施点（Choke Point s），一致地应用到所有服务
对服务如何构建、评审、实施的安全需求的约束力较弱	安全相关的需求一致地应用到所以服务	同上
安全组件的可观测性较弱	有安全策略及其是否生效的全局视图	同上
发布不标准，发布频率较低	标准化的构建和发布流程，每个微服务变更独立，变更更频繁	简单、自动、标准化的变更发布流程
工作负载通常作为虚拟机部署或部署到物理主机，并使用物理机或管理程序进行隔离	封装的工作负载（就是Pod）及其进程在共享的操作系统中运行，并有管理平台提供的某种机制来进行隔离	在共享的操作系统的工作负载之间进行隔离

两种安全模型的对比以及其中引发的需求

这个表格其实相当于系统地阐述了零信任安全在微服务、云原生环境中的具体落地过程了，后续的整篇论文（除了介绍谷歌自己的实现框架外）就是以此为主线来展开论述的，由于论文原文写的较为分散晦涩，笔者对其中主要观点按照自己的理解转述如下：

- **零信任不等同于放弃在边界上保护网络**：虽然防火墙等位于网络边界的设施是边界安全而非零信任安全中的概念，但它仍然是一种行之有效的提升安全性的做法。在微服务集群的前端部署防火墙，把内部服务节点间的流量与来自互联网的网络攻击和未经授权的流量隔离开来依然是值得提倡的，这能够使其避开来自互联网的未经授权的流量和潜在的攻击，如最典型的[DDOS拒绝服务攻击](#)。
- **身份只来源于服务**：以前传统应用都是部署在特定的服务器上，这些机器的IP、MAC地址很少发生变化，此时在安全角度来看系统是相对静态的。基于这个前提，安全策略才使用IP地址、主机名等作为身份标识符（Identifiers）。如今在微服务尤其是云原生环境中，虚拟化基础设施被大范围应用，这使得服务部署的IP、数量等随时都可能发生改变，因此，身份只能来源于服务本身，而不是服务所在的IP地址、主机名或者其他元素。
- **服务之间也没有固有的信任关系**：只有已知的、明确授权的调用者才能访问服务。这样可以阻止攻击者通过某个服务节点中的代码漏洞来越权调用到其他服务。如果某个服务节点被成功入侵，这一原则可阻止攻击者执行扩大其入侵范围，与微服务设计模式中使用断路器、舱壁隔离实现容错来避免雪崩效应类似，在安全方面也应当采用这种“互不信任”的模式来隔离入侵危害的影响面。
- **集中、共享的安全策略实施点**：这点与微服务的“分散治理”刚好相反，微服务提倡每个服务自己独立的负责自身的功能性、非功能性需求。而谷歌这个观点相当于为分散治理原则做了一个补充——涉及安全的非功能性需求（如身份管理、安全传输层、数据安全层）最好除外。一方面，要写出高度安全的代码极为不易，为此付出的精力甚至可能远高于业务逻辑本身，如果你有兴趣阅读[基于Spring Cloud的Fenix's Bookstore的源码](#)，很容易就会发现在Security工程中的代码量是所有微服务中最多的。另一方面，而且还是更重要的一个方面是让服务各自处理安全问题很容易会出现实现不一致或者出现问题时要修改多处地方，甚至有些安全问题如果不立足于全局是很难彻底解决的（下一节面向于具体操作的“[服务安全](#)”中将会详细说明），因此谷歌明确提出应该有集中式的“安全策略实施点”（原文中称之为Choke Points），安全需求应该从微服务的应用代码下沉至云原生的基础设施里，这也契合其论文的标题“Cloud-Native Security”。
- **受信的机器运行来源已知的代码**：限制了服务只能使用认证过的代码和配置，并且只能运行在认证过的、验证过的环境中。
- **自动化、标准化的变更管理**：这点也是为何提倡通过基础设施而不是应用代码去实现安全功能的另一个重要理由。如果将安全放在应用上，由于应用本身的分散治理，这决定了安全也必然是难以统一和标准化的。做不到标准化就意味着做不到自动化，相反，一

套独立于应用的安全基础设施，可以让运维人员轻松地了解基础设施变更对安全性的影
响，并且可以在几乎不影响生产环境的情况下发布安全补丁程序。

谷歌认为零信任安全网络的最终目标是实现整个基础设施之上的自动化安全控制，服务所
需的安全能力可以与服务自身一起，以相同方式自动进行伸缩扩展。对于服务来说，做到
安全是日常，风险是例外（Secure by Default and Insecure by Exception），对于人类来
说，做到袖手旁观是日常，主动干预是例外（Human Actions Should Be by Exception, No
t Routine），这的确是很美好的愿景，但问题在于其代价是什么？代价有多大？很显然，
零信任网络模型之所以在今天才真正严肃地讨论，并不是因为它本身有多么巧妙、有什么
此前没有想到的好办法，而在于前文中提到的边界安全模型的“合理之处”，即“安全设施对
整个应用系统复杂度的影响，以及网络传输性能的额外损耗”。以前的软件开发架构是很承
受担零信任安全模型的代价的，只有在云原生时代，虚拟化的基础设施长足发展，可以将
复杂性隐藏于基础设施之内，虚拟化的网络性能足够高，可以弥补安全通讯的额外损耗的
前提下，零信任网络的安全模型才有它生根发芽的土壤。

综上，谷歌所诠释的云原生下的零信任安全，在引入比边界安全更细致更复杂安全措施的
同时，也强调着自动化的重要性。换句话说，是既要保证系统中各个微服务之间安全通
讯，同时也不削弱微服务架构本身的属性，如集中式的安全并不抵触于分散治理原则，安
全机制并不影响服务的自动伸缩和有效的封装，等等。总而言之，所有这些都不是“大
饼”，都可以实现，而且不会在底层基础架构的安全性和实现细节方面给微服务开发者带来
额外的负担。

如何构建零信任网络安全是一个非常大而且比较新的话题，受文字篇幅所限，在这里就不
做更多的展开了。下一节，笔者将从实践角度出发，介绍在前微服务时代（以Spring Clou
d为例）和后微服务时代（即服务网格架构，以Kubernetes with Istio为例），具体是如何
实现安全传输、认证和授权的，通过这两者的对比，我们能够更具体、更量化地体会到本
文中所说的零信任安全模型的价值与权衡。

服务安全

前置知识

本文涉及到SSL/TLS、PKI、CA、OAuth2、RBAC、JWT等概念，直接依赖于“安全架构”部分的[认证](#)、[授权](#)、[凭证](#)、[传输](#)四节对安全基础知识的铺垫，这四篇文章中介绍过的内容，笔者在本篇中将不再重复，建议读者先行阅读。

在“安全架构”部分，我们了解过那些跟具体架构形式无关的、业界主流的安全概念和技术标准，在上一节“零信任网络”里，我们立足于微服务架构，探讨了与微服务运作特点相适应的安全模型。在本文中，我们将从实践和编码的角度出发，探讨在微服务架构下，如何将业界的安全技术标准引入并实际落地，实现零信任网络下安全的服务访问。

建立信任

零信任网络里不存在默认的信任关系，一切服务调用、资源访问成功与否，均需以调用者与提供者间已建立的信任关系为前提。此前我们曾讨论过，真实世界里，能够达成信任的基本途径不外乎[基于共同私密信息的信任](#) 和 [基于权威公证人的信任](#) 两种；网络世界里，因为客户端和服务端之间并没有什么共同私密信息，所以真正能采用的就只能是基于权威公证人的信任，它有个标准的名字：[公开密钥基础设施](#)（Public Key Infrastructure，PKI）。

PKI是构建[传输安全层](#)（Transport Layer Security，TLS）的必要基础。在任何网络设施都不可信任的假设前提下，无论是DNS、代理服务器、负载均衡器还是路由器，传输路径上的每一个节点都有可能监听或者篡改通讯双方传输的信息。要保证通讯过程不受到中间人攻击的威胁，启用TLS对传输通道本身进行加密，让发送者发出的内容只有接受者可以解密是唯一具备可行性的方案。建立TLS传输，说起来似乎不复杂，只要在部署服务器时预置好CA根证书，以后用该CA为部署的服务签发TLS证书便是。但落到实际操作上，这种事情就属于典型的“必须集中在基础设施中自动进行的安全策略实施点”，面对数量庞大且能够自动扩缩的服务节点，依赖运维人员人工去部署和轮换根证书必定是难以为继的。除了服务节点扩缩与数量带来的复杂性外，微服务中TLS认证的频次也会显著高于传统的应

用，比起公众互联网中主流单向的TLS认证，在零信任网络中，往往要启用[双向的TLS认证](#)（ Mutual TLS，常简写为mTLS ），即不仅要确认服务端的身份，还需要确认调用者的身份。

- **单向TLS认证**：只需要服务端提供证书，客户端通过服务端证书验证服务器的身份，但服务器并不验证客户端的身份。单向TLS用于公开的服务，即任何客户端都被允许连接到服务进行访问，它保护的是客户端免遭冒牌服务器的欺骗。
- **双向TLS认证**：客户端、服务端双方都要提供证书，双方各自通过对方提供的证书来验证对方的身份。双向TLS用于私密的服务，即服务只允许特定身份的客户端访问，它除了保护客户端不连接到冒牌服务器外，也保护服务端不遭到非法用户的越权访问。

对于以上提到的围绕TLS而展开的密钥生成、证书分发、[签名请求](#)（ Certificate Signing Request，CSR ）、更新轮换等是一套非常繁琐的流程，稍有疏忽就会产生安全漏洞，所以尽管理论上可行，但实践中如果没有自动化的基础设施的支持，仅靠应用程序和运维人员的努力，是很难成功实施零信任安全模型的。

关于自动化基础设施在密钥、证书等方面该提供哪些支持、具体是如何运作的，我们还会在“不可变基础设施”中继续去探讨，这里先聚焦“认证”和“授权”两个最基本的安全需求，看它们在微服务架构下，有或者没有基础设施支持时，各是如何实现的。

认证

根据认证的目标对象可以把认证分为两种类型，一种是以机器作为认证对象，即访问服务的流量来源是另外一个服务，称为服务认证（ Peer Authentication ）；另一种是以人类作为认证对象，即访问服务的流量来自于最终用户，称为请求认证（ Request Authentication ）。无论哪一种认证，无论是否没有基础设施的支持，均需要有可行的方案来确定服务调用者的身份，建立起信任关系才能调用服务，我们先从服务认证说起。

服务认证

Istio版本的Fenix's Bookstore采用了双向TLS认证作为服务调用双方的身份认证手段。得益于Istio提供的基础设施的支持，我们无需改动任何代码就可以启用mTLS认证。不过，如果你准备在自己的生产系统中启用mTLS，要先想一下是否整个服务集群都受Istio管理？如果每一个服务提供者、调用者均受Istio管理，那mTLS就是最理想的认证方案，没有之一。你

只需要参考以下简单的PeerAuthentication CRD配置，即可对某个Kubernetes名空间范围内所有的流量均启用mTLS：

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: authentication-mtls
  namespace: bookstore-servicemesh
spec:
  mtls:
    mode: STRICT
```

yaml

如果你的分布式系统中存在不受Istio管理（即未注入Sidecar）的服务端或者客户端——这其实在大型系统中是颇为常见的，你也可以将传输声明为“宽容模式”（Permissive Mode）。宽容模式的含义是受Istio管理的服务会允许同时接受纯文本和mTLS两种流量，纯文本流量仅用于与那些不受Istio管理的节点进行交互，你需要自行想办法解决纯文本流量的认证问题；而对于服务网格内部的流量，就可以使用mTLS认证。宽容模式为普通微服务向服务网格迁移提供了极大的灵活性，运维人员能够逐个服务进行mTLS升级，原本没有启用mTLS的服务在启用mTLS时甚至允许不中断现存已建立的纯文本传输连接。一旦所有服务都完成迁移，便可将整个系统设置为严格TLS模式（即上面代码中的mode: STRICT）。

在Spring Cloud版本的Fenix's Bookstore里，因为没有基础设施的支持，一切认证工作就不得不在应用层面去实现。笔者选择的方案是借用OAuth2协议的客户端模式来进行认证，其大体思路有如下两步：

- 每一个要调用服务的客户端都与认证服务器约定好一组只有自己知道的密钥（Client Secret），这个约定过程应该是由运维人员在线下自行完成，通过参数传给服务（而不是由开发人员在源码或配置文件中直接设定，笔者在演示工程的代码注释中也专门强调了这点，以免有读者被示例代码误导）。密钥就是客户端的身份证明，客户端调用服务时，会先使用该密钥向认证服务器申请到JWT令牌。如以下代码定义了五个客户端，其中四个是集群内部的微服务，均使用客户端模式，且注明了授权范围是“SERVICE”（后面介绍授权会用到）。

```
/***
 * 客户端列表
```

java

```

*/
private static final List<Client> clients = Arrays.asList(
    new Client("bookstore_frontend", "bookstore_secret", new String[]
{GrantType.PASSWORD, GrantType.REFRESH_TOKEN}, new String[]
{Scope.BROWSER}),
    // 微服务一共有Security微服务、Account微服务、Warehouse微服务、Payment微
服务四个客户端
    // 如果正式使用，这部分信息应该做成可以配置的，以便快速增加微服务的类型。
clientSecret也不应该出现在源码中，应由外部配置传入
    new Client("account", "account_secret", new String[]
{GrantType.CLIENT_CREDENTIALS}, new String[]{Scope.SERVICE}),
    new Client("warehouse", "warehouse_secret", new String[]
{GrantType.CLIENT_CREDENTIALS}, new String[]{Scope.SERVICE}),
    new Client("payment", "payment_secret", new String[]
{GrantType.CLIENT_CREDENTIALS}, new String[]{Scope.SERVICE}),
    new Client("security", "security_secret", new String[]
{GrantType.CLIENT_CREDENTIALS}, new String[]{Scope.SERVICE}))
);

```

- 每一个对外提供服务的服务端都是OAuth2中的资源服务器，它们均声明为要求提供客户端模式的凭证，如以下代码所示。客户端要调用受保护的服务，就必须先出示能证明调用者身份的JWT令牌，否则就会遭到拒绝。这个操作本质上是授权，但是在授权过程中也已实现了身份认证。

```

public ClientCredentialsResourceDetails
clientCredentialsResourceDetails() {
    return new ClientCredentialsResourceDetails();
}

```

java

由于每一个微服务都同时具有服务端和客户端两种身份，所以这些代码在每个微服务中都有包含（放在公共infrastructure工程里）。Spring Security提供的过滤器会自动拦截请求，驱动认证、授权检查的执行，申请和验证JWT令牌等操作无论在开发期对程序员，还是运行期对用户都能做到相对透明。尽管如此，以上仍然是一种应用层面的、不加密传输的解决方案。前文提到在零信任网络中，面对可能的中间人攻击，TLS是唯一可行的办法，言下之意是即使应用层的认证能一定程度上保护服务不被身份不明的客户端越权调用，但对传输过程中内容被监听、篡改、以及被攻击者在传输途中拿到JWT令牌后去再冒认调用者身份调用其他服务都是无法防御的。简而言之，这种方案不适用于零信任安全模型，只能在默认内网节点间具备信任关系的边界安全模型上才能良好工作。

用户认证

对于来自最终用户的请求认证，Istio版本的Fenix's Bookstore仍然能做到单纯依靠基础设施解决问题，整个认证过程无需应用程序参与（生成JWT令牌还是在应用中生成的，因为我们并没有使用独立的用户认证服务器，只有应用本身才拥有用户信息）。当来自最终用户的请求进入服务网格时，Istio会自动根据配置中的[JWKS](#)（JSON Web Key Set）来验证令牌的合法性，如果令牌没有被篡改过且在有效期内，就信任Payload中的用户身份，并从令牌的Iss字段中获得Principal。

关于Iss、Principals等概念，在[安全架构](#)中都有介绍过，这里不再重复。JWKS倒是之前没有提到过的概念，它代表了一个密钥仓库。我们知道分布式系统中，JWT应采用非对称的签名算法（RSA SHA256、ECDSA SHA256等，默认的HMAC SHA256属于对称加密），认证服务器使用私钥对Payload进行签名，资源服务器使用公钥对签名进行验证。常与JWT配合使用的JWK就是一种存储密钥的纯文本格式，本质上和[JKS](#)（Java Key Storage）、[P12](#)（Predecessor of PKCS#12）、[PEM](#)（Privacy Enhanced Mail）这些常见的密钥格式在功能上并没有什么差别。JKWS顾名思义就是一组JWK的集合，通过JWT令牌Header中的KID（Key ID）来匹配使用哪个JWK做签名验证。

以下是Istio版本的Fenix's Bookstore中的用户认证配置，其中“jwks”字段配的就是JWKS全文（实际生产中并不推荐这样做，应该使用jwksUri来配置一个JWKS地址，以方便密钥轮换），根据这里配置的密钥信息，Istio就能够验证请求中附带的JWT是否合法。

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: authentication-jwt-token
  namespace: bookstore-servicemesh
spec:
  jwtRules:
    - issuer: "icyfenix@gmail.com"
      # Envoy默认只认“Bearer”作为JWT前缀，之前其他地方用的都是小写，这里专门兼容一下
      fromHeaders:
        - name: Authorization
          prefix: "bearer "
```

```

# 在rsa-key目录下放了用来生成这个JWKS的证书，最初是用java keytool生成的
jks格式，一般转jwks都是用pkcs12或者pem格式，为方便使用也一起附带了
jwks: |
{
  "keys": [
    {
      "e": "AQAB",
      "kid": "bookstore-jwt-kid",
      "kty": "RSA",
      "n": "i-
htQPOTvNMccJj0kCAzd3YlqBE1URzkaeRLDoJYskyU59JdG0-
p_q4JEH0DZOM2BbonGI4lIHFKiZL04IBBZ5j2P7U6QYURt6-
AyjS6RGw9v_wFdIRlyBI9D3E07u8rCA4RktBLPavfEc5BwYX2Vb9wX6N63tV48cP1CoGU0G
tIq9HTqbEQs5KVmme5n4X0uzxQ6B2AGaPBJgdq_K0ZWdkXiqPz6921X3oiNYPCQ22bvFxb4
yFX8ZfbxeYc-1rN7PaUsK009q0x-qRenHpWgPVfagMbNYkm0TOHN0wXqukxE-soCDI_Nc-
-1khWCmQ9E2B82ap7IXsVBAnBIaV9WQ"
    }
  ]
}
forwardOriginalToken: true

```

Spring Cloud版本的Fenix's Bookstore就略微麻烦一些，它依然是采用JWT令牌作为用户身份凭证的载体，认证过程依然在Spring Security的过滤器里中自动完成（因讨论主题不在这里，详细过程就不表了，主要路径是过滤器→令牌服务→令牌实现）。Spring Security已经做好了认证所需的绝大部分的工作，真正要用户编写的代码就是令牌实现，即代码中名为 RSA256PublicJWTAccessToken 的实现类。它的作用是加载Resource目录下的公钥证书 public.cert （实在是怕“抄作业不改名字”的行为，笔者再一次强调不要将密码、密钥、证书这类敏感信息打包到程序中，示例代码只是为了演示，实际生产应该由运维人员管理密钥），验证请求中的JWT令牌是否合法。

```

@Named
public class RSA256PublicJWTAccessToken extends JWTAccessToken {
  RSA256PublicJWTAccessToken(UserDetailsService userDetailsService)
throws IOException {
  super(userDetailsService);
  Resource resource = new ClassPathResource("public.cert");
  String publicKey = new
String(FileCopyUtils.copyToByteArray(resource.getInputStream()));
  setVerifierKey(publicKey);
}

```

```
    }  
}
```

如果JWT令牌合法，Spring Security的过滤器就会放行调用请求，并从令牌中提取出Principals，放到自己的安全上下文中（即 `SecurityContextHolder.getContext()`）。开发实际项目时，你可以根据需要决定Principals的形式，即可以像Istio中那样直接从令牌中取出来，以字符串形式原样存放，节省一些数据库或者缓存的查询开销；也可以统一做些额外的转换处理，如将Principals转换自动为系统中用户对象以方便后续业务使用。Fenix's Bookstore转换操作是在JWT令牌的父类 `JWTAccessToken` 中完成的。由此可见，尽管由应用自己来做请求验证会有一定的代码量和侵入性，但灵活性确实也要更高一些。

为方便不同版本实现之间的对比（偷懒少改代码），在Istio版本中保留了Spring Security自动从令牌转换Principals为用户对象的逻辑，因此必须在YAML中包含 `forwardOriginalToken: true` 的配置，告诉Istio验证完JWT令牌后不要丢弃掉请求中的Authorization Header，原样转发给后面的服务处理。

授权

经过认证之后，合法的调用者就有了可信任的身份，此时就已经不再需要区分调用者到底是机器（服务）还是人类（最终用户）了，只根据其身份角色来进行权限访问控制，即我们常说的RBAC。不过为了更便于理解，Fenix's Bookstore提供的示例代码仍然沿用此前的思路，针对来自“服务”和“用户”的流量来控制权限和访问范围。

举个具体例子，如果我们准备把一部分微服务视为私有服务，限制它只接受来自集群内部其他服务的请求，另外一部分微服务视为公共服务，允许它可接受来自集群外部的最终用户发出的请求；又或者我们想要控制一部分服务只允许移动应用调用，另外一部分服务只允许浏览器调用。那一种可行的方案就是为不同的调用场景设立角色，进行授权控制（另一种常用的方案是做BFF网关）。

在Istio版本的Fenix's Bookstore中，通过以下配置，限制了来自 `bookstore-servicemesh` 名空间的内部流量只允许访问 `accounts`、`products`、`pay` 和 `settlements` 四个端点的GET、POST、PUT、PATCH方法，而对于来自 `istio-system` 名空间（Istio Ingress Gateway所在的名空间）的外部流量就不作限制，直接放行。

```

apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: authorization-peer
  namespace: bookstore-servicemesh
spec:
  action: ALLOW
  rules:
    - from:
        - source:
            namespaces: ["bookstore-servicemesh"]
      to:
        - operation:
            paths:
              - /restful/accounts/*
              - /restful/products/*
              - /restful/pay/*
              - /restful/settlements*
            methods: ["GET", "POST", "PUT", "PATCH"]
    - from:
        - source:
            namespaces: ["istio-system"]

```

yaml

但对外部的请求（不来自 bookstore-servicemesh 名空间的流量），又进行了另外一层控制，如果请求中没有包含有效的登录信息，就限制不允许访问 accounts、pay 和 settlements 三个端点，如以下配置所示：

```

apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: authorization-request
  namespace: bookstore-servicemesh
spec:
  action: DENY
  rules:
    - from:
        - source:
            notRequestPrincipals: ["*"]
            notNamespaces: ["bookstore-servicemesh"]
      to:
        - operation:

```

yaml

```
paths:
  - /restful/accounts/*
  - /restful/pay/*
  - /restful/settlements*
```

Istio已经提供了比较完善的目标匹配工具，如上面配置中用到的源 `from`、目标 `to`，还有未用到的条件匹配 `when`，以及其他如通配符、IP、端口、名空间、JWT字段，等等。要说灵活和功能强大，肯定还是不可能跟在应用中由代码实现的授权相媲美，但对绝大多数场景已经够用了。在便捷性、安全性、无侵入、统一管理等方面，Istio这种在基础设施上实现授权方案显然要更具优势。

Spring Cloud版本的Fenix's Bookstore中，授权控制自然还是使用Spring Security、通过应用程序代码来实现的。常见的Spring Security授权方法有两种，一种是使用它的 `ExpressionUrlAuthorizationConfigurer`，即类似如下编码所示的写法来进行集中配置，这与上面在Istio的AuthorizationPolicy CRD中写法在体验上是非常相似的，这也是几乎所有Spring Security资料中都有介绍的最主流方式，适合对批量端点进行控制，不过在示例代码中并没有采用（没有什么特别原因，就是笔者的习惯而已）。

```
http.authorizeRequests()
    .antMatchers("/restful/accounts/**").hasScope(Scope.BROWSER)
    .antMatchers("/restful/pay/**").hasScope(Scope.SERVICE)
```

java

另一种写法，也是实例代码中采用的方法通过是Spring的[全局方法级安全](#)（Global Method Security）以及JSR 250的 `@RolesAllowed` 注解来做授权控制。这种写法对代码的侵入性更强，但也能以更方便的形式做出更加精细的控制效果。譬如要控制服务中某个方法只允许来自服务或者来自浏览器的调用，只许直接在该方法上标注 `@PreAuthorize` 注解即可，支持[SpEL表达式](#)。表达式中用到的 `SERVICE`、`BROWSER` 代表授权范围，是在声明客户端列表时传入的（见开头声明客户端列表的代码清单）。

```
/**
 * 根据用户名获取用户详情
 */
@GET
@Path("/{username}")
@Cacheable(key = "#username")
@PreAuthorize("#oauth2.hasAnyScope('SERVICE', 'BROWSER')")
```

java

```
public Account getUser(@PathParam("username") String username) {  
    return service.findAccountByUsername(username);  
}  
  
/**  
 * 创建新的用户  
 */  
@POST  
@CacheEvict(key = "#user.username")  
@PreAuthorize("#oauth2.hasAnyScope('BROWSER')")  
public Response createUser(@Valid @UniqueAccount Account user) {  
    return CommonResponse.op(() -> service.createAccount(user));  
}
```

可观测性

随着分布式架构渐成主流，[可观测性](#)（Observability）一词也日益频繁地被提起。它与[可控制性](#)（Controllability）是Rudolf E. Kálmán针对线性动态控制系统提出的一组对偶属性，原本的含义是“可以由其外部输出推断其内部状态的程度”。

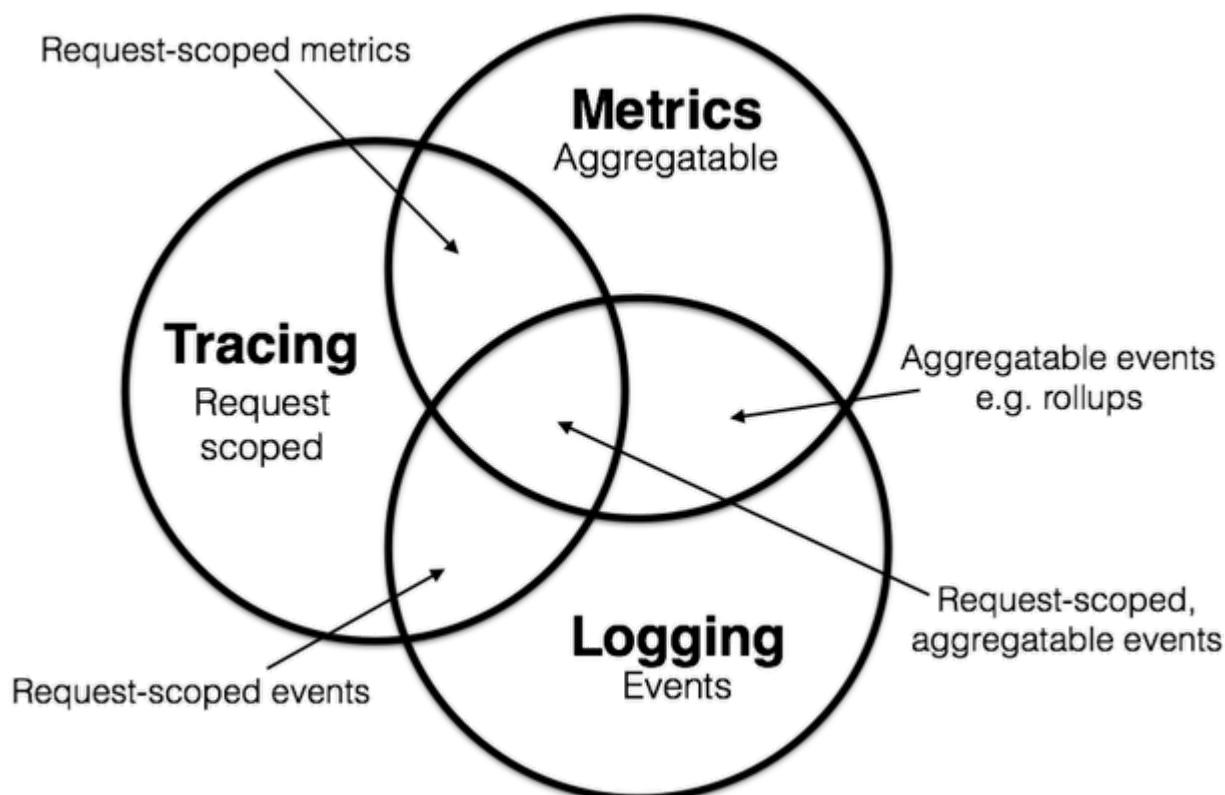
可观测性是近几年才从控制理论中借用的舶来概念，不过其内容实际在计算机科学中已有多年的实践积累。如今一般会将可观测性具体分解为三个更具体方向进行研究：[日志收集](#)（Logging）、[链路追踪](#)（Tracing）和[聚合度量](#)（Metrics），如果此前你从来没接触过分布式系统的观测工作，那可能只会对日志比较了解，对追踪和度量都相对陌生。尽管在分布式系统中追踪和度量必要性和复杂程度确实比单体系统时要更高，但此前单体系统中你肯定已经做过类似的工作，譬如：

- **日志**：日志的主要职责是记录离散事件，通过日志事后分析出程序的行为，譬如曾经调用过什么方法，操作过哪些数据。日志功能非常基础，不论是单体还是微服务，今天你也许还能找到不使用Spring的Java系统，却应该很难找到不使用Slf4j或Common Logging的系统了，Log4j、Slf4j这些日志实现及外观（Facade）类库已事实上成了Java程序标配的基础类库。但你是否曾想过什么是日志？日志的边界在哪里？输出日志是为了解决什么问题？代价是什么？打印日志被认为是程序中最简单的工作之一，在调试问题时常有人说“当初这里记得打点日志就好了”，可见是一项举手之劳的工作。尽管输出日志很简单，但收集和分析日志却可能会很复杂，也许对大多数程序员来说，分析日志就是最常遇见也最有实践可行性的“大数据”系统了。用大数据的视角来观察日志，会有许多应该考虑、可以分析挖掘的内容。
- **追踪**：单体系统时代追踪的概念只局限于[栈追踪](#)（Stack Tracing），你调试程序时，在IDE打个断点，看到的Call Stack视图上的内容便是跟踪；你编写代码时，处理异常调用了Exception::printStackTrace()方法，它输出的堆栈信息也是追踪。微服务时代，追踪就不只局限于调用栈了，一个外部请求需要内部若干服务的共同响应，这时候完整的调用轨迹将跨越了多个服务，包括服务间的网络交互与各个服务内部的调用栈，因此，分布式系统中的追踪在国内常被称为“全链路追踪”（本文就直接称“链路追踪”了），国外一般不叫“全链路”，习惯就称做“分布式追踪”，即[Distributed Tracing](#)。追踪的主要目

的是故障排查，如分析调用链的哪一部分、哪个方法出现错误、阻塞，输入输出是否符合预期等等。

- **度量**：度量是指对系统中某一类信息的总结聚合，譬如，证券市场的每一只股票都会定期公布财务报表，通过财报上的营收、净利、毛利、资产、负载等等一系列数据来体现过去一个财务周期中公司的经营状况，这便是一种信息聚合。Java里有一种很基本的度量便是由虚拟机直接提供的JMX（Java Management eXtensions）度量，诸如内存大小、各分代的用量、峰值的线程数、垃圾收集的吞吐量、频率，等等都可以从JMX中获得。度量的主要目的是监控预警（Monitoring），如某些度量指标达到风险阈值时触发事件，以便自动处理或者提醒管理员介入。

日志、追踪、度量三者并不是完全互相独立的，它之间有有许多天然重合或者可以结合之处，2017年Peter Bourgon撰写的文章《Metrics, tracing, and logging》讲述了这三者之间的关系，如下图所示。日志、追踪、度量三者融合是趋势，像OpenTelemetry这样三者兼备的融合框架，被CNCF视为可观测性的终极方案。



日志、追踪、度量的目标与结合（图片来源）

在具体产品层面，经过几年的群雄混战，日志、度量两个领域的胜利者已经基本尘埃落定。日志领域早就已经统一到Elastic Stack（ELK）技术栈上，未来要是能出现变化的话，也就是其中的Logstash能看到有被Fluentd取代的趋势，让ELK变成EFK，但整套Elastic Stack技术栈的地位还是相当稳固的。

度量领域，跟随着Kubernetes统一容器编排的步伐，Prometheus也击败了度量领域里以Zabbix为代表的众多前辈，成为了云原生时代度量监控的事实标准。从市场角度来说Prometheus还没有达到Kubernetes那样一统天下的程度，但从社区活跃度上看，Prometheus已占有绝对的优势，在Google和CNCF的推动下，未来前途可期。

额外知识：Kubernetes与Prometheus的关系

Kubernetes是CNCF第一个孵化成功的项目，Prometheus是CNCF第二个孵化成功的项目。

Kubernetes起源于Google的编排系统Borg，Prometheus起源于Google为Borg做的度量监控系统BorgMon。

追踪领域与日志、度量不同，它是与具体网络协议、程序语言相关的，收集日志你不必关心某段日志是由Java程序输出的还是由Golang程序输出的，对你来说它们都是一段非结构化文本，同理，度量对你来说也只是一个聚合的数据指标而已。但是，链路追踪就不一样，各个服务之间是使用HTTP还是gRPC来进行网络访问会直接影响追踪的实现，各个服务是使用Java、Golang还是Node.js来编写，也会直接影响到进程内调用栈的追踪方式。这决定了追踪工具本身有更强烈的探针插件化的需求，也决定了追踪领域很难出现一家独大的情况。近年来各种链路追踪产品层出不穷，市面上主流的工具既有像Datadog这样的一揽子商业方案，也有AWS X-Ray和Google Stackdriver Trace这样的云厂商产品，还有像SkyWalking、Zipkin、Jaeger这样来自开源社区的产品。2016年，CNCF牵头制定了[Open Tracing规范](#)，这是CNCF接受的第三个项目，OpenTracing致力于推进追踪领域的工具与协议标准化，未来我们很可能会根据服务具体情况的差异，组合使用不同厂商或组织的追踪工具。

Monitoring



Logging



Tracing

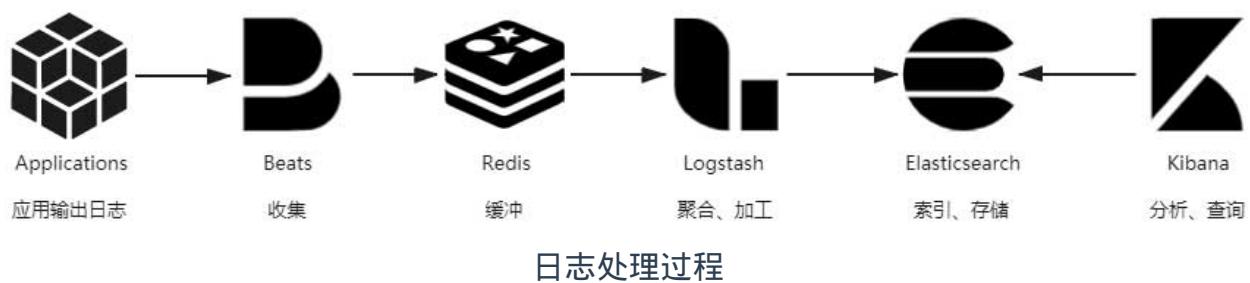


日志、追踪、度量的相关产品（图片来源[CNCF](#)）

上图是CNCF Interactive Landscape [中列出的日志、追踪、度量领域的常见产品](#)，其实里面很多不同领域的产品是跨界的，譬如ELK可以通过Metricbeat来实现度量的功能，Apache SkyWalking的探针就有同时支持度量和追踪两方面的数据来源，等等。接下来到三个小节，笔者将详细介绍这三个领域中具体需要考虑的问题及其解决方案。

事件日志

日志用来记录系统运行期间发生过的离散事件。相信没有哪一个生产系统会认为日志功能是可以缺少的，然而也很少人会把日志作为多关键功能来看待。日志就像阳光与空气，经常被使用，却不怎么被重视。程序员们会说日志简单，其实这是在说“打印日志”这个操作简单，打印日志的目的是为了日后从中得到有价值的信息，而今天只要稍微复杂点的系统，尤其是复杂的分布式系统，就很难只依靠tail、grep、awk来从日志中挖掘信息了，往往还要有专门的全局查询和可视化功能。此时，从打印日志到分析查询之间，还隔着收集、缓冲、聚合、加工、索引、存储等若干个步骤，如下图所示。



这一整个链条中涉及到大量值得注意的细节，复杂性并不亚于任何一项技术或业务功能的实现。下面，笔者将以此为线索，以最成熟的Elastic Stack技术栈为例子，介绍该链条每个步骤的目的与注意事项。

输出

要是说好的日志能像文章一样，能让人读起来身心舒畅，这话肯定有夸大的成分，不过好的日志应该能做到像“流水账”一样，无有遗漏地记录信息，格式统一，内容恰当。其中，“恰当”是指日志不应该多，也不应该少。“多少”不是指输出的日志行数，尽管笔者听过夸张的系统是单节点INFO级别下每天的日志都能到TB量级（这是代码有问题的），给网络与磁盘I/O带来了不小压力，但笔者通常不以数量来衡量日志是否恰当，“多少”主要是指日志中不该出现的内容不要有，该有的不要少，下面笔者先列出一些常见的“不应该有”的例子：

- **避免打印敏感信息。** 日志中完全可以打印当前用户的ID（最好是内部ID，避免登录名或者用户名称），笔者见过有系统就直接用[MDC](#)（Mapped Diagnostic Context）将用户ID自动打印在Pattern Layout上。不用专门提醒，你肯定也知道不该将密码，银行账号，身份证这些敏感信息打到日志里，但笔者同样见过不止一个系统的日志中直接能找到这些信息的。一旦这些敏感信息随日志流到了后续的索引、存储、归档等步骤后，清理起来将非常麻烦（Elasticsearch要到段文件合并的时候才会物理删除文档）。
- **避免引用慢操作。** 日志中打印的信息应该是上下文中可以直接取到的信息，如果当前上下文中根本没有这项数据，需要专门调用远程服务、或从数据库，或者通过大量计算才能取到的话，那应该先考虑这项信息放到日志中是不是必要且恰当的。
- **避免误导他人。** 日志中给日后调试除错的人挖坑是十分恶劣却又常见的行为。笔者相信开发人员并不是专门要去误导别人，只是很多人都无意识地这样做了。譬如明明已经在逻辑中妥善处理好了某个异常，偏习惯性地把堆栈打到日志中，有时候更恶劣的还以ERROR或者WARN级别去打，日后要是出问题了，其他人来除错的话往往就使劲盯着这段堆栈找线索了。
- **避免打印追踪诊断信息。** 日志中不要打印方法输入参数、输出结果、方法执行时长之类的调试信息。这点是反直觉的，笔者知道不少公司甚至会将其作为最佳实践来提倡，但是笔者仍坚持将其归入反模式中。日志的职责是记录事件，追踪诊断有追踪系统去处理，哪怕贵公司完全没有开发追踪诊断方面功能的打算，笔者也建议使用[BTrace](#)或者[Arthas](#)这类“On-The-Fly”的工具来解决调试问题。之所以将其归为反模式，是因为上面说的敏感信息、慢操作等的主要源头就是这些原本想用于调试的日志。譬如，当前方法入口参数有个User对象，如果要输出这个对象的话，常见做法是将它序列化成JSON字符串然后打到日志里，这时候User里面的Password字段、BankCard字段就全暴露出来了；再譬如，当前方法的返回值是个Map，开发期的调试数据只做了三五个Entity，你觉得遍历一下把具体内容打到日志里面没什么问题，生产期这个Map里面搞不好就放成了千上万个Entity，这时候打印日志就相当于引用慢操作。
-

另一方面，日志中不该缺少的内容也“不应该少”，以下是部分笔者建议应该输出到日志中的内容：

- **处理请求时的TraceID。** 服务收到请求时，如果该请求没有附带TraceID，就应该自动生成唯一的TraceID来对请求进行标记，并使用MDC自动输出到日志。TraceID会贯穿整条调用链，目的是可以通过它把外部请求在分布式系统中的执行过程串联起来。TraceID

通常也会随着请求的响应返回到客户端，如果响应内容出现了异常，用户便能通过此ID快速找到与问题相关的信息。TraceID是OpenTracing中的概念，类似的还有用于标识进程内调用状况的SpanID，它们都可以用Spring Cloud Sleuth自动生成，这些概念会放到下一节“链路追踪”里再详细介绍。尽管TraceID是为分布式跟踪而提出的，但即使对单体系统，生成TraceID、记录到日志并返回给最终用户对快速定位错误也是有益的。

- **系统运行过程中的关键事件。** 日志的职责就是记录事件，譬如发生了与预期不符的业务流程、运行期间出现未能处理的异常或警告、定期自动执行的任务，等等，都应该在日志中记录下来。原则上业务领域中发生的事件只要有价值就应该去记录，并不提倡因考虑日志数量多少对性能的影响而投鼠忌器。开发阶段只要判断清楚事件的重要程度，选定相匹配的日志的级别即可，至于如何快速处理大量日志，这是后面步骤要考虑的问题。退一步说，日志实在太多，由运维人员去调整全局或单个类的日志级别也是可以的。
- **启动时输出配置信息。** 与避免输出诊断信息不同，对于系统启动时或者检测到配置中心变化时更新的配置，应将非敏感的配置信息输出到日志中，譬如连接的数据库地址、临时目录的路径等等，初始化配置的逻辑一般只会执行一次，不便于诊断时复现，所以应该输出到日志中。
-

收集与缓冲

写日志是在服务节点中进行的，但我们不可能在每个节点都单独建设日志查询功能。这不是资源或工作量的问题，而是分布式系统处理一个请求要跨越多个服务节点，为了能看到跨节点的全部日志，就要有能覆盖整个链路的全局日志系统。这个需求决定了每个节点输出日志到文件后，必须将日志文件统一收集起来集中存储、索引，由此便催生了专门的日志收集器。

最初，ELK中日志收集与下一节要讲的加工聚合的职责都是由Logstash来承担的，Logstash既部署在各个节点中作为收集的客户端（Shipper），也同时有独立部署的节点，扮演归集转换日志的服务端（Master）。Logstash有良好的插件化设计，收集、转换、输出都支持插件化定制，应对多重角色本身并没有什么困难。但是Logstash与它的插件是基于JRuby编写的，要跑在单独的Java虚拟机进程上（Logstash的默认的堆大小就到了1GB），对于归集部分（Master）这种消耗并不是什么问题，但作为每个节点都要部署的日志收集器就显得太过负重了。后来，Elastic.co公司将所有需要在服务节点中处理的工作整理成以Lib

beat 为核心的 Beats 框架，并使用 Golang 重写了一个功能较少，但也更轻量高效的日志收集器，就是今天流行的 Filebeat。

现在的 Beats 已经是一个很大的家族了，除了 Filebeat 外，Elastic.co 还提供有用于收集 Linux 审计数据的 Auditbeat、用于无服务计算架构的 Functionbeat、用于心跳检测的 Heartbeat、用于聚合度量的 Metricbeat、用于收集 Linux Systemd Journal 日志的 Journalbeat、用于收集 Windows 事件日志的 Winlogbeat，用于网络包嗅探的 Packetbeat 等等，如果再算上大量由社区维护的 Community Beats，那几乎是你能想像到的数据都可以收集到，以至于 ELK 也可以一定程度上代替度量和跟踪系统，实现它们的部分职能，这对于中小型分布式系统来说是便利的，但对于大型系统，建议还是让专业的工具去做专业的事情。

日志收集器不仅要保证能覆盖全部数据来源，还要尽力保证日志数据的连续性，这其实并不容易做到。譬如淘宝这类大型的互联网系统，每天的日志量超过了 10,000TB（10PB）量级，日志收集器的数量能到达百万量级（[数据来源](#)），此时归集到系统中的日志要与实际产生的日志保持绝对的一致性是非常困难的，也不应该为此付出过高成本。换而言之，日志是不追求绝对的一致、完整、精确的，而追求在代价可承受的范围内保证尽可能地保证较高的数据质量。最常见的做法是将压力从 Logstash 和 Elasticsearch 转移到抗压能力更强的消息队列缓存中，譬如在 Logstash 之前架设一个 Kafka 或者 Redis 作为缓冲层，面对突发流量，Logstash 或 Elasticsearch 处理能力出现瓶颈时自动削峰填谷，甚至它们完全停顿时也不至于丢失日志数据。

加工与聚合

将日志集中收集之后，存入 Elasticsearch 之前，一般还要对它们进行加工转换和聚合处理。这是因为日志是非结构化数据，一行日志中往往包含多项信息，如果不做处理，那在 Elasticsearch 就只能以全文检索的原始方式去使用日志，既不利于统计对比，也不利于条件过滤。举个具体例子，下面是一行 Nginx 服务器的 Access Log，代表了一次页面访问操作：

```
14.123.255.234 - - [19/Feb/2020:00:12:11 +0800] "GET /index.html  
HTTP/1.1" 200 1314 "https://icyfenix.cn" "Mozilla/5.0 (Windows NT 10.0;  
WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.163  
Safari/537.36"
```

在这一行日志里面，包含了下表所列的这10项独立的数据项：

数据项	值
IP	14.123.255.234
Username	null
Datetime	19/Feb/2020:00:12:11 +0800
Method	GET
URL	/index.html
Protocol	HTTP/1.1
Status	200
Size	1314
Refer	https://icyfenix.cn
Agent	Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.163 Safari/537.36

Logstash最常用的职能就是把日志行中的非结构化数据，通过Grok表达式语法转换为上面表格那样的结构化数据，进行结构化的同时，还可能会根据需要调用其他插件来完成时间处理（统一时间格式）、类型转换（如字符串、数值的转换）、查询归类（譬如将IP地址根据地理信息库按省市归类）等种额外处理，然后再以JSON格式输出到Elasticsearch中（这是最普遍的输出形式，Logstash输出也有很多插件可以具体定制）。这些经过Logstash转换，已经结构化的日志，Elasticsearch便可针对不同的数据项来建立索引，进行条件查询、统计、聚合等操作的了。

提到聚合，这也是Logstash的常见职能之一。日志中存储的是离散事件，离散的意思是每个事件都是相互独立的，譬如有10个用户访问服务，他们操作所产生的事件都在日志中会分别记录。如果想知道这些用户中正常返回（200 OK）的有多少、出现异常的（500 Internal Server Error）的有多少，再生成个可视化统计图表什么的，一种解决方案是通过Elasticsearch本身的处理能力做实时的聚合统计，这很便捷，不过要消耗Elasticsearch服务器的运算资源。另一种解决方案是在收集日志后自动生成某些常用的、固定的聚合指标，这种聚合就会在Logstash中通过聚合插件来完成。这两种聚合方式都有不少实际应用，前者一般用于应对即席查询，后者一般用于应对固定查询。

存储与查询

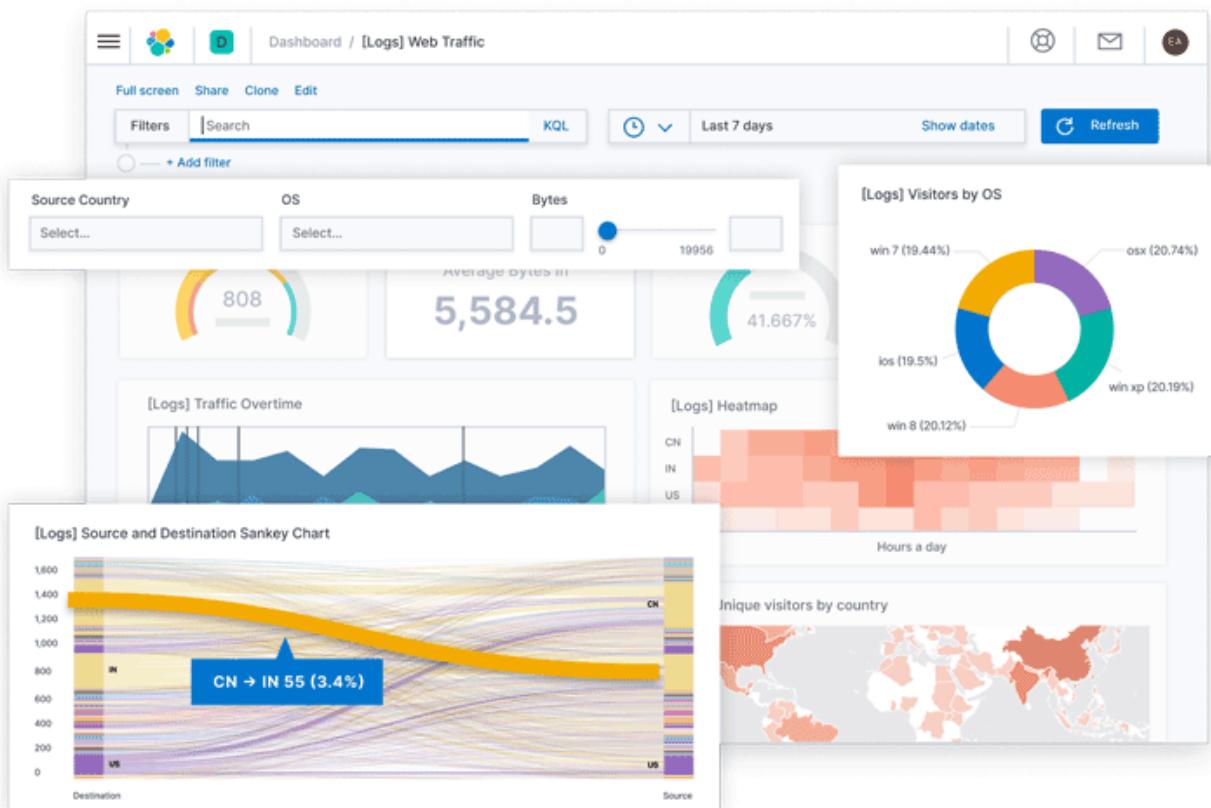
经过收集、缓冲、加工、聚合的日志数据，终于可以放入Elasticsearch中索引存储了。Elasticsearch是整个Elastic Stack技术栈的核心，其他步骤的工具，如Filebeat、Logstash、Kafka都有替代品，有自由选择的余地，唯独Elasticsearch在日志分析这方面没有什么值得一提的竞争者，几乎成了解决此问题的标准答案。这样的结果肯定与Elasticsearch本身是一款优秀产品有关，然而更关键的是Elasticsearch的优势正好与日志分析的需求完美契合：

- 从数据特征的角度看，日志是典型的基于时间的数据流，但它与其他时间数据流，譬如你的新浪微博、微信朋友圈这种社交网络数据又稍有区别：日志虽然增长速度很快，但已写入的数据几乎没有再发生变动的可能。日志的数据特征决定了所有用于日志分析的Elasticsearch都会使用时间范围作为索引，根据实际数据量的大小可能是按月、按周或者按日、按时。以按天索引为例，由于你能准确地预知明天、后天的日期，因此全部索引都可以预先创建，这免去了动态创建的寻找节点、创建分片、在集群中广播变动信息等开销。又由于所有新的日志都是“今天”的日志，所以你建立了“logs_current”这样的索引别名来指向当前索引，接收新的日志，避免代码因日期而变。
- 从数据价值的角度看，日志基本上只会以最近的数据为检索目标，随着时间推移，早期的数据将逐渐失去价值。这点决定了可以很容易根据时间区出分冷数据和热数据，进而对不同数据采用不一样的硬件策略。譬如为热数据配备SSD磁盘和更好的处理器，为冷数据配备HHD磁盘和较弱的处理器，甚至可以放到更为廉价的对象存储（如阿里云的OSS，腾讯云的COS，AWS的S3）中。

注意，本文的主题是日志在可观测性方面的作用，另外还有一些基于日志的其他类型应用，譬如从日志记录的事件中去挖掘业务热点，分析用户习惯等等，这属于大数据挖掘的范畴，并不在我们讨论“价值”的范围之内，事实上它们更可能采用的技术栈是HBase与Spark的组合，而不是Elastic Stack。

- 从数据使用的角度看，分析日志很依赖全文检索和即席查询，对实时性的要求是处于实时与离线两者之间的“近实时”，即不强求日志产生后立刻能查到，但也不能接受日志产生之后按小时甚至按天的频率来更新，这些检索能力和近实时性，也正好都是Elasticsearch的强项。

Elasticsearch只提供了API层面的查询能力，它通常搭配同样出自Elastic.co之手的Kibana一起使用，你可以将Kibana理解为Elastic Stack的GUI部分。Kibana尽管只负责图形界面和展示，但它提供的能力远不止让你能在界面上执行Elasticsearch的查询那么简单。Kibana宣传的核心能力是“探索数据并可视化”，即把存储在Elasticsearch中的数据被检索、聚合、统计后，定制形成各种图形、表格、指标、统计，以此观察系统的运行状态，找出日志事件中潜藏的规律和隐患。按Kibana官方的宣传语来说就是“一张图片胜过千万行日志”。



Kibana可视化界面（图片来自Kibana官网 [↗](#)）

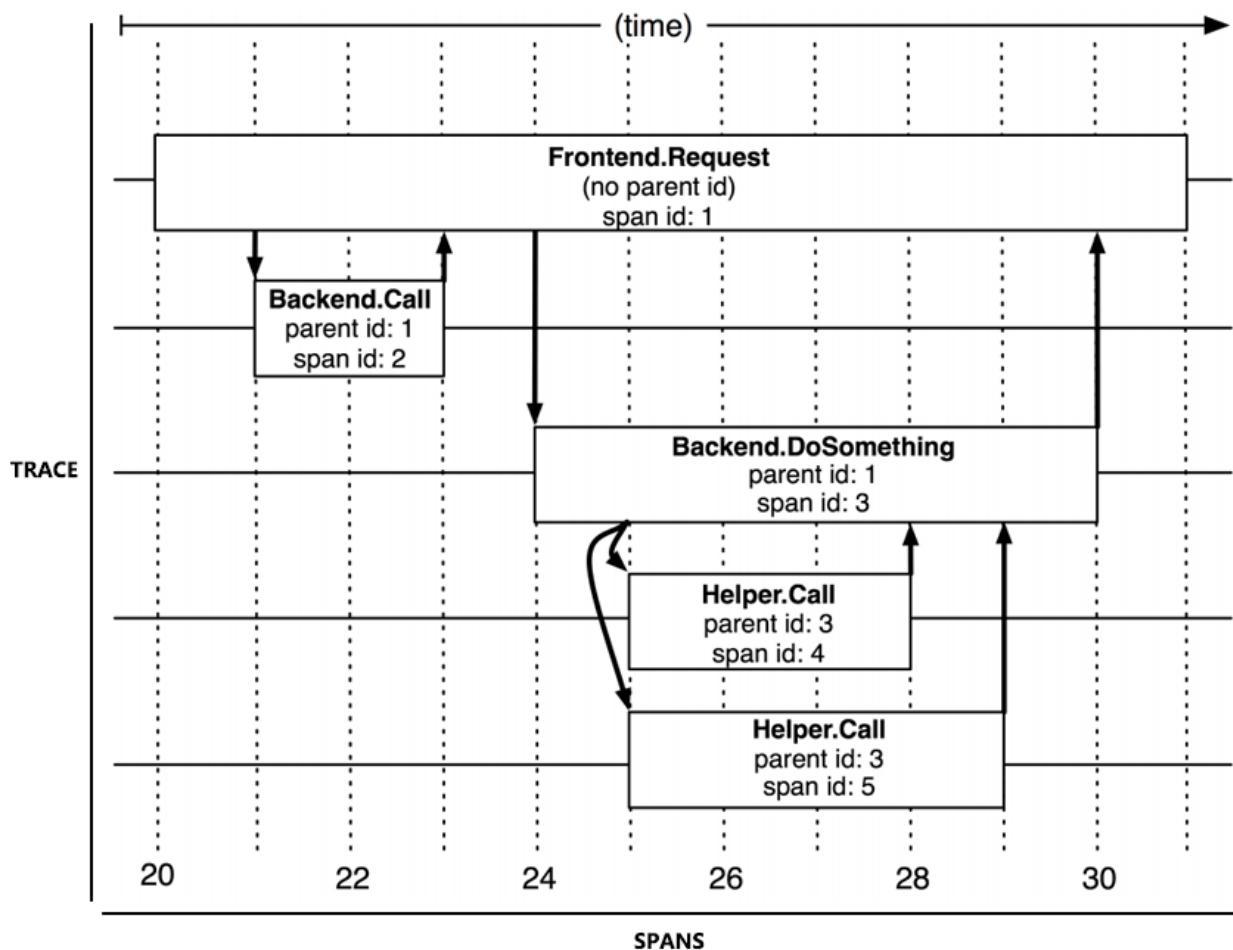
链路追踪

尽管2010年之前就已经有了X-Trace、Magpie等追踪系统了，但分布式链路追踪公认的起源是Google在2010年发表的论文《[Dapper : a Large-Scale Distributed Systems Tracing Infrastructure](#)》，这篇论文介绍了Google自己从2004年开始使用的分布式追踪系统Dapper的实现原理。此后，所有著名的追踪系统，无论是国外Twitter的[Zipkin](#)、Naver的[Pinpoint](#)（Naver是Line的母公司，Pinpoint出现其实早于Dapper论文发表，在Dapper论文中还提到了Pinpoint），抑或是国内阿里的鹰眼、大众点评的[CAT](#)、个人开源的[SkyWalking](#)（后进入Apache基金会孵化毕业）都受到Dapper的直接影响。

广义上讲，一个完整的分布式追踪系统应该由数据收集、数据存储和数据展示三个相对独立的子系统构成，而狭义上讲的追踪则就只是特指链路追踪数据的收集部分。譬如[Spring Cloud Sleuth](#)就属于狭义的追踪系统，通常会搭配Zipkin作为数据展示，搭配Elasticsearch作为数据存储来组合使用，而前面提到的那些Dapper的徒子徒孙们大多都属于广义的追踪系统，广义的追踪系统又常被称为“APM系统”（Application Performance Management）。

追踪与跨度

为了有效地进行分布式追踪，Dapper提出了“追踪”与“跨度”的概念。链路追踪从客户发起请求抵达系统的边界开始，记录流经的每一个服务，直到到被追踪系统向客户返回响应为止，这整个过程就称为一次“追踪”（Trace，为了不与系统名称产生混淆，后文就直接使用英文Trace来指代了）。由于每个Trace都可能会调用数量不定、位置不定的服务，为了能够记录具体调用了哪些服务，以及每次调用的顺序、开始时点、执行时长等信息，调用服务时需要埋入一个调用记录，这个记录称为一个“跨度”（Span），每一个Trace实际上都是由若干个有顺序、有层级关系的Span所组成一颗追踪树（Trace Tree）。Span的数据结构应该足够简单，以便于能放在日志或者网络协议的报文头里；也应该足够完备，起码应含有时间戳、起止时间、Trace的ID、当前Span的ID、父Span的ID等能够满足追踪需要的信息，譬如下图所示。



Trace和Spans (图片来源于[Dapper论文](#))

从目标来看，链路追踪的目的是为排查故障和分析性能提供数据支持，系统对外提供服务的过程中，持续地接受请求并处理响应，同时持续地生成Trace，按次序整理好Trace中每一个Span所记录的调用关系，便能绘制出一幅系统的服务调用拓扑图。根据拓扑图中Span记录的时间信息、请求结果（正常或异常返回）就可以定位到缓慢或者出错的服务，将Trace与历史记录进行对比统计，就可以从系统整体层面分析服务性能，定位性能优化的目标。

从实现来看，为每次服务调用记录Trace和Span，并以此构成追踪树结构，听着好像也不是很复杂，然而考虑到实际情况，追踪系统在功能和非功能性上都有不小的挑战。功能上的挑战来源于服务的异构性，各个服务采用不同程序语言，服务间交互采用不同的网络协议，每兼容一种场景，都会增加功能实现方面的工作量。而非功能性的挑战具体就包括以下这四个方面：

- **低性能损耗**：分布式追踪不能对服务本身产生明显的性能负担。追踪的主要目的之一就是为了寻找性能缺陷，越慢的服务越是需要追踪，所以工作场景都是性能敏感的地方。

- **对应用透明**：追踪系统通常是运维期才事后加入的系统，应该尽量以非侵入的方式来实现追踪，对开发人员做到透明化。
- **随应用扩缩**：现代的分布式服务集群都有根据流量压力自动扩缩的能力，这要求当业务系统扩缩时，追踪系统也能自动跟随，不需要运维人员人工参与。
- **持续的监控**：要求追踪系统必须能够7x24小时工作，否则就难以定位到系统偶尔抖动的行为。

数据收集

目前，追踪系统根据数据收集方式的差异，可分为三种主流的实现方式，分别是 基于日志的追踪（Log-Based Tracing），基于服务的追踪（Service-Based Tracing）和 基于边车代理的追踪（Sidecar-Based Tracing），笔者分别介绍如下：

- 基于日志的追踪的思路是将TraceID、Span等信息直接输出到应用日志中，然后随着所有节点的日志归集过程汇聚到一起，再从全局日志信息中反推出完整的调用链拓扑。日志追踪对网络消息完全没有侵入性，对应用服务只有很少量的侵入性，对性能影响也非常低。但其缺点是对日志归集过程的直接依赖，这使得它往往不如其他两种追踪系统实现来的精确。业务服务的调用与日志的归集并不是同时完成的，也通常不由同一个进程完成，完全可能发生业务调用已经顺利结束了，但由于日志归集不及时或者精度丢失，导致日志出现延迟或缺失记录，进而导致追踪失真。这也是前面笔者介绍Elastic Stack时提到的观点，ELK在日志、追踪和度量方面都可以发挥作用，这对中小型应用确实有一定便利，但是大型系统最好还是由专业的工具做专业的事，让ELK只专注于日志处理。

目前，日志追踪的代表是Spring Cloud Sleuth，下面是一段由Sleuth在调用时自动生成的日志记录，你可以从中观察到TraceID、SpanID、父SpanID等追踪信息。

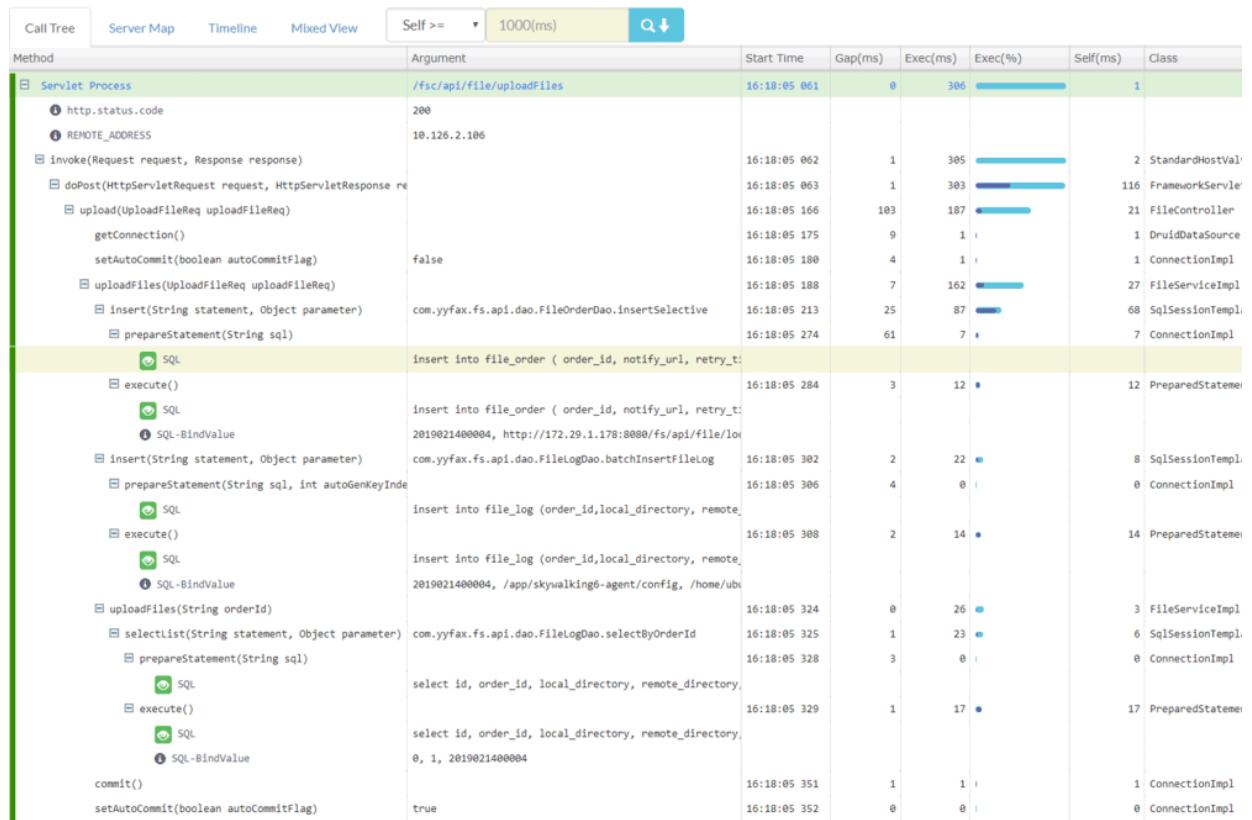
```
# 以下为调用端的日志输出：  
Created new Feign span [Trace: cbe97e67ce162943, Span:  
bb1798f7a7c9c142, Parent: cbe97e67ce162943, exportable:false]  
2019-06-30 09:43:24.022 [http-nio-9010-exec-8] DEBUG  
o.s.c.s.i.web.client.feign.TraceFeignClient - The modified request  
equals GET http://localhost:9001/product/findAll HTTP/1.1  
  
X-B3-ParentSpanId: cbe97e67ce162943  
X-B3-Sampled: 0
```

```
X-B3-TraceId: cbe97e67ce162943
X-Span-Name: http:/product/findAll
X-B3-SpanId: bb1798f7a7c9c142

# 以下为服务端的日志输出：
[findAll] to a span [Trace: cbe97e67ce162943, Span: bb1798f7a7c9c142,
Parent: cbe97e67ce162943, exportable:false]
Adding a class tag with value [ProductController] to a span [Trace:
cbe97e67ce162943, Span: bb1798f7a7c9c142, Parent: cbe97e67ce162943,
exportable:false]
```

- 基于服务的追踪是目前最为常用的追踪实现方式，被Zipkin、SkyWalking等主流追踪系统广泛采用。服务追踪的实现思路是通过某些手段（针对Java应用一般就是通过Java Agent注入）给目标应用注入追踪探针（Probe），你可以把探针视为一个寄生在目标服务身上的独立微服务系统了，它一般会有自己专用的服务注册、心跳检测等功能，有专门的数据收集协议，把从目标系统中监控得到的服务调用信息，通过另一次独立的HTTP或者RPC请求发送给追踪系统。因此，基于服务的追踪会比基于日志的追踪消耗更多的资源，也有更强的侵入性，以此换来的收益是追踪的精确性与稳定性都有所保证，不必再依靠日志归集来传输追踪数据。

下面是一张Pinpoint的追踪效果截图，从图中可以看到参数、变量等相当详细方法级调用信息。笔者在上一节“[日志分析](#)”里把“打印追踪诊断信息”列为一种反模式，如果需要诊断需要方法参数、返回值、上下文信息，或者方法调用耗时这类数据，通过追踪系统来实现是比通过日志系统实现更加恰当的解决方案。



Pinpoint的追踪截图（图片来自网络）

当然，也必须说明清楚的是像Pinpoint这种详细程度的追踪对应用系统的性能压力是相当大的，应该仅在除错时开启，Pinpoint本身就是比较重负载的系统（运行它必须先维护一套HBase）。目前服务追踪的其中一个发展趋势是轻量化，SkyWalking是这方面的佼佼者。

- 基于边车代理的追踪目前实际上只有Envoy一家能做（如果把“实际上”改成“理论上”的话可算上Linkerd/Conduit和NginMesh），毕竟当前Sidecar领域还没有什么能和Envoy真正竞争的产品。现在主流的服务网格框架，如Istio和微软的OSM（Open Service Mesh）都是基于Envoy的。基于边车代理的追踪毫无疑问是最符合理想中的分布式追踪模型的，它对应用完全透明，无论是日志还是服务本身都不会有任何变化；它与程序语言无关，无论应用采用什么编程语言实现，只要它还是通过网络（HTTP或者gRPC）来访问服务就可以被追踪到；它有自己独立的数据通道，追踪数据通过xDS协议进行上报，避免了对服务本身网络或者日志归集的依赖，保证了精确性。如果说这种追踪方式还有什么缺点的话，那就是服务网格现在还不够普及，未来随着云原生的发展，相信它会成为追踪系统的主流实现方式之一。另外就是Envoy本身的工作原理决定了它只能实现服务调用层面的追踪，像上面Pinpoint截图那样本地方法调用级别的追踪诊断是做不到的。

Envoy也没有提供自己的界面端和存储端，所以Envoy和Sleuth一样都属于狭义的追踪系

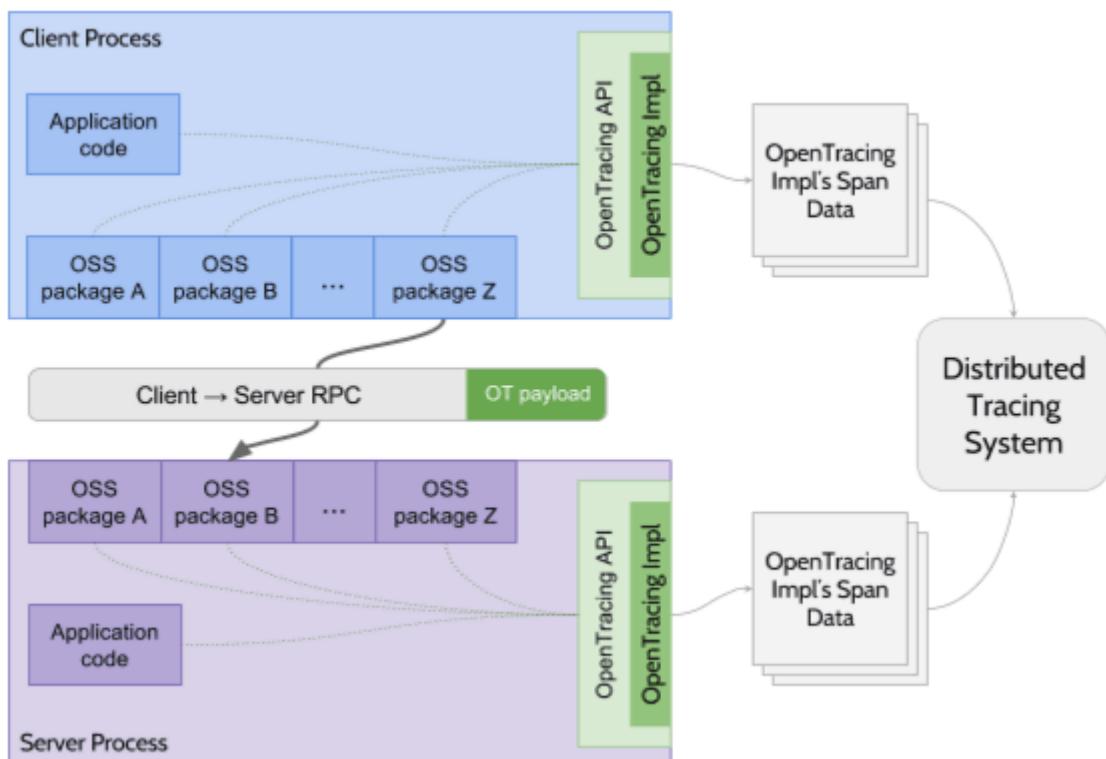
统，需要配合专门的UI来使用，现在SkyWalking、Zipkin、Jaeger[↗]、LightStep Tracing等系统都可以接受来自于Envoy的追踪数据，充当它的界面端。

追踪规范化

比起日志与度量，追踪这个领域的产品竞争要相对激烈。一方面，目前还没有像日志、度量那样出现明显具有统治力的产品，仍处于群雄混战的状态。另一方面，几乎市面上所有的追踪系统都是以Dapper的论文为原型发展出来的，基本上都算是同门师兄弟，却囿于实现细节，彼此互斥。这只能怪当初Google发表的Dapper是论文而不是有约束力的规范，说白了就是只提了思路，并没有规定细节，譬如该怎样进行埋点、Span上下文具体该有什么数据结构，怎样设计追踪系统与探针或者界面端的API接口等等。

为了平息追踪产品的混乱状况，2016年11月，CNCF技术委员会接受了OpenTracing作为基金会第三个项目。OpenTracing是一套与平台无关、与厂商无关、与语言无关的追踪协议规范，只要遵循OpenTracing规范，追踪产品就可以随时更换，也可以相互搭配使用。

OpenTracing的具体工作是制定了一个很薄的标准化层，位于应用程序与追踪系统之间，这样探针与追踪系统完全可以不是同一个厂商的产品，只要它们都支持OpenTracing协议即可互相通讯。另外，OpenTracing还规定了微服务之间发生调用时，应该如何传递Span信息（OpenTracing Payload），以上这些都如下图绿色部分所示。



符合OpenTracing的软件架构（[图片来源](#)）

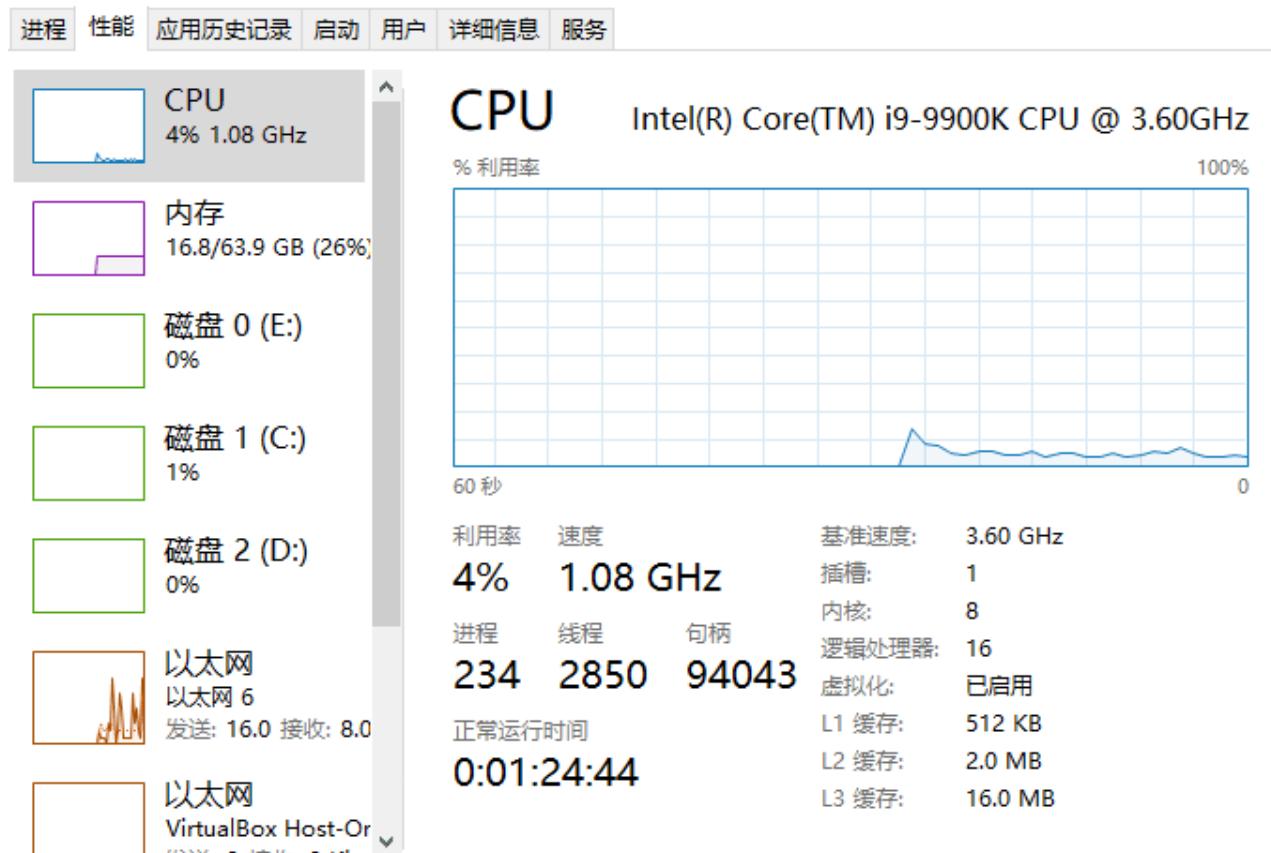
OpenTracing规范一公布，业界著名的追踪系统，如Zipkin、Jaeger、SkyWalking等都很快宣布支持OpenTracing协议，但谁也没想到的是，Google自己却在此时出来反对，提出了与OpenTracing很类似的OpenCensus规范，并随后得到了Microsoft的支持和参与。OpenCensus不仅涉及到追踪，还把指标度量也纳入进来；内容上不仅涉及到规范制定，还把数据采集的探针和收集器都一起以SDK（目前支持五种语言）的形式提供出来。一下子大家都有点手足无措了，OpenTracing和OpenCensus迅速形成了可观测性的两大阵营，一边是在这方面深耕多年的众多老牌APM系统厂商，另一边是分布式追踪概念的提出者Google，以及与Google同样庞大的Microsoft。对追踪系统的规范化工作，看来并没有平息厂商竞争的混乱，反倒是把水搅得更加浑了。

正当群众们买好西瓜搬好板凳的时候，2019年，OpenTracing和OpenCensus又忽然握手言和了，它们共同发布了可观测性的终极解决方案[OpenTelemetry](#)，并宣布会各自冻结OpenTracing和OpenCensus的发展。OpenTelemetry野心颇大，不仅包括追踪规范，还包括度量规范、各种语言的SDK、以及采集系统的参考实现，但是不包括界面端和指标预警这些会与用户直接接触的后端功能，将它们留给具体产品去实现，算是没有对一众APM厂商赶尽杀绝，留下了一条活路。

OpenTelemetry一诞生就带着无比炫目的光环，直接进入CNCF的孵化项目，它的目标是统一追踪、度量和日志三大领域（目前主要关注的是追踪和度量，日志方面，官方表示将放到下一阶段再去处理）。不过，OpenTelemetry毕竟是2019年才出现的新生事物，尽管背景深厚，前途无量，但未来究竟如何发展，能否打败现在已经有的众多成熟系统，目前仍然言之尚早。

聚合度量

度量 (Metrics) 的目的是揭示系统的总体运行状态。相信大家应该见过这样的场景：舰船的驾驶舱或者卫星发生中心的控制室，在整个房间最显眼的位置，布满整面墙壁的巨型屏幕里显示着一个个指示器、仪表板与统计图表，沉稳端坐中央的指挥官看着屏幕上闪烁变化的指标，果断决策，下达命令……如果以上场景被改成指挥官双手在键盘上飞舞，双眼紧盯着日志或者追踪系统，试图判断出系统工作是否正常。这光想像一下，都能感觉到一股身份与行为不一致的违和气息。由此可见度量与日志、追踪的差别，度量是用经过聚合统计后的高维度信息，以最简单直观的形式来总结复杂的过程，为监控、预警提供决策支持。



Windows系统的任务管理器界面

如果你人生经历比较平澹，没有驾驶航母的经验，甚至连一颗卫星或者导弹都没有发射过，那就打开电脑吧，按 **CTRL + ALT + DEL** 呼出任务管理器，看看上面这个熟悉的界面，它也是一个非常典型度量系统。

度量总体上可分为客户端的指标收集、服务端的存储查询以及终端的监控预警三个相对独立的过程，每个过程在系统中一般也会设置对应的组件来实现，你不妨先翻到下面，看一眼Prometheus的组件流程图作为例子，图中在Prometheus Server左边的部分都属于客户端过程，右边的部分就属于终端过程。

Prometheus 在度量领域的统治力虽然还暂时不如日志领域中Elastic Stack的统治地位那么稳固，但基本也已经能算是事实标准了，接下来，笔者将主要以Prometheus为例，介绍这三部分组件的总体思路、大致内容与理论标准。

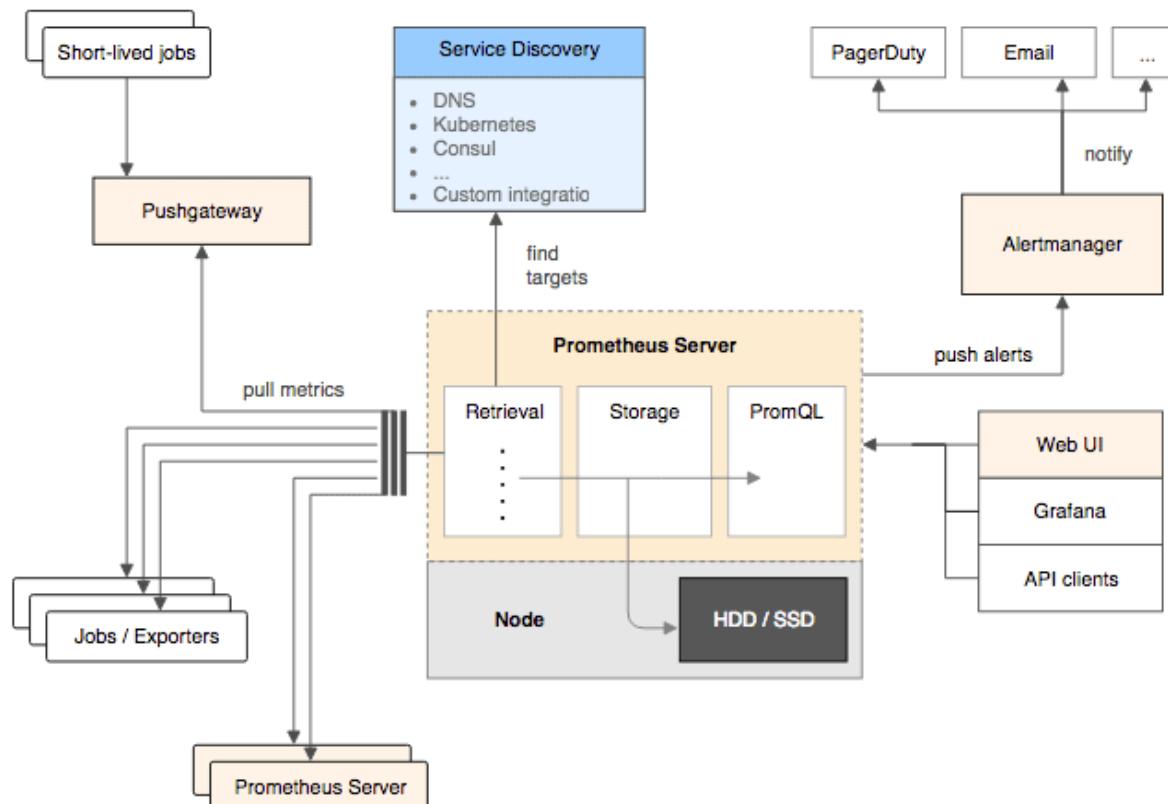
指标收集

指标收集部分要解决两个问题：“如何定义指标”以及“如何将这些指标告诉服务端”，这两个问题听起来应该是与目标系统密切相关，必须根据实际情况才能讨论，其实也并不绝对，无论目标是何种系统，都还是具备一些的共性的。譬如，在确定目标系统前我们无法决定要收集什么指标，但指标的数据类型（Metrics Types）来来去去也基本就以下这几类，对于一个通用的度量系统来说，面向指标的数据类型来设计服务即可满足各种目标系统的监控所需：

- 计数度量器（Counter）：这是最好理解也是最常用的指标形式，计数器就是对有相同量纲、可加减数值的合计量，譬如业务指标像销售额、货物库存量、职工人数等等；技术指标像服务调用次数、网站访问人数等都属于计数器指标。
- 瞬态度量器（Gauge）：瞬态度量器比计数器还要简单，它就表示某个指标在某个时点的数值，连加减统计都不需要。譬如当前Java虚拟机堆内存的使用量，这就是一个瞬态度量器；又譬如，网站访问人数是计数器，而网站在线人数则是瞬态度量器。
- 吞吐率度量器（Meter）：吞吐率度量器顾名思义是用于统计单位时间的吞吐量，即单位时间内某个事件的发生次数。譬如交易系统中常以“TPS”衡量吞吐率，即每秒发生了多少笔事务交易；又譬如港口的货运吞吐率常以“吨/每天”为单位计算，10万吨/天的港口要比1万吨/天的港口的货运规模更大。
- 直方图度量器（Histogram）：直方图在是常见二维统计图，它的两个坐标分别是统计样本和该样本对应的某个属性的度量，以长条图的形式具体数值。举个具体例子，譬如经济报告中要衡量某个地区历年的GDP变化情况，常会以GDP为纵坐标，时间为横坐标构成直方图来呈现。

- 采样点分位图度量器（Quantile Summary）：分位图是统计学中通过比较各分位数的分布情况的工具，用于验证实际值与理论值的差距，评估理论值与实际值之间的拟合度。譬如，我们说“高考成绩一般符合正态分布”，这句话的意思是：高考成绩高低分的人数都较少，中等成绩的较多，将人数按不同分数段统计，得出的统计结果一般能够与正态分布的曲线较好地拟合。
- 除了以上常见的度量器之外，还有Timer、Set、Fast Compass、Cluster Histogram等其他各种度量器，采用不同的度量系统，支持度量器类型的范围肯定会有差别，譬如Prometheus支持了上面提到五种度量器中的Counter、Gauge、Histogram和Summary四种，其他未提及的度量器囿于篇幅就不再逐一介绍了。

对于“如何将这些指标告诉服务端”这个问题，有两种解决方案：拉取式采集（Pull-Based Metrics Collection）和推送式采集（Push-Based Metrics Collection）。所谓Pull是指度量系统主动从目标系统中拉取指标，相对地，Push就是由目标系统主动向度量系统推送指标。这两种方式并没有绝对的好坏优劣，以前很多老牌的度量系统，如Ganglia[↗]、Graphite[↗]、StatsD[↗]等是基于Push的，而以Prometheus、Datadog[↗]、Collectd[↗]为代表的另一派度量系统则青睐Pull式采集（Prometheus官方解释选择Pull的原因[↗]）。Push还是Pull的抉择，不仅仅在度量中才有，所有涉及到Client和Server通讯的场景，都会涉及到该谁主动（或者双方都可以主动）的问题，譬如上一节中讲的追踪系统也是如此。



Prometheus组件流程图（图片来自Prometheus官网 [↗](#)）

一般来说，度量系统只会支持其中一种指标采集方式，因为度量系统的网络连接数量（以及对应的线程或者协程数）可能非常庞大，如何采集指标将直接影响到整个度量系统的架构设计。Prometheus基于Pull架构的同时还能够（有限度 [↗](#) 地）兼容Push式采集，是因为它有Push Gateway（如上图所示）的存在，这是一个位于Prometheus Server外部的相对独立的中介模块，将外部推送来的指标放到Push Gateway中暂存，然后再等候Prometheus Server从Push Gateway中去拉取。Prometheus设计Push Gateway的本意是为了解决Pull的一些固有缺陷，譬如目标系统位于内网（NAT）之中，外网的Prometheus是无法主动连接目标系统的，只能由目标系统主动推送数据；又譬如对小型短生命周期服务，可能还等不及Prometheus来拉取，服务就已经结束运行了，因此也只能由服务自己Push来保证度量的及时准确。

由推和拉决定该谁主动之后的另一个问题是指标应该以怎样的网络访问协议、取数接口、数据类型来获取？这个问题曾经的解决方向是“标准化”，跟计算机科学的很多其他领域一样，定义一个专门用于度量的协议，目标系统按照协议与度量系统交互。譬如网络管理中的[SNMP ↗](#)、Windows硬件的[WMI ↗](#)、以及此前提到的Java的[JMX ↗](#)都属于这种思路的产物。结果是这些协议都只会在自己那一块小块领域上流行，一方面业务系统要使用这些协议并不容易，你可以想像一下，让订单金额存到SNMP中，让Golang的系统把指标放到JMX Bean里，即便技术上可行，这些也不像是正常程序员会做的事；另一方面，度量系统并不甘心局限于某个领域，成为某项业务的附属品，度量面向的是广义上的信息系统，横跨存储（日志、文件、数据库）、通讯（消息、网络）、中间件（HTTP服务、API服务），直到系统本身的业务指标，甚至还会包括度量系统本身（部署两个独立的Prometheus互相监控是很常见的）。所以，上面这些度量协议都没有成为最终答案的希望。

既然没有标准，有一些度量系统，譬如老牌的Zabbix就支持了SNMP、JMX、IPMI等多种常见的协议，另一些度量系统……说的就是Prometheus相对任性，选择任何一种协议都不去支持，只设计了通过HTTP访问度量端点这一种访问方式。如果目标提供了HTTP的度量端点（如Kubernetes、Etcd等本身就带有Prometheus的Client Library）便直接访问，否则就需要Exporter来充当媒介。Exporter是目标应用的代表，既可以独立运行，也可以与应用运行在同一个进程中（只要集成Prometheus的Client Library便可）。Exporter以HTTP协议（Prometheus在2.0版本之前支持过Protocol Buffer，目前已不再支持）返回符合Prometheus格式要求的文本数据给Prometheus Server。

得益于Prometheus的良好社区生态，现在已经有大量各种用途的Exporter，让Prometheus的监控范围几乎能涵盖所有你所关心的目标，如下表所示。通常你只需要针对自己系统业务方面的度量指标编写Exporter即可。

范围	常用Exporter
数据库	MySQL Exporter、Redis Exporter、MongoDB Exporter、MSSQL Exporter等
硬件	Apcupsd Exporter, IoT Edison Exporter, IPMI Exporter、Node Exporter等
消息队列	Beanstalkd Exporter、Kafka Exporter、NSQ Exporter、RabbitMQ Exporter等
存储	Ceph Exporter、Gluster Exporter、HDFS Exporter、ScaleIO Exporter等
HTTP服务	Apache Exporter、HAProxy Exporter、Nginx Exporter等
API服务	AWS ECS Exporter, Docker Cloud Exporter、Docker Hub Exporter、GitHub Exporter等
日志	Fluentd Exporter、Grok Exporter等
监控系统	Collectd Exporter、Graphite Exporter、InfluxDB Exporter、Nagios Exporter、SNMP Exporter等
其它	Blockbox Exporter、JIRA Exporter、Jenkins Exporter, Confluence Exporter等

顺便一提，前文提到了一堆没有希望成为最终答案的协议，一种名为[OpenMetrics](#)的度量规范正在从Prometheus的数据格式中逐渐分离出来，有望成为监控数据格式的国际标准，最终结果如何，要看Prometheus本身的发展情况，还有OpenTelemetry与OpenMetrics的关系如何协调。

存储查询

指标从目标系统采集过来之后，应存储在度量系统中，以便被后续的分析界面、监控预警所使用。存储数据对于计算机软件来说是最常见的操作之一，但如果用传统关系数据库的思路来解决度量系统的存储，结果可能不会太理想。举个具体例子，假设你建设一个中等

规模的、有着200个节点的微服务系统，每个节点要采集的存储、网络、中间件和业务等各种指标加一起，也按200个来计算，监控的频率如果按秒为单位的话，一天时间内就会产生超过34亿条记录：

$$200 \text{ (节点)} \times 200 \text{ (指标)} \times 86400 \text{ (秒)} = 3,456,000,000 \text{ (记录)}$$

也许这种200节点规模的系统，本身一天的业务发生数据都远到不了34亿条，建设度量系统，肯定不能让度量反倒成了系统性能的负担，因此，度量的存储是需要专门研究解决的问题。至于如何解决，让我们先来观察一段Prometheus的真实度量数据，如下所示：

```
{  
    // 时间戳  
    "timestamp": 1599117392,  
    // 指标名称  
    "metric": "total_website_visitors",  
    // 标签组  
    "tags": {  
        "host": "icyfenix.cn",  
        "job": "prometheus"  
    },  
    // 指标值  
    "value": 10086  
}
```

json

每一个度量指标由时间戳、名称、值和一组标签构成，除了时间之外，指标不与任何其他因素相关。指标的数据量固然是不小的，但没有嵌套、没有关联、没有主外键，不必关心范式和事务，这些都是可以针对性优化的地方。事实上，业界早就已经存在了专门针对该类型数据的数据库了，即时序数据库（说明一下，时序数据库对度量来说是很良好的选择，但并不是只有用时序数据库才能解决问题，Prometheus流行之前最老牌的度量系统Zabbix用的就是传统关系数据库）。

额外知识：时序数据库（Time Series Database）

时序数据库用于存储跟随时间而变化的数据，并且以时间（时间点或者时间区间）来建立索引的数据库。

时序数据库最早是应用于工业（电力行业、化工行业）应用的各类型实时监测、检查与分析设备所采集、产生的数据，这些工业数据的典型特点是产生频率快（每一个监测点一秒

钟内可产生多条数据）、严重依赖于采集时间（每一条数据均要求对应唯一的时间）、测点多信息量大（常规的实时监测系统均可达到成千上万的监测点，监测点每秒钟都在产生数据）。

时间序列数据是历史烙印，具有不变性、唯一性、有序性。时序数据库同时具有数据结构简单，数据量大的特点。

针对时序数据的热点只集中在近期数据（Facebook有研究表明85%的查询都与最近26个小时的数据写入有关）、多写少读（95%以上操作是写操作）、几乎无删改（很少删改或者根本不允许删改）、数据只顺序追加等特点，时序数据库可以做出很激进的存储、访问和保留策略（Retention Policies）。譬如以[LSM-Tree](#)代替传统关系型数据库中的[B+Tree](#)作为存储结构（LSM最适合的场景就是写多读少，且几乎不删改的数据，关于数据库存储结构方面的知识比较繁琐，笔者就不展开了）；譬如根据过期时间（TTL）自动删除相关数据以节省存储空间，同时提高查询性能；譬如对数据进行再采样（Resampling）——最近几天的数据可能需要精确到秒，而查询一个月前的时，只需要精确到天，查询一年前的数据，只要精确到周就够了，这样将数据重新采样汇总就可以极大节省了存储空间。时序数据库中甚至还有一种并不罕见却比较极端的形式，叫做[轮替型数据库](#)（Round Robin Database，RRD），以环形缓冲（在“缓存”中介绍过）的思路实现，只能存储固定数量的最新数据，超期的数据就会被轮替覆盖，因此也有着固定的数据库容量。

Prometheus Server自己就内置了一个时序数据库实现，且颇为强大，近几年在[DB-Engines的排名](#)中不断提升，目前已经跃居TSDB的前三。该时序数据库提供了名为PromQL的数据查询语言，能对时序数据进行丰富的查询、聚合以及逻辑运算。某些时序库（如排名第一的[InfluxDB](#)）会提供类SQL风格查询，但PromQL不是，它是一套完全由Prometheus自己定制的数据查询[DSL](#)，写起来风格有点像……嗯像带运算与函数支持的CSS选择器。譬如要查找网站“icyfenix.cn”访问人数，会是如下写法：

```
// 查询命令：  
total_website_visitors{host="icyfenix.cn"}  
  
// 返回结果：  
total_website_visitors{host="icyfenix.cn", job="prometheus"}=(10086)
```

通过PromQL可以实现指标之间的运算、聚合、统计等操作，在查询界面中往往需要通过PromQL计算多种指标的统计结果才能满足监控的需要，语法方面的细节笔者就不详细展开

了，具体可以参考[Prometheus的文档手册](#)。

监控预警

总的来讲，分析和预警是度量的两个最终目的，界面分析和监控预警也是与用户更加贴近的模块，但对度量系统来说，它们都属于相对外围的功能。与追踪系统的情况类似，广义上的度量系统由面向目标系统进行指标采集的客户端（Client，与目标系统进程在一起的Agent，或者代表目标系统的Exporter等都可归为客户端），负责调度、存储和提供查询能力的服务端（Server，Prometheus的服务端是带存储的，但也有很多度量服务端需要配和独立的存储来使用的），以及面向最终用户的终端（Backend，UI界面、监控预警功能等都归为终端）。狭义上的度量系统就只包括客户端和服务端，不包含终端。

严格地讲Prometheus处于狭义和广义的度量系统之间，尽管它确实内置了一个界面解决方案“Console Template”，以模版和JavaScript接口的形式提供了一系列预设的组件（菜单、图表等），让用户编写一段简单的脚本就可以实现可用的监控功能。不过这种可用程度，往往不足以支撑正规的生产部署，只能说是为把度量功能嵌入到系统的某个子系统中提供了一定便利。在生产环境下，大多是Prometheus配合Grafana来进行展示的，这是Prometheus官方推荐的组合方案。但该组合并非唯一选择，如果要搭配Kibana甚至SkyWalking（8.x版之后的SkyWalking支持从Prometheus获取度量数据）来使用也都是可以的。

良好的可视化能力对于度量系统提升产品力十分重要，长期趋势分析（譬如根据对磁盘增长趋势的观察判断什么时候需要扩容）、对照分析（譬如版本升级后对比新旧版本的性能、资源消耗等方面的差异）、故障分析（不仅从日志、追踪自底向上可以分析故障，高维度的度量指标也可能自顶向下寻找到问题的端倪）等分析工作，不仅需要度量指标的持续收集、统计，往往还需要对数据进行可视化，才能让人更容易地从数据中挖掘规律，毕竟数据最终还是要为人类服务的。

除了为分析、决策、故障定位等提供支持的UI界面外，度量信息的另一种主要的消费途径是用来做预警。譬如你希望当磁盘消耗超过90%时给你发送一封邮件甚至是一条微信消息，通知你过来处理，这就是一种预警。Prometheus提供了专门用于预警的Alert Manager，将Alert Manager与Prometheus关联后，可以设置某个指标在多长时间内达到何种条件就会触发预警状态，触发预警后，根据路由中配置的接收器，譬如邮件接收器、Slack接收器、微信接收器、或者更通用的[WebHook](#)接收器等来自动通知用户。

从微服务到云原生

云原生定义（Cloud Native Definition）

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式API。

这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。

—— Cloud Native Definition [↗](#), CNCF [↗](#), 2018

“不可变基础设施”这个概念由来已久。2012年Martin Fowler设想的“[凤凰服务器](#)”与2013年Chad Fowler正式提出的“[不可变基础设施](#)”，都阐明了基础设施不变性所能带来的益处。在CNCF [↗](#)定义的“云原生”概念中，“不可变基础设施”提升到了与微服务平级的重要程度，此时它的内涵已不再局限于方便运维、程序升级和部署的手段，而是升华为向应用代码隐藏分布式架构复杂度、让分布式架构得以成为一种可普遍推广的普适架构风格的必要前提。

前一章以“分布式的基石”为题，介绍了微服务中关键的技术问题与解决方案，在本章，笔者会以“不可变基础设施”的发展为主线，介绍虚拟化容器与服务网格是如何模糊掉软件与

硬件之间的界限，如何在基础设施与通讯层面上帮助微服务隐藏复杂性，解决原本只能由程序员通过软件编程来解决的分布式问题。

虚拟化容器

容器技术可以说是云计算、微服务等许多当前软件业界热门词汇的共同前提，容器最基本的目标是让软件分发部署过程从传统的分发安装包、由人工部署转变为直接分发已经部署好的、包含整套运行环境的虚拟化镜像。在容器技术成熟之前，软件部署过程多由系统管理员拿到静态的二进制安装包，根据软件的部署说明文档准备好正确的操作系统、动态链接库、配置文件、资源权限等各种前置依赖以后，才能将程序正确地运行起来。[Chad Fowler](#)提出“不可变基础设施”这个概念的文章《[Trash Your Servers and Burn Your Code](#)》中开篇就直接吐槽：要把一个不知道打过多少个升级补丁，不知道经历了多少任管理员的系统迁移到其他机器上，无疑是一场灾难。

让软件能够在任何环境、任何物理机器上达到“一次编译，到处运行”是Java早年提出的宣传口号，这不是一个简单的目标，通用意义的“到处运行”仅靠Java语言和Java虚拟机也是很难达成的。一个计算机软件要能够正确运行，需要有以下三方面的兼容性保证：

- **ISA兼容**：目标机器指令集兼容性，譬如ARM架构的计算机无法直接运行面向x86架构编译的程序。
- **ABI兼容**：目标系统或者依赖库的二进制兼容性，譬如Windows系统环境中无法直接运行Linux的程序，又譬如DirectX 12的游戏无法运行在DirectX 9之上。
- **环境兼容**：目标环境的兼容性，譬如没有正确设置的配置文件、服务注册中心、数据库地址、文件系统的权限等等，任何一个环境因素出现错误，都会让你的程序无法正常运行。

人们把使用仿真（Emulation，代表为[QEMU](#)和[Bochs](#)）或者虚拟化（Virtualization，主要包括软件的二进制翻译和硬件虚拟化两种，代表是[VMware](#)和[VirtualBox](#)）技术来解决以上三项兼容性的方法都统称为虚拟化。

额外知识：ISA与ABI

[指令集架构](#)（Instruction Set Architecture，ISA）是计算机体系结构中与程序设计有关的部分，包含了基本数据类型，指令集，寄存器，寻址模式，存储体系，中断，异常处理以

及外部I/O。指令集架构包含一系列的Opcode操作码（即通常所说的机器语言），以及由特定处理器执行的基本命令。

[应用二进制接口](#) (Application Binary Interface , ABI) 是应用程序与操作系统之间或其他依赖库之间的低级接口。ABI涵盖了各种底层细节，如数据类型的宽度大小、对象的布局、接口调用约定等等。ABI不同于应用程序接口 (Application Programming Interface , API) ，API定义的是源代码和库之间的接口，因此同样的代码可以在支持这个API的任何系统中编译，然而ABI允许编译好的目标代码在使用兼容ABI的系统中无需改动就能直接运行。

我们今天所讨论的虚拟化容器并没有提供以上全部三项兼容性，容器化 (Containerization) 只是虚拟化的一个子集，在“容器”这个名称流行之前，它更多被称做[操作系统层虚拟化](#) (OS-Level Virtualization)，只提供了操作系统内核以上的部分ABI兼容性与完整的环境兼容性。这决定了如果没有其他虚拟化手段的帮助，我们在Windows系统上是不能运行Linux的Docker镜像的（现在可以，是因为有其他虚拟机或者[WSL2](#)的支持），反之亦然。也决定了如果你Docker宿主机的内核版本是Linux Kernel 5.5，那无论上面运行的镜像是Ubuntu、RHEL、Fedora、Mint或者任何发行版，看到的内核一定都是相同的Linux Kernel 5.5。

容器的崛起

容器的最初的目的不是为了部署软件，而是为了隔离计算机中的各类资源，以便降低软件开发、测试阶段可能产生的误操作风险，或者专门充当蜜罐^[1]，吸引黑客的攻击，以便监视黑客的行为。

隔离文件：chroot

容器的起点可以追溯到1979年[Version 7 Unix](#)^[2]系统中提供的 `chroot` 命令，它是英文单词“Change Root”的缩写，功能是当某个进程经过 `chroot` 操作之后，它的根目录就会被锁定在参数指定的位置，以后它或者它的子进程将不能再访问和操作该目录之外的其他文件。

1991年，世界上第一个监控黑客行动的蜜罐程序就是使用 `chroot` 来实现的，那个参数指定的根目录当时被作者被戏称为“Chroot监狱”（Chroot Jail，黑客突破 `chroot` 限制的方法就称为Jailbreak）。后来，FreeBSD 4.0系统使用虚拟化技术重新实现了 `chroot` 命令，用它作为系统中进程沙箱隔离的基础，并将其命名为[FreeBSD jail](#)^[3]，再后来，苹果公司又以FreeBSD为基础研发出了举世闻名的iOS操作系统，此时，黑客们就将绕过iOS沙箱机制以root权限任意安装程序的方法称为“越狱”（Jailbreak^[4]），这些故事都是后话了。

2000年，Linux Kernel 2.3.41版内核引入了 `pivot_root` 技术来实现文件隔离，`pivot_root` 直接切换了[根文件系统](#)^[5]（rootfs），有效地避免 `chroot` 命令的安全性问题。本文后续提到的容器技术，如LXC、Docker等也都是使用 `pivot_root` 来实现根文件系统的切换的。

时至今日，`chroot` 命令依然活跃在Unix和几乎所有主流的Linux发行版中，同时以命令行工具（`chroot(8)`^[6]）或者系统调用（`chroot(2)`^[7]）的形式存在，但无论是 `chroot` 命令抑或是 `pivot_root`，都并不能提供完美的隔离性。按照[Unix的设计哲学](#)^[8]（In Unix, Everything is a File），一切资源都可以视为文件，一切处理都可以视为对文件的操作，隔离了文件系统本该安枕无忧才对。可是，哲学归哲学，现实归现实，从硬件层面暴露的低层次资源，如磁盘、网络、内存、处理器，到经操作系统层面封装的高层次资源，如Unix分时

(Unix Time-Sharing , UTS) 、进程ID (Process ID , PID) 、用户ID (User ID , UID) 、进程间通讯 (Inter-Process Communication , IPC) 都存在大量以非文件形式暴露的操作入口，以 chroot 为代表的文件隔离，仅仅是容器崛起之路的第一步。

隔离访问：namespaces

2002年，Linux Kernel 2.4.19版内核引入了一种全新的隔离机制：[Linux名称空间](#)（Linux Namespaces）。名称空间的概念在很多现代的高级程序语言中都存在，常用于避免不同开发者提供的API相互冲突，相信作为一名开发人员的你应该不会陌生。

Linux的名称空间是一种由内核直接提供的全局资源封装，是内核针对进程设计的访问隔离机制。进程在一个独立的Linux名称空间中朝系统看去，会觉得自己仿佛就是这方天地的主人，拥有这台Linux主机上的一切资源，不仅文件系统是独立的，还有着独立的PID编号（譬如拥有自己的0号进程，即系统初始化的进程）、UID/GID编号（譬如拥有自己独立的root用户）、网络（譬如完全独立的IP地址、网络栈、防火墙等设置），等等，此时进程的心情简直不能再好了。

Linux的名称空间是受“[贝尔实验室九号项目](#)”（一个分布式操作系统，“九号”项目并非代号，操作系统的名字就叫“Plan 9 from Bell Labs”，满满的赛博朋克风格）的启发而设计的，最初的目的依然只是为了隔离文件系统，而非为了什么容器化的实现。这点从2002年发布时只提供了Mount名称空间，并且其构造参数为“CLONE_NEWNS”（即Clone New Namespace）而非“CLONE_NEWMOUNT”便能看出一些端倪。后来，要求系统隔离其他访问操作的呼声愈发强烈，从2006年起，内核陆续添加了UTS、IPC等名称空间隔离，直到目前最新的Linux Kernel 5.6版内核为止，Linux名称空间支持以下八种资源的隔离（内核的官网[Kernel.org](#)上仍然只列出了前六种，从Linux的Man命令能查到全部八种）：

名称空间	隔离内容	内核版本
Mount	隔离文件系统，功能上大致可以类比 chroot	2.4.19
UTS	隔离主机的Hostname、Domain names	2.6.19
IPC	隔离进程间通讯的渠道（详见“ 远程服务调用 ”中对IPC的介绍）	2.6.19

名称空间	隔离内容	内核版本
PID	隔离进程编号，无法看到其他名称空间中的PID，意味着无法其他进程产生影响	2.6.24
Network	隔离网络资源，如网卡、网络栈、IP地址、端口，等等	2.6.29
User	隔离用户和用户组	3.8
Cgroup	隔离 <code>cgroups</code> ，进程有自己的 <code>cgroups</code> 的根目录视图（在/ <code>proc/self/cgroup</code> 不会看到整个系统的信息）。 <code>cgroups</code> 的话题很重要，稍后笔者会安排一整节来介绍	4.6
Time	隔离系统时间，2020年3月最新的5.6内核开始支持进程独立设置系统时间	5.6

如今，对文件、进程、用户、网络等各类信息的访问，都被囊括在Linux的名称空间中，即使一些今天仍有没被隔离的访问（譬如[syslog](#)就还没被隔离，容器内可以看到容器外其他进程产生的内核syslog），日后也可以随内核版本的更新纳入到这套框架之内，现在距离完美的隔离性就只差最后一步了：资源的隔离。

隔离资源：`cgroups`

如果要让一台物理计算机中的各个进程看起来像独享整台虚拟计算机的话，不仅要隔离各自进程的访问操作，还必须能够隔离或者说分配好各个进程对资源的使用配额，不然的话，一个进程发生了内存溢出或者占满了处理器，其他进程就莫名其妙地被牵连挂起，这样肯定算不上是完美的隔离。

Linux系统解决以上问题的方案是[控制群组](#)（Control Groups，目前常用的简写为 `cgroups`），它与名称空间一样都是直接由内核提供的功能，用于隔离或者说限制、分配一个进程组能够使用的资源配额，譬如处理器时间、内存大小、磁盘I/O速度，等等，具体可以参见下表所列：

控制组子系统	功能
blkio	为块设备（如磁盘，固态硬盘，USB 等等）设定I/O限额
cpu	控制 <code>cgroups</code> 中进程的处理器占用比率

控制组子系统	功能
cpuacct	自动生成 cgroups 中进程所使用的处理器时间的报告
cpuset	为 cgroups 中的进程分配独立的处理器（包括多路系统的处理器，多核系统的处理器核心）
devices	设置 cgroups 中的进程访问某个设备的权限（读、写、创建三种权限）
freezer	挂起或者恢复 cgroups 中的进程
memory	设定 cgroups 中进程使用内存的限制，并自动生成内存资源使用报告
net_cls	使用等级识别符标记网络数据包，可允许Linux流量控制程序识别从具体 cgroups 中生成的数据包
net_prio	用来设置网络流量的优先级
hugetlb	主要针对于HugeTLB系统进行限制
perf_event	允许Perf工具基于 cgroups 分组做性能监测

cgroups 项目最早是由Google的工程师（主要是Paul Menage和Rohit Seth）在2006年发起的，当时取的名字就叫做“进程容器”（Process Containers），不过“容器”（Container）这个名词的定义在那时候尚不如今天清晰，不同场景中常有不同所指，为避免混乱，2007年这个项目才被重命名为 cgroups，在2008年合并到2.6.24版的内核后对外发布，这个阶段的 cgroups 被称为“第一代 cgroups”。在2016年3月发布的Linux Kernel 4.5中，搭载了由Facebook工程师（主要是Tejun Heo）重新编写的“第二代 cgroups”（关键改进是支持Unified Hierarchy^[2]），目前这两个版本的 cgroups 在Linux内核中是并存的，Docker暂时仅支持第一版的 cgroups。

封装系统：LXC

当文件系统、访问、资源都可以被隔离后，容器已经有它降生所需的全部前置支撑条件，而且Linux的开发者们也已经明确地看到了这一点。为降低普通用户综合使用 namespaces、cgroups 这些低级特性的门槛，2008年Linux Kernel 2.6.24内核刚刚开始提供 cgroups 的第一时间，就一并发布了名为Linux容器^[2]（LinuX Containers，LXC）的系统级虚拟

化▣特性。此前，Linux也并不是没有系统级虚拟化的解决方案，如传统的OpenVZ▣和Linux-VServer▣都能够实现容器隔离，并且只会有很低的性能损失（按OpenVZ提供的数据，只会有1-3%的损失），但它们都是非官方的技术，最大的阻碍是系统级虚拟化必须要有内核的支持，它们就只能通过非官方内核补丁的方式修改标准内核才能获得那些原本不存在的能力。

LXC带着令人瞩目的光环登场，它的出现促使“容器”从一个阳春白雪的只流传于开发人员口中的技术词汇，逐渐向整个信息业界的公共概念共同语言发展，就如同今天的“服务器”、“客户端”、“互联网”一样。相信你肯定会好奇为什么今天提到容器，大家首先联想到的是Docker而不是LXC，为什么去问10个开发人员，至少有9个听过Docker，但如果问LXC，可能只有1个人会听说过。

LXC的出现肯定受到了OpenVZ和Linux-VServer的启发，摸着巨人的肩膀过河这并没有什么不对。可惜的是，LXC在设定自己的发展目标时，也被前辈们的影响所局限。LXC眼中的容器的定义与OpenVZ和Linux-VServer并无差别，是一种**封装系统**的轻量级虚拟机，Docker眼中的容器的定义则是一种**封装应用**的技术手段。这两种封装理念在技术层面并没有什么差别，但应用效果就差异巨大。举个具体例子，如果你要建设一个LAMP▣（Linux、Apache、MySQL、PHP）应用，按照LXC的思路，你应该寻找到LAMP的template▣（可以暂且不准确地类比为LXC版本的Dockerfile吧），以此构造出一个LAMP容器。如果按部署虚拟机的角度来说，这的还算挺方便的，作为那个时代（距今也就十年）的系统管理员，所有软件、补丁、配置都是自己搞定的，部署一台新虚拟机要花费一两天时间都很正常，有LXC的template，一下子帮你把LAMP都安装好了，还想要啥自行车？但是，作为一名现代的系统管理员，这里问题就相当大，如果我想把LAMP改为LNMP（Linux、Nginx、MySQL、PHP）该怎么办？如果我想把LAMP里的MySQL 5调整为MySQL 8该怎么办？都得找到或者自己编写新的template来解决。好吧，那这台机的软件、版本都配置对了，下一台机我要构建LYME▣或者MEAN▣，又该怎么办？以封装系统为出发点，仍是按照先装系统然再装软件的思路，就永远无法做到一两分钟甚至几秒钟就构造出一个合乎要求的软件运行环境，也决定了LXC不可能形成今天的容器生态的，所以，接下来舞台的聚光灯就打向了Docker身上。

封装应用：Docker

2013年宣布开源的Docker毫无疑问是容器发展里程中里程碑式的发明，然而Docker的成功似乎没有太多技术驱动的成分。至少对开源早期的Docker而言，确实没有什么能构成壁垒的技术。它的容器化能力直接来源于LXC，它镜像分层组合的文件系统直接来源于AUFS¹，Docker开源后不久，就有人仅用了一百多行Shell脚本便实现了Docker的核心功能（名为Bocker²，实现了docker build/pull/images/ps/run/exec/logs/commit/rm/rmi等功能）。

为何历史选择了Docker，而不是LXC或者其他容器技术呢？对于这个问题，笔者引用（转述非直译，有所精简）DotCloud公司（当年创造Docker的公司，已于2016年倒闭）创始人Solomon Hykes在Stackoverflow上的一段问答³：

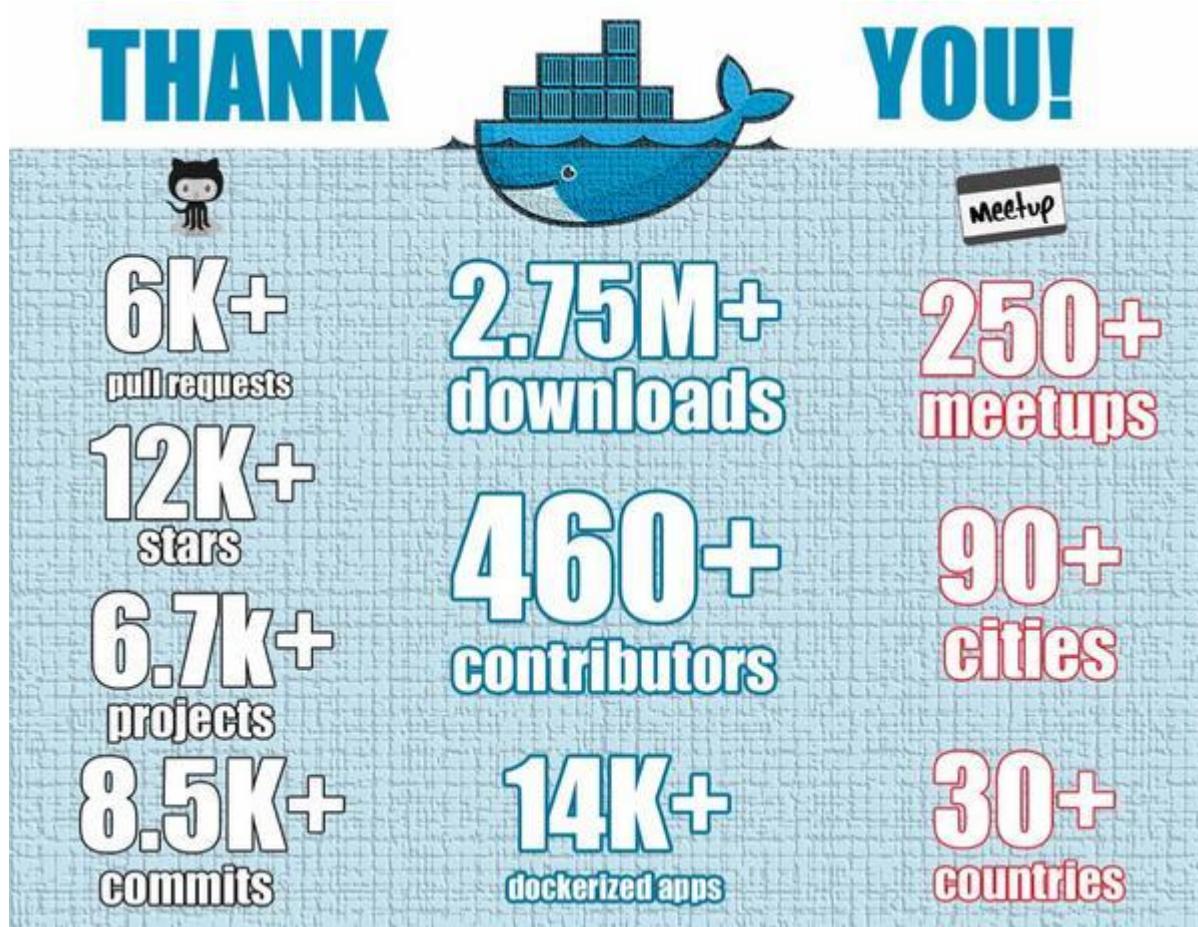
为什么要用Docker而不是LXC？（Why would I use Docker over plain LXC？）

Docker除了包装来自Linux内核的特性之外，它的价值还在于：

- **跨机器的绿色部署**：Docker定义了一种将应用及其所有的环境依赖都打包到一起的格式，仿佛它原本就是绿色软件⁴一样。LXC并没有提供这样的能力，使用LXC部署的新机器很多细节都依赖人的介入，虚拟机的环境几乎肯定会跟你原本部署程序的机器有所差别。
- **以应用为中心的封装**：Docker封装应用而非封装机器的理念贯穿了它的设计、API、界面、文档等多个方面。相比之下，LXC将容器视为对系统的封装，这局限了容器的发展。
- **自动构建**：Docker提供了开发人员从在容器中构建产品的全部支持，开发人员无需关注目标机器的具体配置，即可使用任意的构建工具链，在容器中自动构建出最终产品。
- **多版本支持**：Docker支持像Git一样管理容器的连续版本，进行检查版本间差异、提交或者回滚等操作。从历史记录中你可以查看到该容器是如何一步一步构建成的，并且只增量上传或下载新版本中变更的部分。
- **组件重用**：Docker允许将任何现有容器作为基础镜像来使用，以此构建出更加专业的镜像。
- **共享**：Docker拥有公共的镜像仓库，成千上万的Docker用户在上面上传了自己的镜像，同时也使用他人上传的镜像。
- **工具生态**：Docker开放了一套可自动化和自行扩展的接口，在此之上有可以实现很多工具来扩展其功能，譬如容器编排、管理界面、持续集成等等。

—— Solomon Hykes，Stackoverflow⁵，2013

以上内容也同时被收录到了Docker官网的FAQ上，从开源起至今未变。促使Docker的一问世就惊艳世间的，不是什么黑科技式的秘密武器，而是其符合历史潮流的创意与设计理念，还有充分开放的生态运营。在正确的时候，正确的人手上有一个好点子，确实有机会引爆一个时代。



受到广泛认可的Docker

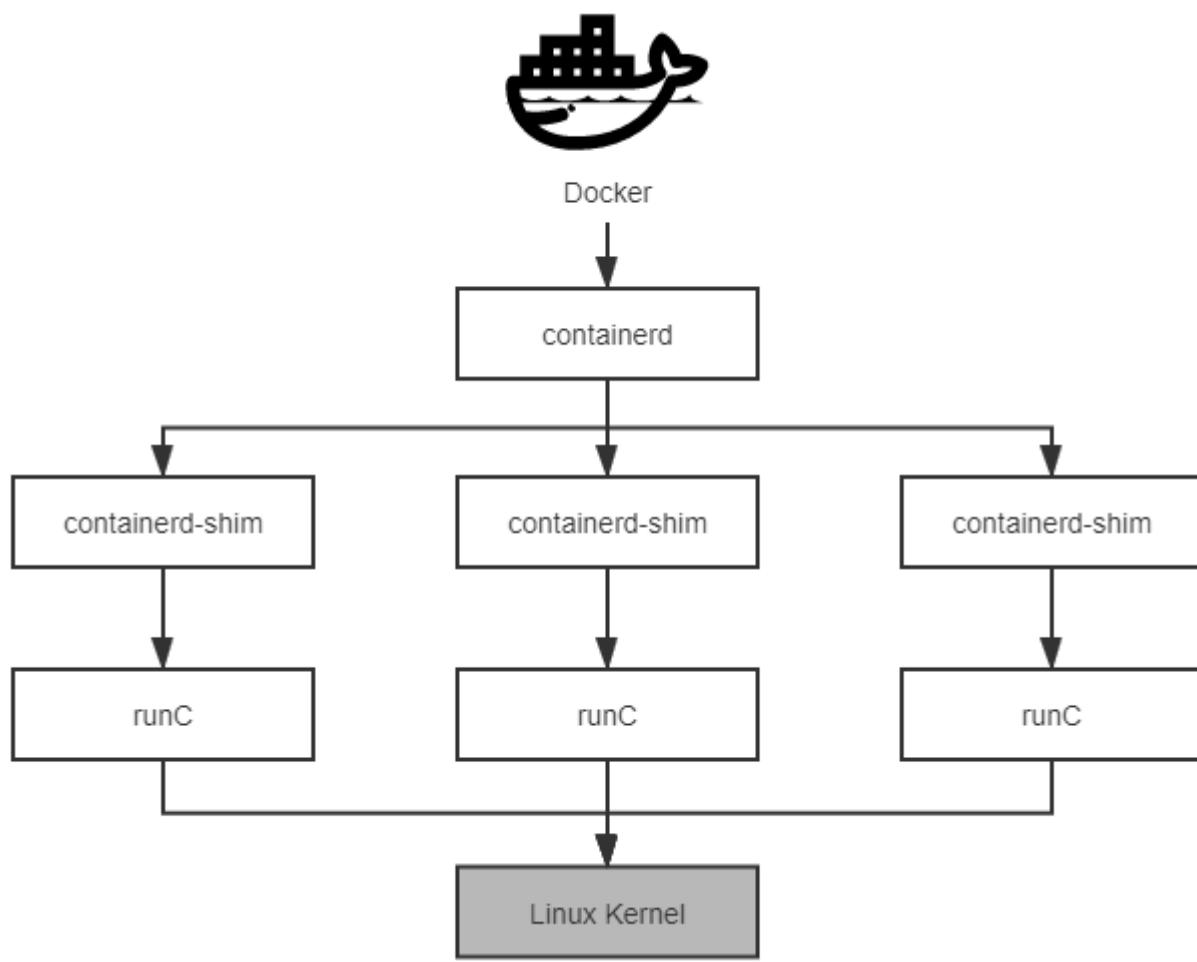
(以上是Docker截至2014年12月获得的成绩，图片来自Docker官网)

从开源到现在也只过了短短数年时间，Docker已成为软件开发、测试、分发、部署等各个环节都难以或缺的基础支撑，自身的架构也发生了相当大的改变，Docker被分解为由Docker Client、Docker Daemon、Docker Registry、Docker Container等子系统，以及Graph、Driver、libcontainer等各司其职的模块组成，此时再说一百多行脚本能实现Docker核心功能，再说Docker没有太高的技术含量，就已经不再合适了。

2014年，Docker开源了自己采用Golang开发的libcontainer，这是一个越过LXC直接操作 namespaces、cgroups 的核心模块，有了libcontainer以后，Docker就能直接与系统内核打交道，不必依赖LXC来提供容器化隔离能力了。

2015年，在Docker的主导和倡议下，多家公司联合制定了[开放容器交互标准](#)（Open Container Initiative，OCI），这是一个关于容器格式和运行时的规范文件，其中包含运行时标准（[runtime-spec](#)）、容器镜像标准（[image-spec](#)）和镜像分发标准（[distribution-spec](#)，分发标准还未正式发布）。运行时标准定义了应该如何运行一个容器、如何管理容器的状态和生命周期、如何使用操作系统的底层特性（namespaces、cgroup、pivot_root等等）；容器镜像标准规定了容器镜像的格式、配置、元数据的格式，可以理解为对镜像的静态描述；镜像分发标准规定了镜像推送和拉取的网络交互过程。

为了符合OCI标准，Docker推动自身的架构继续演变，首先将libcontainer独立出来，封装重构成[runC](#)并捐献给了Linux基金会管理，runC是OCI Runtime的首个参考实现，提出了“让标准容器无所不在（Make Standard Containers Available Everywhere）”的口号。为了能够兼容所有符合标准的OCI Runtime实现，Docker进一步重构了Docker Daemon子系统，将其中与运行时交互的部分抽象为[containerd](#)，这是一个管理容器执行、分发、监控、网络、构建、日志等功能的核心模块，内部会为每个容器运行时创建一个containerd-shim适配进程，默认与runC搭配工作，但也可以切换到其他OCI Runtime实现上（然而实际并没做到，containerd仍是紧密绑定于runC）。在2016年，Docker把containerd捐献给了CNCF，runC与containerd两个项目的捐赠托管，即带有Docker对开源信念的追求，也带有Docker在众多云计算大厂夹击下自救的无奈，它们将成为未来Docker消亡和存续的伏笔（看到本文末尾你就能理解这句矛盾的话了）。



Docker、containerd和runC的交互关系

以上笔者列举的这些Docker推动的开源与标准化工作，既是对Docker为开源乃至整个软件业做出贡献的赞赏，也是为后面介绍容器编排时，讲述当前容器引擎的混乱关系做的一些铺垫。Docker目前无疑处于容器领域的统治地位，但不仅没到高枕无忧，说是危机四伏都不为过。目前已经有了可见的、足以威胁动摇Docker地位的潜在可能性正在酝酿，风险源于虽然Docker赢得了容器战争，但Docker Swarm却输掉了容器编排战争。从结果回望当初，Docker赢得容器战争有一些偶然，Docker Swarm输掉的编排战争却是必然的。

封装集群：Kubernetes

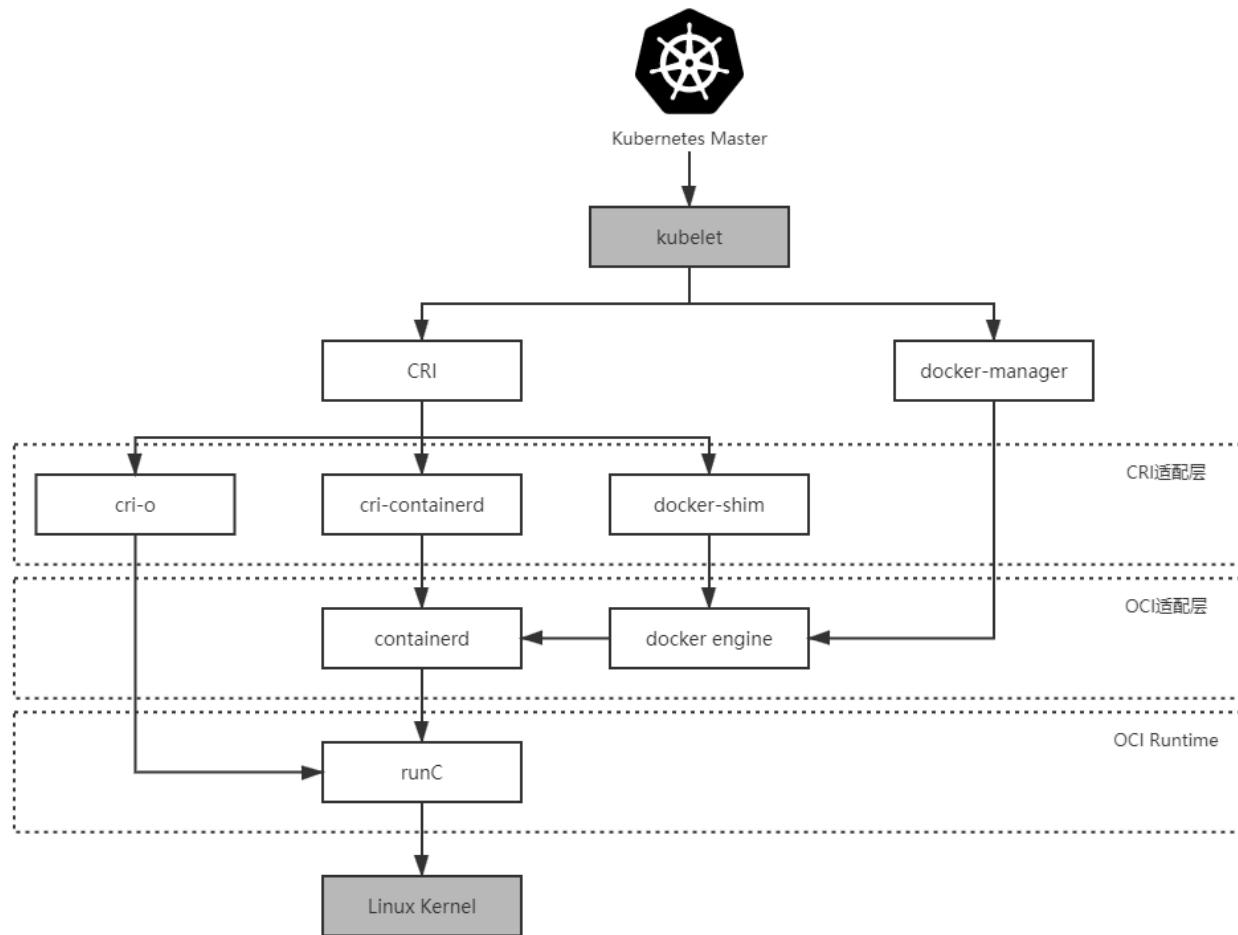
如果说以Docker为代表的容器引擎将软件的发行过程从分发二进制安装包转变为分发虚拟化后的整个运行环境，令应用得以实现跨机器的绿色部署；那以Kubernetes为代表的容器编排框架，就是把大型软件系统运行所依赖的集群环境也进行了虚拟化，令集群得以实现跨数据中心的绿色部署，并能够根据实际情况自动扩缩。

容器的崛起之路讲到Docker和Kubernetes这阶段，已经不再是介绍历史了，从这里开始发生的变化，都是近几年软件业界中的热点事件，也是这章要讨论的主要话题。这里我们先不去介绍Kubernetes的技术细节，它们将会被留到后面的文章中更详细的解析。这节里，我们首先从宏观层面去理解Kubernetes的诞生与演变的驱动力，这对我们正确理解未来云原生的发展方向至关重要。

Kubernetes可谓出身名门，前身是Google内部已运行多年的集群管理系统Borg，2014年6月使用Golang完全重写后开源。自诞生之日起，只要与云计算能稍微扯上关系的业界巨头都对Kubernetes争相追捧，IBM、RedHat、Microsoft、VMware，华为都是它很早期的代码贡献者。此时，云计算从实验室到工业化应用已经有十个年头，然而大量应用使用云计算的方式仍停滞在传统IDC（Internet Data Center）时代，仅仅是用云端的虚拟机代替了传统的物理机。尽管早在2013年，Pivotal（开发Spring Framework和Cloud Foundry的公司）就提出了“云原生”的概念，但是要实现服务化、具备韧性（Resilience）、弹性（Elasticity）、可观测性（Observability）的软件系统十分困难，在当时基本只能依靠架构师和程序员高超的个人能力，云计算本身帮不上什么忙。在云的时代不能充分利用云的强大能力，这让云计算厂商无比遗憾，也无比焦虑，直到Kubernetes诞生，大家才看到了破局的希望。

2015年7月，Kubernetes发布了第一个正式版本1.0版，更重要的事件是Google宣布与Linux基金会共同筹建[云原生基金会](#)（Cloud Native Computing Foundation，CNCF），并且将Kubernetes托管到CNCF，成为其第一个项目。随后，Kubernetes以摧枯拉朽之势覆灭了容器编排领域的其他竞争对手，哪怕Docker Swarm有着Docker在容器引擎方面的先天优势，DotCloud后来甚至将Swarm直接内置入Docker之中都未能阻挡Kubernetes前进的脚步。

Kubernetes的成功与Docker不同，Docker靠的是优秀的理念，以一个“好点子”引爆了一个时代。笔者相信就算没有Docker也会有Cocker或者Eocker的出现，但由成立仅3年的DotCloud公司（3年后又倒闭）做成了这样的产品确实有一定的偶然性。Kubernetes的成功不仅有Google深厚的技术功底作支撑，有领先时代的设计理念，更加关键的是Kubernetes的出现符合所有云计算大厂的切身利益，有着业界巨头不遗余力的广泛支持，它的成功便是一种必然。



在Kubernetes开源的早期，它是完全依赖且绑定Docker的，并没有考虑够日后有使用其他容器引擎的可能性。直至Kubernetes 1.5之前，Kubernetes管理容器的方式都是通过内部的DockerManager向Docker Engine以HTTP方式发送指令，通过Docker来操作镜像的增删改查的，如上图最右边线路的箭头所示（图中的kubelet是集群节点中的代理程序，负责与管理集群的Master通讯，其他节点的含义在后文中都会有解释）。将这个阶段Kubernetes与容器引擎的调用关系捋直，并结合上一节提到的Docker捐献containerd与runC后重构的调用，完整的调用链条如下所示：

Kubernetes Master → kubelet → DockerManager → Docker Engine → containerd → runC

2016年，Kubernetes 1.5版本开始引入[容器运行时接口](#)（Container Runtime Interface，CRI），这是一个定义容器运行时应该如何接入到kubelet的规范标准，从此Kubernetes内部的DockerManager就被更为通用的KubeGenericRuntimeManager所替代（实际上在1.6.6之前都仍然可以选择到DockerManager），kubelet与KubeGenericRuntimeManager之间通过gRPC协议通讯。由于CRI在Docker之后才发布，Docker是肯定不支持CRI的，所以Kubernetes又提供了DockerShim服务作为CRI的适配层，由它与Docker Engine以HTTP形

式通讯，实现了原来DockerManager的全部功能。此时，Docker对Kubernetes来说只是一项默认依赖，而非之前的无可或缺了，它们的调用链为：

```
Kubernetes Master → kubelet → KubeGenericRuntimeManager → DockerShim → Docker  
Engine → containerd → runC
```

2017年，由Google、RedHat、Intel、SUSE、IBM联合发起的[ORI-O](#)（Container Runtime Interface Orchestrator）项目发布了首个正式版本。从名字就可以看出，它肯定是完全遵循CRI标准进行实现的，另一方面，它可以支持所有符合OCI运行时标准的容器引擎，默然仍然是与runC搭配工作的，若要换成[Clear Containers](#)、[Kata Containers](#)等其他OCI运行时也完全没有问题。虽然Kubernetes是使用ORI-O、cri-containerd抑或是DockerShim作为CRI实现，完全可以由用户自由选择，但在RedHat自己扩展定制的Kubernetes企业版，即[OpenShift 4](#)中，调用链已经没有了Docker Engine的身影：

```
Kubernetes Master → kubelet → KubeGenericRuntimeManager → CRI-O → runC
```

由于Docker在容器引擎中的市场份额仍然占有绝对优势，对于普通用户来说，如果没有明确的收益，并没有什么动力要把Docker换成别的引擎，所以CRI-O即使摆出了直接挖掉Docker根基的凶悍姿势，其实也并没有给Docker带来太大的影响，不过能够想像此时Docker心中肯定充斥了难以言喻的危机感。

2018年，由Docker捐献给CNCF的containerd，在CNCF的精心孵化下发布了1.1版，1.1版与1.0版的最大区别是此时它已完美地支持了CRI标准，这意味着原本用作CRI适配器的cri-containerd从此不再需要。此时，再观察Kubernetes到容器运行时的调用链，你会发现调用步骤会比通过DockerShim、Docker Engine与containerd交互的步骤要少两步，这又意味着用户只要愿意抛弃掉Docker情怀的话，在容器编排上便可至少省略一次HTTP调用，获得性能上的收益，且根据Kubernetes官方给出的[测试数据](#)，这些免费的收益还相当可观。Kubernetes的1.10版本宣布开始支持containerd 1.1，在调用链中已经完全抹去Docker Engine的存在：

```
Kubernetes Master → kubelet → KubeGenericRuntimeManager → containerd → runC
```

今天，要使用哪一种容器运行时取决于你安装Kubernetes时机器上的容器运行时环境，但对于云计算厂商，譬如国内的[阿里云ACK](#)、[腾讯云TKE](#)等直接提供的Kubernetes容器

环境，采用的容器运行时普遍都已是containerd，毕竟运行性能对它们来说就是核心生产力和竞争力。

未来，随着Kubernetes的持续发展壮大，Docker Engine经历从不可或缺、默认依赖、可选择、直到淘汰是大概率事件，这件事情表面上是Google、RedHat等云计算大厂联手所为，实际淘汰它的是技术发展的潮流趋势，就如同Docker诞生时依赖LXC，到最后用libcontainer取代掉LXC一般。然而我们也该看到事情的另外一面，现在连LXC都还没有挂掉，反倒还发展出了更加专注于与OpenVZ等系统级虚拟化竞争的LXD¹，相信Docker本身也很难彻底消亡的，已经养成习惯的CLI界面，已经形成成熟生态的镜像仓库等都应该会长期存在，只是在容器编排领域，未来的Docker很可能只会以runC和containerd的形式续存下去，毕竟它们最初都源于Docker的血脉。

以容器构建系统

自从Docker提出“以封装应用为中心”的容器发展理念，成功取代了“以封装系统为中心”的LXC以后，一个容器封装一个单进程应用已经成为被广泛认可的最佳实践。然而单体时代过去之后，分布式系统里应用的概念已不再等同于进程，此时的应用需要多个进程共同协作，以集群的形式对外提供服务，以虚拟化方法实现这个目标的过程就被称为容器编排（Container Orchestration）。

容器之间顺畅地交互通讯是协作的核心需求，但容器协作并不仅仅是将容器以高速网络互相连接而已。如何调度容器，如何分配资源，如何扩缩规模，如何最大限度地接管系统中的非功能特性，让业务系统尽可能免受分布式复杂性的困扰都是容器编排框架必须考虑的问题，只有恰当解决了这一系列问题，云原生应用才有可能获得比传统应用更高的生产力。

隔离与协作

笔者并不打算过多介绍Kubernetes具体有哪些功能，向你说明Kubernetes由Pod、Node、Deployment、ReplicaSet等各种类型的资源组成可用的服务、集群管理平面与节点之间如何工作、每种资源该如何配置使用等等并不是笔者的本意，如果你希望了解这方面信息，可以从Kubernetes官网的文档或任何一本以Kubernetes为主题的使用手册中得到。

笔者真正希望说清楚的问题是“为什么Kubernetes会设计成现在这个样子？”、“为什么以容器构建系统应该这样做？”，寻找这些问题的答案需从它们设计的实现意图出发，为此，笔者虚拟了一系列从简单到复杂的场景供你代入其中，理解并解决这些场景中的问题，并不要求你对Kubernetes已有多深入的了解，但要求你至少使用过Docker，熟悉它的核心功能与命令；此外还会涉及到一点儿Linux系统内核资源隔离的基础知识，别担心，只要你仔细读懂了上一节“[容器的崛起](#)”，就已经足够用了。

现在开始试想一下，如果让你来设计一套容器编排系统，协调各种容器来共同来完成一项工作，会遇到什么问题？会如何着手解决？让我们从最简单的场景出发：

场景一：假设你现在有两个应用，其中一个是Nginx，另一个是为该Nginx收集日志的Filebeat，你希望将它们封装为容器镜像，以方便日后分发。

如果将Nginx和Filebeat编译成同一个容器镜像是可以做到的，而且并不复杂，然而这样做会埋下隐患：这种镜像违背了Docker提倡的单个容器封装单进程应用的理念，Docker设计的Dockerfile只允许有一个ENTRYPOINT，这并非无故添加的人为限制，而是因为Docker只会通过监视PID为1的进程（即由ENTRYPOINT启动的进程）的运行状态来判断容器的状态是否正常，容器退出执行清理，容器崩溃自动重启等操作都必须先判断状态。设想一下，即使我们使用了supervisord¹之类的进程控制器来解决同时启动Nginx和Filebeat进程的问题，如果因某种原因它们不停发生崩溃、重启，那Docker也无法察觉到，它只能观察到supervisord的运行状态，因此，以上需求会理所当然地演化成场景二。

场景二：假设你现在有两个Docker镜像，其中一个封装了HTTP服务，为便于称呼，我们叫它Nginx容器，另一个封装了日志收集服务，我们叫它Filebeat容器。现在要求Filebeat容器能收集Nginx容器产生的日志信息。

这依然很容易解决，只要在Nginx容器和Filebeat容器启动时，分别将它们的日志目录和收集目录挂在为主机同一个磁盘位置的Volume即可，这种操作在Docker中是极为常用的容器间信息交换手段。那我们继续，假如此时我又引入了一个新的工具confd²——Linux下的一种配置管理工具，作用是根据配置中心（Etcd、ZooKeeper、Consul）的变化自动更新Nginx的配置。这里又会遇到一个新问题，confd需要向Nginx发送HUP信号以便通知Nginx³配置已经发生了变更，这要求confd与Nginx能够进行IPC通讯才行。尽管共享IPC名称空间远不如共享Volume常用，但Docker确实支持了该功能，`docker run`提供了`--ipc`参数，用于把多个容器挂在到同一个父容器的IPC名称空间之下，实现共享IPC名称空间。类似地，如果要共享UTS名称空间，可以使用`--uts`参数，要共享网络名称空间的话，就使用`--network`参数，等等。

以上便是Docker针对场景二这种不跨机器的多容器协作所给出的解决方案，这也是Docker Compose⁴的基本工作原理，它可以工作，却并不够优雅，也谈不上有什么系统性。容器的本质是对cgroups和namespaces所提供的隔离能力的一种封装，在Docker提倡的单进程封装的理念影响下，容器蕴含的隔离性也多了仅针对于单个进程的额外局限，然而Linux的cgroups和namespaces原本都是针对进程组而不仅仅是单个进程来设计的，同一个进程组中的多个进程天然就可以共享着相同的访问权限与资源配额。如果现在我们把容器与进程在概念上对应起来，那容器编排的第一个扩展点，就是要找到容器领域中与“进程组”相对

应的概念，这是实现容器从隔离到协作的第一步，在Kubernetes的设计里，这个对应物叫做Pod。

额外知识：Pod名字的由来与含义

Pod的概念在容器化出现之前的Borg系统中就已经存在，从Google发表的《[Large-Scale Cluster Management at Google with Borg](#)》可以看出，Kubernetes时代的Pod整合了Borg时代的“Prod”（Production Task的缩写）与“Non-Prod”的职能。由于Pod一直没有权威的中文翻译，笔者在后续文章中会尽量用英文指代，偶尔需要中文的场合就使用Borg中Prod的译法，即“生产任务”。

扮演“容器组”的角色，解决场景二中的容器共享名称空间的需求，是Pod的两大最基本职责之一，同处于一个Pod内的多个容器，相互之间以超亲密的方式协作。请注意，“超亲密”在这里并非带强烈感情色彩的形容词，而是有一种有具体定义的协作程度，对于普通非亲密的容器，它们一般以网络交互方式（其他譬如共享分布式存储来交换信息也算跨网络）协作；对亲密协作的容器，是指它们被调度到同一个集群节点上，可以通过共享本地磁盘等方式协作；而超亲密的协作是特指多个容器位于同一个Pod这种特殊关系，它们将共享：

- UTS名称空间：所有容器都有相同的主机名和域名。
- 网络名称空间：所有容器都共享一样的网卡、网络栈、IP地址，等等。因此，同一个Pod中不同容器占用的端口不能冲突。
- IPC名称空间：所有容器都可以通过信号量或者POSIX共享内存等方式通讯。
- 时间名称空间：所有容器都共享相同的系统时间。

同一个Pod的容器，只有PID名称空间和文件名称空间默认是隔离的。PID的隔离令每个容器都有独立的进程ID编号，它们封装的应用进程就是PID 1进程，可以通过Pod元数据定义中的 `spec.shareProcessNamespace` 来改变这点。一旦要求共享PID名称空间，容器封装的应用进程就不再具有PID 1了，这有可能导致部分依赖该特性的应用出现异常。在文件名称空间方面，容器要求文件名称空间的隔离是很理所当然的需求，因为容器需要相互独立的文件系统以避免冲突。但容器间可以共享存储卷，这就是通过Kubernetes的Volume来实现。

额外知识：Kubernetes中Pod名称空间共享的实现细节

Pod内部多个容器共享UTS、IPC、网络等名称空间是通过一个名为Infra Container的容器来实现的，这个容器是整个Pod中第一个启动的容器，只有几百KB大小（代码只有很短的几十行，见[这里](#)），Pod中的其他容器都会以Infra Container作为父容器，UTS、IPC、网络等名称空间实质上都是来自Infra Container的。

如果容器设置为共享PID名称空间的话，Infra Container中的进程将作为PID 1进程，其他容器的进程将以它的子进程的方式存在，此时将由Infra Container来负责进程管理（譬如清理[僵尸进程](#)）、感知状态和传递状态。

由于Infra Container的代码除了注册SIGINT、SIGTERM、SIGCHLD等信号的处理器外，就只是一个以pause()方法为循环体的无限循环，永远处于Pause状态，所以也常被称为Pause Container。

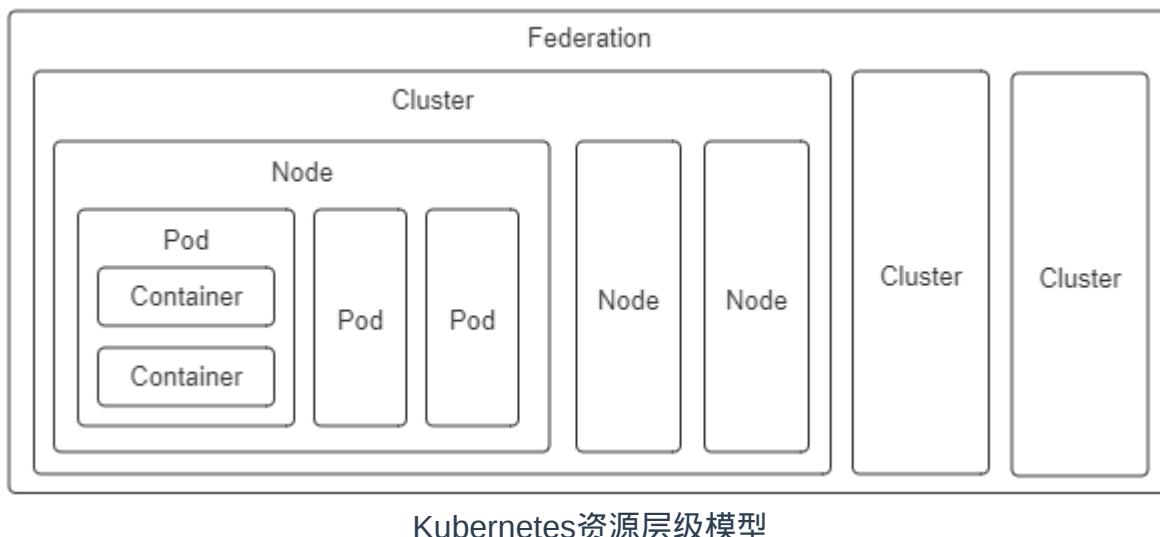
Pod存在的另外一个基本职责是实现原子性调度，如果容器编排不跨越集群节点，是否具有原子性都无关紧要。但是在集群环境中，容器可能跨机器调度时，以容器为单位来调度的话，不同容器就可能被分配到不同机器上。两台机器之间本来就是物理隔离、网络连接的，这时候谈什么名称空间共享、cgropus配额共享都没有意义了，我们由此从场景二又演化出以下场景三。

场景三：假设你现在有Filebeat、Nginx两个Docker镜像，在一个具有多个节点的集群环境下，要求每次调度都必须让Filebeat和Nginx容器运行于同一个节点上。

两个关联的协作任务必须一起调度的需求在容器出现之前就存在已久，譬如在[并发系统](#)的调度中，如果两个线程或者进程是紧密依赖的，单独给谁分配处理时间、而另一个被挂起都会导致不能工作，如此就有了[协同调度](#)（Coscheduling）的概念，以保证一组紧密联系的任务能够被同时分配资源。如果我们在容器编排中仍然坚持将容器视为调度的最小粒度，那对容器运行所需资源的需求声明就要设定在容器上，这样集群每个节点剩余资源越紧张，单个无法容纳全部协同容器的概率就越大，协同的容器被分配到不同节点的可能性就越高。

协同调度是很麻烦的，实现起来要么很低效，譬如Apache Mesos的Resource Hoarding调度策略，就要等所有需要调度的任务都完备后才会开始分配资源；要么就会实现得很复杂，譬如Google就曾针对Borg的下一代Omega系统发表过论文《[Omega: Flexible, Scalable Schedulers for Large Compute Clusters](#)》介绍它如何使用通过乐观调度、冲突回滚的方式做到高效率（也高度复杂）的协同调度。但是如果将运行资源的需求声明定义在Pod

上，直接以Pod为最小的原子单位来实现调度的话，由于Pod之间绝不存在超亲密的协同关系，只通过网络非亲密地协作，那就根本没有协同的说法，自然也不需要考虑复杂的调度了。



Kubernetes资源层级模型

Pod是Kubernetes资源层级模型中唯一仅在逻辑上存在、没有物理对应的概念（因为对应的“进程组”也只是个逻辑概念），也是其他编排系统没有的概念，所以笔者用较大量的篇幅去介绍它的设计意图，而不是像帮助手册那样直接给出它的作用和特性。对于Kubernetes资源层级模型的其他方面，像Node、Cluster等有切实的物理对应物，很容易就能形成共同的认知，笔者也就不逐一详细介绍了。Kubernetes资源层级模型上至最大的Federation，下至最小的Container，每一项都有自己的设计意图，笔者简要列出如下：

- **容器（Container）**：延续了自Docker以来一个容器封装一个应用进程的理念，是镜像管理的最小单位。
- **生产任务（Pod）**：补充了容器化后缺失的与进程组对应的“容器组”的概念，Pod中容器共享UTS、IPC、网络等名称空间，是资源调度的最小单位。
- **节点（Node）**：对应于集群中的单台机器，这里的机器即可以是生产环境中的物理机，也可以是云计算环境中的虚拟节点，节点是处理器和内存等资源的资源池，是硬件单元的最小单位。
- **集群（Cluster）**：对应于整个集群，Kubernetes提倡理念是面向集群来管理应用。当你部署应用的时候，只需要通过声明式API将你的意图写成一份元数据（Manifests），将它提交给集群即可，而无需关心它具体分配到哪个节点（尽管通过标签选择器你也完全可以在控制它分配到哪个节点）、如何实现Pod间通讯、如何保证韧性与弹性，等等，所以集群是处理元数据的最小单位。

- **集群联邦 (Federation)**：对应于多个集群，通过联邦可以统一管理多个Kubernetes集群，联邦的一种常见应用是支持跨可用区域多活、跨地域容灾的需求。

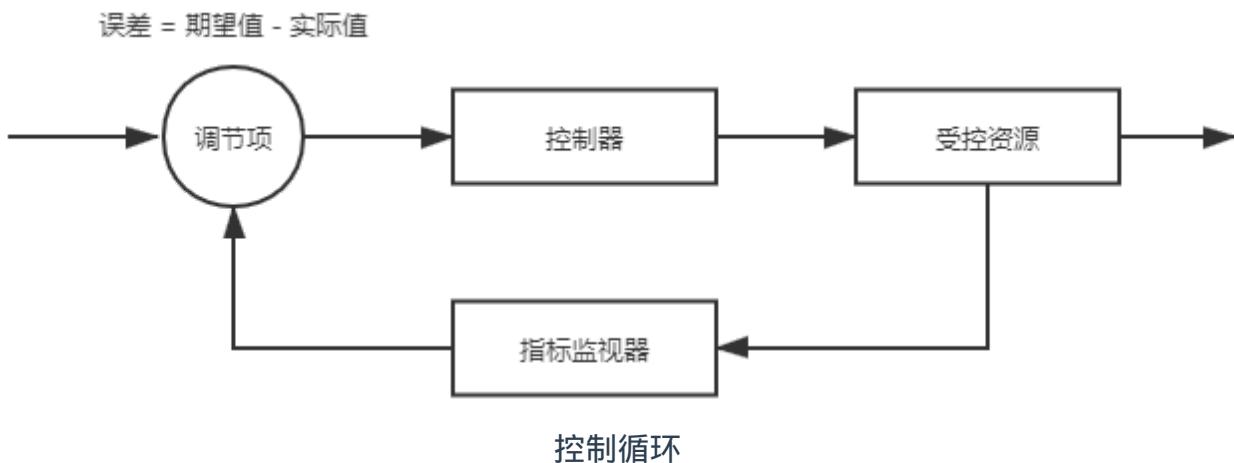
韧性与弹性

笔者曾看过一部叫做《Bubble Boy》的电影，讲述了一个体内没有任何免疫系统的小男孩，终日只能生活在无菌的圆形气球里，对常人来说不值一提的细菌，都能够直接威胁到他的性命。小男孩尽管能够降生于世间，但并不能真正与世界交流，这种生命是极度脆弱的。

真实世界的软件系统与电影世界中的小男孩具有可比性。让容器得以相互连通，相互协作仅仅是以容器构建系统的第一步，我们不仅希望得到一个能够运行起来的系统，而且还希望得到一个能够健壮运行的系统、能够抵御意外与风险的系统。在Kubernetes，你确实可以直接创建Pod将应用运行起来，但这样的应用就如同电影中只能存活在气球中的小男孩一般脆弱，无论是软件代码、意外操作或者硬件故障，都可能导致在复杂协作的过程中某个容器出现异常，进而出现系统性的崩溃。为此，架构师专门设计了服务容错的策略和模式，Kubernetes作为云原生时代的基础设施，也尽力帮助程序员以最小的代价来实现容错，为系统健壮运行提供底层支持。此外，如何实现具有韧性与弹性的系统也是展示Kubernetes控制器设计模式的最好示例，控制器模式是继资源层级模型之后，本节介绍的另一个Kubernetes核心设计理念。下面，我们就从如何解决以下场景中的问题开始。

场景四：假设有一个由数十个Node、数百个Pod、近千个Container所组成的分布式系统，要避免系统因为外部流量压力、代码缺陷、软件更新、硬件升级、资源分配等各种原因而出现中断，作为管理员，你希望编排系统能为你提供何种支持？

我当然最希望编排系统能把所有意外因素都消灭掉，让任何每一个服务都永远健康永不出错，不过这个愿望大概只有凑齐七颗龙珠才有望办到。那退而求其次，让编排系统在这些服务出现问题，状态不正确的时候，能自动将它们调整成正确的状态。这种需求听起来也挺贪心的，但起码合理些，应对的解决办法[工业控制系统](#)里已经很常见，叫做**控制回路**（Control Loop）。[Kubernetes官网](#)中以房间中空调自动调节温度为例子介绍了控制回路的一般工作过程：当你设置好了温度，就是告诉空调你对温度的“期望状态”（Desired State），而传感器测量出的房间实际温度是“当前状态”（Current State）。根据当前状态与期望状态的差距，控制器对空调制冷的开关进行调节控制，就能让其当前状态逐渐接近期望状态。



为了能够以声明式的API来描述资源的期望状态，Kubernetes中已经内置了有相当多的资源对象，它们有一部分是已经出现在上节的资源层级模型中对容器运行环境不同粒度的抽象，另一部分则对应于安全、服务、令牌、网络等许多具体功能的抽象。每种资源都可以向Kubernetes声明它的期望状态，如果你已有实际操作Kubernetes的经验，那在元数据文件中的 `spec` 字段所描述的便是资源的期望状态。

额外知识：Kubernetes的资源对象与控制器

目前，Kubernetes已内置支持相当多的资源对象，并且还可以使用CRD（Custom Resource Definition）来自定义扩充，你可以使用 `kubectl api-resources` 来查看它们。笔者根据用途分类列举了以下常见的资源：

- 用于描述如何创建、销毁、更新、扩缩Pod，包括：Autoscaling（HPA）、CronJob、DaemonSet、Deployment、Job、Pod、ReplicaSet、StatefulSet
- 用于配置信息的设置与更新，包括：ConfigMap、Secret
- 用于持久性地存储文件或者Pod之间的文件共享，包括：Volume、LocalVolume、PersistentVolume、PersistentVolumeClaim、StorageClass
- 用于维护网络通讯和服务访问的安全，包括：SecurityContext、ServiceAccount、Endpoint、NetworkPolicy
- 用于定义服务与访问，包括：Ingress、Service、EndpointSlice
- 用于划分虚拟集群、节点和资源配置，包括：Namespace、Node、ResourceQuota

这些资源在控制器管理框架中一般都会有相应的控制器来管理，笔者列举常见的控制器，按照它们的启动情况分类如下：

- 必须启用的控制器：EndpointController、ReplicationController、PodGCController、ResourceQuotaController、NamespaceController、ServiceAccountController、GarbageC

ollectorController、DaemonSetController、JobController、DeploymentController、ReplicaSetController、HPAController、DisruptionController、StatefulSetController、CronJobController、CSRSigningController、CSRApprovingController、TTLController

- 默认启用的可选控制器，可通过选项禁止：TokenController、NodeController、Service Controller、RouteController、PVBinderController、AttachDetachController
- 默认禁止的可选控制器，可通过选项启用：BootstrapSignerController、TokenCleanerController

相应地，只要是状态会自动发生变化的资源对象，通常都会有对应的控制器进行追踪，每个控制器至少会追踪一种类型的资源。此外，Kubernetes还设计了统一的控制器管理框架（[kube-controller-manager](#)）来维护这些控制器的正常运作，以及统一的指标监视器（[kube-apiserver](#)）来为控制器工作时提供其追踪资源的度量数据。

由于篇幅所限，笔者无法将每个控制器都详细展开讲解，毕竟不是在写Kubernetes的操作手册。尽管如此，只要将场景四进一步具体化，转换成场景五，便可以得到一个很好的例子，以部署控制器（Deployment Controller）、副本集控制器（ReplicaSet Controller）和自动扩缩控制器（HPA Controller）为例来介绍Kubernetes控制器模式的工作原理。

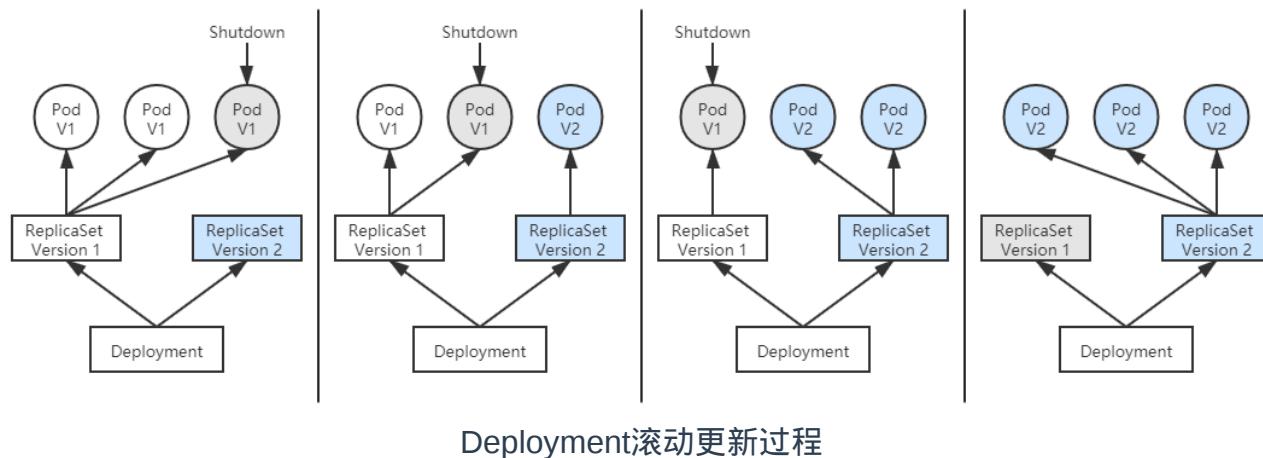
场景五：通过服务编排，对任何分布式系统自动实现以下三种通用的能力：

1. Pod出现故障时，能够自动恢复，不中断服务；
2. Pod更新程序时，能够滚动更新，不中断服务；
3. Pod遇到压力时，能够水平扩展，不中断服务；

之前提到，虽然Pod本身也是资源，完全可以直接创建，但这样做是十分脆弱的，并不提倡。正确的做法是通过副本集（ReplicaSet）来创建Pod。ReplicaSet是一种资源，代表一个或多个Pod副本的集合，你可以在ReplicaSet资源的元数据中描述你期望Pod副本的数量（即 `spec.replicas` 的值）。当ReplicaSet成功创建之后，副本集控制器就会持续跟踪该资源，如果有Pod崩溃退出，或者状态异常（你可以在Pod中设置探针，以自定义的方式告诉Kubernetes出现何种情况Pod才算状态异常），ReplicaSet都会自动创建新的Pod来替代它；如果异常多出现了额外数量的Pod，也会被ReplicaSet自动回收掉，总之就是确保任何时候集群中这个Pod副本的数量都向期望状态靠拢。

ReplicaSet本身就足以满足场景五中的第一项能力，可以保证Pod出现故障时自动恢复，但是在升级程序版本时，ReplicaSet不得不主动中断旧Pod的运行，重新创建新版的Pod，这

会造成服务中断。对于那些不允许中断的业务，以前的Kubernetes曾经提供过 `kubectl rolling-update` 命令来辅助实现滚动更新。所谓滚动更新（ Rolling Updates ）是指先停止少量旧副本，维持大量旧副本继续提供服务，当停止的旧副本更新成功、可以提供服务以后，再重复以上操作，直至所有的副本都更新成功。将这个过程放到ReplicaSet上，就是先创建新版本的ReplicaSet，然后一边让新ReplicaSet逐步创建新版Pod的副本，一边让旧的ReplicaSet逐渐减少旧版Pod的副本。



之所以 `kubectl rolling-update` 命令会被淘汰，是因为这样的命令式交互完全不符合Kubernetes的设计理念（这是台面上的说法，笔者觉得淘汰的根本原因肯定是因为它不好用呀），如果你希望改变某个资源的某种状态，应该将期望状态告诉Kubernetes，而不是告诉Kubernetes具体该如何操作。为此，部署资源（ Deployment ）与部署控制器被设计出来，由Deployment来创建ReplicaSet，再由ReplicaSet来创建Pod，当你更新Deployment中的信息（典型的如程序版本）以后，部署控制器就会跟踪到你新的期望状态，自动地创建新ReplicaSet，并逐渐缩减旧的ReplicaSet的副本数，直至升级完成后彻底删除掉旧ReplicaSet，如上图所示。

场景五中最后一种情况，遇到流量压力时，你可以通过手动修改Deployment中的副本数量，或者通过 `kubectl scale` 命令指定副本数量，促使Kubernetes部署更多的Pod副本来应对压力，然而这种扩容方式不仅需要人工参与，且只靠人类经验来判断需要扩容的副本数量的难以做到精确、及时。为此Kubernetes提供了Autoscaling资源和自动扩缩控制器，能够实现自动根据度量指标（如处理器、内存占用率、用户自定义的度量值等）来设置Deployment（或者ReplicaSet）的期望状态，实现当度量指标出现变化时，系统自动按照“Autoscaling → Deployment → ReplicaSet → Pod”这样的层层变更，最终实现根据度量指标自动扩容缩容。

故障恢复、滚动更新、自动扩缩这些特性，在云原生中时代里常被概括成服务的弹性（Elasticity）与韧性（Resilience），这些资源的操作在Kubernetes中也属于是所有教材资料都会讲到的“基础必修课”。如果你准备学习Kubernetes，建议不要死记硬背地学习每个资源的元数据文件该如何编写、有哪些指令、有哪些操作，而是站在更高层次去理解为什么Kubernetes要设计这些资源和控制器，为什么这些资源和控制器会被设计成这种样子。

如果你觉得已经理解了前面的几种资源和控制器的例子，那不妨思考一下：如果我想限制某个Pod持有的最大存储卷数量，应该会如何设计？如果集群中某个Node发生硬件故障，Kubernetes要让调度任务避开这个Node，应该如何设计？如果一旦这个Node重新恢复，Kubernetes要能尽快利用上面的资源，又该如何去设计？只要你真正接受了资源与控制器是贯穿整个Kubernetes的基本理念，即使不去查文档手册，也应该能推想出个大概轮廓。如此以后，当你再去看手册或者源码时，必定能够事半功倍。

应用为中心的封装

容器技术发展的历程有时候不免会让人感到有种“套娃式”的迷惑：容器的崛起缘于chroot、namespaces、cgroups等内核提供的隔离能力，系统级虚拟化技术使得同台机器上互不干扰地运行多个服务成为可能；为了降低用户使用内核隔离能力的门槛，随即又出现了LXC，它是namespaces、cgroups特性的上层封装，使得“容器”一词真正走出实验室，走入工业界实际应用；为了实现跨机器的软件绿色部署，双出现了Docker，它（最初）是LXC的上层封装，彻底改变了软件打包分发的方式，迅速被大量企业广泛采用；为了满足大型系统对服务集群化的需要，轰出现了Kubernetes，它（最初）是Docker的上层封装，让以多个容器共同协作构建出健壮的分布式系统，成为今天云原生时代的技术基础设施。

Kubernetes会是容器化崛起之路的终点线吗？它达到了人们对云原生时代技术基础设施的期望了吗？从能力角度讲，是可以说是的，Kubernetes被誉为云原生的“操作系统”，自诞生之日起就因其出色管理能力、扩展性的与以声明代替命令的交互理念收获了无数喝彩声；但是，从易用角度讲，坦白说差距还非常大，云原生基础设施的其中一个重要目标是接管掉业务系统复杂的非功能特性，让业务研发与运维工作变得足够简单，然而Kubernetes被诟病得最多的就是复杂，自诞生之日起就以陡峭的学习曲线而闻名。

举个具体例子，用Kubernetes部署一套[Spring Cloud版的Fenix's Bookstore](#)，你需要分别部署一个到多个的配置中心、注册中心、服务网关、安全认证、用户服务、商品服务、交易服务，再对每个微服务都配置好相应的Kubernetes工作负载与服务访问，为每一个微服务的Deployment、ConfigMap、StatefulSet、HPA、Service、ServiceAccount、Ingress等资源都编写好元数据。这个过程最难的地方在于要写出合适的元数据描述文件，既需要懂的开发（网关中服务调用关系、使用容器的镜像版本、运行依赖的环境变量这些参数等等，只有开发最清楚），又需要懂运维（要部署多少个服务，配置何种扩容缩容策略、数据库的密钥文件地址等等，只有运维最清楚），有时候还需要懂平台（需要什么的调度策略，如何管理集群资源，一般只有平台组、中间件组或者核心系统组的同学才会关心），一般企业根本找不到合适的角色来为它管理应用。

这个事儿Kubernetes心里其实也挺委屈，因为以上复杂性不能说是Kubernetes带来的，而是分布式系统本身的原罪。对于大规模的分布式集群，无论是最终用户部署应用，还是软

件公司管理应用都存在痛点。应用难以部署的实质是Docker容器镜像封装了单个服务、Kubernetes资源封装了服务集群，却没有一个载体真正封装整个应用，将原本属于应用内部的技术细节圈禁起来，不要暴露给最终用户，让使用者去埋单；应用难以管理矛盾在于封装应用的方法没能将开发、运维、平台等各种角色的关注点恰当地分离。既然微服务时代，应用的形式已经不再是单个进程，那也该到了重新定义“以应用为中心的封装”这句话的时候了。至于具体怎样的封装才算是正确，今天还未有特别明确结论，不过经过人们的尝试探索，已经窥见未来容器的一些雏形，笔者将近几年来研究的几种主流思路列出供你参考。

Kustomize

最初，由Kubernetes官方给出“如何管理应用”的解决方案是“用配置文件来配置配置文件”（这不是绕口令），你可以理解为一种针对YAML的模版引擎的变体。Kubernetes官方认为应用就是一组具有相同目标的Kubernetes资源的集合，如果逐一管理、部署每项资源过于繁琐的话，那就提供一种便捷的方式，把应用中不变的信息与易变的信息分离开来解决管理问题，把应用所有涉及的资源自动生成一个多合一（All-in-One）的整合包来解决部署问题。

完成这项工作的工具叫做[Kustomize](#)，它原本是一个独立的小程序，从Kubernetes 1.14起，已经吸纳入kubectl命令之中，成为随着Kubernetes提供的内置功能。Kustomize使用[Kustomization文件](#)来组织与应用相关的所有资源，Kustomization本身也是一个以YAML格式编写的配置文件，里面定义了构成应用的全部资源，以及资源中需根据情况被覆盖的变量值。

```
k8s
├── base
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   └── service.yaml
└── overlays
    └── prod
        ├── load-loadbalancer-service.yaml
        └── kustomization.yaml
    └── debug
        └── kustomization.yaml
```

只要建立多个Kustomization文件，开发人员就能以[基于基准进行派生](#)（Base and Overlay）的方式，对不同的模式（譬如生产模式、调试模式）、不同的项目（同一个产品对不同客户的定制化）定制出不同的资源整合包。在配置文件里，无论是开发关心的信息，还是运维关心的信息，只要是在元数据中有描述的内容，最初都是由开发人员来编写的，然后在编译期间由负责CI/CD的产品人员针对项目进行定制，最后在部署期间由运维人员通过kubectl的补丁（Patch）机制更改其中需要运维去关注的属性，譬如构造一个补丁来增加Deployment的副本个数，构造另外一个补丁来设置Pod的内存限制，等等。

Kustomize使用Base、Overlay、Patch生成最终配置文件的思路与Docker中分层镜像的思路颇为相似，既规避了以“字符替换”对资源元数据文件的入侵，也不需要用户学习额外的DSL语法（譬如Lua）。从效果来看，使用由Kustomize编译生成的All-in-One整合包还是比较方便的，只要一行命令就能够把应用安装好，本文档附带的[Kubernetes版本](#)和[Istio版本](#)的Fenix's Bookstore都使用了这种方式来发布应用的，你不妨实际体验一下。

对于开发人员，Kustomize只是简化了产品针对不同情况的重复配置，其实并没有真正解决应用管理复杂的问题，要做的事、要写的配置，最终都没有减少；对于运维人员，应用维护不仅仅只是部署那一下，应用的整个生命周期，除了安装外还有更新、回滚、卸载、多版本、多实例、依赖项维护等诸多问题都很麻烦。这些问题需要更加强大的管理工具去解决，即下节介绍的Helm。不过Kustomize能够以极小的成本，在一定程度上分离了开发和运维的工作，不像Helm那样需要一套独立的体系来管理应用，这种轻量便捷，本身也是可算一种价值。

Helm与Chart

另一种更具系统性的管理和封装应用的解决方案参考了各大Linux发行版管理应用的思路，代表为[Deis公司](#)开发的[Helm](#)和它的应用格式Chart。Helm一开始的目标就很明确：如果说Kubernetes是云原生操作系统的话，那Helm就要成为这个操作系统上面的应用商店与包管理工具。

Linux下的包管理工具和封装格式，如Debian系的apt-get命令与dpkg格式、RHEL系的yum命令与rpm格式相信大家肯定不陌生。有了包管理工具，你只要知道应用的名称，就可以很方便地从应用仓库中下载、安装、升级、部署、卸载、回滚程序，而且包管理工具自己知道应用的依赖信息和版本变更情况，具备自管理能力，每个应用需要依赖哪些前置的第三方库，在安装的时候都会一并处理好。

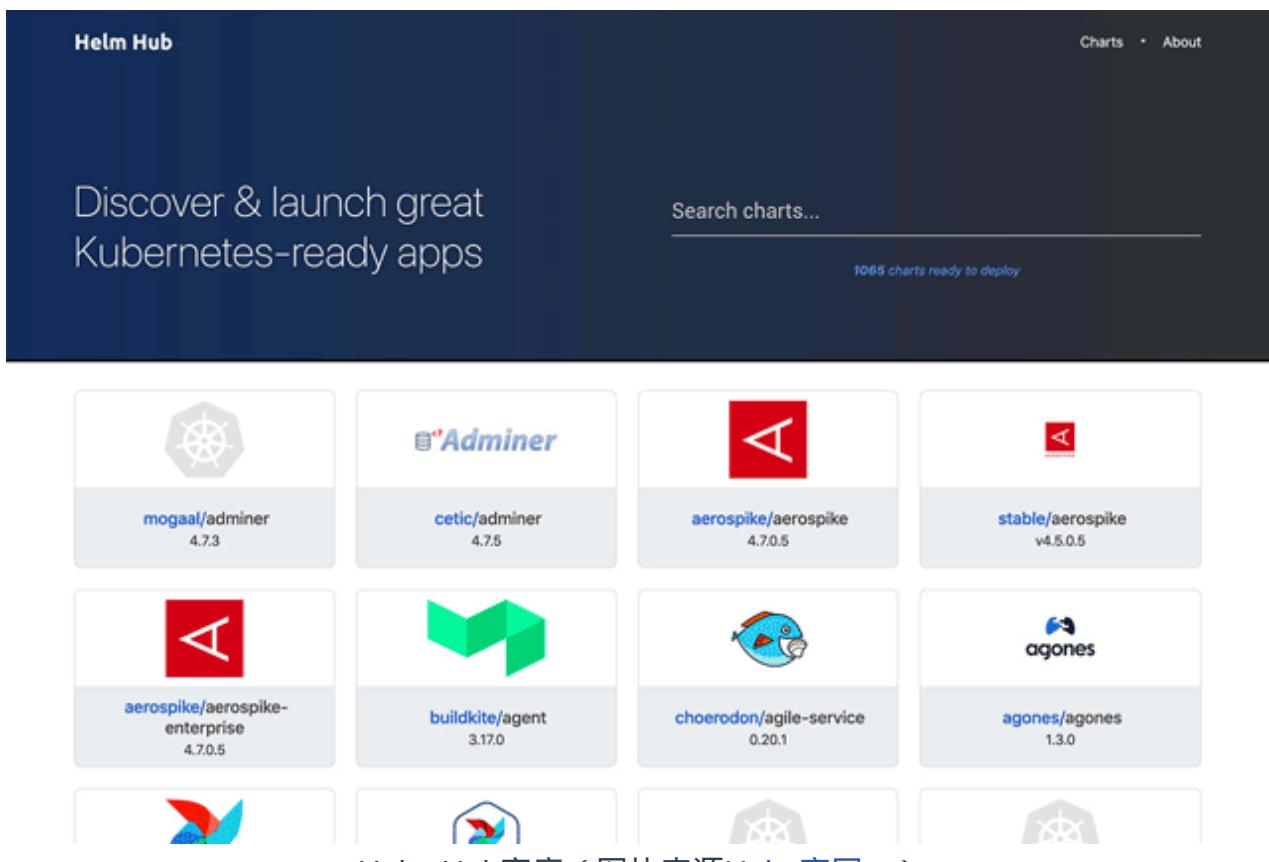
Helm模拟的就是上面这种做法，它提出了与Linux包管理直接对应的Chart格式和Repository应用仓库，针对Kubernetes中特有的一个应用经常要部署多个版本的特点，也提出了Release的概念。

Chart用于封装Kubernetes应用涉及到的所有资源，通常以目录内的文件集合的形式存在。目录名称就是Chart的名称（没有版本信息），譬如官方仓库中WordPress Chart的目录结构是这样的：

```
WordPress
├── templates
│   ├── NOTES.txt
│   ├── deployment.yaml
│   ├── externaldb-secrets.yaml
│   └── 版面原因省略其他资源文件
│       └── ingress.yaml
└── Chart.yaml
└── requirements.yaml
└── values.yaml
```

其中，`Chart.yaml` 给出了应用自身的详细信息（名称、版本、许可证、自述、说明、图标，等等），`requirements.yaml` 给出了应用的依赖关系，依赖项指向的是另一个应用的坐标（名称、版本、Repository地址）。`values.yaml` 给出了所有可配置项目的预定义值。可配置项就是需要部署期间由运维人员调整的那些参数，它们以花括号包裹在`templates` 目录下的资源文件中，部署应用时，Helm会先将管理员设置的值覆盖到`values.yaml` 的默认值上，然后以字符串替换的形式传递给`templates` 目录的资源模版，最后生成要部署到Kubernetes的资源文件。由于Chart封装了足够丰富的信息，所以Helm除了支持命令行操作外，也能很容易地根据这些信息自动生成图形化的应用安装、参数设置界面。

Repository仓库用于实现Chart的搜索与下载服务，Helm社区维护了公开的Stable和Incubator的中央仓库，也支持其他人或组织搭建私有仓库和公共仓库，并能够通过Hub服务把不同个人或组织搭建的公共仓库聚合起来，形成一个大型分布式的应用仓库，有利于Chart的查找与共享，界面如下图所示。



Helm提供了应用全生命周期、版本、依赖项的管理能力，同时，Helm还支持额外的扩展插件，能够加入CI/CD或者其他方面的辅助功能，定位已经从单纯的工具升级为应用管理平台。强大的功能让Helm收到了不少支持，有很多应用主动入驻到官方的仓库中。从2018年起，Helm项目被托管到CNCF，成为其中的一个孵化项目。

Helm以模仿Linux包管理器的思路去管理Kubernetes应用，一定程度上是可行的，不过，Linux与Kubernetes中部署应用还是存在一些差别，最重要的一点是在Linux中99%的应用都只会安装一份，而Kubernetes里为了保证可用性，同一个应用部署多份副本才是更常见的操作。Helm也允许对同一个Chart包进行多次部署，每次安装它都会产生一个Release，这个Release相当于该Chart的安装实例。对于无状态的服务，只要Release能保证多个服务能够并行运行就已经足够了。但对于有状态的服务来说，服务会与特定资源或者服务彼此之间要产生依赖关系，事情就变得复杂起来，Helm无法很好地管理这种有状态的依赖关系，这个问题就是要留待Operator解决的痛点了。

Operator与CRD

Operator不应该被称作是工具或者系统，它应该算是一种封装、部署和管理Kubernetes应用的方法，尤其是针对最复杂的有状态应用去封装运维能力的解决方案，最早由CoreOS公司（于2018年被RedHat收购）的华人程序员邓洪超所提出。如果上一节“[以容器构建系统](#)”介绍Kubernetes资源与控制器模式时你没有开小差的话，那么Operator中最核心的理念你其实已经理解了，简单地说，Operator就是通过Kubernetes 1.7开始支持的自定义资源（Custom Resource Definitions，CRD，此前曾经以TPR，即Third Party Resource的形式提供过类似的能力），把应用封装为更高层次的另一种资源，再把Kubernetes的控制器模式从面向于内置资源，扩展到了面向所有自定义资源。以下笔者引用了一段RedHat官方对Operator设计理念的阐述：

Operator设计理念

Operator是使用自定义资源（CR，笔者注：CR即Custom Resource，是CRD的实例）管理应用及其组件的自定义Kubernetes控制器。高级配置和设置由用户在CR中提供。Kubernetes Operator基于嵌入在Operator逻辑中的最佳实践将高级指令转换为低级操作。Kubernetes Operator监视CR类型并采取特定于应用的操作，确保当前状态与该资源的理想状态相符。

—— [什么是 Kubernetes Operator](#)，RedHat

以上这段文字是由RedHat官方撰写和翻译的，准确严谨，对于没接触过Operator的人却并不那么友好，什么叫做“高级指令”？什么叫做“低级操作”？两者之间具体如何转换？为了能够理解这些问题，我们不妨先理解有状态和无状态应用的含义及影响，然后再来理解Operator所做的工作。

有状态应用（Stateful Application）与无状态应用（Stateless Application）说的是应用程序是否要自己持有其运行所需的数据，如果应用每次执行都跟首次执行一样，不会依赖之前任何操作所遗留下来的痕迹，那它就是无状态的；反之，如果应用推倒重来之后，用户能察觉到该应用已经发生变化，那它就是有状态的。无状态应用在分布式系统中具有非常高的价值，我们都应该知道分布式中CAP不兼容原理，如果无状态，自然就不必考虑状态一致性，没有了C，A和P便兼得，换而言之，只要资源足够，无状态应用天生就是高可用的。但不幸的是现在的分布式系统中多数基础设施都是有状态的，如缓存、数据库、对象存储、消息队列，等等，只有像Web服务器这类设施属于无状态服务。

站在Kubernetes的角度看，是否有状态的本质差异在于有状态应用会对某些外部资源有绑定性的直接依赖，譬如Elasticsearch建立实例时必须依赖特定的存储位置，重启后仍然指向同一个数据文件的实例才能被认为是相同的实例。此外，有状态应用多个应用实例之间往往有着特定到拓扑关系与顺序关系，譬如Etcd的节点间选主、投票，节点们都需要得知彼此的存在。为了管理好那些与应用实例密切相关的状态信息，Kubernetes从1.9版本开始正式发布了StatefulSet及对应的StatefulSetController，与普通ReplicaSet中的Pod相比，StatefulSet中创建的Pod具备以下几个额外特性：

- **Pod会按顺序创建和按顺序销毁**：StatefulSet中的各个Pod会按顺序地创建出来，创建后续的Pod前，必须要保证前面的Pod已经转入就绪状态。删除StatefulSet中的Pod时会按照与创建顺序的逆序来执行。
- **Pod具有稳定的网络名称**：Kubernetes中的Pod都具有唯一的名称，在普通的副本集中这是靠随机字符产生的，而在StatefulSet中管理的Pod，会以带有顺序的编号作为名称，且能够在重启后依然保持不变。。
- **Pod具有稳定的持久存储**：StatefulSet中的每个Pod都可以拥有自己独立的PersistentVolumeClaim资源。即使Pod被重新调度到其它节点上，它所拥有的持久磁盘也依然会被挂载到该Pod。

如果把ReplicaSet中的Pod比喻为养殖场中的肉猪，那StatefulSet就是被家庭当宠物圈养的荷兰猪，不同的肉猪在食用功能上并没有什么区别，但每只宠物猪都是无独一无二的，有专属于自己的名字、习性与记忆，事实上StatefulSet就曾经有一段时间用的名字就是PetSet。StatefulSet出现以后，Pod就能满足Pod重新创建后仍然保留上一次运行状态的需求，不过有状态应用的维护并不仅限于此，譬如对于一套Elasticsearch集群来说，通过StatefulSet只能做到创建集群、删除集群、扩容缩容等基础操作，其他的运维操作，譬如备份恢复数据、创建删除索引、调整平衡策略等操作也很常用，StatefulSet却并不能提供什么帮助。

为了解释清楚Operator在这个问题上能提供的帮助，笔者举个具体例子来说明：要部署一套Elasticsearch集群，通常要在StatefulSet中定义相当多的细节，譬如服务的端口、Elasticsearch的配置、更新策略、内存大小、虚拟机参数、环境变量、数据文件位置，等等，为了便于你对Kubernetes的“复杂”有更加直观的体验，咱就奢侈一回，挥霍一点儿版面，将YAML全文贴出如下：

```
yaml
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch-cluster
spec:
  clusterIP: None
  selector:
    app: es-cluster
  ports:
    - name: transport
      port: 9300
---
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch-loadbalancer
spec:
  selector:
    app: es-cluster
  ports:
    - name: http
      port: 80
      targetPort: 9200
  type: LoadBalancer
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: es-config
data:
  elasticsearch.yml: |
    cluster.name: my-elastic-cluster
    network.host: "0.0.0.0"
    bootstrap.memory_lock: false
    discovery.zen.ping.unicast.hosts: elasticsearch-cluster
    discovery.zen.minimum_master_nodes: 1
    xpack.security.enabled: false
    xpack.monitoring.enabled: false
    ES_JAVA_OPTS: -Xms512m -Xmx512m
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: esnode
```

```
spec:  
  serviceName: elasticsearch  
  replicas: 3  
  updateStrategy:  
    type: RollingUpdate  
  template:  
    metadata:  
      labels:  
        app: es-cluster  
  spec:  
    securityContext:  
      fsGroup: 1000  
    initContainers:  
      - name: init-sysctl  
        image: busybox  
        imagePullPolicy: IfNotPresent  
        securityContext:  
          privileged: true  
        command: ["sysctl", "-w", "vm.max_map_count=262144"]  
    containers:  
      - name: elasticsearch  
        resources:  
          requests:  
            memory: 1Gi  
        securityContext:  
          privileged: true  
          runAsUser: 1000  
        capabilities:  
          add:  
            - IPC_LOCK  
            - SYS_RESOURCE  
        image: docker.elastic.co/elasticsearch/elasticsearch:7.9.1  
        env:  
          - name: ES_JAVA_OPTS  
            valueFrom:  
              configMapKeyRef:  
                name: es-config  
                key: ES_JAVA_OPTS  
        readinessProbe:  
          httpGet:  
            scheme: HTTP  
            path: /_cluster/health?local=true  
            port: 9200  
            initialDelaySeconds: 5  
        ports:
```

```

    - containerPort: 9200
      name: es-http
    - containerPort: 9300
      name: es-transport
  volumeMounts:
    - name: es-data
      mountPath: /usr/share/elasticsearch/data
    - name: elasticsearch-config
      mountPath: /usr/share/elasticsearch/config/elasticsearch.yml
      subPath: elasticsearch.yml
  volumes:
    - name: elasticsearch-config
      configMap:
        name: es-config
        items:
          - key: elasticsearch.yml
            path: elasticsearch.yml
  volumeClaimTemplates:
    - metadata:
        name: es-data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 5Gi

```

需要大量的细节配置的根本原因在于Kubernetes并不知道Elasticsearch是什么东西，所有Kubernetes不知道的信息、不能启发式推断出来的信息，都必须在资源的元数据定义中明确列出。你必须一步一步手把手地“教会”Kubernetes如何部署Elasticsearch，这种形式就属于RedHat在Operator设计理念介绍中所说的“低级操作”。

如果我们使用[Elastic.co官方提供的Operator](#)，那就会便捷得多了，Elasticsearch Operator提供了一种 kind: Elasticsearch 的自定义资源，在它的帮助下，你仅需10行代码，将你的意图是“部署三个版本为7.9.1的ES集群节点”说清楚即可，便能实现与前面StatefulSet那一大堆配置相同的效果，如下面代码所示。显然此时Kubernetes已经学会怎样操作了Elasticsearch，知道所有它相关的参数含义与默认值，无需你再手把手地教了，这种就是所谓的“高级指令”。

```

apiVersion:.elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch

```

yaml

```
metadata:  
  name: elasticsearch-cluster  
spec:  
  version: 7.9.1  
  nodeSets:  
    - name: default  
      count: 3  
      config:  
        node.master: true  
        node.data: true  
        node.ingest: true  
        node.store.allow_mmap: false
```

Operator将简洁的“高级指令”转化为Kubernetes中实际操作的方法与前面Helm或者Kustomize并不相同。Helm和Kustomize最终仍然是依靠Kubernetes的内置资源来跟Kubernetes打交道的，Operator则是让开发者自己实现一个专门针对该CRD的控制器，在控制器中维护自定义资源的期望状态。譬如当需要更新集群中某个Pod对象的时候，由Operator开发者自己编码实现的控制器完全可以在原地对Pod进行重启，而无需像Deployment那样必须先删除旧Pod，然后再创建新Pod。

使用CRD定义高层次资源，使用配套的控制器来维护期望状态，带来的好处不仅仅是“高级指令”的便捷，而是遵循Kubernetes一贯基于资源与控制器的设计原则的同时，又不必再受制于Kubernetes内置资源的表达能力。只要Operator的开发者愿意编写代码，前面曾经提到那些StatefulSet不能支持的能力，如备份恢复数据、创建删除索引、调整平衡策略等操作，都完全可以实现出来。

把运维的操作封装在应用代码上，表面上最大的受益者是运维人员，开发人员要付出更多劳动。然而Operator并没有被开发者抵制，让它变得小众，反而由于代码相对于资源配置的表达能力提升，让开发与运维之间的协作成本降低而备受开发者的好评。Operator变成了近两、三年容器封装应用的一股新潮流，现在很多复杂分布式系统都有了官方或者第三方提供的的Operator（[这里收集了其中一部分](#)），RedHat也持续在Operator上面大量投入，推出了简化开发人员编写Operator的[Operator Framework/SDK](#)。

目前来看，应对有状态应用的封装运维问题，Operator也许是最可行的方案，但这依然不是一项轻松的工作，譬如[Etcd的Operator](#)，Etcd本身不算什么特别复杂的应用，Operator实现的功能看起来也挺基础，主要有创建集群、删除集群、扩容缩容、故障转移、滚动更新、备份恢复等，其代码就已经超过一万行了。现在开发Operator的确还是有着相对较高

的门槛，通常由开发而非运维人员去完成，但是Operator符合技术潮流，顺应软件业界所提倡的DevOps一体化理念，等Operator的开发支持和生态进一步成熟之后，开发和运维都能从中受益，未来会是大有可为的。

开放应用模型

本节介绍的最后一种应用封装的方案，是阿里云和微软在2019年10月上海QCon大会上联合发布的[开放应用模型](#)（Open Application Model，OAM），它不仅是中国云计算企业参与制定乃至主导发起的国际技术规范，也是业界首个云原生应用标准定义与架构模型。

开放应用模型思想的核心是如何将开发人员、运维人员与平台人员关注点分离，开发人员关注业务逻辑的实现，运维人员关注程序平稳运行，平台人员关注基础设施的能力与稳定性，长期让几个角色厮混在同一个All-in-One资源文件里，并不能擦出什么火花，反而将配置工作弄得越来越复杂，将“[YAML Engineer](#)”弄成了容器界的嘲讽梗。

开放应用模型把云原生应用定义为“由一组相互关联但又离散独立的组件构成，这些组件实例化在合适的运行时上，由配置来控制行为并共同协作提供统一的功能”。没看明白并没有关系，为了便于跟稍后的概念对应，笔者先把这句话翻译为你更加看不明白的另一种形式：

OAM定义的应用

一个 `Application` 由一组 `Components` 构成，每个 `Component` 的运行状态由 `Workload` 描述，每个 `Component` 可以施加 `Traits` 来获取额外的运维能力，同时我们可以使用 `Application Scopes` 将 `Components` 划分到一或者多个应用边界中，便于统一做配置、限制、管理。把 `Components`、`Traits` 和 `Scopes` 组合在一起实例化部署，形成具体的 `Application Configuration`，以便解决应用的多实例部署与升级。

然后，笔者通过解析上述所列的核心概念来帮助你理解OAM对应用的定义。这句话里面每一个用英文标注出来的技术名词都是OAM在Kubernetes基础上扩展而来概念，每一个名词都有专门的CRD与之对应，换而言之，它们并非纯粹的抽象概念，而是可以被实际使用的资源定义。这些概念的具体含义是：

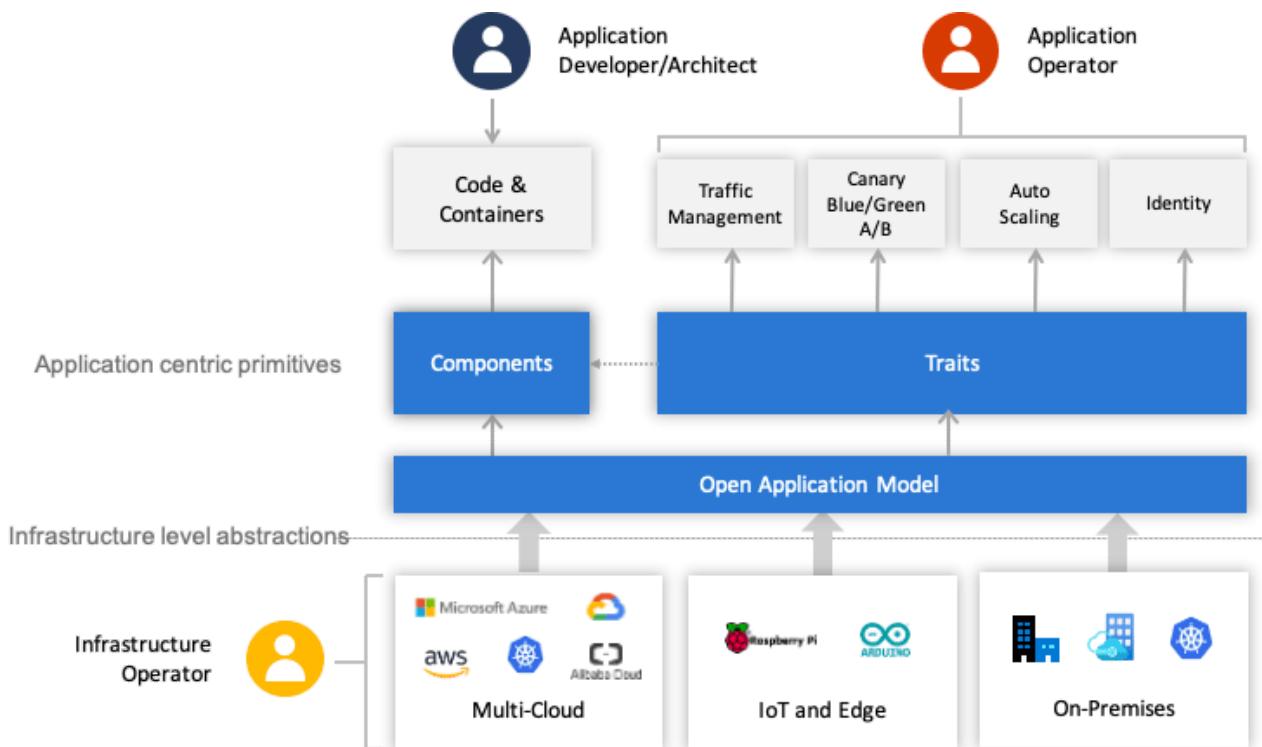
- **服务组件（Components）**：由Component构成应用的思想自SOA以来就屡见不鲜，然而OAM的Component不仅仅是特指构成应用“整体”的一个“部分”，它还有一个重要职责

是抽象那些应该由开发人员去关注的元素。譬如应用的名字、自述、容器镜像、运行所需的参数，等等。

- 工作负荷 (Workload)**：Workload决定了应用的运行模式，每个Component都要设定自己的Workload类型，OAM按照“是否可访问、是否可复制、是否长期运行”预定义了六种Workload类型，如下表所示。如有必要还可以通过CRD与Operator去扩展。

工作负荷	可访问	可复制	长期运行
Server	√	√	√
Singleton Server	√	✗	√
Worker	✗	√	√
Singleton Worker	✗	✗	√
Task	✗	√	✗
Singleton Task	✗	✗	✗

- 运维特征 (Traits)**：开发活动有大量能力复用的技巧，但运维活动却很贫乏，平时能为运维写个Shell脚本或者简单工具已经算是高级的运维人员了。OAM的Traits用于封装模块化后的运维能力，可以针对运维中的可重复操作预先设定好一些具体的Traits，譬如日志收集Trait、负载均衡Trait、水平扩缩容Trait，等等。这些预定义的Traits定义里，会注明它们可以作用于哪种类型的工作负荷、包含能填哪些参数、哪些必填选填项、参数的作用描述是什么，等等。
- 应用边界 (Application Scopes)**：多个Component共同组成一个Scope，你可以根据Component的特性或者作用域来划分Scope，譬如具有相同网络策略的Component放在同一个Scope中，具有相同健康度量策略的Component放到另一个Scope中。同时，一个Component也可能属于多个Scope，譬如一个Component完全可能即需要配置网络策略，也需要配置健康度量策略。
- 应用配置 (Application Configuration)**：将Component（必须）、Trait（必须）、Scope（非必须）组合到一起进行实例化，形成一个完整的应用配置。



OAM角色关系图（图片来自[OAM规范GitHub](#)）

OAM使用上述介绍的这些CRD将原先All-in-One的复杂配置做了一定层次的解耦，开发人员负责管理Component；运维人员将Component组合并绑定Trait变成Application Configuration；平台人员或基础设施提供方负责提供OAM的解释能力，将这些CRD映射到实际的基础设施。不同角色分工协作，整体简化了单个角色关注的内容，使得不同角色可以更聚焦更专业的做好本角色的工作，整个过程如上图所示。

OAM未来能否成功，很大程度上寄希望于云厂商的支持力度，譬如阿里云的[EDAS](#)就已内置了OAM的支持。如果你希望能够应用于私有Kubernetes环境，目前OAM的主要实现是[Rudr](#)（已声明废弃）和[Crossplane](#)，Crossplane是一个仅发起一年多的CNCF沙箱项目，主要参与者包括阿里云、微软、Google、RedHat等工程师。Crossplane提供了OAM中全部的CRD以及控制器，安装后便可用OAM定义的资源来描述应用。

后记：今天容器圈的发展是一日千里，各种新规范、新技术层出不穷，本节根据人气和代表性，列举了其中最出名的四种，其他笔者未提到的应用封装技术还有[CNAB](#)、[Armada](#)、[Pulumi](#)等等。这些封装技术会有一定的重叠之处，但并非都是重复的轮子，实际应用时往往会联合其中多个工具一起使用。应该如何封装应用才是最佳的实践，目前尚且没有定论，但是以应用为中心的理念却已经成为明确的共识。

容器间网络

本节我们准备讨论[虚拟化网络](#)的话题。如果不加任何限定，“虚拟化网络”是一项内容十分丰富，研究历史悠久的计算机技术，是计算机科学中一门独立的分支，完全不依附于虚拟化容器而存在。网络运营商常提及的“[网络功能虚拟化](#)”（ Network Function Virtualization , NFV ），网络设备商、网络管理软件提供商常提及的“[软件定义网络](#)”（ Software Defined Networking , SDN ）等等都属于虚拟化网络的范畴，对于普通的软件开发者而言，要完全理解和掌握虚拟化网络，需要储备大量开发中不常用到的专业知识与消耗大量的时间成本。

然而，本节我们讨论的虚拟化网络是狭义的，它特指“如何基于Linux系统的网络虚拟化技术来实现的容器间网络通讯”，更通俗一点说，就是只关注那些为了相互隔离的Linux Network Namespaces间可实现相互通讯而设计出来的虚拟化网络设施，这里所需的网络知识基本还在普通开发者应该具有的合理知识范畴之内。同时，在这个语境中的“虚拟化网络”就是直接为容器服务的，说它是依附于容器而存在的亦无不可，因此为避免混淆，笔者在后文中都刻意回避“虚拟化网络”这个范畴过大的概念，后续的讨论将会以“容器间网络”题来展开。

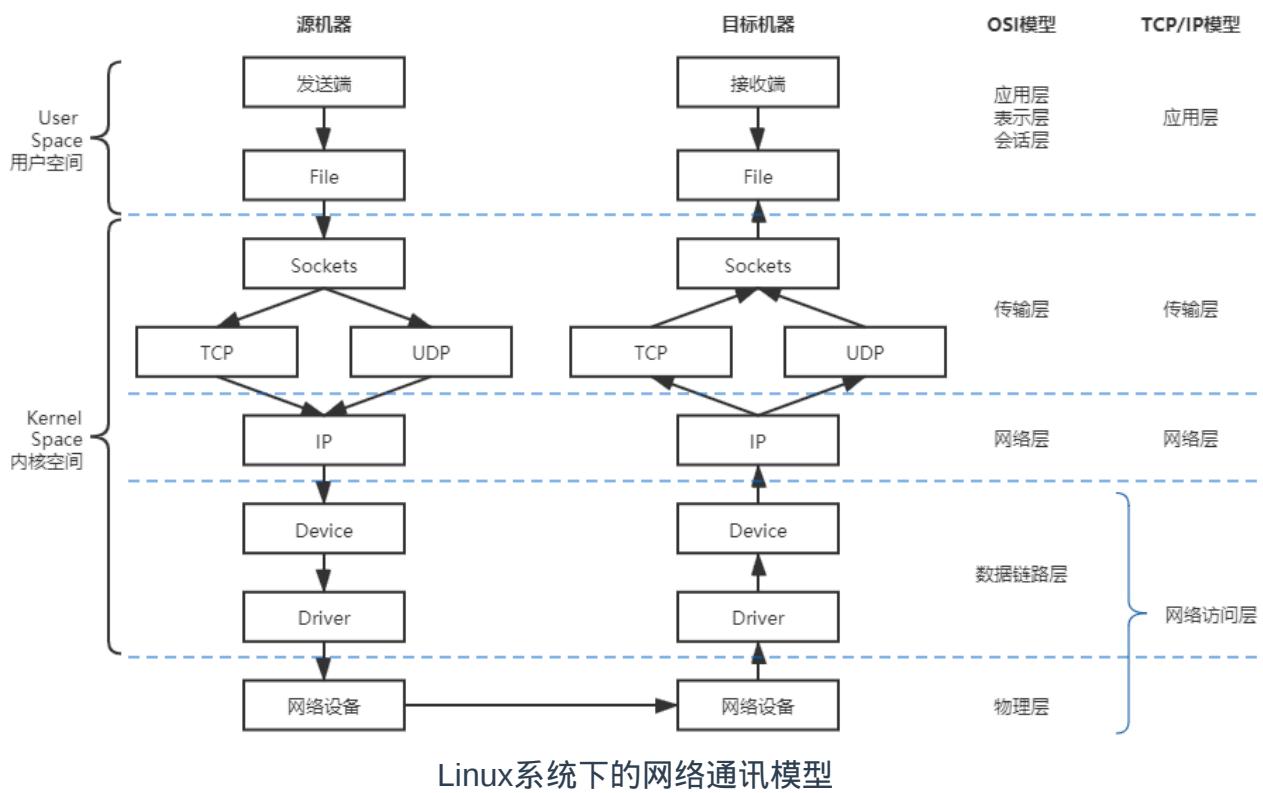
Linux网络虚拟化

Linux目前提供的八种名称空间里，网络名称空间无疑是最为复杂的一种，它为名称空间内的所有进程提供了全套的网络设施，包括独立的设备界面、路由表、ARP表，IP地址表、iptables/ebtables规则、协议栈，等等。虚拟化容器是以Linux名称空间的隔离性为基础来实现的，那解决隔离的容器之间、容器与宿主机之间、乃至跨物理网络的不同容器间通讯问题的责任，很自然也落在了Linux网络虚拟化技术的肩上。本节里，我们暂时放下容器编排、云原生、微服务等这些上层概念，走入Linux网络的底层世界，去学习与设备、协议、通讯相关的基础网络知识。

本节的阅读对象设定为以实现业务功能为主、平常并不直接接触网络设备的普通开发人员，对于平台基础设施的开发者或者运维人员，可能会显得有点过于啰嗦或过于基础了，如果你已经提前掌握了这些知识，完全可以快速阅读，或者直接跳过。

网络通讯模型

笔者先抛开虚拟化，只谈网络，介绍一下Linux系统的网络通讯模型，即信息是如何从程序中发出，通过网络传输，再被另一个程序接收到的。整体上看，Linux系统的通讯过程无论按理论上的OSI七层模型，还是以实际上的TCP/IP四层模型来解构，都明显地呈现出“接口逐层调用，数据逐层封装”的特点，这种逐层处理的方式与程序执行时的方法栈很类似，因此它也常被称为“Linux网络协议栈”，下图体现了Linux网络通讯过程与OSI或者TCP/IP模型的对应关系，也展示了协议栈中的数据流动的路径。



Linux系统下的网络通讯模型

从图中传输模型可见，几乎整个网络栈（应用层以下）都实现在系统内核空间之中，之所以采用这种设计，主要是从安全隔离的角度出发来考虑的，让内核去处理网络栈尽管有较高的开销（数据在内核态和用户态之间的拷贝成本），会损失一些性能，但能够保证应用程序无法窃听到或者去伪造另一个应用程序的通讯内容；如果遇到需要特别关注收发性能的场景，也有直接在用户空间中实现全套协议栈的旁路方案，譬如开源的Netmap²⁵以及Intel的DPDK²⁶，都能做到零拷贝收发包。从数据流动过程来看，信息从程序中发出，将经历如下几个阶段：

- **Socket**：应用层的程序基本上都是通过Socket编程接口来和内核空间的网络协议栈通信的。Linux Socket最初是从BSD Socket发展而来的，现在Socket已经成为各大主流操作系统的通用网络接口，是网络应用程序实际上的交互基础。应用程序通过读写收、发缓冲区（Receive/Send Buffer）来与Socket进行交互，在*nix系统中，出于“一切皆是文件”的设计哲学，对Socket操作被实现为对文件系统（socketfs）的读写访问操作。
- **TCP/UDP**：传输层协议族里最主要就是**传输控制协议**²⁷（Transmission Control Protocol, TCP）和**用户数据报协议**²⁸（User Datagram Protocol, UDP）两种，此外还有**流控制传输协议**²⁹（Stream Control Transmission Protocol, SCTP）、**数据报拥塞控制协议**³⁰（Datagram Congestion Control Protocol, DCCP）等等。

不同的协议处理过程大致是一样的，但封装的信息会有所不同，这里以TCP协议为例：内核发现Socket的发送缓冲区中有新的数据被拷贝进来后，会把数据构建为TCP Segm

ent报文。常见网络协议的报文基本上都是由报文头（Header）和报文体（Body，也叫荷载：Payload）两部分组成。内核将缓冲区中用户要发送出去的数据作为报文体，然后把传输层中的必要控制信息，譬如代表哪个程序发、哪个程序收的源、目标端口号，用于保证可靠通讯（重发与控制顺序）的序列号、用于校验信息是否在传输中出现损失的校验和（Check Sum）等信息封装入报文头中。

- IP：网络层协议最主要就是网际协议（Internet Protocol，IP），其他还有因特网组管理协议（Internet Group Management Protocol，IGMP）、大量的路由协议（EGP、NHRP、OSPF、IGRP、……）等等。

以IP协议为例，它会将来自上一层（例子中的TCP的报文）的数据包作为报文体，再次加入自己的报文头，譬如指明数据应该发到哪里的路由地址、数据包的长度、协议的版本号，等等，封装成IP数据包后发往下一层。关于IP协议报文头的内容，笔者曾在“[负载均衡](#)”中讲解过，有需要的读者可以参考。

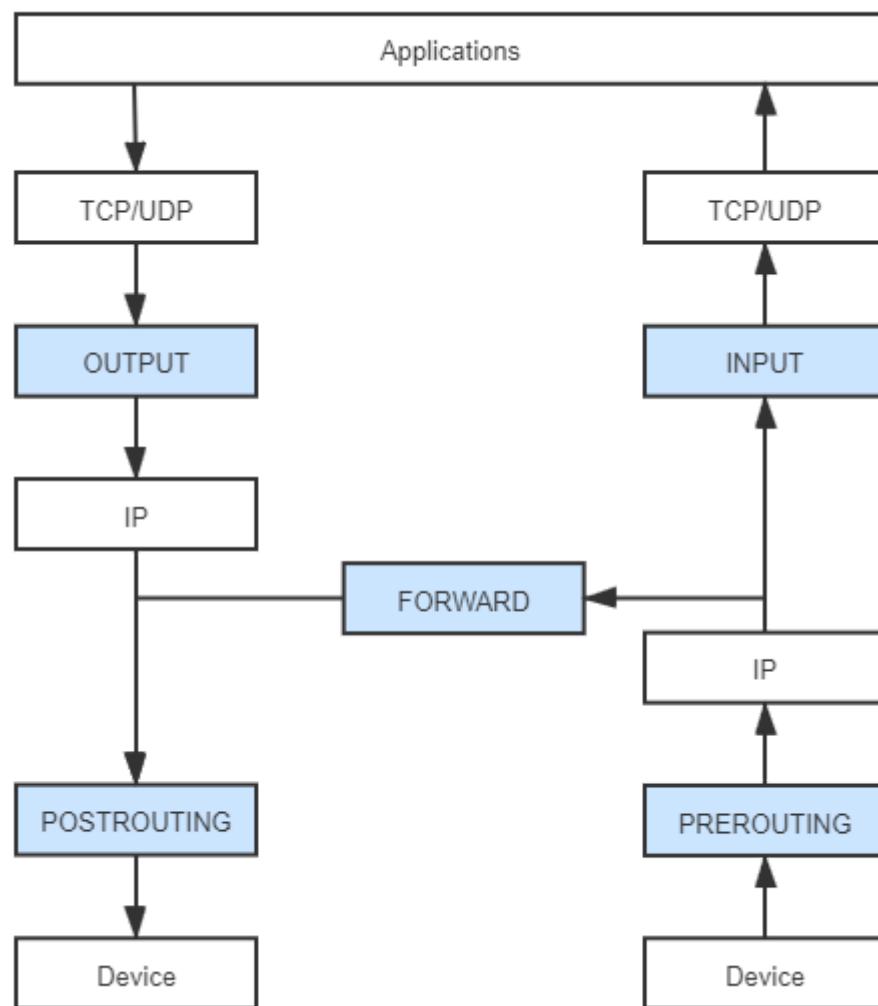
- Device：Device是网络访问层中面向系统一侧的接口，这里所说的设备（Device）与物理硬件设备并不是同一个概念，Device只是一种向操作系统端开放的接口，其背后既可能代表着真实的物理硬件，也可能是某段具有特定功能的程序代码。许多网络抓包工具，如[tcpdump](#)、[Wireshark](#)便是在此工作。前面介绍微服务[流量控制](#)时笔者曾提到过的网络流量整形也通常是在这里完成的。Device主要的作用是预留出一些让代码去介入收发包过程的手段，譬如确定数据应该从哪块网卡设备发送出去；还有就是准备好网卡驱动工作所需的数据，譬如来自上一层的IP数据包、[下一跳](#)（Next Hop）的MAC地址（这个地址是通过[ARP Request](#)得到的）等等。
- Driver：Driver是网络访问层中面向硬件一侧的接口，网卡驱动程序（Driver）会通过[DMA](#)将主存中的待发送的数据包复制到驱动内部的缓冲区之中。数据被复制的同时，也会将上层提供的IP数据包、下一跳MAC地址这些信息，加上网卡的MAC地址、VLAN Tag（这些内容稍后都会介绍到）等信息一并封装成为[以太帧](#)（Ethernet Frame），并自动计算校验和。对于需要确认重发的信息，如果在没有收到确认（ACK）响应，那重发的工作也是在这里自动完成的。

上面这些阶段是信息从程序中对外发出时经过协议栈的过程，接收过程则是从相反方向进行的逆操作。程序发送数据做的是层层封包，加入协议头，传给下一层；接受数据则是层层解包，提取协议体，传给上一层。相信你应该能够举一反三，笔者就不再专门列举一遍数据接收步骤了。

干预网络通讯

网络协议栈的处理过程里，似乎除了在Device这里程序还有一点插手的空间以外，其他过程似乎都没有可供程序介入的余地了。然而事实并非如此，从Linux Kernel 2.4版开始，内核开放了一套通用的、可供代码干预数据在协议栈中流转的过滤器框架，这套名为Netfilter的框架是Linux防火墙和网络的主要维护者Rusty Russell提出并设计的。它围绕网络层（IP协议）的周围，埋下了五个钩子（Hooks），每当有数据包流到网络层，经过这些钩子时，就会自动触发由内核模块注册在这里的回调函数，程序代码就能够通过回调来干预Linux的网络通讯。这五个钩子的名字与含义分别列举如下：

- PREROUTING：来自设备的数据包进入协议栈后立即触发此钩子。PREROUTING钩子在进入IP路由之前触发，这意味着只要接收到的数据包，无论是否真的发往本机，都会触发此钩子。一般用于目标网络地址转换（DNAT）。
- INPUT：报文经过IP路由后，如果确定是发往本机的，将会触发此钩子，一般用于加工发往本地进程的数据包。
- FORWARD：报文经过IP路由后，如果确定不是发往本机的，将会触发此钩子，一般用于处理转发到其他机器的数据包。
- OUTPUT：从本机程序发出的数据包，在经过IP路由前，将会触发此钩子，一般用于加工本地进程的输出数据包。
- POSTROUTING：从本机网卡出去的数据包，无论是本机的程序所发出的，还是由本机转发给其他机器的，都会触发此钩子，一般用于源网络地址转换（SNAT）。



应用收、发数据包所经过的Netfilter钩子

Netfilter允许在同一个钩子处注册多个回调函数，因此向钩子注册回调函数必须提供明确的优先级，以便触发时能按照优先级从高到低进行激活。由于回调函数有多个，看起来就像挂在一个钩子上的一串链条，钩子触发的回调函数集合也被称为“回调链”（Chained Callbacks），这个名字导致了后续基于Netfilter设计的Xtables系工具，如稍后介绍的iptables均有使用到“链”（Chain）的概念。虽然现在看来Netfilter只是一些简单的事件回调机制而已，然而这样一套简单的设计，却成为了整座Linux网络大厦的重要基石，Linux系统提供的许多网络能力，如数据包过滤、封包处理（设置标志位、修改TTL等）、地址伪装、网络地址转换、透明代理、访问控制、基于协议类型的连接跟踪，带宽限速，等等，都是在Netfilter基础之上实现的。

以Netfilter为基础的应用有很多，其中最重要的毫无疑问要数Xtables系列工具，譬如iptables、ebtables、arptables、ip6tables等等。这里面至少iptables应该是用过Linux系统的开发人员都或多或少会使用过，它常被称为是Linux系统自带的防火墙，然而iptables实际能做的事情已远远超出防火墙的范畴，它比较贴切的定位应是能够代替Netfilter多数常规功

能的IP包过滤工具。Netfilter的钩子回调虽然很强大，但毕竟要通过程序编码才够用，并不适合系统管理员用来运维。iptables的价值就便是使用配置去实现原本用Netfilter编码才能做到的事情。它先把用户常用的管理意图总结成行为，部分预置的行为有如下所列：

- DROP：直接将数据包丢弃。
- REJECT：给客户端返回Connection Refused或Destination Unreachable报文。
- QUEUE：将数据包放入用户空间的队列，供用户空间的程序处理。
- RETURN：跳出当前链，该链里后续的规则不再执行。
- ACCEPT：同意数据包通过，继续执行后续的规则。
- JUMP：跳转到其他用户自定义的链继续执行。
- REDIRECT：在本机做端口映射。
- MASQUERADE：地址伪装，自动用修改源或目标的IP地址来做NAT
- LOG：在/var/log/messages文件中记录日志信息。
-

这些行为本来能够被挂载到Netfilter钩子的回调链上，但iptables又抽象了一层，不是把行为与链直接挂钩，而是根据这些底层操作的目的总结为更高层次的规则。举个具体例子，假设你挂载规则目的是实现网络地址转换（NAT），那就要对符合某种特征的流量（譬如来源于某个网段、从某张网卡发送出去）、在某个钩子上（譬如做SNAT通常在POSTROUTING，做DNAT通常在PREROUTING）进行MASQUERADE行为，这样具有相同目的的规则，就应该放到一起形成规则表，才便于管理。iptables内置了五张不可扩展的规则表（其中security表并不常用，很多资料只计算了前四张表），如下所列：

、数据包过滤、修改数据包IP头信息等，形成了五张规则表（security表不常用，很多资料只计算前四张表），如下：

1. raw表：用于去除数据包上的[连接追踪机制](#)（Connection Tracking）。
2. mangle表：用于修改数据包的报文头信息，如服务类型（Type Of Service，TOS）、生存周期（Time To Live，TTL）以及为数据包设置Mark标记，典型的应用是链路的服务质量管理（Quality Of Service，QoS）。
3. nat表：用于修改数据包的源或者目的地址等信息，典型的应用是网络地址转换。
4. filter表：用于对数据包进行过滤，控制到达某条链上的数据包是继续放行、直接丢弃或拒绝（ACCEPT、DROP、REJECT），典型的应用是防火墙。
5. security表：用于在数据包上应用[SELinux](#)，这张表并不常用。

以上五张规则表是具有优先级的：raw → mangle → nat → filter → security，也即是它们出现的顺序。在新增规则时，应该按照规则的意图指定要存入到哪张表中，如果没有指定，默认就是filter表。每张表能够使用到的链也有所不同，具体表与链的对应关系如下所示：

	PREROUTING	POSTROUTING	FORWARD	INPUT	OUTPUT
raw	√	✗	✗	✗	√
mangle	√	√	√	√	√
nat (Source)	✗	√	✗	√	✗
nat (Destination)	√	✗	✗	✗	√
filter	✗	✗	√	√	√
security	✗	✗	√	√	√

预置的五条链直接源自于Netfilter的钩子，它们与五张规则表的对应关系是固定的。用户不能增加自定义的表，但可以增加自定义的链，新增的自定义链与钩子没有天然的对应关系，换而言之就是不会自动触发，必须使用JUMP行为从默认的五条链中跳转过去才能被执行。

iptables不仅仅是Linux系统自带的一个网络工具而已，它在容器间通讯中扮演相当重要的角色，譬如Kubernetes用来管理Service的核心组件kube-proxy，就依赖iptables来完成ClusterIP到Pod的通讯（也可以采用IPVS，IPVS同样是基于Netfilter的），这种通讯就是一种NAT访问。如果你平常较少在Linux系统下工作，可能需要一些用iptables充当防火墙过滤数据、当作路由器转发数据、当作网关做NAT转换的实际例子来帮助理解，由于这些操作在网上很容易找到，笔者就不专门举例了。

笔者用两个小节的篇幅去介绍Linux下网络通讯的协议栈模型，以及程序如何干涉在协议栈中流动的信息，它们与虚拟化并没有什么直接关系，而是整个Linux网络通讯的必要基础。从下一节开始，我们就要开始专注于与网络虚拟化密切相关的内容了。

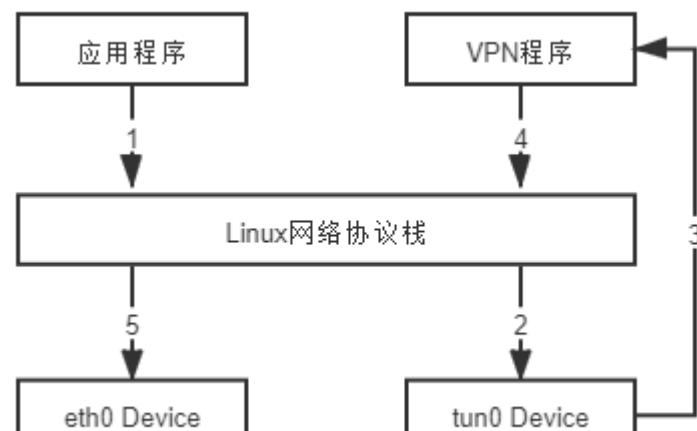
虚拟化网络设备

虚拟化的网络并不需要完全遵照物理网络的样子来设计，不过，由于有大量现成的代码原本就是面向于物理存在的网络设备来实现的，也有出于方便理解和知识继承的方面的考虑，虚拟化网络与物理网络中的设备还是有相当高的可比性。所以，笔者准备从虚拟网络中那些与网卡、交换机、路由器等对应的虚拟设施，以及如何使用这些虚拟设施来组成网络开始说起。

网卡：tun/tap、veth

目前主流的虚拟网卡方案有[tun/tap](#)和[veth](#)两种，在时间上tun/tap出现得更早，它是一组通用的虚拟驱动程序包，里面包含了两个设备，分别是用于网络数据包处理的虚拟网卡驱动，以及用于内核空间与用户空间的间交互的[字符设备](#)（Character Devices，这里具体指`/dev/net/tun`）驱动。大概在2000年左右，Solaris系统为了实现[隧道协议](#)（Tunneling Protocol）开发了这套驱动，从Linux Kernel 2.1版开始移植到Linux内核中，当时是源码中的可选模块，2.4版之后发布的内核都会默认编译tun/tap的驱动。

tun和tap是两个相对独立的虚拟网络设备，其中tap模拟了以太网设备，操作二层数据包（以太帧），tun则模拟了网络层设备，操作三层数据包（IP报文）。使用tun/tap设备的目的是实现把来自协议栈的数据包先交由某个打开了`/dev/net/tun`字符设备的用户进程处理后，再把数据包重新发回到链路中。你可以通俗地将它理解为这块虚拟化网卡驱动一端连接着网络协议栈，另一端连接着用户态程序，而普通的网卡驱动则是一端连接则网络协议栈，另一端连接着物理网卡。只要协议栈中的数据包能被用户态程序截获并加工处理，程序员就有足够的舞台空间去玩出各种花样，譬如数据压缩、流量加密、透明代理等功能都能够以此为基础来实现，以最典型的VPN应用为例，程序发送给tun设备的数据包，会经过如下所示的顺序流进VPN程序：



VPN中数据流动示意图

应用程序通过tun设备对外发送数据包后，tun设备如果发现另一端的字符设备已被VPN程序打开（这就是一端连接着网络协议栈，另一端连接着用户态程序），便会把数据包通过字符设备被发送给VPN程序，VPN收到数据包，便可以修改后再重新封装成新报文，譬如数据包原本是发送给A地址的，VPN把整个包进行加密，然后作为报文体，封装到另一个发送给B地址的新数据包当中。这种将一个数据包套进另一个数据包中的处理方式被形象地形容为“隧道”（Tunneling），隧道技术是在物理网络中构筑逻辑网络的典型做法。而其中提到的加密，也有标准的协议可遵循，譬如IPSec协议。

使用tun/tap设备传输数据需要经过两次协议栈，不可避免地会有一些性能损耗，因此现在容器对容器的直接通讯并不会以tun/tap作为首选方案，一般是基于稍后介绍的veth来实现的。但是tun/tap没有veth那样要求设备成对出现、数据要原样传输的限制，数据包到用户态程序后，程序员就有完全掌控的权力，要进行哪些修改，要发送到什么地方，都可以编写代码去实现，因此tun/tap方案比起veth方案有更广泛的适用范围。

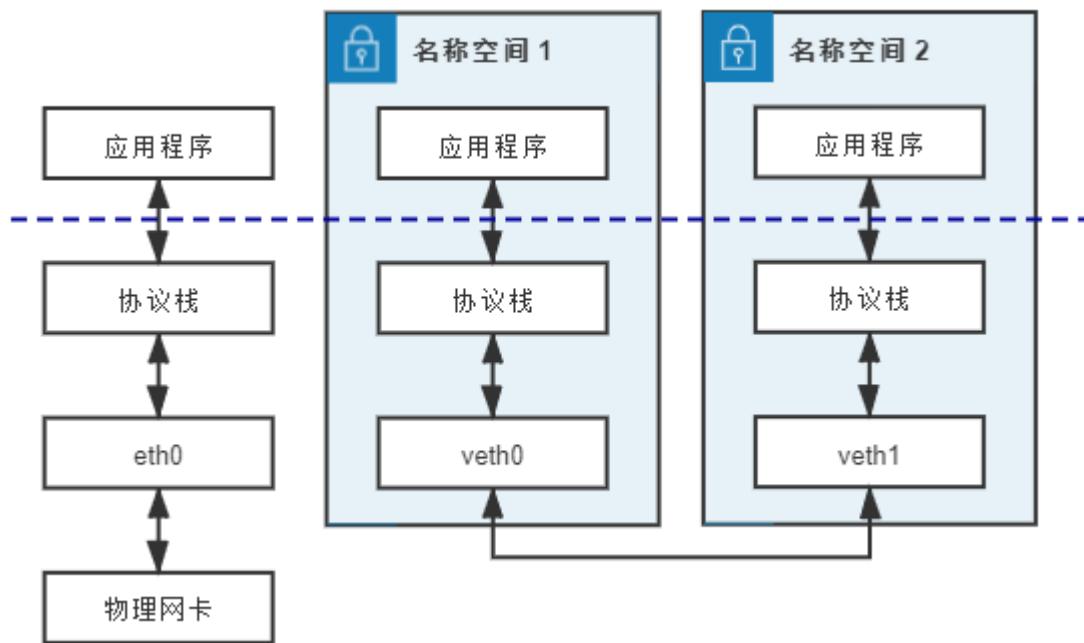
veth是另一种常见的虚拟网卡方案，在Linux Kernel 2.6版本，Linux开始支持网络名空间隔离的同时，也提供了专门的虚拟以太网（Virtual Ethernet，习惯简写做veth）让两个隔离的网络名称空间之间可以互相通讯。直接把veth比喻成是虚拟网卡其实并不十分准确，如果要和物理设备类比，它应该相当于由交叉网线连接的一对物理网卡。

额外知识：直连线序、交叉线序

交叉网线是指一头是T568A标准，另外一头是T568B标准的网线。直连网线则是两头采用同一种标准的网线。

网卡对网卡这样的同类设备需要使用交叉线序的网线来连接，网卡到交换机、路由器就采用直连线序的网线，不过现在的网卡大多带有线序翻转功能，直连线也可以网卡对网卡地连通了。

veth实际上不是一个设备，而是一对设备，因而也常被称作veth pair。要使用veth，就需要在两个独立的网络名称空间中进行才有意义，因为veth pair是一端连着协议栈，另一端彼此相连的，在veth设备的其中一端输入数据，这些数据就会从设备的另外一端原样不变地流出，它工作时数据流动如下图所示：



veth pair工作示意图

由于两个容器之间采用veth通讯不需要经过协议栈，这让veth比起tap/tun具有更高的性能，也让veth pair的实现变的十分简单，内核中只用了几十行代码实现了一个数据复制函数就完成了veth的主体功能。veth以模拟网卡直连的方式很好地解决了两个容器之间的通讯问题，然而对多个容器间通讯，如果仍然单纯只用veth pair的话，事情就变得非常麻烦，让每个容器都为与它通讯的其他容器建立一对专用的veth pair并不实际，这时就需要有一台虚拟化的交换机来解决多容器之间的通讯问题了。

交换机：Linux Bridge

既然有了虚拟网卡，很自然也会联想到让网卡接入到交换机里，实现多个容器间的相互连接。Linux Bridge[↗]便是Linux系统下的虚拟化交换机，虽然它以“网桥”（Bridge）而不是“交换机”（Switch）为名，然而使用过程中，你会发现Linux Bridge的目的看起来像交换机，功能使用起来像交换机、程序实现起来也像交换机，实际就是交换机的Duck Typing[↗]。

Linux Bridge是在Linux Kernel 2.2版本开始提供的二层转发工具，由 brctl 命令创建和管理。Linux Bridge创建以后，便能够接入任何位于二层的网络设备，无论是真实的物理设备（譬如eth0）抑或是虚拟的设备（譬如veth或者tap）都能与网桥配合工作。当有二层数据包（以太帧）从网卡进入网桥，它将根据数据包的类型和目标MAC地址，按如下规则转发处理：

- 如果数据包是广播帧，转发给所有接入网桥的设备。
- 如果数据包是单播帧：
 - 且MAC地址在地址转发表中不存在，则洪泛（Flooding）给所有接入网桥的设备，并将响应设备的接口与MAC地址学习（MAC Learning）到自己的MAC地址转发表中。
 - 且MAC地址在地址转发表中已存在，则直接转发到地址表中指定的设备。
- 如果数据包是此前转发过的，又重新发回到此Bridge，说明冗余链路产生了环路。由于以太帧不像IP报文那样有TTL来约束，因此一旦出现环路，如果没有额外措施来处理的话就会永不停歇地转发下去。对于这种数据包就需要交换机实现生成树协议（Spanning Tree Protocol，STP）来交换拓扑信息，生成唯一拓扑链路以切断环路。

上面提到的这些名词，譬如二层转发、泛洪、STP、MAC学习、地址转发表，等等，都是物理交换机中极为成熟的概念，它们在Linux Bridge中都有对应的实现，所以说Linux Bridge不仅用起来像交换机，实现起来也像交换机。不过，它与普通的物理交换机也还是有一点差别的，普通交换机只会单纯地做二层转发，Linux Bridge却还支持把发给它自身的数据包接入到主机的三层的协议栈中。

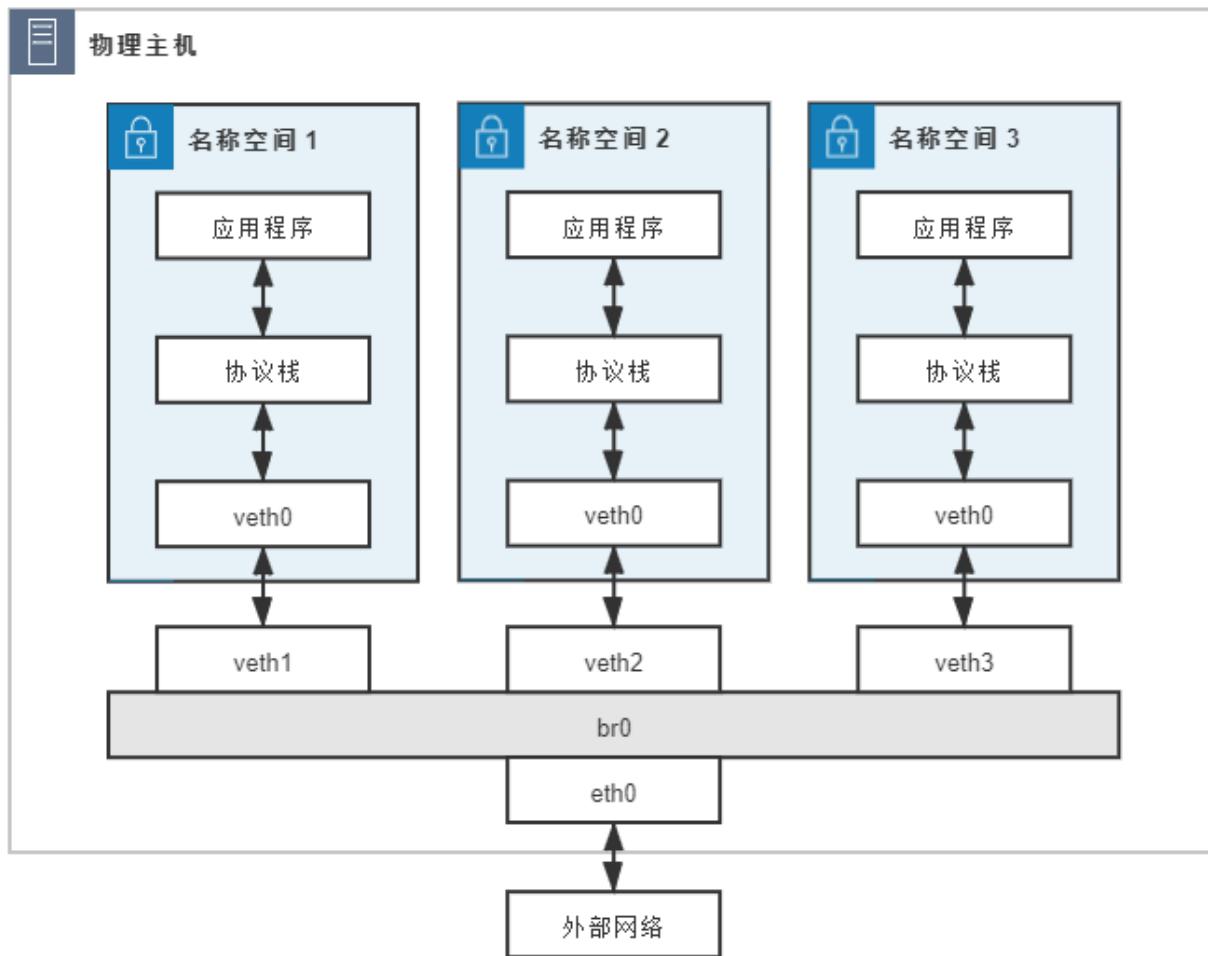
对于通过 `brctl` 命令显式接入网桥的设备，Linux Bridge与物理交换机的转发行为是完全一致的，也不允许给接入的设备设置IP地址，因为网桥是根据MAC地址做二层转发的，就算设置了三层的IP地址也毫无意义。然而Linux Bridge与普通交换机的区别是除了显式接入的设备外，它自己也无可分割地连接着一台有着完整网络协议栈的Linux主机，网桥本身肯定是在某台Linux主机上创建的，可以看作网桥有一个与自己名字相同的隐藏端口，隐式地连接了创建它的那台Linux主机。因此，Linux Bridge允许给自己设置IP地址，比普通交换机多出一种特殊的转发情况：

- 如果数据包的目的MAC地址为网桥本身，并且网桥有设置了IP地址的话，那该数据包即被认为是收到发往创建网桥那台主机的数据包，此数据包将不会转发到任何设备，而是直接交给上层（三层）协议栈去处理。

此时，网桥就取代了eth0设备来对接协议栈，进行三层协议的处理。设置这条特殊转发规则的好处是，只要通过简单的NAT转换，就可以实现一个最原始的单IP容器网络。这种组网是最基本的容器间通讯形式，笔者举个具体例子来帮助你理解。假设现有如下设备及配置，它们的连接情况如图所示：

- 网桥br0：分配IP地址192.168.31.1；

- 容器：三个网络名称空间（容器），分别编号为1、2、3，均使用veth pair接入网桥，且有如下配置：
 - 在容器一端的网卡名为veth0，在网桥一端网卡名为veth1、veth2、veth3；
 - 三个容器中的veth0网卡分配IP地址：192.168.1.10、192.168.1.11、192.168.1.12；
 - 三个容器中的veth0网卡设置网关为网桥，即192.168.31.1；
 - 网桥中的veth1、veth2、veth3无IP地址；
- 物理网卡eth0：分配的IP地址14.123.254.86；
- 外部网络：外部网络中有一台服务器，地址为122.246.6.183



Linux Bridge构建单IP容器网络

如果名称空间1中的应用程序想访问外网地址为122.246.6.183的服务器，由于容器没有自己的公网IP地址，程序发出的数据包将经过如下步骤处理后，才能最终到达外网服务器：

1. 应用程序调用Socket API发送数据，此时生成的原始数据包为：

- 源MAC：veth0的MAC
- 目标MAC：网关的MAC（即网桥的MAC）

- 源IP : veth0的IP , 即192.168.31.1
- 目标IP : 外网的IP , 即122.246.6.183

2. 从veth0发送的数据 , 会在veth1中原样出来 , 网桥将会从veth1中接收到一个目标MAC为自己的数据包 , 并且网桥有配置IP地址 , 由此触发了Linux Bridge的特殊转发规则。这个数据包便不会转发给任何设备 , 而是转交给主机的协议栈处理。
注意 , 从这步以后就是三层路由了 , 已不在网桥的工作范围之内 , 是由Linux主机依靠Netfilter进行IP转发 (IP Forward) 去实现的。

3. 数据包经过主机协议栈 , Netfilter的钩子被激活 , 预置好的iptables nat规则会修改数据包的源IP地址 , 将其改为物理网卡eth0的IP地址 , 并在映射表中记录设备端口及两个IP地址之间的对应关系 , 经过SNAT之后的数据包 , 最终会从eth0出去 , 此时报文头中的地址为 :

- 源MAC : eth0的MAC
- 目标MAC : 下一跳 (Hop) 的MAC
- 源IP : eth0的IP , 即14.123.254.86
- 目标IP : 外网的IP , 即122.246.6.183

4. 可见 , 经过主机协议栈后 , 数据包的源和目标IP地址均为公网的IP , 这个数据包在外部网络中可以根据IP正确路由到目标服务器手上。当目标服务器处理完毕 , 对该请求发出响应后 , 返回数据包的目标地址也是公网IP , 这个数据包经过链路所有跳点 , 由eth0达到网桥时 , 报文头中的地址为 :

- 源MAC : eth0的MAC
- 目标MAC : 网桥的MAC
- 源IP : 外网的IP , 即122.246.6.183
- 目标IP : eth0的IP , 即14.123.254.86

5. 可见 , 这同样是一个以网桥MAC地址为目标的数据包 , 同样会触发特殊转发规则 , 交由协议栈处理。此时Linux将根据映射表中的转换关系做DNAT转换 , 把目标IP地址从eth0替换回veth0的IP , 最终veth0收到的响应数据包为 :

- 源MAC : 网桥的MAC
- 目标MAC : veth0的MAC
- 源IP : 外网的IP , 即122.246.6.183

- 目标IP：veth0的IP，即192.168.31.1

在以上处理过程中，Linux主机承担了三层路由的职责，一定程度上扮演了路由器的角色。由于有Netfilter的存在，对网络层的路由转发，就无需像Linux Bridge一样专门提供 brctl 这样的命令去创建一个虚拟设备，通过Netfilter很容易就能在Linux内核完成根据IP地址进行路由的功能，你也可以理解为Linux内核天然就是一个虚拟的路由器。

限于篇幅，笔者仅介绍Linux Bridge这一种虚拟交换机的方案，此外还有OVS（Open vSwitch）等同样常见，而且更强大、更复杂的方案这里就不再涉及了。

网络：VXLAN

有了虚拟化网络设备后，下一步就是要使用这些设备组成网络，容器分布在不同的物理主机上，每一台物理主机都有物理网络相互通联，然而这种网络的物理拓扑结构是相对固定的，这很难跟上云原生时代的分布式系统的逻辑拓扑结构变动频率，譬如服务的扩缩、断路、限流等等，都可能要求网络跟随做出相应的变化。正因如此，软件定义网络（Software Defined Network，SDN）的需求变得前所未有的迫切，SDN的核心思路是在物理的网络之上再构造一层虚拟化的网络，将控制平面和数据平面分离开来，实现流量的灵活控制，为核心网络及应用的创新提供良好的平台。SDN里位于下层的物理网络被称为Underlay，它着重解决网络的连通性与可管理性，位于上层的逻辑网络被称为Overlay，它着重为上层应用提供与软件需求相符的传输服务和拓扑。

软件定义网络已经发展了十余年时间，远比云原生、微服务这些概念出现得早。网络设备商基于硬件设备开发出了EVI（Ethernet Virtualization Interconnect）、TRILL（Transparent Interconnection of Lots of Links）、SPB（Shortest Path Bridging）等大二层网络技术；软件厂商也提出了VXLAN（Virtual eXtensible LAN）、NVGRE（Network Virtualization Using Generic Routing Encapsulation）、STT（A Stateless Transport Tunneling Protocol for Network Virtualization）等一系列基于虚拟交换机实现的Overlay网络。跨主机的容器间通讯，用的大多是Overlay网络，本节里，笔者会以VXLAN为例去介绍Overlay网络的原理。

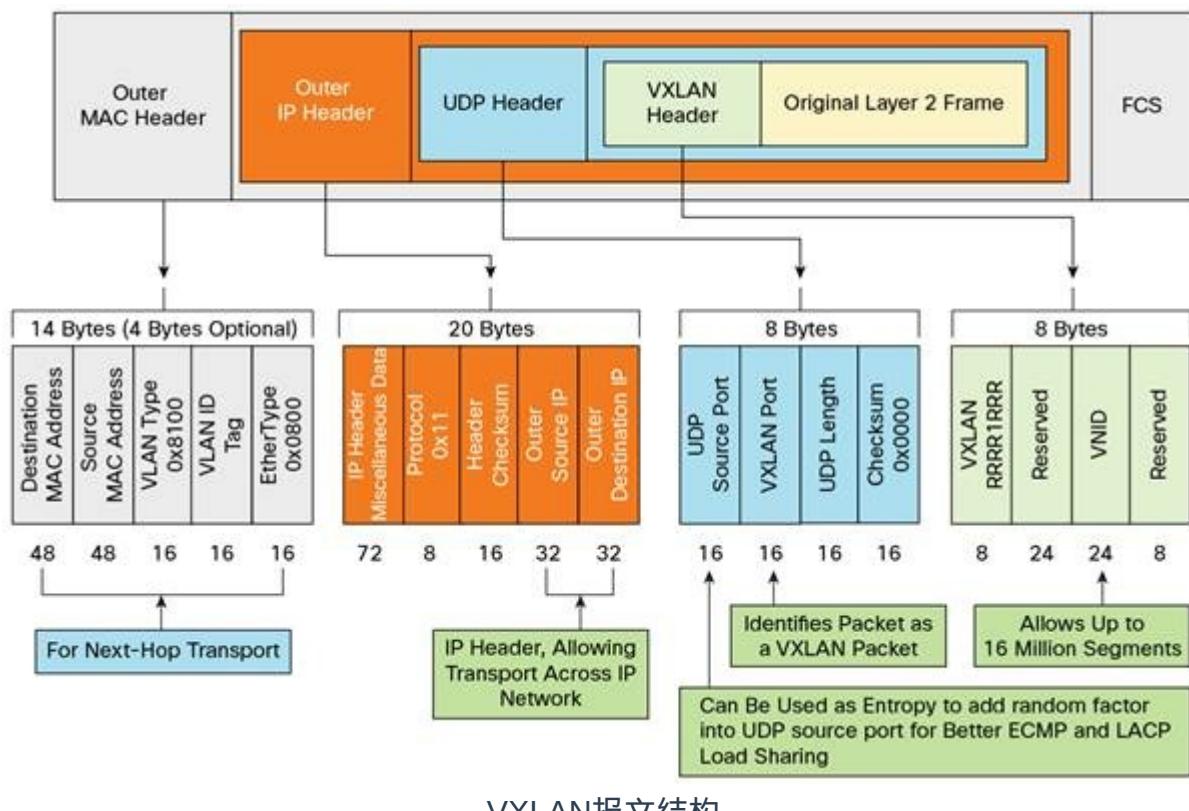
VXLAN你有可能没听说过，但VLAN相信只要从事计算机专业的人都有所了解。VLAN的全称是“虚拟局域网”（Virtual Local Area Network），它也算是网络虚拟化技术的早期成果之一了。由于二层网络本身的工作特性决定了它非常依赖于广播，无论是广播帧（如ARP请求、DHCP、RIP都会产生广播帧），还是泛洪路由，其执行成本都随着接入二层网络设

备数量的增加而等比例增长，当设备太多，广播又频繁的时候，很容易就会形成广播风暴（Broadcast Radiation）。因此，VLAN的首要职责就是划分广播域，将连接在同一个物理网络上的设备区分开来，划分的具体方法是在以太帧的报文头中加入VLAN Tag，让所有广播只针对具有相同VLAN Tag的设备生效。这样即缩小了广播域，也附带有提高了安全性和可管理性，因为两个VLAN之间不能通讯，如果确有通讯的需要，那就必须通过三层设备来进行，如单臂路由（Router on a Stick）或者三层交换机。

然而VLAN有两个明显的缺陷，第一个缺陷在于VLAN Tag的设计，定义VLAN的802.1Q规范是在1998年提出的，当时的工程师完全不可能预料到未来云计算会如此地普及，因而只给VLAN Tag预留了32 Bits的空隙，其中还要分出16 Bits存储标签协议识别符（Tag Protocol Identifier）、3 Bits存储优先权代码点（Priority Code Point）、1 Bits存储标准格式指示（Canonical Format Indicator），剩下的12 Bits才能用来存储VLAN ID（Virtualization Network Identifier，VNI），换而言之，VLAN ID最多只能有 $2^{12}=4096$ 种取值。当云计算数据中心出现后，即使不考虑虚拟化，单是需要分配IP的物理设备都有可能数以十万、百万计，4096个VLAN肯定是不够用的。后来IEEE的工程师们提出802.1AQ规范力图补救这个缺陷，大致思路是给以太帧连续打上两个VLAN Tag，每个Tag里仍然只有12 Bits的VLAN ID，但两个加起来就可以存储 $2^{24}=16777216$ 个不同的VLAN ID了，由于两个VLAN Tag并排放在报文头上，802.1AQ规范还有了个QinQ（802.1Q in 802.1Q）的别名。

QinQ是2011年推出的规范，到现在都并没有特别普及，除了需要设备支持外，它还解决不了VLAN的第二个缺陷：跨数据中心传递。VLAN本身是为二层网络所设计的，但是在两个独立数据中心之间，信息只能跨三层传递，由于云计算的灵活性，大型分布式系统却完全有跨数据中心运作的可能性，此时如何让VLAN Tag在两个数据中心间传递又成了不得不考虑的麻烦事。

为了解决以上两个问题，IETF定义了VXLAN规范，这是NVO3（Network Virtualization over Layer 3）标准技术规范之一，是一种典型的Overlay网络。VXLAN采用L2 over L4（Mac in UDP）的报文封装模式，把原本在二层传输的以太帧放到四层UDP协议的报文体内，同时加入了自己定义的VXLAN Header。在VXLAN Header里就有24 Bits的VLAN ID，同样可以存储1677万个不同的取值，VXLAN让二层网络得以在三层范围内进行扩展，不再受数据中心间传输的限制。VXLAN的整个报文结构如下图所示：



VXLAN报文结构

(图片来源：[Orchestrating EVPN VXLAN Services with Cisco NSO](#))

VXLAN对网络架构的要求很低，不需要硬件提供的特别支持，只要三层可达的网络就能部署VXLAN。VXLAN网络的每个边缘入口上布置有一个VTEP (VXLAN Tunnel Endpoints) 设备，它既可以是物理设备，也可以是虚拟化设备，负责VXLAN协议报文的封包和解包。

[互联网号码分配局](#) (InternetAssigned Numbers Authority , IANA) 专门分配了4789作为VTEP设备的UDP端口 (以前Linux VXLAN用的默认端口是8472) 。

从Linux Kernel 3.7版本起，Linux系统就开始支持VXLAN。到了3.12版本，Linux对VXLAN的支持已达到完全完备的程度，能够处理单播和组播，能够运行于IPv4和IPv6之上，一台Linux主机经过简单配置之后，便可以把Linux Bridge作为VTEP设备使用。

VXLAN带来了很高的灵活性、扩展性和可管理性，同一套物理网络中可以任意创建多个VXLAN网络，每个VXLAN中接入的设备都仿佛是在一个完全独立的二层局域网中一样，不会受到外部广播的干扰，也很难遭受外部的攻击，这使得VXLAN能够良好地匹配分布式系统的弹性需求。不过，VXLAN也带来了额外的复杂度和性能开销，具体表现在：

- 传输效率的下降，如果你仔细数过前面VXLAN报文结构中UDP、IP、以太帧报文头的字节数，会发现经过VXLAN封装后的报文，新增加的报文头部分就有整整占了50 Bytes (VXLAN报文头占8 Bytes，UDP报文头占8 Bytes，IP报文头占20 Bytes，以太帧的MA

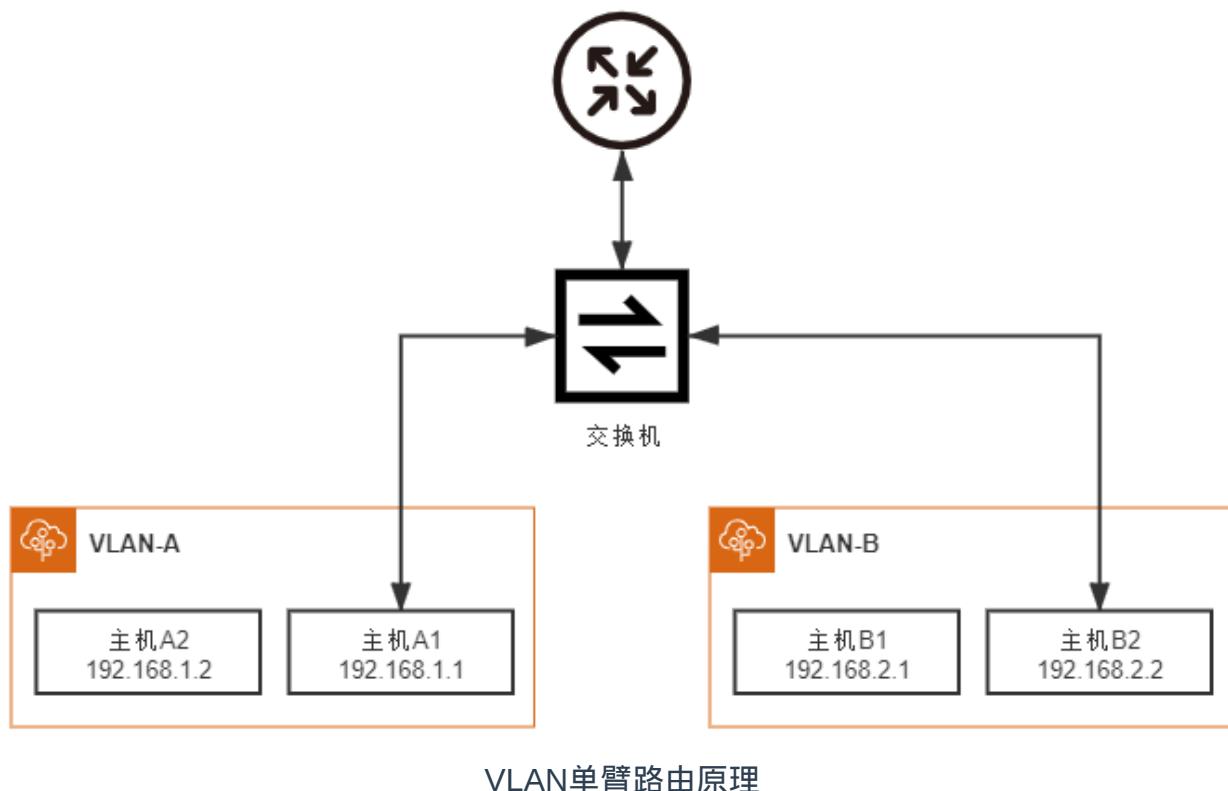
C头占14 Bytes），而原本只需要14 Bytes而已，而且现在这14 Bytes也还在，被封到了最里面的以太帧中。以太网的MTU是1500 Bytes，如果是传输大量数据，额外损耗50 Bytes并不算很高的成本，但如果传输的数据本来就只有几个Bytes的话，那传输消耗在报文头上的成本就很高昂了。

- 传输性能的下降，每个VXLAN报文的封包和解包操作都属于额外的处理过程，尤其是用软件来实现的VTEP，额外的运算资源消耗有时候会成为不可忽略的性能影响因素。

副本网卡：MACVLAN

理解了VLAN和VXLAN的原理后，我们就有足够的前置知识去了解MACVLAN这最后一种网络设备虚拟化的方式了。

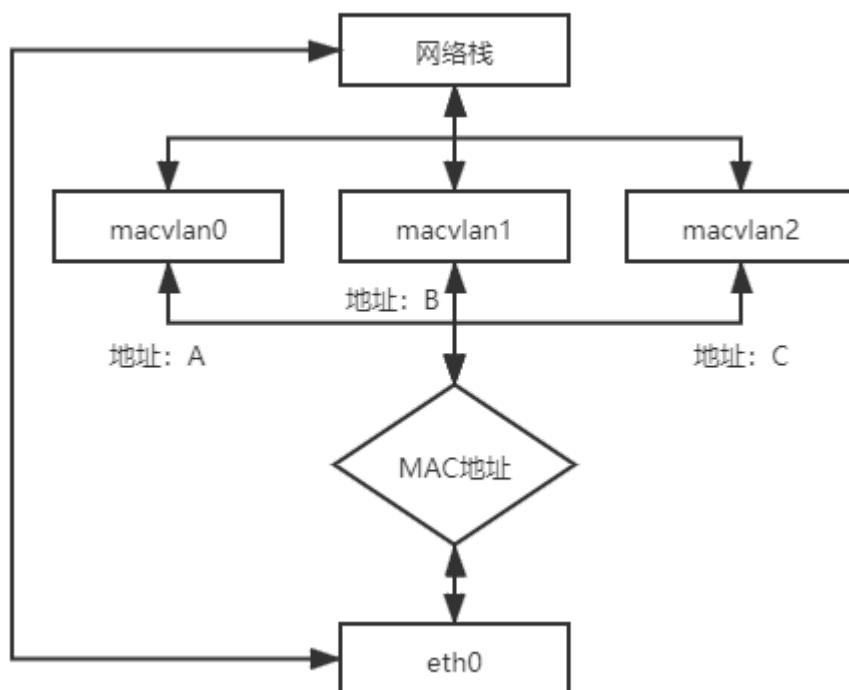
前文中提到了两个VLAN之间位于独立的广播域，是完全二层隔离的，要通讯就只能通过三层设备。最基础的三层通讯就是靠单臂路由了。笔者以下图所示的网络拓扑结构来举个具体例子，介绍单臂路由是如何工作的：



假设位于VLAN-A中的主机A1希望将数据包发送给VLAN-B中的主机B2，由于A、B两个VLAN之间二层链路不通，因此需引入单臂路由，单臂路由不属于任何VLAN，它与交换机之间的链路允许任何VLAN ID的数据包通过，这种接口被称为TRUNK。这样，A1要和B2通

讯，A1就将数据包先发送给路由（把路由设置为网关即可），然后路由根据数据包上的IP地址得知B2的位置，去掉VLAN A的VLAN Tag，改用VLAN B的VLAN Tag重新封装数据包后发回给交换机，交换机收到后就可以顺利转发给B2了。这个过程并没什么复杂的地方，但你是否注意到一个问题，路由器应该设置怎样的IP地址呢？由于A、B处于独立的网段上，它们又各自要将同一个路由作为网关使用，这就要求路由器必须同时具备192.168.1.0/24和192.168.2.0/24的地址。如果真的就只有A、B两个VLAN，那把路由器上的两个接口分别设置IP地址，然后用两条网线分别连接到交换机上也勉强算是一个解决办法，但VLAN最多支持4096个VLAN，如果要接4000多条网线就太离谱了。为了解决这个问题，802.1Q规范中专门定义了子接口（Sub-Interface）的概念，允许在同一张物理网卡上，针对不同的VLAN绑定不同的IP地址。

MACVLAN的思路是在VLAN子接口的基础上更进一步，不仅允许对同一个网卡设置多个IP地址，还允许对同一张网卡上设置多个MAC地址，这也是MACVLAN名字的由来。原本MAC地址是网卡接口的“身份证”，应该是一对一的关系，而MACVLAN的原理是在物理设备之上、网络栈之下生成多个虚拟的Device，每个Device都有一个MAC地址，新增Device的操作相当于在系统中注册了一个收发特定数据包的回调函数，每个回调函数都能对一个MAC地址的数据包进行响应，当物理设备收到数据包时，会先根据MAC地址判断一次，确定交给哪个Device处理，如下图所示。以交换机一侧的视角来看，这个端口后面仿佛是另一台已经连接了多个设备的交换机而已。



MACVLAN原理

用MACVLAN技术虚拟出来的副本网卡，在逻辑上和物理网卡是完全对等的，物理网卡实际上确实承担着类似交换机的职责，收到数据包后，根据目的MAC地址判断这个包应转发给哪块副本网卡处理，由同一块物理网卡虚拟出来的副本网卡，天然处于同一个VLAN之中，可以直接通讯，不需要将流量转发到外部网络。

与Linux Bridge相比，这种模拟交换机的方法在功能层面上看并没有本质的不同，但MACVLAN在内部实现上要比Linux Bridge轻量得多。从数据流来看，副本网卡的通讯只比物理网卡多了一次判断而已，能获得很高的网络性能；从操作步骤来看，由于MAC地址是静态的，所以MACVLAN不需要像Linux Bridge那样考虑MAC地址学习、STP协议等复杂的算法，这进一步加大了MACVLAN的性能优势。

除了模拟交换机的Bridge模式外，MACVLAN还支持虚拟以太网端口聚合模式（Virtual Ethernet Port Aggregator，VEPA）、Private模式、Passthru模式、Source模式等另外几种工作模式，有兴趣可以参考相关资料，笔者就不再一一介绍了。

容器间通讯

经过对虚拟化网络基础知识的一番铺垫后，在最后这个小节里，我们尝试使用这些知识去理解容器间通讯，毕竟运用知识去解决问题才是笔者介绍网络虚拟化的根本目的。这节我们先以Docker为目标，谈一谈Docker所提供的容器通讯方案¹。下节介绍过CNI下的网络插件生态后，你也许会觉得Docker的网络通讯相对简单，对于某些分布式系统的需求来说甚至是过于简陋了，然而，容器间的网络方案多种多样，但通讯主体都是固定的，不外乎没有物理设备的虚拟主体（容器、Pod、Service、Endpoints等等）、不需要跨网络的本地主机、以及通过网络连接的外部主机三种层次，所有的网络通讯问题，都可以归结为本地主机内部的多个容器之间、本地主机与内部容器之间和跨越不同主机的多个容器之间的通讯问题，所以Docker网络的简单，在作为检验前面网络知识有没有理解到位时倒不失为一种优势。

Docker的网络方案在操作层面上是指能够直接通过 `docker run --network` 参数指定的网络，或者先 `docker network create` 创建后再被容器使用的网络。安装Docker过程中会自动在宿主机上创建一个名为docker0的网桥，以及三种不同的网络，分别是bridge、host 和none，你可以通过 `docker network ls` 命令查看到这三种网络，具体如下所示：

```
$ docker network ls
NETWORK ID      NAME      DRIVER
SCOPE
2a25170d4064    bridge    bridge
local
a6867d58bd14    host      host
local
aeb4f8df39b1    none     null
local
```

sh

这三种网络，对应着Docker提供的三种开箱即用的网络方案，它们分别为：

- **桥接模式**，使用 `--network=bridge` 指定，这种也是未指定网络参数时的默认网络。桥接模式下，Docker会为新容器分配独立的网络名称空间，创建好veth pair，一端接入容器，另一端接入到docker0网桥上。Docker为每个容器自动分配好IP地址（默认配置下地址范围是172.17.0.0/24，docker0的地址默认是172.17.0.1），并且设置所有容器的网关均为docker0，这样所有接入同一个网桥内的容器直接依靠二层转发来通讯，在此范围之外的容器、主机就通过网关来访问，具体过程笔者在介绍Linux Bridge时举例详细讲解过。
- **主机模式**，使用 `--network=host` 指定。主机模式下，Docker不会为新容器创建独立的网络名称空间，这样容器一切的网络设施，如网卡、网络栈等都直接使用宿主机上的，容器也就不会有自己独立的IP地址。此模式下与外界通讯无需进行NAT转换，没有性能损耗，但缺点也很明显，没有隔离就无法避免网络资源（譬如端口号）的冲突。
- **空置模式**，使用 `--network=none` 指定，空置模式下，Docker会给新容器创建独立的网络名称空间，但是不会创建任何虚拟的网络设备，此时容器能看到的只有一个回环设备（Loopback Device）而已。提供这种方式是为了方便用户去做自定义的网络配置，如自己增加网络设备、自己管理IP地址，等等。

除了三种开箱即用的网络外，Docker还支持以下自己创建的网络：

- **容器模式**，创建容器后使用 `--network=container:容器名称` 指定。容器模式下，新创建的容器将会加入指定的容器的网络名称空间，共享一切的网络资源（但其他资源，如文件、PID等默认仍然是隔离的），两个容器间可以直接使用回环地址（localhost）通讯，端口号等网络资源不能有冲突。

- **MACVLAN模式**：使用 `docker network create -d macvlan` 创建，此网络允许为容器指定一个副本网卡，容器通过副本网卡的MAC地址来使用宿主机上的物理设备，在追求通讯性能的场合这种网络是比较好的选择。Docker的MACVLAN只支持Bridge通讯模式，因此在功能表现上与桥接模式是类似的。
- **Overlay模式**：使用 `docker network create -d overlay` 创建，Docker说的Overlay网络实际上はVXLAN，主要用于Docker Swarm服务之间进行通讯。然而由于Docker Swarm败于Kubernetes，并未成为主流，所有这种网络模式实际很少使用。

容器网络与生态

容器网络的第一个业界标准源于Docker在2015年发布的[libnetwork](#)项目，如果你还记得在“[容器的崛起](#)”中关于[libcontainer](#)的故事，那从名字上就很容易推断出[libnetwork](#)项目的意义所在。这是Docker用Golang编写的、专门用来抽象容器间网络通讯的一个独立模块，与[libcontainer](#)是作为OCI的标准实现类似，[libnetwork](#)是作为Docker提出的CNM规范（Container Network Model）的标准实现而提出的。不过，与[libcontainer](#)因孵化出runC项目，时至今日仍然广为人知的结局不同，[libnetwork](#)随着Docker Swarm的失败，已经基本失去了实用价值，只具备历史与研究的价值了。

CNM与CNI

如今容器网络的事实标准[CNI](#)（Container Networking Interface）与CNM在目标上几乎是完全重叠的，由此决定了CNI与CNM之间只能是你死我活的竞争关系，这与容器运行时中提及的CRI和OCI的关系明显不同，CRI与OCI目标并不一样，两者有足够空间可以和平共处。

尽管CNM规范已是明日黄花，但它作为容器网络的先行者，对后续的容器网络标准制定有直接的指导意义，提出容器网络标准的目的就是为了把网络功能从容器运行时引擎或者容器编排系统中剥离出去，网络的专业性和针对性极强，如果不把它变成外部可扩展的功能，都由自己来做的话，不仅费时费力，还难以讨好。这个特点从下图所列的一大堆容器网络提供商就可见一斑。



ROMANA



CNI-Genie



部分容器网络提供商

网络的专业性与针对性也决定了CNM和CNI均采用了插件式的设计，需要接入什么样的网络，就设计一个对应的网络插件即可。所谓的插件，在形式上也就是一个可执行文件，再配上相应的Manifests描述。为了方便插件编写，CNM将协议栈、网络接口（对应于veth、tap/tun等）和网络（对应于Bridge、VXLAN、MACVLAN等）分别抽象为Sandbox、Endpoint和Network，并在接口的API中提供了这些抽象资源的读写操作。而CNI中尽管也有Sandbox、Network的概念，含义也与CNM大致相同，不过在Kubernetes资源模型的支持下，它就无需刻意地强调某一种网络资源应该如何描述、如何访问了，因此结构上反而显得更加轻便。

从程序功能上看，CNM和CNI的网络插件提供的能力都能划分为网络的管理与IP地址的管理两类，插件可以选择只实现其中的某一个，也可以全部都实现：

- 管理网络创建与删除。**解决如何创建网络，将容器接入到网络，以及容器退出和删除网络。这个过程实际上是对容器网络的生命周期管理，如果你熟悉Docker命令，可以类比为基本上等同于 `docker network` 命令所做的事情，CNM规范中定义了创建网络、删除网络、容器接入网络、容器退出网络、查询网络信息、创建通讯Endpoint、删除通讯Endpoint等十个编程接口，而CNI中就更加简单了，只要实现对网络的增加、删除操作即可。你甚至不需要学过Golang语言都能轻松看明白接口中方法的含义是什么：

```
type CNI interface {
    AddNetworkList (net *NetworkConfigList, rt *RuntimeConf)
    (types.Result, error)
```

go

```
DelNetworkList (net *NetworkConfigList, rt *RuntimeConf) error

AddNetwork (net *NetworkConfig, rt *RuntimeConf) (types.Result,
error)
DelNetwork (net *NetworkConfig, rt *RuntimeConf) error
}
```

- **管理IP地址分配与回收。**解决如何为三层网络分配唯一的IP地址的问题，二层网络的MAC地址天然就具有唯一性，无需刻意地考虑如何分配。但是三层网络的IP地址只有精心规划，才能保证在全局网络中都是唯一的，否则，如果两个容器之间可能存在相同地址，那它们就最多只能做NAT，而无法做到直接通讯。相比起基于UUID或者数字序列实现的全局唯一ID产生器，IP地址的全局分配工作要更加困难一些，首先是要符合IPv4的网段规则，且保证不重复，这在分布式环境里只能依赖Etcd、ZooKeeper等协调工具来实现，Docker自己也提供了类似的libkv来完成这项工作。其次是必须考虑到回收的问题，否则如果Pod发生持续重启就有可能耗尽某个网段中的所有地址。最后还必须要关注时效性，原本IP地址的获取采用标准的DHCP协议（Dynamic Host Configuration Protocol）即可，但DHCP有可能产生长达数秒的延迟，对于某些生存周期很短的Pod，这已经超出了它的忍受限度，因此在容器网络中，往往Host-Local的IP分配方式会比DHCP更加实用。

CNM到CNI

容器网络标准能够提供一致的网络操作界面，不论是什么网络插件都使用一致的API，提高了网络配置的自动化程度和在不同网络间迁移的体验，对最终用户、容器提供商、网络提供商来说都是三方共赢的事情。

CNM规范发布以后，借助Docker在容器领域的强大号召力，很快就得到了网络提供商与开源组织的支持，不说专门为Docker设计针对容器互联的网络，最起码也会让现有的网络方案兼容于CNM规范，以便能在容器圈中多分一杯羹，譬如Cisco的Contiv、OpenStack的Kuryr、Open vSwitch的OVN（Open Virtual Networking）以及来自开源项目的Calico和Weave等都是CNM阵营中的成员。唯一对CNM持有不同意见的是那些和Docker存在直接竞争关系的产品，譬如Docker的最大竞争对手，来自CoreOS公司的RKT容器引擎。其实凭良心说，并不是其他容器引擎想刻意去抵制CNM，而是Docker制定CNM规范时完全基于Docker本身来设计，并没有考虑CNM用于其他容器引擎的可能性。

为了平衡CNM规范的影响力，也是为了在Docker的垄断背景下寻找一条出路，RKT提出了与CNM目标类似的“[RKT网络提案](#)”（ RKT Networking Proposal ）。一个业界标准成功与否，很大程度上取决于它支持者阵营的规模，对于容器网络这种插件式的规范更是如此。Docker力推的CNM毫无疑问是当时统一容器网络标准的最有力竞争者，如果没有外力的介入，有极大可能会成为最后的胜利者。然而，影响容器网络发展的外力最终还是出现了，容器圈里能够掀翻Docker的“外力”，也就唯有Kubernetes一家而已。

Kubernetes开源的初期（ Kubernetes 1.5提出CRI规范之前），在容器引擎上是选择彻底绑定于Docker的，但是，在容器网络的选择上，Kubernetes一直都坚持独立于Docker自己来维护网络。CNM和CNI提出以前的早期版本里，Kubernetes会使用Docker的空置网络模式（`--network=none`）来创建Pause容器，然后通过内部的kubenet来创建网络设施，再让Pod中的其他容器加入到Pause容器的名称空间中共享这些网络设施。

额外知识：kubenet

[kubenet](#)是kubelet内置的一个非常的简单的网络，采用网桥来解决Pod间通讯。kubenet会自动创建一个名为cbr0的网桥，当有新的Pod启动时，会由kubenet自动将其接入cbr0网桥中，再将控制权交还给kubelet，完成后续的Pod创建流程。kubenet采用Host-Local的IP地址管理方式，具体来说是根据当前服务器对应的Node资源上的 `PodCIDR` 字段所设的网段来分配IP地址。当有新的Pod启动时，会由本地节点的IP段中分配一个空闲的IP给Pod使用。

CNM规范提出以前，Kubernetes自己来维护网络是必然的结果，因为Docker自带的网络基本上只聚焦于如何解决本地通讯，完全无法满足Kubernetes跨集群节点的容器编排的需要。CNM规范提出之后，原本CNM的价值就是能很方便地引入其他网络插件来替代掉Docker自带的网络，Kubernetes却对Docker的CNM规范表现得颇为犹豫，经过一番评估考量，Kubernetes最终决定去支持RKT的网络提案，与CoreOS合作以RKT网络提案为基础发展出CNI规范。

Kubernetes Network SIG的Leader、Google的工程师Tim Hockin专门撰写过一篇文章《[Why Kubernetes doesn't use libnetwork](#)》解释为何Kubernetes要拒绝CNM与libnetwork。当时容器编排战争还处于三国争霸（Kubernetes、Apache Mesos、Docker Swarm）的阶段，即使强势如Kubernetes，拒绝CNM其实也要冒不小的风险，付出颇大的代价，因为这个决定不可避免会引发一系列技术和非技术的问题，譬如网络提供商要为Kubernetes专门

编写不同的网络插件、由 `docker run` 启动的容器将会无法与Kubernetes启动的容器相互通讯，等等。

促使Kubernetes拒绝CNM的理由也同样有来自于技术和非技术方面的。技术上，Docker的网络模型做出了许多对Kubernetes无效的假设，Docker的网络有本地网络（不带任何跨节点协调能力，譬如Bridge模式就没有全局统一的IP分配）和全局网络（跨主机的容器通讯，例如Overlay模式）的区别，本地网络对Kubernetes来说毫无意义，而全局网络又默认依赖[libkv](#)来实现全局IP地址管理等跨机器的协调工作。这里的libkv是Docker建立的lib*家族中另一位成员，用来对标Etcd、ZooKeeper等分布式K/V存储，这对于已经拥有了Etcd的Kubernetes来说如同鸡肋。非技术上，Kubernetes决定放弃CNM的原因很大程度上还是由于他们与Docker在发展理念上的冲突，Kubernetes当时已经开始推进Docker从必备依赖变为可选引擎的重构工作，而Docker则坚持CNM只能基于Docker来设计。Tim Hockin在文章中举了一个例子：CNM的网络驱动没有向外部暴露网络所连接容器的具体名称，只使用一个内部分配的ID来代替，这让外部（网络插件、编排系统）很难将网络连接的容器与自己管理的容器对应关联起来，当他们向Docker开发人员反馈这个问题时，问题被以“工作符合预期结果（Working as Intended）”为由直接关闭掉，Tim Hockin还专门列出了这些问题清单，如[libnetwork #139](#)、[libnetwork #486](#)、[libnetwork #514](#)、[libnetwork #865](#)、[docker #18864](#)。这种设计被Kubernetes认为是人为地给非Docker的第三方容器引擎使用CNM设置障碍。整个沟通过程中，Docker表现得也极为强硬，明确表示他们对偏离当前路线或委托控制的想法不太欢迎。上面这些“非技术”的问题，即使没有Docker的支持，Kubernetes也并非不能从“技术上”去解决，但Docker的理念令Kubernetes感到忧虑，因为Kubernetes在Docker之上扩展的很多功能，Kubernetes却并不想这些功能永远绑定在Docker之上。

CNM与Libnetwork是2015年5月1日发布的，CNI则是在2015年7月发布，两者正式诞生只相差不到两个月时间，这显然是竞争的需要而非单纯的巧合。五年之后的今天，这场容器网络的话语权之争已经尘埃落定，CNI获得全面的胜利，除了Kubernetes和RKT外，Amazon ECS、RedHat OpenShift、Apache Mesos、Cloud Foundry等容器圈中除了Docker之外其他具有影响力的参与者都已宣布支持CNI规范，原本已经加入了CNM阵营的Contiv、Calico、Weave网络提供商也纷纷推出了自己的CNI插件。

网络插件生态

时至今日，支持CNI的网络插件已多达数十种，笔者不太可能逐一细说，不过，跨主机通讯的网络实现方式来去也就下面这几种，我们不妨以网络实现模式为主线，每种模式介绍一个具有代表性的插件，以达到对网络插件生态窥斑见豹的效果。

- **Overlay模式**：我们已经学习过Overlay网络，这是一种虚拟化的上层逻辑网络，好处在于它不受底层物理网络结构的约束，有更大的自由度，更好的易用性；坏处是由于额外的包头封装导致信息密度降低，额外的隧道封包解包会导致传输性能下降。

在虚拟化环境（例如 OpenStack）中的网络限制往往较多，譬如不允许机器之间直接进行二层通讯，只能通过三层转发。在这类被限制网络的环境里，基本上就只能选择Overlay网络插件，常见的Overlay网络插件有Flannel（VXLAN模式）、Calico（IPIP模式）、Weave等等。

以Flannel-VXLAN为例，由CoreOS开发的Flannel可以说是最早的跨节点容器通信解决方案，很多其他网络插件的设计中都能找到Flannel的影子。早在2014年，VXLAN还没有进入Linux内核的时Flannel就已经开始流行，那时候Flannel只能采用自定义的UDP封包实现自己私有的Overlay网络，由于封包、解包的操作只能在用户态中进行，而数据包在内核态的协议栈中流转，这导致数据要反复在用户态、内核态之间拷贝，性能堪忧。从此Flannel就给人留下了速度慢的坏印象。VXLAN进入Linux内核后，这种内核态用户态的转换消耗已经完全消失，Flannel-VXLAN的效率比起Flannel-UDP有了很大提升，目前已经成为最常用的容器网络插件之一。

- **路由模式**：路由模式其实也属于Underlay模式的一种特例，这里将它单独作为一种网络实现模式来介绍。相比起Overlay网络，路由模式的主要区别在于它的跨主机通信是直接通过路由转发来实现的，因而无需在不同主机之间做一个隧道封包。这种模式的好处是性能相比Overlay网络有所提升，坏处是路由转发要依赖于底层网络环境的支持，并不是你想做就能做到的。路由网络要求要么所有主机都位于同一个子网之内，都是二层连通的，要么不同二层子网之间由支持**边界网关协议**（Border Gateway Protocol，BGP）的路由相连，并且网络插件也同样支持BGP协议去修改路由表。

上一节介绍Linux网络基础知识时，笔者提到过Linux下不需要专门的虚拟路由，因为Linux本身就具备路由的功能。路由模式就是依赖Linux之上的路由协议，将路由表分发到子网的每一台物理主机之中，这样当跨主机访问容器时，Linux主机可以根据自己的路由表得知该容器存在于哪台物理主机之中，从而直接将数据包转发过去，避免了VXLAN的封包解包而导致的性能降低。常见的路由网络有Flannel（HostGateway模式）、Calico（BGP模式）等等。

以Flannel-HostGateway为例，Flannel通过在各个节点上运行的Flannel Agent（Flannel

d) 将容器网络的路由信息设置到主机的路由表上，这样一来，所有的物理主机都拥有整个容器网络的路由数据，容器间的数据包可以被Linux主机直接转发，通信效率与裸机直连都相差无几，不过由于Flannel Agent只能修改它运行主机上的路由表，一旦主机之间隔了其他路由设备，譬如路由器或者三层交换机，这个包就会在路由设备上被丢掉，要解决这种问题就必须依靠BGP路由和Calico-BGP这类支持标准BGP协议修改路由表的网络插件共同协作才行。

- **Underlay模式**：这里的Underlay模式特指让容器和宿主机处于同一网络，两者拥有相同的地位的网络方案。Underlay网络要求容器的网络接口能够直接与底层网络进行通讯，因此该模式是直接依赖于虚拟化设备与底层网络能力的。常见的Underlay网络插件有MACVLAN、SR-IOV¹（Single Root I/O Virtualization）等等。

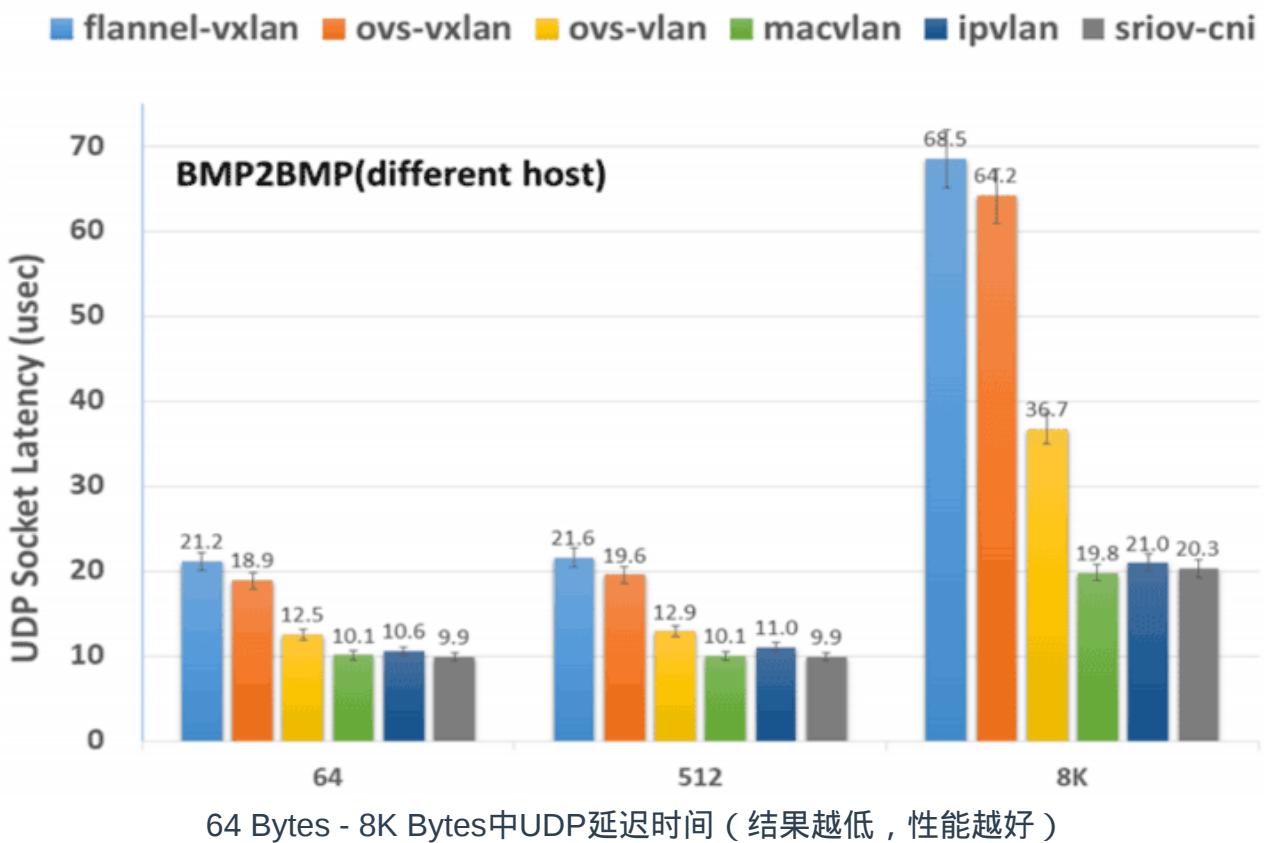
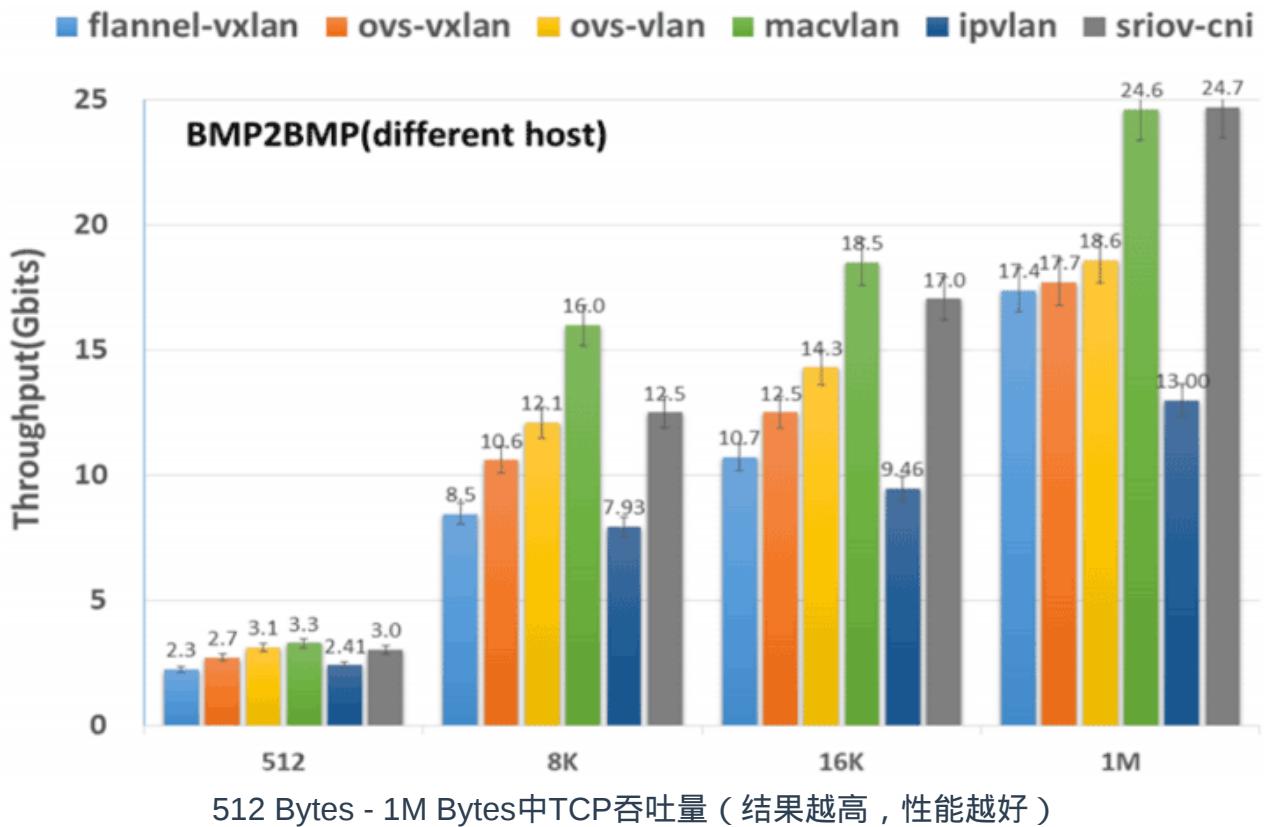
对于真正的大型数据中心、大型系统，Underlay模式是最有发展潜力的网络模式。这种方案能够最大限度地利用硬件的能力，往往有着最优秀的性能表现。但也是由于它直接依赖于硬件与底层网络环境，必须根据软、硬件情况来进行部署，难以做到Overlay网络那样开箱即用的灵活性

以SR-IOV为例，SR-IOV是一种将PCIe²设备共享给虚拟机使用的硬件虚拟化标准，目前用在网络设备上应用比较多，理论上也可以支持其他的PCIe硬件。通过SR-IOV能够让硬件在虚拟机上实现独立的内存地址、中断和DMA流，而无需虚拟机管理系统的介入。对于容器系统来说，SR-IOV的价值是可以直接在硬件层面虚拟多张网卡，并且以硬件直通（Pass-Through）的形式交付给容器使用。但是SR-IOV直通部署起来通常都较为繁琐，现在容器用的SR-IOV方案不少是使用MACVTAP来对SR-IOV网卡进行转接的，MACVTAP提升了SR-IOV的易用性，但是这种转接又会带来额外的性能损失，并不一定会比其他网络方案有更好的表现。

了解过CNI插件的大致的实现原理与分类后，相信你的下一个问题是哪种CNI网络最好？如何选择合适的CNI插件？选择CNI网络插件主要有三方面的考量因素，首先必须是你系统所处的环境是支持的，这点在前面已经有针对性地介绍过。第二、第三个因素就是性能与功能方面是否合乎你的要求。

性能方面，笔者引用一组测试数据供你参考。这些数据来自于2020年8月刊登在IETF的论文《Considerations for Benchmarking Network Performance in Containerized Infrastructures³》，此文中测试了不同CNI插件在裸金属服务器⁴之间（BMP to BMP，Bare Metal Pod）、虚拟机之间（VMP to VMP，Virtual Machine Pod）、以及裸金属服务器与虚拟机

之间 (BMP to VMP) 的本地网络和跨主机网络的通讯表现。囿于篇幅，这里只列出最具代表性的是裸金属服务器之间的跨主机通讯，其结果如下图所示：



从测试结果可见，MACVLAN和SR-IOV这样的Underlay网络插件的吞吐量最高、延迟最低，仅从网络性能上看它们是最优秀的，相对而言Flannel-VXLAN这样的Overlay网络插件，其吞吐量只有MACVLAN和SR-IOV的70%左右，延迟更是高了两至三倍之多。Overlay为了易用性、灵活性所付出的代价还是不小的，但是对于那些不以网络I/O为性能瓶颈的系统而言，这样的代价并非一定不可接受，就看你心中如何权衡取舍。

功能方面的问题就比较简单了，完全取决于你的需求是否能够满足。对于容器编排系统来说，网络并非孤立的功能模块，只提供网络通讯就可以的，譬如Kubernetes的NetworkPolicy资源是用于描述“两个Pod之间是否可以访问”这类ACL策略。但它不属于CNI的范畴，因此不是每个CNI插件都会支持NetworkPolicy的声明，如果有这方面的需求，就应该放弃Flannel，去选择Calico、Weave等插件。类似的其他功能上的选择的例子还有很多，笔者不再一一列举。

容器持久化存储

容器实质上是镜像的运行时实例，为了保证镜像能够重复地产生出一致的运行时实例，必须要求镜像是持久而稳定的，在容器中发生的一切数据变动操作都不能真正地写入到镜像之中，否则就会破坏镜像的持久和稳定的性质。因此，容器中的数据修改操作，都是基于[写入时复制](#)（Copy-on-Write）策略来实现的，容器会利用[叠加式文件系统](#)（OverlayFS）的特性，当需要对根镜像进行修改时，容器会将变更内容写入到独立区域并“覆盖”原有内容。这种改动通常都是临时的，当容器被删除时，这些改动也将一并移除，不复存在。因此，如果不进行额外的处理，容器默认是没有永久存储的。

而另一方面，容器作为信息系统的运行载体，必定会产生出有价值、应该被持久保存的信息，譬如扮演数据库角色的容器，大概没有什么数据库能够接受像缓存服务一样重启之后数据会全部丢失；多个容器之间也往往需要通过共享存储来实现某些交互，譬如[以前曾经举过的例子](#)，Nginx容器产生日志、Filebeat容器收集日志，两者就需要共享同一块存储区域才能协同工作。因此便有了本节的话题，容器的持久化存储。

Kubernetes存储设计

Kubernetes在规划存储能力的时候，依然遵循着它的一贯设计哲学，用户负责以资源和声明式API来描述自己的意图，Kubernetes负责根据意图完成具体的操作。然而，即使是遵循了Kubernetes一贯的设计哲学，相比起提供其他能力的资源，Kubernetes内置的存储能力依然显得格外地繁琐，甚至可以说是有些混乱的。如果你是Kubernetes的拥趸，无法认同笔者对Kubernetes的批评，那不妨来看一看下列围绕着“Volume”所衍生的概念，它们仅是Kubernetes存储相关的概念的一个子集，请你思考一下这些概念是否全都是必须的，是否还有整合的空间，是否有化繁为简的可能性：

- **概念**：[Volume](#)、[PersistentVolume](#)、[PersistentVolumeClaim](#)、[Provisioner](#)、[StorageClass](#)、[Volume Snapshot](#)、[Volume Snapshot Class](#)、[Ephemeral Volumes](#)、[FlexVolume Driver](#)、[Container Storage Interface](#)、[CSI Volume Cloning](#)、[Volume Limits](#)、[Volume Mode](#)、[Access Modes](#)、[Storage Capacity](#)……
- **操作**：[Mount](#)、[Bind](#)、[Use](#)、[Provision](#)、[Claim](#)、[Reclaim](#)、[Reserve](#)、[Expand](#)、[Clone](#)、[Schedule](#)、[Reschedule](#)……

诚然，Kubernetes摆弄出如此多关于存储的术语概念，很重要的一个原因是想要尽可能多地兼容各种存储系统，为此不得不预置了很多In-Tree（意思是在Kubernetes的代码树里）插件来对接，让用户根据自己业务的需要来选择。同时，为了兼容那些不在预置范围内的需求场景，支持用户使用FlexVolume或者CSI来定制Out-of-Tree（意思是在Kubernetes的代码树之外）的插件，实现更加丰富多样的存储能力。下表列出了部分Kubernetes目前提供的存储与扩展：

Temp	Ephemeral (Local)	Persistent (Network)	Extension
------	-------------------	----------------------	-----------

Temp	Ephemeral (Local)	Persistent (Network)	Extension
EmptyDir	HostPath GitRepo Local Secret ConfigMap DownwardAPI	AWS Elastic Block Store GCE Persistent Disk Azure Data Disk Azure File Storage vSphere CephFS and RBD GlusterFS iSCSI Cinder Dell EMC ScaleIO	FlexVolume CSI

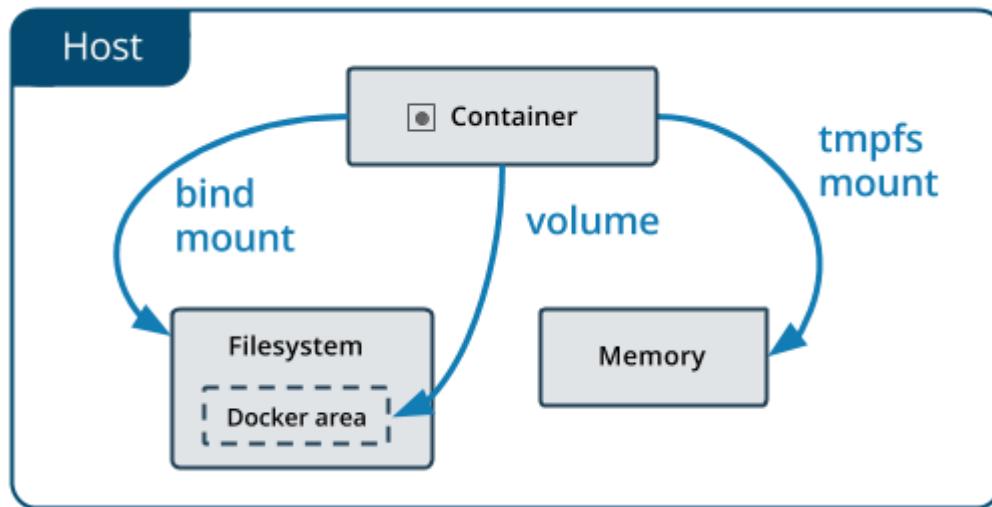
迫使Kubernetes存储设计成如此繁复，其实还有另外一个非技术层面的原因。复杂性归根结底源于Kubernetes是一个工业级的、面向大规模生产应用的容器编排系统，这意味着即使发现某些已存在的功能有更好的实现方式，直到旧版本被基本淘汰出生产环境以前，原本已支持的功能都不允许突然间被移除或者替换掉，否则，如果生产系统一更新版本，已有的功能就出现异常，那对产品的累积的良好信誉是颇为不利的。

为了兼容性而导致的复杂性在一定程度上可以被谅解，但这样的设计的确令Kubernetes的学习曲线变得更加陡峭。Kubernets官方文档的主要作用是参考手册，它并不会告诉你Kubernetes中各种概念的演化历程、版本发布新功能的时间线、改动的缘由与背景等信息。Kubernetes的文档系统只会以“平坦”的方式来陈述目前可用的所有功能，这有利于熟练的管理员快速查询到关键信息，却不利于初学者去理解Kubernetes的设计思想，由于难以理解那些概念和操作的本意，往往只能死记硬背，也就很难分辨出它们应该如何被“更正确”的使用。介绍Kubernetes设计理念的职责，应该要由[Kubernetes官方的Blog](#)与定位超越了帮助手册的非官方资料去完成。本节中，笔者就将以Volume概念的演化历程为主线去介绍前面提及的那些概念与操作，以此帮助大家窥探Kubernetes的存储设计理念。

Mount和Volume

Mount和Volume最初都是来自操作系统的基础概念，Mount是动词，表示将某个外部存储挂载到系统中，Volume是名词，它是物理存储的逻辑抽象，目的是为物理存储提供更有弹性的分割方式。容器发源于对操作系统层的虚拟化，为了满足容器内生成数据的外部存储需求，也很自然地会将Mount和Volume的概念延拓至容器中。因此，关于Volume的发展演进，笔者就以Docker的Mount（挂载）为起点来介绍。

目前，Docker内建支持三种挂载类型，分别是Bind（`--mount type=bind`）、Volume（`--mount type=volume`）和tmpfs（`--mount type=tmpfs`），如下图所示。其中tmpfs用于在内存中读写临时数据，鉴于本节主要讨论的对象是持久化的存储，所以后面我们将着重关注Bind和Volume两种挂载类型。



Docker的三种挂载类型（图片来自Docker官网文档[↗](#)）

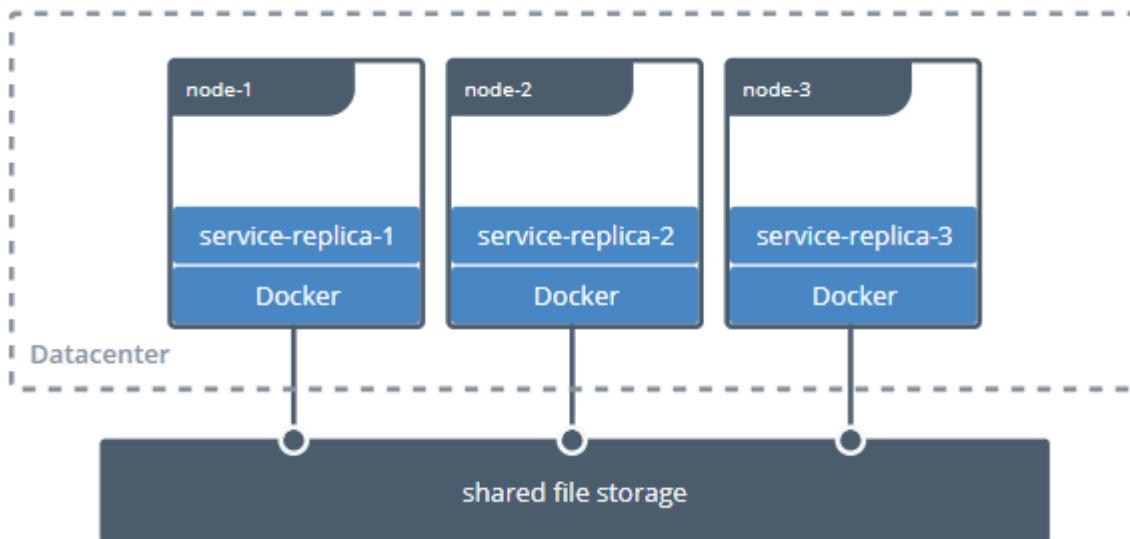
Bind Mount是Docker最早提供的（发布时就支持）挂载类型，它就是指把宿主机的某个目录（或文件）挂载到容器的指定目录（或文件）下，譬如以下命令中参数`-v`表达的意思就是将外部的HTML文档挂到Nginx容器的默认网站根目录下：

```
sh
docker run -v /icyfenix/html:/usr/share/nginx/html nginx:latest
```

这里有一点值得注意，虽然命令中`-v`参数是`--volume`的缩写，但`-v`最初只是用来创建Bind Mount而不是创建Volume Mount的，这种迷惑的行为并非Docker的本愿，只是因为Docker刚发布时考虑得不够周全，随随便便就在参数中占用了“Volume”这个词，到后来需要扩展Volume的概念来支持Volume Mount时，前面的`-v`已经被用户广泛使用了，所以也就只得如此将就着继续用。从Docker 17.06版本开始，它在Docker Swarm中借用了`--mount`参数过来，这个参数默认创建的是Volume Mount，可以通过明确的type子参数来指定另外两种挂载类型。上面命令可以等价于`--mount`版本如下形式：

```
sh
docker run --mount
type=bind,source=/icyfenix/html,destination=/usr/share/nginx/html
nginx:latest
```

为什么Docker后来要扩展新的Volume Mount类型呢？或者说，Bind Mount有什么不足之处？如果只限于在开发期间上使用，Bind Mount并不会有什么问题，但是用在生产环境就颇为不便了。Bind Mount只是让容器与本地主机之间建立了某个目录的映射关系，如果不同主机上的容器需要共享同一份存储的话，必须先把共享存储挂载到每一台宿主机操作系统的某个目录下，然后才能逐个挂载到容器内使用。跨主机共享存储的场景如下图所示。



跨主机的共享存储需求（图片来自Docker官网文档[↗](#)）

即使只考虑本地主机的场景，Docker出于管理需求也有提出Volume的必要。Bind Mount的设计里，Docker只有容器的控制权，存放容器生产数据的主机目录是完全独立的，与Docker没有什么关系，既不受Docker保护，也不受Docker管理。数据很容易被其他进程访问到，甚至是被修改和删除。如果用户想对挂载的目录进行备份、迁移等管理运维操作，也只能在Docker之外靠管理员人工进行，这都增加了数据安全与操作意外的风险。Docker希望能有一种抽象的资源代表数据在主机中的存储位置，以便让Docker来管理这些资源，由此就很自然地想到了对操作系统里Volume的概念进行延伸。

提出Volume还有最后也是最重要的一个目的：为了提升Docker对不同存储系统的支撑能力，同时也是为了减轻Docker本身的工作量。如果Docker要越过操作系统去支持挂载某种存储系统中的目录，首先必须要知道该如何访问它，然后才能将容器中的读写操作自动转移到该位置。Docker把解决如何访问存储的功能模块称为存储驱动（Storage Driver），只要使用 `docker info` 命令，就能查看到当前Docker所支持的存储驱动。尽管Docker已经内置了市面上主流的OverlayFS驱动，譬如Overlay、Overlay2、AUFS、BTRFS、ZFS，等等。但面对云计算的崛起，仅靠Docker自己来支持全部云计算厂商的存储系统是不太现实的，因此Docker提出了与Storage Driver相对应的Volume Driver（卷驱动）的概念。用户可以通过 `docker plugin install` 命令安装外部的卷驱动[↗](#)，并在创建Volume的时候

指定一个其存储系统匹配的卷驱动，譬如希望存储在AWS Elastic Block Store上，就找一个AWS EBS的驱动，如果想存储在Azure File Storage上，也找一个对应的Azure File Storage驱动即可。创建Volume时不指定卷驱动的话，那默认就是local类型，在Volume中存放的数据会存储在宿主机的 `/var/lib/docker/volumes/` 目录中。

到了容器编排系统里，Kubernetes同样也将Docker中Volume的概念延续了下来，并且进一步强化了它。Kubernetes中的Volume有明确的生命周期——与挂载它的Pod相同的生命周期，这意味着Volume比Pod中运行的任何容器的存活期都更长，Pod中不同的容器能自动共享相同的Volume，当容器重新启动时，Volume中的数据也会自动得到保留。当然，一旦整个Pod被销毁，Volume也将不再存在，而数据是否存在，取决于具体的回收策略以及驱动是如何实现的。

Kubernetes原本内置了相当多In-Tree的卷驱动，且时间上还早于Docker宣布支持卷驱动功能，这种策略使得Kubernetes能够在云存储提供商发布官方驱动之前就将其纳入到支持范围中，同时减轻了管理员维护的工作量，为它在诞生初期快速占领市场做出了一定的贡献。但是，这种策略也让Kubernetes丧失了添加或修改卷驱动的灵活性，只能在更新大版本时才能加入或者修改驱动，导致云存储提供商被迫与Kubernetes的发布节奏保持一致。此外，还涉及到第三方存储代码混杂在Kubernetes二进制文件中可能引起的可靠性和安全性问题。因此，当Kubernetes成为市场主流以后——准确的时间点是从1.14版本开始，Kubernetes启动了In-Tree卷驱动的CSI外置迁移工作，按照计划，在1.21到1.22版本（大约在2021年中期）时，Kubernetes中主要的卷驱动，如AWS EBS、GCE PD、vSphere等都会迁移至CSI的Out-of-Tree实现，不再提供In-Tree的支持。这种做法在设计上无疑是最正确的，然而，这又面临了该如何兼容旧功能的问题，譬如下面YAML定义了一个Pod：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-example
spec:
  containers:
  - name: nginx
    image: nginx:latest
    volumeMounts:
    - name: html-pages-volume
      mountPath: /usr/share/nginx/html
    - name: config-volume
      mountPath: /etc/nginx
```

```

volumes:
  - name: html-pages-volume
    hostPath:          # 来自本地的存储
      path: /srv/nginx/html
      type: Directory
  - name: config-volume
    awsElasticBlockStore:    # 来自AWS ESB的存储
      volumeID: vol-0b39e0b08745caef4
      fsType: ext4

```

其中用到了类型为 `hostPath` 的Volume，这相当于Docker中驱动类型为local的Volume，不需要专门的驱动；而类型为 `awsElasticBlockStore` 的Volume，从名字上就能看出是指存储驱动为AWS EBS的Volume，当CSI迁移完成，`awsElasticBlockStore` 从In-Tree卷驱动中移除掉之后，它就应该按照CSI的写法改写成如下形式：

```

- name: config-volume
  csi:
    driver: ebs.csi.aws.com
    volumeAttributes:
      - volumeID: vol-0b39e0b08745caef4
      - fsType: ext4

```

这样的要求显然有悖于升级版本不得影响还在大范围使用的已有功能的原则，所以Kubernetes 1.17中提出了[CSIMigration的解决方案](#)，让Out-of-Tree的驱动能够自动伪装成In-Tree的接口来提供服务。

笔者专门花几百字来介绍Volume的CSI迁移，并非由于它是多么重要的特性，而是这种兼容性设计本身就是Kubernetes设计理念的一个缩影，在Kubernetes的代码与功能中随处可见。好的设计需要权衡多个方面的利益，很多时候都得顾及现实的影响，而不能仅仅考虑理论最优的方案。

Static Provisioning

从操作系统里继承下来的Volume概念，在Docker和Kubernetes中继续按照一致的逻辑延伸拓展，这种有传承的概念通常会显得清晰易懂，没有歧义。如果仅用Volume就解决所有问题，Kubernetes的存储便不会如此繁琐，可惜的是容器编排系统里仅有Volume并不能够

满足全部的需要，最明显的缺陷是应用于规模较大的系统时，Volume将很难被自动化，这是由于Pod创建过程挂载某个Volume时，要求该Volume必须是真实存在的，否则Pod启动所依赖的数据（如一些配置、外部资源等等）都可能无从读取。Kubernetes有能力随着流量压力和硬件资源状况，自动扩缩Pod的数量，但是当Kubernetes自动扩展出一个新的Pod时，并没有办法让Pod去自动挂载一个还未被分配资源的Volume。想解决这个问题，要么要求多个不同的Pod去共用相同的Volume，这种方案确实只用Volume就能解决，却损失了隔离性；要么就要求每个Pod用到的Volume都是已经被预先建立并分配好的，这种方案管理员手工分配存储也可以实现，却损失了自动化能力。无论哪种情况，都算不上完美，难以符合Kubernetes工业级编排系统的产品定位，不过即使再不完美，也要先解决有无的问题才行，因此Kubernetes给出的第一个解决方案依然是继续从Volume延伸，拓展出了PersistentVolume与PersistentVolumeClaim的概念。

Persistent Volume and PersistentVolumeClaim

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator.

A PersistentVolumeClaim (PVC) is a request for storage by a user.

—— Kubernetes Reference Documentation , Persistent Volumes ↗

从定义上讲，PersistentVolume是抽象Volume的具象化表现，通俗地说就是已经被管理员分配好的具体的（这里的“具体的”是指明确的容量、访问模式、存储位置等信息）存储；PersistentVolumeClaim是Pod对其所需存储能力的声明，通俗地说就是满足这个Pod正常运行要满足怎样的条件，譬如要消耗多大的容量、要支持怎样的访问方式。定义中特别强调了PersistentVolume是由管理员（运维人员）负责维护的，用户（开发人员）通过PersistentVolumeClaim来匹配到合乎需求的PersistentVolume，两者配合工作过程是：

1. 管理员准备好要使用的存储系统，它应是某种网络文件系统（NFS）或者云储存系统，应该具备跨主机共享的能力。
2. 管理员根据存储系统的实际情况，手工预先分配好若干个PersistentVolume，并定义好每个PersistentVolume可以提供的具体能力。譬如以下例子所示：

```
apiVersion: v1
kind: PersistentVolume
```

yaml

```

metadata:
  name: nginx-html
spec:
  capacity:
    storage: 5Gi          # 最大容量为5GB
  accessModes:
    - ReadWriteOnce       # 访问模式为RXO
  persistentVolumeReclaimPolicy: Recycle # 回收策略是Recycle
  nfs:
    path: /html           # 存储驱动是NFS
    server: 172.17.0.2

```

以上YAML中定义的存储能力具体为：

- 最大容量是5GB。
- 访问模式是“只能被一个节点读写挂载”(ReadWriteOnce, RWO) , 另外两种可选的访问模式是“可以被多个节点以只读方式挂载”(ReadOnlyMany, ROX) 和“可以被多个节点读写挂载”(ReadWriteMany, RWX)。
- 回收策略是Recycle , 即在Pod被销毁时 , 自动执行 `rm -rf /volume/*` 这样的命令来自动删除资料 , 另外两种可选的回收策略是Retain (即人工回收) 以及Delete (用于AWS EBS、GCE PersistentDisk、OpenStack Cinder这些云存储的删除)。
- 存储驱动是NFS , 其他常见的存储驱动还有AWS EBS、GCE PD、iSCSI、RBD (Ceph Block Device) 、GlusterFS、HostPath , 等等。

3. 用户根据业务系统的实际情况 , 创建PersistentVolumeClaim , 列出所需的存储能力。譬如以下例子所示 :

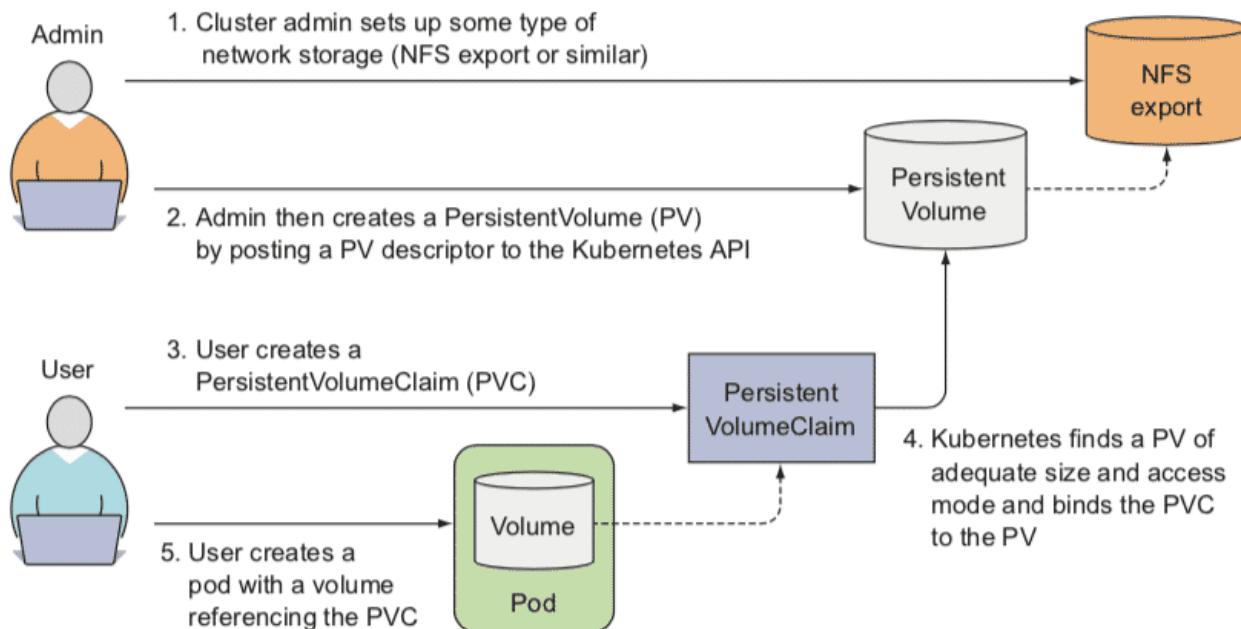
```

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nginx-html-claim
spec:
  accessModes:
    - ReadWriteOnce      # 支持RXO访问模式
  resources:
    requests:
      storage: 5Gi      # 最小容量5GB

```

以上YAML中声明了要求容量不得小于5GB，必须支持RWO的访问模式。

4. Kubernetes创建Pod的过程中，会根据系统中PersistentVolume与PersistentVolumeClaim的供需关系对两者进行撮合，撮合成功则将它们绑定。
5. 以上几步都顺利完成的话，意味着Pod的存储需求得到满足，继续Pod的创建过程，整个过程如下图所示。



PV/PVC运作过程（图片来自[Kubernetes in Action](#)）

Kubernetes对PersistentVolumeClaim与PersistentVolume撮合的结果是产生一对一的绑定关系，“一对一”的意思是PersistentVolume一旦绑定在某个PersistentVolumeClaim上，直到释放以前都会被这个PersistentVolumeClaim所独占，不能再与其他PersistentVolumeClaim进行绑定。这意味着即使PersistentVolumeClaim申请的存储空间比PersistentVolume能够提供的要少，依然要求整个存储空间都为该Pod所用，这有可能会造成资源的浪费。譬如，某个PersistentVolumeClaim要求3GB的存储容量，当前Kubernetes手上只剩下一个5GB的PersistentVolume了，此时Kubernetes只好将这个PersistentVolume与申请资源的PersistentVolumeClaim进行绑定，平白浪费了2GB空间。假设后续有另一个PersistentVolumeClaim申请2GB的存储空间，那它也只能等待管理员分配新的PersistentVolume，或者其他PersistentVolume被回收之后才能成功分配。

Dynamic Provisioning

纯粹靠管理员手工分配PersistentVolume对中小型系统来说尚可一用，但对与大型系统，面对成百上千，来自成千上万的Pod，靠管理员手工分配存储实在是捉襟见肘疲于应付。在2017年Kubernetes发布1.6版本后，终于提供了今天被称为Dynamic Provisioning的解决方案，让系统管理员摆脱了依靠人工分配的PersistentVolume的窘境，与之相对，人们把此前的分配方式称为Static Provisioning。

所谓的Dynamic Provisioning方案，是指在用户声明存储能力的需求时，不是期望通过Kubernetes撮合来获得一个人工预置的PersistentVolume，而是由特定的资源分配器（Provisioner）自动地在资源池或者云存储中分配符合用户存储需要的PersistentVolume，然后挂载到Pod中使用，完成这件事情的资源被命名为StorageClass。Dynamic Provisioning的具体工作过程是：

1. 管理员根据储系统的实际情况，先准备好对应的Provisioner。Kubernetes官方已经提供了一系列[预置的In-Tree Provisioner](#)，放置在 `kubernetes.io` API组之下。其中部分Provisioner已经有了官方的CSI驱动，譬如vSphere的Kubernetes自带驱动为 `kubernetes.io/vsphere-volume`，VMware的官方驱动为 `csi.vsphere.vmware.com`。
2. 管理员不再是手工去分配PersistentVolume了，而是根据存储去配置StorageClass。Pod是可以自动扩缩的，而存储则是相对固定的，哪怕使用的是具有扩展能力的云存储，也会将它们视为存储容量、访问IOPS等参数可变的固定存储，譬如你可以将来自不同云存储提供商、不同性能、支持不同访问模式的存储配置为各种类型的StorageClass，这也是它名字中“Class”的含义，譬如以下例子所示：

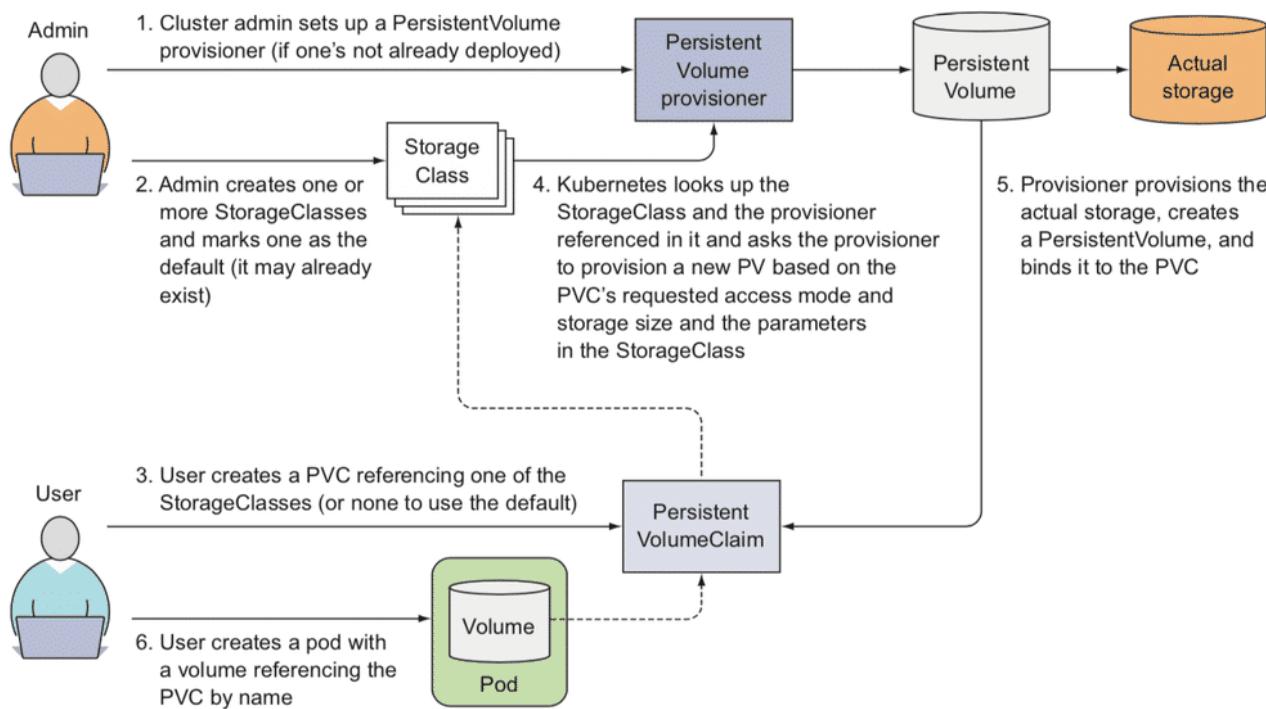
```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs  #AWS EBS的Provisioner
parameters:
  type: gp2
reclaimPolicy: Retain
```

yaml

3. 用户依然通过PersistentVolumeClaim来声明所需的存储，但是应在声明中明确指出该由哪个StorageClass来代替Kubernetes处理该PersistentVolumeClaim的请求，譬如以下例子所示：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: standard-claim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: standard  #明确指出该由哪个StorageClass来处理该PersistentVolumeClaim的请求
  resources:
    requests:
      storage: 5Gi
```

4. 如果PersistentVolumeClaim中要求的StorageClass及它用到的Provisioner均是可用的话，那这个StorageClass就会接管掉原本由Kubernetes撮合PersistentVolume与PersistentVolumeClaim的操作，按照PersistentVolumeClaim中声明的存储需求，自动产生出满足该需求的PersistentVolume描述信息，并发送给Provisioner处理。
5. Provisioner接收到StorageClass发来的创建PersistentVolume请求后，会操作其背后存储系统去分配空间，如果分配成功，就生成并返回符合要求的PersistentVolume给Pod使用。
6. 以上几步都顺利完成的话，意味着Pod的存储需求得到满足，继续Pod的创建过程，整个过程如下图所示。



StorageClass运作过程 (图片来自 [Kubernetes in Action](#))

Dynamic Provisioning与Static Provisioning并不是各有用途的互补设计，而是对同一个问题先后出现的两种解决方案。你完全可以只用Dynamic Provisioning来实现所有的存储需求，包括那些不需要动态分配的场景，譬如之前例子里使用HostPath在本地静态分配存储，便可以用指定 `no-provisioner` 作为Provisioner的StorageClass来代替，譬如以下例子所示：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

yaml

使用Dynamic Provisioning来分配存储由很多优点，不仅省去了管理员的人工操作的中间层，也不再需要将PersistentVolume这样的概念暴露给最终用户，因为Dynamic Provisioning里的PersistentVolume只是处理过程的中间产物，用户不再需要接触和理解它，只需要知道由PersistentVolumeClaim去描述存储需求，由StorageClass去满足存储需求即可。只描述意图而不关心具体处理过程是声明式编程的精髓，也是流程自动化的必要保障。由Dynamic Provisioning来分配存储还能获得更高的可管理性，譬如前面提到的回收策略，当Volume跟随Pod一同被销毁时，以前经常会配置回收策略为Recycle来回收空间，即让系统自

动执行 `rm -rf /volume/*` 命令，这种方式往往过于粗暴，遇到更精细的管理需求，譬如“删除到回收站”或者“粉碎式彻底删除”这样的功能实现起来就很麻烦。而Dynamic Provisioning中由于有Provisioner的存在，如何创建、如何回收都是由Provisioner的代码所管理的，这就带来了更高的灵活性。现在Kubernetes官方已经明确建议废弃掉Recycle策略，如有这类需求就改由Dynamic Provisioning去实现了。

不过笔者相信，从Kubernetes发布到现在，直至目前可见的将来，Kubernetes都还将会把Static Provisioning作为用户分配存储的一种可选方案，尽管这已经不是最佳设计，而同样是对历史现实的兼容。

容器存储与生态

调度硬件资源

服务网格

网格化服务

xDS协议

Envoy代理

向微服务迈进

没有银弹

传说里，能从普通人忽然变身的狼人是梦靥中最为可怕的怪物，人们一直尝试寻找
到能对狼人一枪毙命的银弹。

软件亦有着狼人的特性，平常看似人畜无害的技术研发工作，转眼间就能变成一只
工期延误、预算超支、产品满身瑕疵的怪兽。我听到了管理者、程序员与用户都在
绝望地呼唤，大家都渴望能找到某种可以有效降低软件开发的成本的银弹，让软件
开发的成本也能如同电脑硬件的成本那样，稳定且快速地下降。

—— Fred Brooks [\[1\]](#) , No Silver Bullet : Essence and Accidents of Software Engineering [\[2\]](#)
, 1987

这部文档的主体内容是务实的，多谈具体技术，少谈方向理论。只在本章中会集中讨论几
点与分布式、微服务、架构等相关的相对务虚的话题。

IBM大型机之父Fred Brooks [\[1\]](#) 在他的两本著作《**没有银弹：软件工程的本质性与附属性工
作** [\[2\]](#)》和《**人月神话：软件项目管理之道** [\[3\]](#)》里都反复强调着一个观点：“**软件研发中任何
一项技术、方法、架构都不可能是银弹**”，这个结论已经被软件工程里无数事实所验证，现
在对于微服务也依然成立。在本节中，笔者将会谈到哪些场景适合使用微服务，以及一些
已经被验证过、被总结为经验的最佳的实践方式；而更主要的是想讨论什么场景不适合微
服务，微服务存在哪些理解误区、应用前提，等等。

作为一部技术文档的作者，如果有同学是因为看了此文档，然后被带进微服务的沟里，那
作者只强调一句“微服务不是银弹”也难以免责，所以，在你准备发起实际行动向微服务迈
进前，希望你能阅读一遍本章，向微服务迈进——的避坑指南。

目的：微服务的驱动力

微服务的目的

The goal of microservices is to sufficiently decompose the application in order to facilitate agile application development and deployment.

微服务的目的是有效的拆分应用，实现敏捷开发和部署。

—— Chris Richardson, Founder of CloudFoundry, [Introduction to Microservices](#)

在讨论什么时候开始、以及如何向微服务迁移之前，我们先来理清为什么要迈向微服务。凡事总该先有目的，有预期收益再谈行动才显得合理。有人说迈向微服务的目的是为了追求更先进的架构形式。这话对，但没有什么信息量可言，任何一次架构演进的目的都是为了更加先进，应该没谁是为“追求落后”而重构系统的。有人说微服务是信息系统发展的必然阶段，为了应对日益庞大的压力，获得更好的性能，自然会演进至能够扩缩自如的微服务架构，这个观点看似合理、具体、正确，实则争议颇大。笔者个人的态度是旗帜鲜明地反对以“获得更高性能”为目的将系统重构为微服务架构的，性能有可能会作为辅助性的理由，但仅仅为了性能而进行微服务的话，那应该是40年前“[原始分布式时代](#)”所追求的目标。现代的单体系统同样会采用可扩缩的设计，同样能够集群部署，更重要的是云计算数据中心的处理能力几乎可以认为是无限的，那能够通过扩展硬件的手段解决问题就尽量别使用复杂的软件方法，其中原因在前面引用的《没有银弹》中已经解释过：**硬件的成本能够持续稳定地下降，而软件开发的成本则不可能**。此外，性能也不会因为采用了微服务架构而凭空产生。把系统拆成多个微服务，一旦在某个关键地方依然卡住了业务流程，其整体的结果往往还不如单体，没有清晰的职责划分，导致扩展性失效，多加机器往往还不如单机。前面这句话将性能替换为代码质量、生产力等词语往往也同样适用，这些方面笔者就不再展开了。

软件系统选择微服务架构，通常比较常见的、合理的驱动力来自组织外部、内部两方面，笔者先列举一些外部因素：

- 当意识到没有什么技术能够包打天下。

举个具体例子，某个系统选用了处于[Tiobe排行榜](#)榜首多年的Java语言来开发，也会遇到很多想做但Java不擅长的事情。譬如想去做人工智能，进行深度学习时，发现大量的库和开源代码都离不开Python；想要引入分布式协调工具时，发现近几年ZooKeeper已经有被后起之秀Golang的Etcd蚕食替代的趋势；想要做集中式缓存，发现无可争议的首选是ANSI C编写的Redis，等等。很多时候为异构能力进行的分布式部署，并不是你想或者不想的问题，而是无可避免的。

- 当个人能力因素成为系统发展的明显制约。

对于北上广深的信息技术企业这个问题可能不会成为主要矛盾，在其他地区，不少软件公司即使有钱也很难招到大量的靠谱的高端开发者。此时，无论是引入外包团队，抑或是让少量技术专家带着大量普通水平的开发者去共同完成一个大型系统，微服务都是一个更有潜力的选择。在单体架构下，没有什么有效阻断错误传播的手段，系统中“整体”与“部分”的关系没有物理的划分，系统质量只能靠研发与项目管理措施来尽可能地保障，少量的技术专家很难阻止大量螺丝钉式的程序员或者不熟悉原有技术架构的外包人员在某个不起眼的地方犯错并产生全局性的影响，并不容易做出整体可靠的大型系统。这时微服务可以作为专家掌控架构约束力的技术手段，由高水平的开发、运维人员去保证关键的技术和业务服务靠谱，其他大量外围的功能即使不靠谱（默认它们必定不靠谱），也能保证整体的稳定和局部的容错、自愈与快速迭代。

- 当遇到来自外部商业层面对内部技术层面提出的要求。

对于那些以“自产自销”为主的互联网公司来说这一点体验不明显，但对于很多为企业提供信息服务的软件公司来说，甲方爸爸的要求往往才是具决定性的推动力。技术、需求上困难也许能变通克服，但当微服务架构变成大型系统先进性的背书时，甲方的招投标文件技术规范明文要求系统必须支持微服务架构、支持分布式部署，那就没有多少讨价还价的余地。

-

在系统和研发团队内部，也会有一些因素促使其向微服务靠拢：

- 变化发展特别快的创新业务系统往往会自主地向微服务架构靠近。

需求喊着“要试错！要拥抱变化！”，开发喊着“资源永远不够！活干不完！”，运维喊着“你见过凌晨四点的洛杉矶吗！”，对于那种“一个功能上线平均活不过三天”的系统，如果团队本身能力能够支撑在合理的代价下让功能有快速迭代的可能，让代码能避免在类库层面的直接依赖而导致纠缠不清，让系统有更好的可观测性和回弹性（自愈能力），

需求、开发、运维肯定都是很乐意接受微服务的，毕竟此时大家的利益一致，微服务自然会水到渠成。

- 大规模的、业务复杂的、历史包袱沉重的系统也可能主动向微服务架构靠近。这类系统最后的结局不外乎三种：

第一种是日渐臃肿，客户忍了，系统持续维持着，直到谁也替代不了却又谁也维护不了。笔者曾听说过国外有公司招聘60、70岁以上程序员去维护上个世纪的COBOL编写的系统，没有求证过这到底是网络段子还是确有其事。

第二种是日渐臃肿，客户忍不了了，痛下决心，宁愿付出一段时间内业务双轨运行，忍受在新、旧系统上重复操作，期间业务发生震荡甚至短暂停顿的代价，也要将整套旧系统彻底淘汰掉，第二种情况笔者亲眼看见过不少。

第三种是日渐臃肿，客户忍不了，系统也很难淘汰。此时迫于外部压力，微服务会作为一种能够将系统部分地拆除、修改、更新、替换的技术方案被严肃地论证，若在重构阶段有足够的靠谱的技术人员参与，该大型系统的应用代码和数据库都逐渐分离独立，直至孵化出一个个可替换可重生的微服务，微服务的先驱Netflix曾在多次演讲中介绍说自己公司属于第三种的成功案例。

-

以上列举的这些内外部原因并不是全部，促使你的产品最终选择微服务的具体理由可能是多种多样，你做出先微服务迈进的决策时肯定经过恰当的权衡，认为收益大于成本。微服务最主要的目的是对系统进行有效的拆分，实现物理层面的隔离。微服务的核心价值就是拆分之后的系统能够让局部的单个服务**有可能**实现敏捷地卸载、部署、开发、升级。局部的持续更迭，是系统整体具备Phoenix特性的必要条件。

前提：微服务需要的条件

康威定律

Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.

系统的架构趋同于系统设计团队的沟通结构。

—— Melvin Conway [↗](#), Conway's Law [↗](#), 1968

无论是上面列举或者没有列举到的哪种原因，现在笔者假设你所在的组织已经作出了要向微服务迈进的决定。那下一件你应要弄明白的事情就是，什么情况下可以开始微服务化，或者说，微服务需要哪些前提条件？对于此问题，Martin Fowler [↗](#) 曾经撰写过文章《Micro service Prerequisites [↗](#)》从技术角度专门讨论过该问题，不过笔者认为微服务的前提条件首要还是该先解决非技术方面的问题，准确地说是人的问题。微服务不是一项纯粹的技术性工作，如果不能满足以下条件，就应该尽量避免采用微服务。

微服务化的第一个前提条件是决策者与执行者都能意识到康威定律在软件设计中的关键作用。

康威定律尝试使用社会学的方法去解释软件研发中的问题，其核心观点是“沟通决定设计”（Communication Dictates Design），如果技术层面紧密联系在一起特性，在组织层面上强行分离开来，那结果是沟通成本的上升，因为会产生大量的跨组织的沟通；如果技术层面本身没什么联系的特性，在组织层面上强行安放一块，那结果是管理成本的上升，因为成员越多越不利于一致决策的形成。这些社会学、管理学的规律决定了假如产品和组织能够经受住考验，长期发展的话，最终都会自发地调整成组织与产品互相匹配的状态。哪些特性在团队内部沟通，哪些特性需要跨团队的协作，将最终都会在产品中分别映射成与组织结构一致的应用内、外部的调用与依赖关系。

尽管稍微有工作经验的员工和管理者只要稍微思考一下都能理解康威定律所描述的现象，但是为了推进软件架构的微服务化而配合地调整组织架构，通常不是一件容易的事情。西

方有一句谚语叫做“所有的技术上的决策实际都是政治上的决策（All Technical Decisions Are Political Decisions）”，这里“政治”是泛指如何与其他人协作将事情搞定。架构不仅仅是技术问题，更是一种社交活动，甚至还会涉及利益重新分配，譬如，将某个组织的一部分权利、职能和人员拆分出来，该组织的领导愿不愿意？将两个团队合并成一个新的团队，总会有一个团队负责人要考虑该怎么安置？这些问题不仅需要执行者有良好的社交能力，还需要更上层的决策者充分理解架构演变同步调整组织结构的必要性，为微服务化打破局部的利益藩篱。

微服务化的第二个前提条件是组织中具备一些的对微服务有充分理解、有一定实践经验的技术专家。

笔者在“[微服务时代](#)”中曾写到“作为一个普通的服务开发者，作为一个螺丝钉式的程序员，微服务架构是友善的。可是，微服务对架构者是满满的恶意，对架构能力要求已提升到史无前例的程度”。即使对微服务最乐观的支持者也无法否认它在架构方面的技术复杂性。对于开发业务逻辑的普通程序员，即使代码出现缺陷也可以被快速修复升级，甚至有可能在Kubernetes的帮助下自动回弹，哪怕不能自愈，最起码错误也会被系统自动地隔离，而不至于影响全局弄崩整个系统。开发业务的普通程序员可以不去深究跟踪治理、负载均衡、故障隔离、认证授权、伸缩扩展这些系统性的问题，它们被隐藏于软件架构的最底层，被掩埋在基础设施之下。与此相对的另外一面，靠谱的软件架构应该要由深刻理解微服务的技术专家来设计建造，健壮的基础设施也离不开有经验的运维专家的持续运维，Kubernetes、Istio、Spring Cloud、Dubbo等现成的开源工具能在此过程发挥很大的作用，但它们本身也具备不低的复杂性。如果整个团队中缺乏能够在微服务架构中撑起系统主干的技术和运维专家，强行进行微服务化并不会有任何好处，至少收益不足以抵消复杂性增加而导致的成本。这些技术专家不需要多（能多当然更好），但是一定必须有，如今在软件职场中阿里、腾讯等大厂出来的程序员受到追捧，除了本身企业带来的光环外，有大型系统浸染的经验，更有可能是技术专家也是其中主要原因之一。

微服务对普通程序员友善的背后，预示着未来的信息技术行业很可能也会出现“阶级分层”的现象，由于更先进的软件架构已经允许更平庸的开发者也同样能写出可运行、可用于生产的软件产品，同时又对精英开发者提出更多、更复杂的技术要求，长此以往，在开发者群体中会出现比现在还要显著的[马太效应](#)。如果把整个软件业界这个看作一个巨大组织的话，它也应会符合康威定律，软件架构的趋势将导致开发者的分层，即从如今所有开发者都普遍被认为是“高智商人群”的状态，转变为大部分工业化软件生产工人加上小部分软件设计专家的金字塔结构。

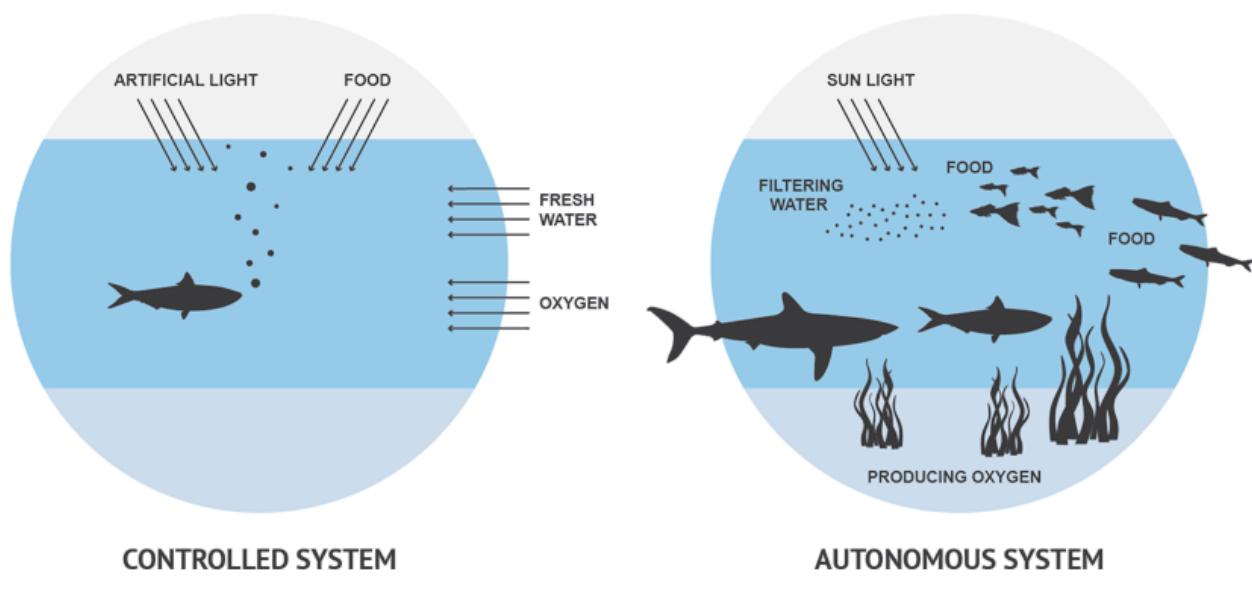
微服务化的第三个前提条件是系统应具有以自治为目标的自动化与监控度量能力。

微服务是由大量松耦合服务互相协作构成的系统，将自动化与监控度量作为它的建设前提是顺理成章的。Martin Fowler在《Microservice Prerequisites》中提出了微服务系统的三个技术前提都跟自动化与监控度量有关，分别是：

- 环境预置（Rapid Provisioning）：即使不依赖云计算数据中心的支持，也有能力在短时间（原文是几个小时，如今Kubernetes重启一个Pod只需要数十秒）内迅速地启动好一台新的服务器。
- 基础监控（Basic Monitoring）：监控体系有能力迅速捕捉到系统中出现的技术问题（如异常、服务可用性变化）和业务问题（如订单交易量下降）。
- 快速部署（Rapid Application Deployment）：有能力通过全自动化的部署管道，将服务的变更迅速部署到测试或生产环境中。

请注意Martin Fowler撰写这篇文章的时间是2014年，彼时连Kubernetes都还没有从闭源的Borg中诞生，虚拟化、自动化技术仍是较初级水平。近年来，许多公司都构建起了DevOps文化，虚拟化与开发运维自动化有了长足发展，2014年要专门强调的“前提条件”对今天的系统来说都算不上有什么困难。在这里笔者更希望强调的重点是“以自治为目标”，如果不是朝这个方向去努力的话，自动化最终会导向一个套娃式的悖论：即使所有运维都实现了自动化，同时有一个监控系统来随时恢复出现故障的服务，然而这个监控系统本身也需要被监控。如果启用另一个监控系统，同样这个监控系统需要被监控。最终，不论自动化实现了多少层，顶层仍然必须是人，只有人能确保整体运维的连续性，所以永远也无法达到完全的自动化。而且，这些自动化与监控措施本身也会消耗资源，也会带来更高的复杂性。

微服务自动化的最终目的是一個构筑可持续的生态系统。这句话听起来很抽象，有点像主席台上领导的演讲词。笔者用一个具体的比喻加以说明：如果将微服务比作水族馆里养的鱼，为了维持鱼的生存，管理员需要不断向水族馆内添加各种自动化设施：人工照明、氧化剂、水过滤器、加热器，等等。这些设施最终仍然需要人花费精力去照料维护，本身就耗费了大量成本。如果我们换一种思路，通过种植海洋植物以提供氧气、通过藻类过滤水质、通过放养螺类来清理鱼缸等等。这样的水族馆就不再是依靠人工维护才能存在的水族馆了，它变成了一个小型的湖泊或海洋，理想状态下，这里的鱼类可以不需要人的干预就能长期存活。

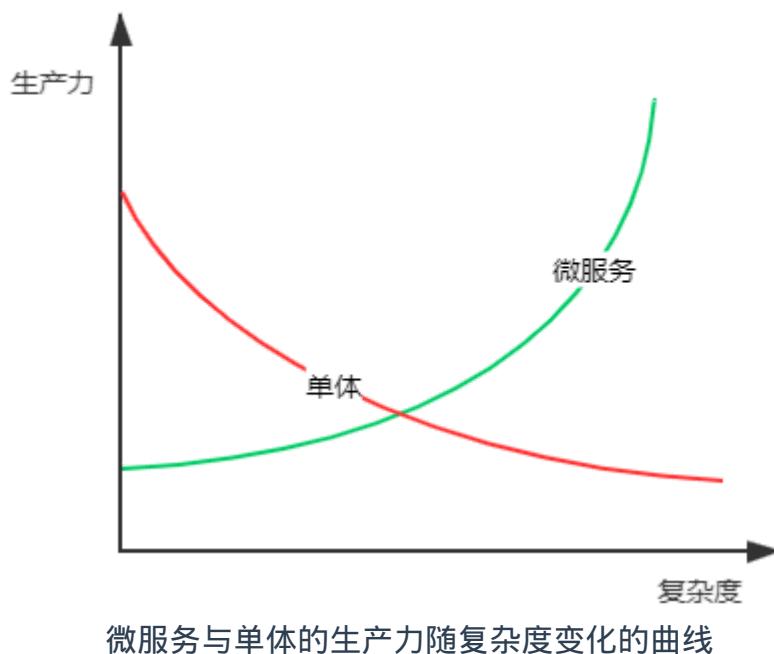


从人工控制系统到自动化生态系统（[图片来源](#)）

以生态自治为目标的自动化，并不是指要达到如此高的自动化程度之后才能开始微服务，只要满足与系统规模和目标相匹配的自动化能力，建设微服务的早期由人工参与运维完全是合情合理的。退一步说，即使在信息化水平最高的大型互联网企业中，生态自治在当前技术水平下仍然是一个过于理想化、难以全面落地的目标，不过，只有朝着这个目标去发展自动化与监控度量，才能避免屠龙少年最终变成恶龙，避免自动化与度量监控反过来成为人与系统的负担。

微服务化的第四个前提条件是**复杂性已经成为制约生产力的主要矛盾**。

在“[单体系统时代](#)”的开篇笔者就阐述了一个观点：“对于小型系统，单体架构就是最好的架构”。系统进行微服务化的根本动力是这样做有收益，一般情况下，引入新技术在解决问题之前会带来复杂度的提升，反而导致生产力下降。只有在业务已经发展到一定的程度，单体架构与微服务架构的生产力曲线已经到达交叉点，此时开始进行微服务化才是有收益的。关于复杂度的问题，将在“[治理：理解系统复杂性](#)”一节中更具体地探讨。



关于复杂性、生产力的性价比问题并不难理解，然而现实中很多架构师却不得不在这上面主动犯错。新项目在立项之初，往往都会定下令人心动的目标愿景，远景规划在战略上有益的，不过多数技术决策都属于战术范畴，应该依据以现实情况而不是远景规划去做决定。遗憾的是管理者、乃至技术架构师都不能真正地接受[演进式设计](#)（Evolutionary Design），尤其不能接受一个具有良好设计的系统，应该是能够被报废的，潜意识中总会希望一步到位，至少是“少走几步能到位”。

演进式设计

Many services to be scrapped rather than evolved in the longer term.

长期来看，多数服务的结局都是报废而非演进。

—— Martin Fowler [\[1\]](#), Microservices [\[2\]](#)

笔者举个“主动犯错”具体例子，试想你就是一名架构师，项目立项中坚持要选择单体架构，此时你就要考虑到日后评审时，别的团队说他的产品采用了微服务，架构上比你的先进；考虑到招聘人员时，程序员听见你这里连微服务都没用，觉得制约了自己的发展前景；考虑到项目成功火爆了，几个月后你再提出进行微服务化，老板听了心里觉得你水平的确不行，之前采用单体是错误决定，导致现在要返工。

以上，便是笔者总结的开始微服务化的四个前提条件，如果你做技术决策时，能仅以技术上的收益为度量标准，根据这些前提就能判断应该或者不应该采用微服务，那你工作的氛

围是比较开明的；如果你做技术决策要考虑的收益不仅限于技术范畴之内，我也完全能够理解，毕竟，所有的技术上的决策实际都是政治上的决策。

边界：微服务的粒度

勿行极端，过犹不及

子贡问：“师与商也孰贤？”子曰：“师也过，商也不及。”曰：“然则师愈与？”子曰：“过犹不及。”

子贡问：“颛孙师和卜商谁更贤德？”孔子说：“颛孙师常常作得有些过头，卜商常常达不到要求。”子贡说：“如此说来，是不是颛孙师要好一些呢？”孔子说：“过头和达不到同样不好。”

—— 论语·先进

当今软件业界，对本节的话题“识别微服务的边界”其实已取得了较为一致的观点，也找到了指导具体实践的方法论，即领域驱动设计¹（Domain-Driven Design，DDD）。囿于主题，在这部文档中甚少涉及该如何抽象业务、分析流程、识别边界、建立模型、映射到服务和代码等偏重理论的务虚话题，即使在这一章中，笔者也尽量规避了DDD中需要专门学习才能理解的概念，如界限上下文（Bounded Context）、语境映射（Context Map）、通用语言（Ubiquitous Language）、领域和子域（Domain、Sub Domain）、聚合（Aggregate）、领域事件（Domain Event）等等。并非笔者认为业务流程与设计方法论不重要，而是如果要严谨、深刻地讨论这些话题，其篇幅足以独立地写出一本书。事实上，市场上已经有不少这样的书了，DDD的发明人Eric Evans撰写的同名书籍《领域驱动设计：软件核心复杂性应对之道²》便是其中翘楚。笔者个人是更推荐Chris Richardson撰写的颇具口碑的入门书《微服务架构设计模式³》，其叙述的主线就是在DDD指导下，如何将一个单体服务逐步拆分为微服务结构，如果你对这方面感兴趣，不妨一读。这两节中，笔者会从业务之外的其他角度，从非功能性、研发效率等方面来探讨微服务的粒度与拆分。

不可能每一位架构师设计的服务粒度全都相同，微服务的大小、边界不应该只有唯一正确的答案或绝对的标准，但是应该有个合理的范围，笔者称其为微服务粒度的上下界。我们可以分析如果微服务的粒度太小或者太大会出现哪些问题，从而得出服务上下界应该定在哪里。

可能是受微服务名字中“微”的“蛊惑”，笔者听过不少人提倡过微服务越小越好，最好做到一个REST Endpoint就对应于一个微服务，这种极端的理解肯定是错误的，如果将微服务粒度定的过细，会受到以下几个方面的反噬：

- 从性能角度看，一次进程内的方法调用（仅计算调用，与方法具体内容无关），耗时在零（按方法完全内联的场景来计算）到数百个[时钟周期](#)（按最慢的虚方法调用无内联缓存要查虚表的场景来计算）之间；一次跨服务的方法调用里，网络传输、参数序列化和结果反序列化都是不可避免的，耗时要达到毫秒级别，你可以算一下这两者有多少个数量级的差距。[远程服务调用](#)里已经解释了“透明的分布式”是不存在的，因此，服务粒度大小必须考虑到消耗在网络上的时间与方法本身执行时间的比例，避免设计得的过于琐碎，客户端要多次调用服务才能完成一项业务操作，譬如，将字符串处理这样的功能设计为一个微服务便是不合适的。这点要求微服务应该是完备的。
- 从数据一致性角度看，每个微服务都有自己独立的数据源，如果多个微服务要协同工作，我们可以采用[很多办法](#)来保证它们处理数据的最终一致性，但如果某些数据必须要求保证强一致性的话，那它们本身就应当聚合在同一个微服务中，而不是强行启用[XA事务](#)来实现，因为在参与协作的微服务越多，XA事务的可用性就越差。这点要求微服务应该是[内聚](#)（Cohesion）的。
- 从服务可用性角度看，服务之间是松散耦合的依赖关系，微服务架构中无法也不应该假设被调用的服务具有绝对的可用性，服务可能因为网络分区、软件重启升级、硬件故障等任何原因发生中断。如果两个微服务都必须依赖对方可用才能正常工作，那就应当将其合并到同一个微服务中（注意这里说的是“彼此依赖对方才能工作”，单向的依赖是必定存在的）。这条要求微服务应该是独立的。

综合以上，我们可以得出第一个结论：**微服务粒度的下界是它至少应满足独立——能够独立发布、独立部署、独立运行与独立测试，内聚——强相关的功能与数据在同一个服务中处理，完备——一个服务包含至少一项业务实体与对应的完整操作。**

我们再来想想，如果微服务的粒度太大，会出现什么问题？从技术角度讲，并不会有什麼问题，每个能正常工作的单体系统都能满足独立、内聚、完备的要求，世界上又有那么多运行良好的单体系统。微服务的上界并非受限于技术，而是受限于人，更准确地说，受限于人与人之间的社交协作。《人月神话》中最反直觉的一个结论是：“为进度给项目增加人力，如同用水去为油锅灭火”（Adding Manpower to a Late Software Project Makes It Later）。为什么？Fred Brooks给出了简洁而有力的答案：

软件项目中的沟通成本= $n \times (n-1)/2$, n为参与项目的人数

为了让你能更直观地理解这个答案，笔者已经算好了一组数字：15人参与的项目，沟通成本大约是5个人时的十倍，150人参与的项目，沟通成本大约是5个人时的一万倍。你不妨回想一下自己在公司的工作体验，不可能有150人的团队而不划分出独立小组来管理的，除非这些人都从事流水线式的工作，协作时完全不需要沟通。此外，你也不妨回想一下自己的生活体验，我敢断言你的社交上界是不超过5个知己好友，15个可信任的伙伴，35个普通朋友，150个说得上话的人。这句话的信心底气源于此观点是人类学家Robin Dunbar
在1992年给出的科学结论，今天已被普遍认可，被称为“邓巴数”（Dunbar's Number），据说是人脑的新皮质大小限制了人能承受的社交数量，决定了邓巴数这个社交的上界。

有了以上铺垫，你应该能更能理解前面的许多文章中笔者为何采用“2 Pizza Team”作为微服务团队规模的“量词”了，并不是因为制造这个梗的人是Jeff Bezos，是亚马逊CEO、世界首富。而是因为两个Pizza能喂饱的人数大概就是6-12人，符合软件开发中团队管理的理想规模。

康威定律约束了软件的架构与组织的架构要保持一致，所以微服务的上界应该与2 Pizza Team能够开发的最大程序规模保持一致。2 Pizza Team能开发多大规模的程序？人员数量固定的前提下，这个答案不仅与开发者的能力水平相关，更是与研发模式和周期相关。如果你的软件产品是瀑布开发，可能需要一个月、两个月迭代一次；如果采用Scrum，可能会一周、两周完成一次冲刺；如果追求日构建、精益，甚至可能一天、两天就会集成构建出一个小版本，以上不同的研发方法，都会产生相应规模的上界。

综合以上，我们得出了第二个结论：**微服务粒度的上界是一个2 Pizza Team能够在一个研发周期内完成的全部需求范围。**

在上下界范围内，架构师会根据业务和团队的实际情况来灵活划定微服务的具体粒度。譬如下界的完备性要求微服务至少包含一项完整的务，不超过上界的前提下，这个微服务包含了两项、三项业务操作是否合理，那需要根据这些操作本身是否有合理的逻辑关系来具体讨论。又譬如上界要求单个研发周期内能处理掉一个微服务的全部需求，不超过下界的前提下，一个周期就能完成分属于两个、三个微服务的全部需求时，是缩短研发周期更合理，还是允许这个周期内同时开发几个微服务，也可以根据实际情况具体讨论。

治理：理解系统复杂性

治理 (Governance)

Ensuring and validating that assets and artifacts within the architecture are acting as expected and maintaining a certain level of quality.

治理就是让产品能够符合预期地稳定运行，并能够持续保持在一定的质量水平上。

—— Gartner[↗]，Magic Quadrant for SOA Governance[↗], 2007

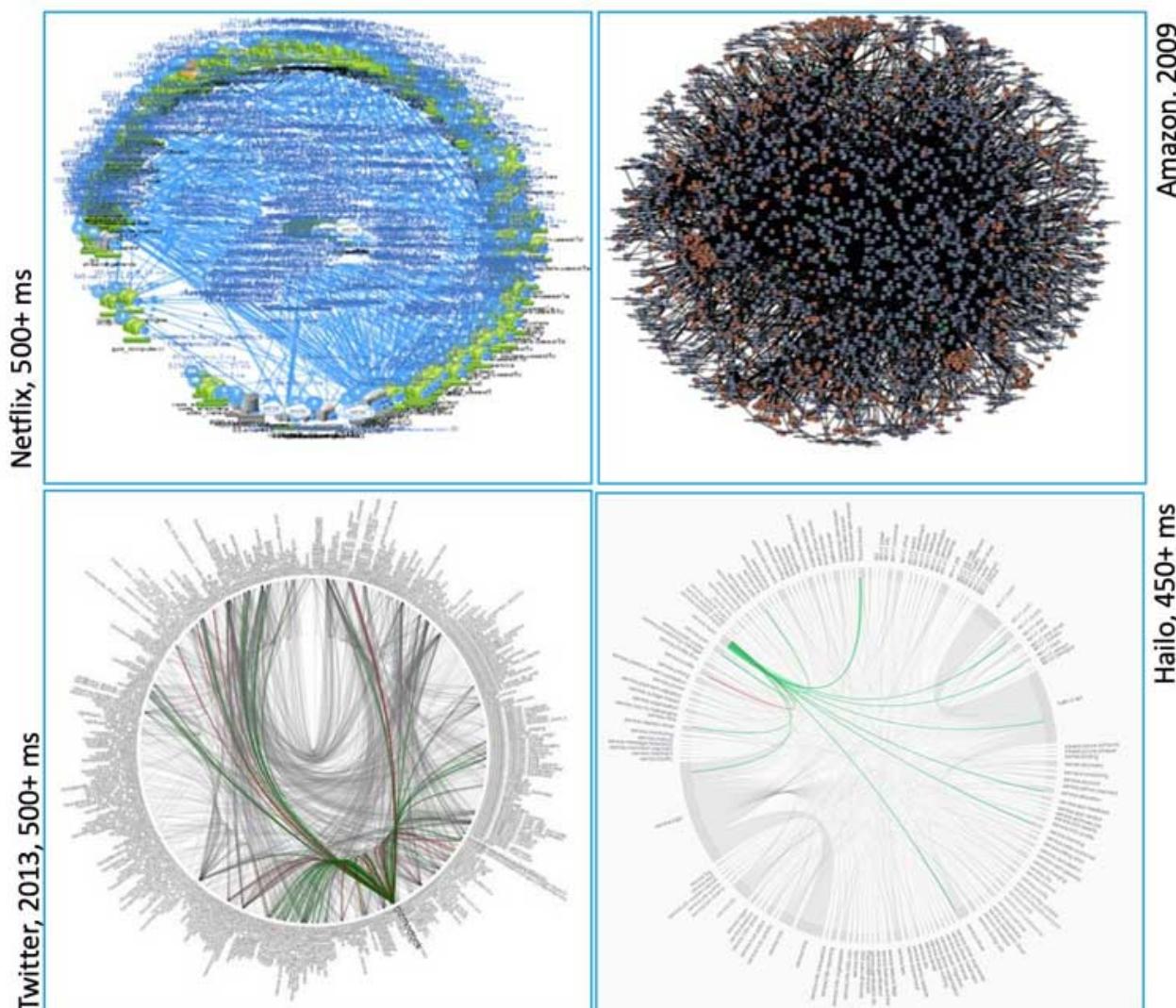
行文至此，本章的字里行间都有意无意地流露出微服务架构的复杂性，反复提醒读者三思后行、慎重决策，却还没有解释过复杂性具体是什么、有多么复杂、有没有解决的办法。对于最后这个问题，笔者其实并不能给出具体的能解决复杂性的灵丹妙药，药到病除的仙方在软件研发中估计永远无法求到。这节里，笔者将重点分析前两个问题，正确理解软件的复杂性，日后实际解决问题时方能有的放矢。

软件业的确经常会使用到“治理[↗]” (Governance) 这个词，听着高级，用着贴切，譬如系统治理、业务治理、流程治理、服务治理，等等。这个词的确切含义是让产品（系统、业务、流程、服务）能够符合预期地稳定运行，并能够持续保持在一定的质量水平上。该定义把治理具体分解为“正确执行”（让软件符合预期地运行）和“持续保持”（让软件持续保持一定质量水平地运行）两个层次的要求，笔者分别从静与动两种角度解释治理与复杂性的关系。

静态的治理

要求一个信息系统能够符合预期地运行，这听起来无论如何不算上什么“高标准”。不过，当复杂性高到一定程度的时候，能达到正常运行确实就已经离不开治理。笔者举例类比一下便于你理解：一只存活的蜂王或者蚁后就能够满足一个昆虫族群稳定运行的需要，一位厨艺精湛的饭店老板也能够满足一家小饭馆稳定运行的需要，一个君圣臣贤的统治集团才能满足一个庞大帝国稳定运行的需求。治理好蜂群只要求蜂王活着即可，治理好饭馆要依

赖老板个人的高明厨技，到了治理国家社稷就要求皇帝圣明大臣贤良才行，可见族群运作的复杂度就越高，治理难度也越高。如果你还是没能将族群与个体的关系跟系统与服务的关系联系起来，那看看以下这张图，仅凭直观感觉也能体会到这些著名企业的由数百上千微服务互相调用依赖构成的系统能够正常运行就并不简单。



服务间交互关系 ([图片来源](#))

说服你认可治理国家比治理一群蚂蚁要更复杂应该不太困难，说服两个软件系统各自的拥护者哪个系统更复杂却不容易。决定复杂度高低的是微服务多少吗？是类或文件的个数吗？是代码行数吗？是团队人员规模吗？答案很模糊，复杂是相当对于人而言的，是一个主观标准，每个人都可以有不同的裁量。基于大型软件都是由开发者们互相协作完成的这个基本出发点，笔者用以下两个心理学概念来解释复杂性的来源，受到较广泛的认可：

- 复杂性来自认知负荷 ([Cognitive Load](#))：在软件研发中表现为接受业务、概念、模型、设计、接口、代码等信息所带来的负担大小。系统中个体的认知负担越大，系统

越复杂，这点解释了为什么蚂蚁族群和国家的人口可能一样多，但治理国家比治理一群蚂蚁要更复杂。

- 复杂性来自协作成本 [\(Collaboration Cost \)](#)：在软件研发中表现为团队共同研发时付出的沟通、管理成本高低。系统个体间协作的成本越高，系统越复杂，这点解释了为什么小饭馆和国家的构成个体都同样是人类，但治理国家比治理一家饭馆要更复杂。

根据这两个概念，我们可以量化地推导出前文中使用过的一个结论：**软件规模小时微服务的复杂度高于单体系统，规模大时则相反，原因是微服务的认知负荷较高，但是协作成本较低。**

软件研发的协作成本，其实就是来自协作的沟通复杂度。前一节的讨论微服务粒度时已经使用过沟通成本的公式：沟通成本 $=n \times (n-1)/2$ ，这是一种随着规模增长呈平方级增长复杂度，借用[算法复杂度](#)的表示方法那就是 $O(N^2)$ 。在微服务架构下，组织的拆分与微服务的拆分对齐（康威定律），微服务系统的交互分为了服务内部的进程内调用和服务之间的网络调用，组织的沟通也被拆分为团队内部的沟通与团队之间的协作，这种分治措施有利于控制沟通成本的增长速度，此时沟通成本的复杂度，也就等同于经典分治算法的时间复杂度，即 $O(N \log N)$ 。

软件研发的认知负荷，其实就是来自技术的认知复杂度。每次技术进步都伴随着新知识、新概念的诞生，说技术进步会伴随复杂度升级也无可。只是微服务或者说分布式系统所提倡许多理念，都选择偏向于机器而不是人，有意无意地加剧了该现象。举个具体例子，心理学研究告诉我们，与现实世界不符合的模型会带来更高的认知负荷，因此面向对象编程（OOP）这种以人类观察世界的视角去抽象系统的设计方式是利于降低认知负荷的，但分布式系统提倡面向资源编程（服务间交互是REST，服务内部并不反对你使用OOP），服务之间的交互绝不提倡面向对象来进行，Martin Fowler曾经撰文《[Microservices and the First Law of Distributed Objects](#)》强调分布式的第一原则就是不要分发对象（Don't Distribute Your Objects）。微服务加剧认知负荷还体现在很多其他方面，如异步通讯（异步比同步更难理解）、粗粒度服务接口（粗粒度API比细粒度API更难使用，关于这点在Martin Fowler的原文中有详细的解释）、容错处理（服务容错比异常更为复杂）、去中心化（尽管中心化设计会降低可用性，但确实比非中心化有更高的可管理性）等等。该结果并不让人感到意外，在原始分布式时代中笔者就提到过，分布式系统早已放弃了Unix所追求的简单性是系统第一属性的设计哲学。

由于认知负荷是与概念、模型、业务、代码的规模呈正比关系，这些工作都是由人来做的，最终都能被某种比例系数放大之后反应到人员规模上，可以认为认知负荷的复杂度是 $O(k \times N)$ （为便于讲解，这里复杂度刻意写成未除消系数的形式），单体与微服务的差别是复杂度比例系数k的大小差别，微服务架构的k要比单体架构的k更大。软件研发的整体复杂度是认知负荷与协作成本两者之和，对于单体架构是 $O(k \times N) + O(N^2)$ ，对于微服务架构，整体复杂度就是 $O(k \times N) + O(N \log N)$ ，由于高次项的差异，N的规模增加时单体架构的复杂度增长更快，这就定量地论证了“件规模小时微服务的复杂度高于单体系统，规模大时则相反”的观点。

笔者用了千余字的篇幅，目的不是为了证明这个观点的正确，很多架构师仅凭经验也能直观感受出它是正确的。笔者的目的是想解释清楚软件研发的复杂性的来源与差距程度，并说明微服务中分治思想对控制软件研发复杂性的价值。假如只能用一个词来形容微服务核心思想，笔者给的答案就是“分治”，这即是微服务的基本特征，也是微服务应对复杂性的手段。

发展的治理

我们再来看治理对动态发展方面的要求，它指采取某些措施，让软件系统能够持续保持一定的质量。“持续保持”听起来只是守成，应该至少不比建设困难。可是一个令人感到意外的结论是此目标其实不可能实现，如果软件系统长期接受新的需求输入，它的质量必然无法长期保持。软件研发中有一个概念“[架构腐化](#)”（Architectural Decay）专门形容此现象：架构腐化只能延缓，无法避免。

架构腐化与生物的衰老过程很像，原因都来自于随时间发生的微妙变化，如果你曾经参与过多个项目或产品的研发，应该能对以下场景有所共鸣：项目在开始的时候，团队会花很多时间去决策该选择用什么技术体系、哪种架构、怎样的平台框架、甚至具体到开发、测试和持续集成工具。此时就像小孩子们在选择自己所钟爱的玩具，笔者相信无论决策的结果如何，团队都会欣然选择他们所选择的，并且坚信他们的选择是正确的。事实也确实如此，团队选择的解决方案通常能够解决技术选型时就能预料到的那部分困难。但真正困难的地方在于，随着时间的流逝，团队对该项目质量的持续保持能力会逐渐下降，一方面是高级技术专家不可能持续参与软件稳定之后的迭代过程（反过来，如果持续绑定在同一个项目上，也很难培养出技术专家），老人的退出新人的加入使得团队总是需要理解旧代码的同时完成新功能，技术专家偶尔来评审一下或救一救火，充其量只能算临时抱佛脚；另

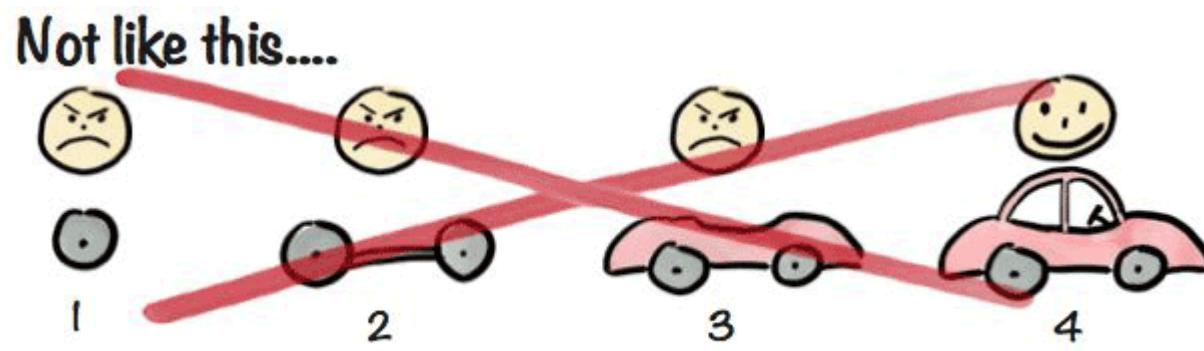
一方面是代码会逐渐失控，时间长了一定会有某些并不适合放进最初设计中的需求出现，工期紧任务重业务复杂代码不熟悉都会成为欠下一笔技术债的妥协理由，原则底线每一次被细微地突破，都可能被[破窗效应](#)撕裂放大成触目惊心的血痕，最终累积到每个新人到来就马上能嗅出老朽腐臭味道的程度。

架构腐化是软件动态发展中出现的问题，任何静态的治理方案都只能延缓，不能根治，必须在发展中才能寻找到彻底解决的办法。治理架构腐化唯一有效的办法是演进式的设计，这点与生物族群的延续也很像，只有流水，才能不腐。

演进式设计这个词语此前的文章中已经提到过多次，它是微服务中提倡的主要特征之一，也是作为技术决策者的架构师应该具备的发展式思维。架构师（Architect）一词是软件行业从建筑行业引进的舶来词，Arch本身就是拱形建筑的含义。有很多资料都把软件架构师类比解释为给建筑设计骨架、绘制图纸的建筑架构师，这里面其实潜藏着极大的误导。一个复杂的软件与一栋复杂的建筑看似有可比性，两者的演进过程却截然不同。万丈高楼也是根据预先设计好的完整详尽图纸准确施工而建成的，但是任何一个大型的软件系统都绝不可能这样建造出来。演进式设计与建筑设计的关键区别是，它不像是“造房子”，更像是“换房子”。举个具体的例子你就能明白：

在校求学的你住着六人间宿舍；
初入职场的你搬进了单间出租屋；
新婚燕尔的你买下属于自己的两室一厅；
孩子上学时，你换上了大户型的学区房；
孩子离家读书时，你也终于走上人生巅峰，换了一套梦想中的大别墅。

对于你住进大别墅的过程，后一套房子并不是前一套房子的“升级版本”，两套房子之间只有法律上继承关系，没有血源上的继承关系。同理，大型软件的建设是一个不断推倒从来的演进过程，前一个版本对后一个版本的价值在于它满足了这个阶段用户的需要，让团队成功适应了这个阶段的复杂度，可以向下一个台阶迈进。对于最终用户来说，一个能在演进过程中逐步为用户提供价值的系统，体验也要远好于一个憋大招的系统（哪怕这大招最终能成功憋出来），如下图这幅关于理想交通工具的漫画所示。



理想交通工具（图片来源[此处](#)）

额外知识：演进式设计

演进式设计是ThoughtWorks提出的架构方法，无论是代际的演进还是渐进的演进，都带有不少争议，它不仅是建造的学问，也是破坏的学问。Neal Ford撰写的《Building Evolutionary Architectures: Support Constant Change[此处](#)》一书比较详细地阐述了演进式架构的思想，受到不少关注，却不见得其中所有观点都能得到广泛认可。如果你是管理者，大概很难接受正是那些正常工作的系统带来了研发效率的下降；如果你是程序员，估计不一定能接受代码复用性越高、可用性越低这样与之前认知相悖的结论。

笔者强调的演进式设计，不应被过度解读成系统最终都是会腐化，项目最终是要被推倒重建的，针对特定阶段的努力就没有什么作用。静态的治理措施当然有它的价值，我们无法避免架构腐化，却完全有必要依靠良好的设计和治理，为项目的质量维持一段合理的“保质期”，让它在合理的生命周期中发挥价值。

复杂性本身不是洪水猛兽，无法处理的复杂性才是。刀耕火种的封建时代无法想像机器大生产中的复杂协作，蒸汽革命时代同样难以想像数字化社会中信息的复杂流动。先进的生产力都伴随着更高的复杂性，需要有与生产力符合的生产关系来匹配，敏锐地捕捉到生产力的变化，随时调整生产关系，这就是架构师治理复杂性的终极方法。

事件驱动架构

事件驱动架构（Event Driven Architecture，EDA）本质上是一种应用、组件间的集成架构模式，事件和传统的消息不同，事件具有Schema，所以可以校验事件的有效性，同时EDA具备QoS保障机制，也能够对事件处理失败进行响应。事件驱动架构不仅用于服务解耦，还可应用于下面的场景中：

- **增强服务韧性**：由于服务间是异步集成的，也就是下游的任何处理失败甚至宕机都不会被上游感知，自然也就不会对上游带来影响；
- **CQRS**（Command Query Responsibility Segregation）：把对服务状态有影响的命令用事件来发起，而对服务状态没有影响的查询才使用同步调用的API接口；结合事件驱动架构中的事件溯源（Event Sourcing）可以用于维护数据变更的一致性，当需要重新构建服务状态时，把事件驱动架构中的事件重新“播放”一遍即可；
- **数据变化通知**：在服务架构下，往往一个服务中的数据发生变化，另外的服务会感兴趣，比如用户订单完成后，积分服务、信用服务等都需要得到事件通知并更新用户积分和信用等级；
- **构建开放式接口**：在事件驱动架构下，事件的提供者并不用关心有哪些订阅者，不像服务调用的场景——数据的产生者需要知道数据的消费者在哪里并调用它，因此保持了接口的开放性；
- **事件流处理**：应用于大量事件流（而非离散事件）的数据分析场景，典型应用是基于Kafka的日志处理；
- **基于事件触发的响应**：在IoT时代大量传感器产生的数据，不会像人机交互一样需要等待处理结果的返回，天然适合用事件驱动架构来构建数据处理应用。

事件溯源

查询职责分离

复杂事件处理

扩展性立方体

AFK Scalability Cube

编排与协同

Graal VM

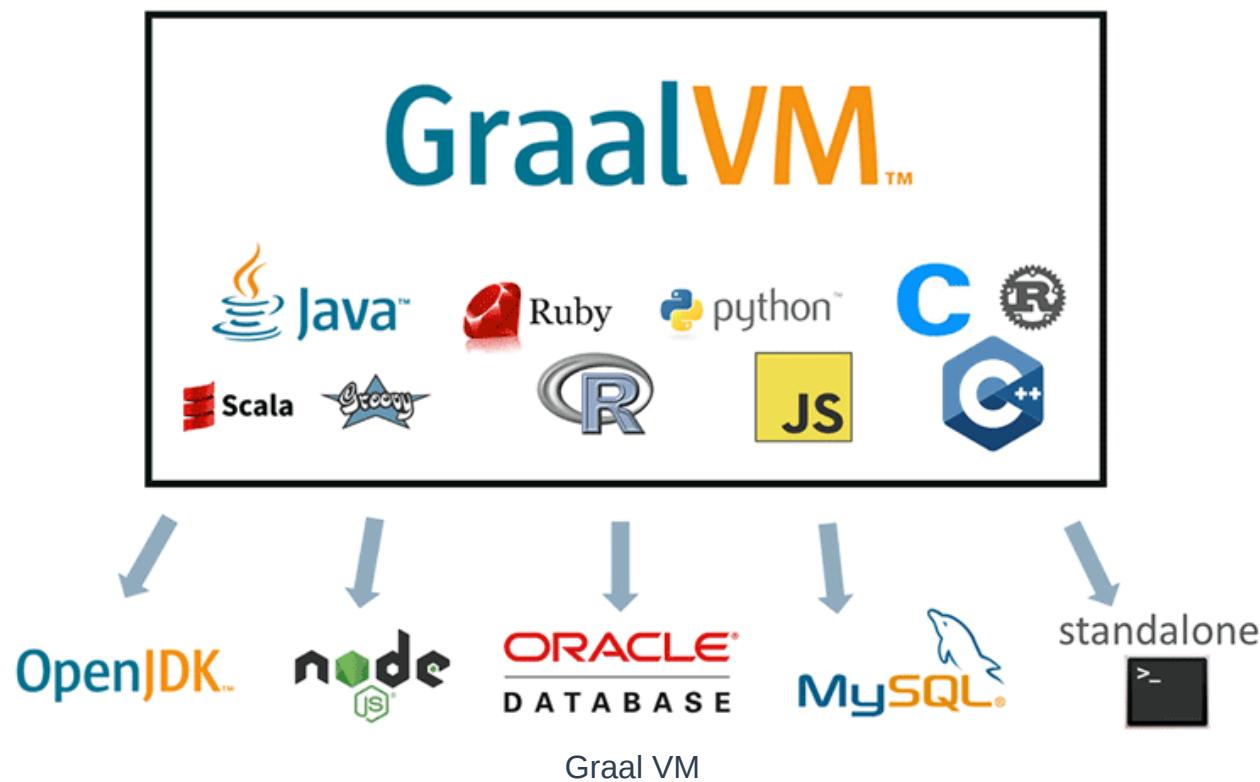
视频公开课

本节笔者有公开课介绍：《GraalVM：云原生时代的Java》

网上每隔一段时间就能见到几条“未来X语言将会取代Java”的新闻，此处“X”可以用Kotlin、Golang、Dart、JavaScript、Python……等各种编程语言来代入。这大概就是长期占据**编程语言榜单**第一位的烦恼，天下第一总避免不了挑战者相伴。

如果Java有拟人化的思维，它应该从来没有惧怕过被哪一门语言所取代，Java“天下第一”的底气不在于语法多么先进好用，而是来自它庞大的用户群和极其成熟的软件生态，这在朝夕之间难以撼动。不过，既然有那么多新、旧编程语言的兴起躁动，说明必然有其需求动力所在，譬如互联网之于JavaScript、人工智能之于Python，微服务风潮之于Golang等等。大家都清楚不太可能有哪门语言能在每一个领域都尽占优势，Java已是距离这个目标最接近的选项，但若“天下第一”还要百尺竿头更进一步的话，似乎就只能忘掉Java语言本身，踏入无招胜有招的境界。

2018年4月，Oracle Labs新公开了一项黑科技：[Graal VM](#)，从它的口号“Run Programs Faster Anywhere”就能感觉到一颗蓬勃的野心，这句话显然是与1995年Java刚诞生时的“Write Once，Run Anywhere”在遥相呼应。



Graal VM被官方称为“Universal VM”和“Polyglot VM”，这是一个在HotSpot虚拟机基础上增强而成的跨语言全栈虚拟机，可以作为“任何语言”的运行平台使用，这里“任何语言”包括了Java、Scala、Groovy、Kotlin等基于Java虚拟机之上的语言，还包括了C、C++、Rust等基于LLVM的语言，同时支持其他像JavaScript、Ruby、Python和R语言等等。Graal VM可以无额外开销地混合使用这些编程语言，支持不同语言中混用对方的接口和对象，也能够支持这些语言使用已经编写好的本地库文件。

Graal VM的基本工作原理是将这些语言的源代码（例如JavaScript）或源代码编译后的中间格式（例如LLVM字节码）通过解释器转换为能被Graal VM接受的中间表示¹（Intermediate Representation，IR），譬如设计一个解释器专门对LLVM输出的字节码进行转换来支持C和C++语言，这个过程称为“程序特化²”（Specialized，也常称为Partial Evaluation）。Graal VM提供了Truffle工具集³来快速构建面向一种新语言的解释器，并用它构建了一个称为Sulong⁴的高性能LLVM字节码解释器。

以更严格的角度来看，Graal VM才是真正意义上与物理计算机相对应的高级语言虚拟机，理由是它与物理硬件的指令集一样，做到了只与机器特性相关而不与某种高级语言特性相关。Oracle Labs的研究总监Thomas Wuerthinger在接受InfoQ采访⁵时谈到：“随着Graal VM 1.0的发布，我们已经证明了拥有高性能的多语言虚拟机是可能的，并且实现这个目标的最佳方式不是通过类似Java虚拟机和微软CLR那样带有语言特性的字节码”。对于一些本来就不以速度见长的语言运行环境，由于Graal VM本身能够对输入的中间表示进行自动优

化，在运行时还能进行即时编译优化，往往使用Graal VM实现能够获得比原生编译器更优秀的执行效率，譬如Graal.js要优于Node.js、Graal.Python要优于CPython，TruffleRuby要优于Ruby MRI，FastR要优于R语言等等。

针对Java而言，Graal VM本来就是在HotSpot基础上诞生的，天生就可作为一套完整的符合Java SE 8标准Java虚拟机来使用。它和标准的HotSpot差异主要在即时编译器上，其执行效率、编译质量目前与标准版的HotSpot相比也是互有胜负。但现在Oracle Labs和美国大学里面的研究院所做的最新即时编译技术的研究全部都迁移至基于Graal VM之上进行了，其发展潜力令人期待。如果Java语言或者HotSpot虚拟机真的有被取代的一天，那从现在看来Graal VM是希望最大的一个候选项，这场革命很可能会在Java使用者没有明显感觉的情况下悄然而来，Java世界所有的软件生态都没有发生丝毫变化，但天下第一的位置已经悄然更迭。

新一代即时编译器

对需要长时间运行的应用来说，由于经过充分预热，热点代码会被HotSpot的探测机制准确定位捕获，并将其编译为物理硬件可直接执行的机器码，在这类应用中Java的运行效率很大程度上是取决于即时编译器所输出的代码质量。

HotSpot虚拟机中包含有两个即时编译器，分别是编译时间较短但输出代码优化程度较低的客户端编译器（简称为C1）以及编译耗时长但输出代码优化质量也更高的服务端编译器（简称为C2），通常它们会在分层编译机制下与解释器互相配合来共同构成HotSpot虚拟机的执行子系统的。

自JDK 10起，HotSpot中又加入了一个全新的即时编译器：Graal编译器，看名字就可以联想到它是来自于前一节提到的Graal VM。Graal编译器是作为C2编译器替代者的角色登场的。C2的历史已经非常长了，可以追溯到Cliff Click大神读博士期间的作品，这个由C++写成的编译器尽管目前依然效果拔群，但已经复杂到连Cliff Click本人都不愿意继续维护的程度。而Graal编译器本身就是由Java语言写成，实现时又刻意与C2采用了同一种名为“Sea-of-Nodes”的高级中间表示（High IR）形式，使其能够更容易借鉴C2的优点。Graal编译器比C2编译器晚了足足二十年面世，有着极其充沛的后发优势，在保持能输出相近质量的编译代码的同时，开发效率和扩展性上都要显著优于C2编译器，这决定了C2编译器中优秀的代码优化技术可以轻易地移植到Graal编译器上，但是反过来Graal编译器中行之有效的优化在C2编译器里实现起来则异常艰难。这种情况下，Graal的编译效果短短几年间迅速追平了C2，甚至某些测试项中开始逐渐反超C2编译器。Graal能够做比C2更加复杂的优化，如“[部分逃逸分析](#)”（Partial Escape Analysis），也拥有比C2更容易使用“[激进预测性优化](#)”（Aggressive Speculative Optimization）的策略，支持自定义的预测性假设等等。

今天的Graal编译器尚且年幼，还未经过足够多的实践验证，所以仍然带着“实验状态”的标签，需要用开关参数去激活，这让笔者不禁联想起JDK 1.3时代，HotSpot虚拟机刚刚横空出世时的场景，同样也是需要用开关激活，也是作为Classic虚拟机的替代品的一段历史。

Graal编译器未来的前途可期，作为Java虚拟机执行代码的最新引擎，它的持续改进，会同时为HotSpot与Graal VM注入更快更强的驱动力。

向原生迈进

对不需要长时间运行的，或者小型化的应用而言，Java（而不是指Java ME）天生就带有一些劣势，这里并不光是指跑个HelloWorld也需要百多兆的JRE之类的问题，而更重要的是指近几年从大型单体应用架构向小型微服务应用架构发展的技术潮流下，Java表现出来的不适应。

在微服务架构的视角下，应用拆分后，单个微服务很可能就不再需要再面对数十、数百GB乃至TB的内存，有了高可用的服务集群，也无须追求单个服务要7×24小时不可间断地运行，它们随时可以中断和更新；但相应地，Java的启动时间相对较长、需要预热才能达到最高性能等特点就显得相悖于这样的应用场景。在无服务架构中，矛盾则可能会更加突出，比起服务，一个函数的规模通常会更小，执行时间会更短，当前最热门的无服务运行环境AWS Lambda所允许的最长运行时间仅有15分钟。

一直把软件服务作为重点领域的Java自然不可能对此视而不见，在最新的几个JDK版本的功能清单中，已经陆续推出了跨进程的、可以面向用户程序的类型信息共享（Application Class Data Sharing，AppCDS，允许把加载解析后的类型信息缓存起来，从而提升下次启动速度，原本CDS只支持Java标准库，在JDK 10时的AppCDS开始支持用户的程序代码）、无操作的垃圾收集器（Epsilon，只做内存分配而不做回收的收集器，对于运行完就退出的应用十分合适）等改善措施。而酝酿中的一个更彻底的解决方案，是逐步开始对提前编译（Ahead of Time Compilation，AOT）提供支持。

提前编译是相对于即时编译的概念，提前编译能带来的最大好处是Java虚拟机加载这些已经预编译成二进制库之后就能够直接调用，而无须再等待即时编译器在运行时将其编译成二进制机器码。理论上，提前编译可以减少即时编译带来的预热时间，减少Java应用长期给人带来的“第一次运行慢”不良体验，可以放心地进行很多全程序的分析行为，可以使用时间压力更大的优化措施。

但是提前编译的坏处也很明显，它破坏了Java“一次编写，到处运行”的承诺，必须为每个不同的硬件、操作系统去编译对应的发行包。也显著降低了Java链接过程的动态性，必须

要求加载的代码在编译期就是全部已知的，而不能再是运行期才确定，否则就只能舍弃掉已经提前编译好的版本，退回到原来的即时编译执行状态。

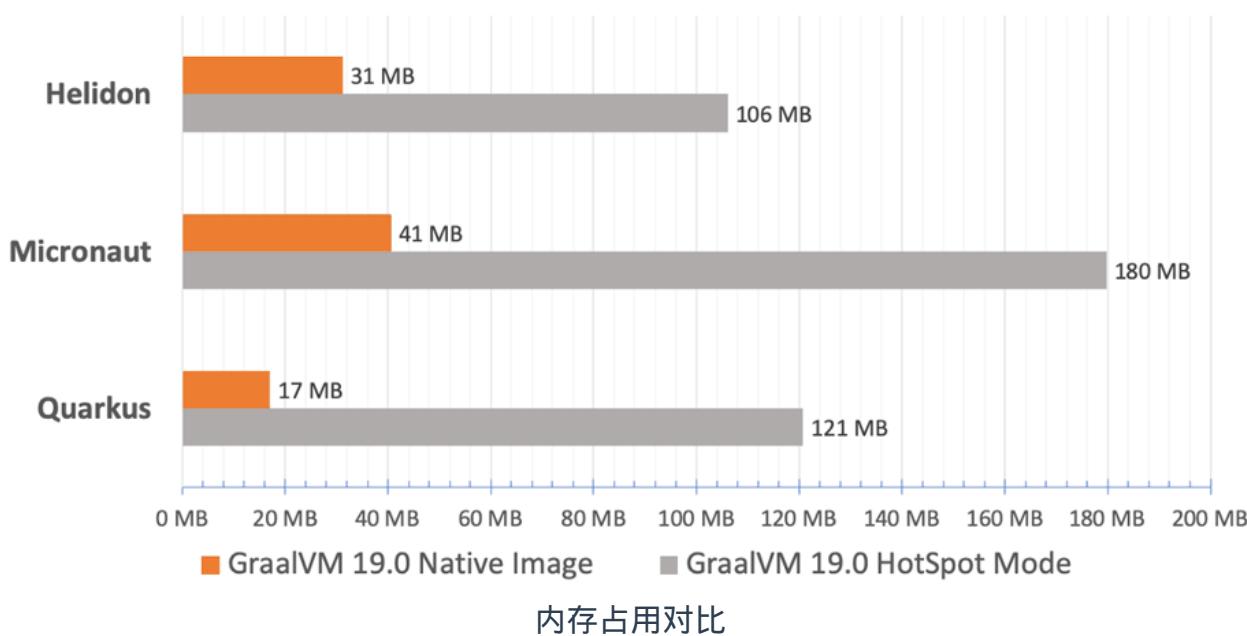
早在JDK 9时期，Java 就提供了实验性的Jaotc命令来进行提前编译，不过多数人试用过后都颇感失望，大家原本期望的是类似于Excelsior JET那样的编译过后能生成本地代码完全脱离Java虚拟机运行的解决方案，但Jaotc其实仅仅是代替掉即时编译的一部分作用而已，仍需要运行于HotSpot之上。

直到Substrate VM¹出现，才算是满足了人们心中对Java提前编译的全部期待。Substrate VM是在Graal VM 0.20版本里新出现的一个极小型的运行时环境，包括了独立的异常处理、同步调度、线程管理、内存管理（垃圾收集）和JNI访问等组件，目标是代替HotSpot用来支持提前编译后的程序执行。它还包含了一个本地镜像的构造器（Native Image Generator）用于为用户程序建立基于Substrate VM的本地运行时镜像。这个构造器采用指针分析（Points-To Analysis）技术，从用户提供的程序入口出发，搜索所有可达的代码。在搜索的同时，它还将执行初始化代码，并在最终生成可执行文件时，将已初始化的堆保存至一个堆快照之中。这样一来，Substrate VM就可以直接从目标程序开始运行，而无须重复进行Java虚拟机的初始化过程。但相应地，原理上也决定了Substrate VM必须要求目标程序是完全封闭的，即不能动态加载其他编译期不可知的代码和类库。基于这个假设，Substrate VM才能探索整个编译空间，并通过静态分析推算出所有虚方法调用的目标方法。

Substrate VM带来的好处是能显著降低了内存占用及启动时间，由于HotSpot本身就会有一定的内存消耗（通常约几十MB），这对最低也从几GB内存起步的大型单体应用来说并不算什么，但在微服务下就是一笔不可忽视的成本。根据Oracle官方给出的测试数据²，运行在Substrate VM上的小规模应用，其内存占用和启动时间与运行在HotSpot相比有了5倍到50倍的下降，具体结果如下图所示：

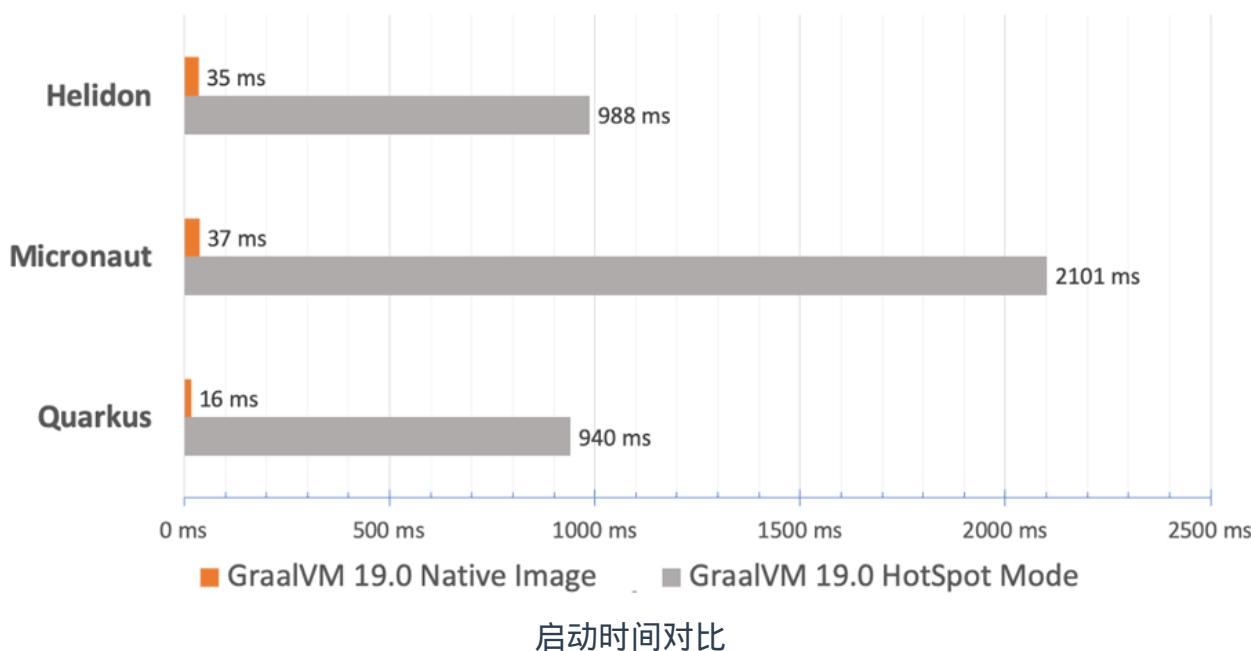
Java Microservice: Memory Footprint

~5x lower



Java Microservice: Startup Time

~50x faster



Substrate VM补全了Graal VM“Run Programs Faster Anywhere”愿景蓝图里最后的一块拼图，让Graal VM支持其他语言时不会有重量级的运行负担。譬如运行JavaScript代码，Node.js的V8引擎执行效率非常高，但即使是最简单的HelloWorld，它也要使用约20MB的内存，而运行在Substrate VM上的Graal.js，跑一个HelloWorld则只需要4.2MB内存而已，且运行速度与V8持平。Substrate VM 的轻量特性，使得它十分适合于嵌入至其他系统之中，譬如Oracle自家的数据库就已经开始使用这种方式支持用不同的语言代替PL/SQL来编写存储过程。

没有虚拟机的Java

尽管Java已经看清楚了在微服务时代的前进目标，但是，Java语言和生态在微服务、微应用环境中的天生的劣势并不会一蹴而就地被解决，通往这个目标的道路注定会充满荆棘；尽管已经有了放弃“一次编写，到处运行”、放弃语言动态性的思想准备，但是，这些特性并不单纯是宣传口号，它们在Java语言诞生之初就被植入到基因之中，当Graal VM试图打破这些规则的同时，也受到了Java语言和在其之上的生态生态的强烈反噬，笔者选择其中最主要的一些困列举如下：

- 某些Java语言的特性，使得Graal VM编译本地镜像的过程变得极为艰难。譬如常见的反射，除非使用[安全管理器](#)去专门进行认证许可，否则反射机制具有在运行期动态调用几乎所有API接口的能力，且具体会调用哪些接口，在程序不会真正运行起来的编译期是无法获知的。反射显然是Java不能放弃不能妥协的重要特性，为此，只能由程序的开发者明确地告知Graal VM有哪些代码可能被反射调用（通过JSON配置文件的形式），Graal VM才能在编译本地程序时将它们囊括进来。

```
[  
  {  
    name: "com.github.fenixsoft.SomeClass",  
    allDeclaredConstructors: true,  
    allPublicMethods: true  
  },  
  {  
    name: "com.github.fenixsoft.AnotherClass",  
    fields: [{name: "foo"}, {name: "bar"}],  
    methods: [  
      {  
        name: "<init>",  
        parameterTypes: ["char[]"]  
      }]  
  },  
  // something else .....  
]
```

这是一种可操作性极其低下却又无可奈何的解决方案，即使开发者接受不厌其烦地列举出自己代码中所用到的反射API，但他们又如何能保证程序所引用的其他类库的反射行为都已全部被获知，其中没有任何遗漏？与此类似的还有另外一些语言特性，如动态代理等。另外，一切非代码性质的资源，如最典型的配置文件等，也都必须明确加入配置中才能被Graal VM编译打包。这导致了如果没有专门的工具去协助，使用Graal VM编译Java的遗留系统即使理论可行，实际操作也将是极度的繁琐。

- 大多数运行期对字节码的生成和修改操作，在Graal VM看来都是无法接受的，因为Substrate VM里面不再包含即时编译器和字节码执行引擎，所以一切可能被运行的字节码，都必须经过AOT编译成为原生代码。请不要觉得运行期直接生成字节码会很罕见，误以为导致的影响应该不算很大。事实上，多数实际用于生产的Java系统都或直接或间接、或多或少引用了ASM、CGLIB、Javassist这类字节码库。举个例子，CGLIB是通过运行时产生字节码（生成代理类的子类）来做动态代理的，长期以来这都是Java世界里进行类增强的主流形式，因为面向接口的增强可以使用JDK自带的动态代理，但对类的增强则并没有多少选择的余地。CGLIB也是Spring用来做类增强的选择，但Graal VM明确表示是不可能支持CGLIB的，因此，这点就必须由用户（面向接口编程）、框架（Spring这些DI框架放弃CGLIB增强）和Graal VM（起码得支持JDK的动态代理，留条活路可走）来共同解决。自Spring Framework 5.2起，@Configuration注解中加入了一个新的proxyBeanMethods参数，设置为false则可避免Spring对与非接口类型的Bean进行代理。同样地，对应在Spring Boot 2.2中，@SpringBootApplication注解也增加了proxyBeanMethods参数，通常采用Graal VM去构建的Spring Boot本地应用都需要设置该参数。
- 一切HotSpot虚拟机本身的内部接口，譬如JVMTI、JVMCI等，在都将不复存在了——在本地镜像中，连HotSpot本身都被消灭了，这些接口自然成了无根之木。这对使用者一侧的最大影响是再也无法进行Java语言层次的远程调试了，最多只能进行汇编层次的调试。在生产系统中一般也没有人这样做，开发环境就没必要采用Graal VM编译，这点的实际影响并不算大。
- Graal VM放弃了一部分可以妥协的语言和平台层面的特性，譬如Finalizer、安全管理器、InvokeDynamic指令和MethodHandles，等等，在Graal VM中都被声明为不支持的，这些妥协的内容大多倒并非全然无法解决，主要是基于工作量性价比的原因。能够被放弃的语言特性，说明确实是影响范围非常小的，所以这个对使用者来说一般是可以接受的。

-

以上，是Graal VM在Java语言中面临的部分困难，在整个Java的生态系统中，数量庞大的第三方库才是真正最棘手的难题。可以预料，这些第三方库一旦脱离了Java虚拟机，在原生环境中肯定会暴露出无数千奇百怪的异常行为。Graal VM团队对此的态度非常务实，并没有直接硬啃。要建设可持续、可维护的Graal VM，就不能为了兼容现有JVM生态，做出过多的会影响性能、优化空间和未来拓展的妥协牺牲，为此，应该也只能反过来由Java生态去适应Graal VM，这是Graal VM团队明确传递出对第三方库的态度：

3rd party libraries

Graal VM native support needs to be sustainable and maintainable, that's why we do not want to maintain fragile patches for the whole JVM ecosystem.

The ecosystem of libraries needs to support it natively.

—— Sébastien Deleuze , DEVOXX 2019 ↗

为了推进Java生态向Graal VM兼容，Graal VM主动拉拢了Java生态中最庞大的一个派系：Spring。从2018年起，来自Oracle的Graal VM团队与来自Pivotal的Spring团队已经紧密合作了很长的一段时间，共同创建了[Spring Graal Native](#)项目来解决Spring全家桶在Graal VM上的运行适配问题，在不久的将来（预计应该是2020年10月左右），下一个大的Spring版本（Spring Framework 5.3、Spring Boot 2.3）的其中一项主要改进就是能够开箱即用地支持Graal VM，这样，用于微服务环境的Spring Cloud便会获得不受Java虚拟机束缚的更广阔舞台空间。

Spring over Graal

前面几部分，我们以定性的角度分析了Graal VM诞生的背景与它的价值，在最后这部分，我们尝试进行一些实践和定量的讨论，介绍具体如何使用Graal VM之余，也希望能以更加量化的角度去理解程序运行在Graal VM之上，会有哪些具体的收益和代价。

尽管需要到2020年10月正式发布之后，Spring对Graal VM的支持才会正式提供，但现在的我们其实已经可以使用Graal VM来（实验性地）运行Spring、Spring Boot、Spring Data、Netty、JPA等等的一系列组件（不过SpringCloud中的组件暂时还不行）。接下来，我们将尝试使用Graal VM来编译一个标准的Spring Boot应用：

- **环境准备：**

- 安装Graal VM，你可以选择直接[下载](#)安装（版本选择Graal VM CE 20.0.0），然后配置好PATH和JAVA_HOME环境变量即可；也可以选择使用[SDKMAN](#)来快速切换环境。个人推荐后者，毕竟目前还不适合长期基于Graal VM环境下工作，经常手工切换会很麻烦。

```
# 安装SDKMAN  
$ curl -s "https://get.sdkman.io" | bash  
  
# 安装Graal VM  
$ sdk install java 20.0.0.r8-grl
```

- 安装本地镜像编译依赖的LLVM工具链。

```
# gu命令来源于Graal VM的bin目录  
$ gu install native-image
```

请注意，这里已经假设你机器上已有基础的GCC编译环境，即已安装过build-essential、libz-dev等套件。没有的话请先行安装。对于Windows环境来说，这步是需要Windows SDK 7.1中的C++编译环境来支持。我个人并不建议在Windows上进行Java应

用的本地化操作，如果说在Linux中编译一个本地镜像，通常是为了打包到Docker，然后发布到服务器中使用。那在Windows上编译一个本地镜像，你打算用它来干什么呢？

- **编译准备：**

- 首先，我们先假设你准备编译的代码是“符合要求”的，即没有使用到Graal VM不支持的特性，譬如前面提到的Finalizer、CGLIB、InvokeDynamic这类功能。然后，由于我们用的是Graal VM的Java 8版本，也必须假设你编译使用Java语言级别在Java 8以内。
- 然后，我们需要用到尚未正式对外发布的Spring Boot 2.3，目前最新的版本是Spring Boot 2.3.0.M4。请将你的pom.xml中的Spring Boot版本修改如下（假设你编译用的是Maven，用Gradle的请自行调整）：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.0.M4</version>
    <relativePath/>
</parent>
```

由于是未发布的Spring Boot版本，所以它在Maven的中央仓库中是找不到的，需要手动加入Spring的私有仓库，如下所示：

```
<repositories>
    <repository>
        <id>spring-milestone</id>
        <name>Spring milestone</name>
        <url>https://repo.spring.io/milestone</url>
    </repository>
</repositories>
```

- 最后，尽管我们可以通过命令行（使用native-image命令）来直接进行编译，这对于没有什么依赖的普通Jar包、写一个Helloworld来说都是可行的，但对于Spring Boot，光是在命令行中写Classpath上都忙活一阵的，建议还是使用[Maven插件](#)来驱动Graal VM编译，这个插件能够根据Maven的依赖信息自动组织好Classpath，你只需

要填其他命令行参数就行了。因为并不是每次编译都需要构建一次本地镜像，为了不干扰使用普通Java虚拟机的编译，建议在Maven中独立建一个Profile来调用Graal VM插件，具体如下所示：

```
<profiles>
  <profile>
    <id>graal</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.graalvm.nativeimage</groupId>
          <artifactId>native-image-maven-plugin</artifactId>
          <version>20.0.0</version>
          <configuration>
            <buildArgs>-Dspring.graal.remove-unused-autoconfig=true
--no-fallback -H:+ReportExceptionStackTraces --no-
server</buildArgs>
          </configuration>
          <executions>
            <execution>
              <goals>
                <goal>native-image</goal>
              </goals>
              <phase>package</phase>
            </execution>
          </executions>
        </plugin>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

这个插件同样在Maven中央仓库中不存在，所以也得加上前面Spring的私有库：

```
<pluginRepositories>
  <pluginRepository>
    <id>spring-milestone</id>
```

```
<name>Spring milestone</name>
<url>https://repo.spring.io/milestone</url>
</pluginRepository>
</pluginRepositories>
```

至此，编译环境的准备顺利完成。

- 程序调整：

- 首先，前面提到了Graal VM不支持CGLIB，只能使用JDK动态代理，所以应当把Spring对普通类的Bean增强给关闭掉：

```
@SpringBootApplication(proxyBeanMethods = false)
public class ExampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(ExampleApplication.class, args);
    }

}
```

- 然后，这是最麻烦的一个步骤，你程序里反射调用过哪些API、用到哪些资源、动态代理，还有哪些类型需要在编译期初始化的，都必须使用JSON配置文件逐一告知Graal VM。前面也说过了，这事情只有理论上的可行性，实际做起来完全不可操作。Graal VM的开发团队当然也清楚这一点，所以这个步骤实际的处理途径有两种，第一种是假设你依赖的第三方包，全部都在Jar包中内置了以上编译所需的配置信息，这样你只要提供你程序里用户代码中用到的配置即可，如果你程序里没写过反射、没用过动态代理什么的，那就什么配置都无需提供。第二种途径是Graal VM计划提供一个Native Image Agent的代理，只要将它挂载在在程序中，以普通Java虚拟机运行一遍，把所有可能的代码路径都操作覆盖到，这个Agent就能自动帮你根据程序实际运行情况来生成编译所需要的配置，这样无论是你自己的代码还是第三方的代码，都不需要做预先的配置。目前，第二种方式中的Agent尚未正式发布，只有方式一是可用的。幸好，Spring与Graal VM共同维护的在[Spring Graal Native](#)项目已经提供了大多数Spring Boot组件的配置信息（以及一些需要在代码层面处理的Patch），我们只需要简单依赖该工程即可。

```
<dependencies>
    <dependency>
        <groupId>org.springframework.experimental</groupId>
        <artifactId>spring-graal-native</artifactId>
        <version>0.6.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-indexer</artifactId>
    </dependency>
</dependencies>
```

另外还有一个小问题，由于目前Spring Boot嵌入的Tomcat中，WebSocket部分在JMX反射上还有一些瑕疵，在[修正该问题的PR](#)被Merge之前，暂时需要手工去除掉这个依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.apache.tomcat.embed</groupId>
                <artifactId>tomcat-embed-websocket</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

- 最后，在Maven中给出程序的启动类的路径：

```
<properties>
    <start-class>com.example.ExampleApplication</start-class>
</properties>
```

- **开始编译：**

- 到此一切准备就绪，通过Maven进行编译：

```
$ mvn -Pgraal clean package
```

sh

编译的结果默认输出在target目录，以启动类的名字命名。

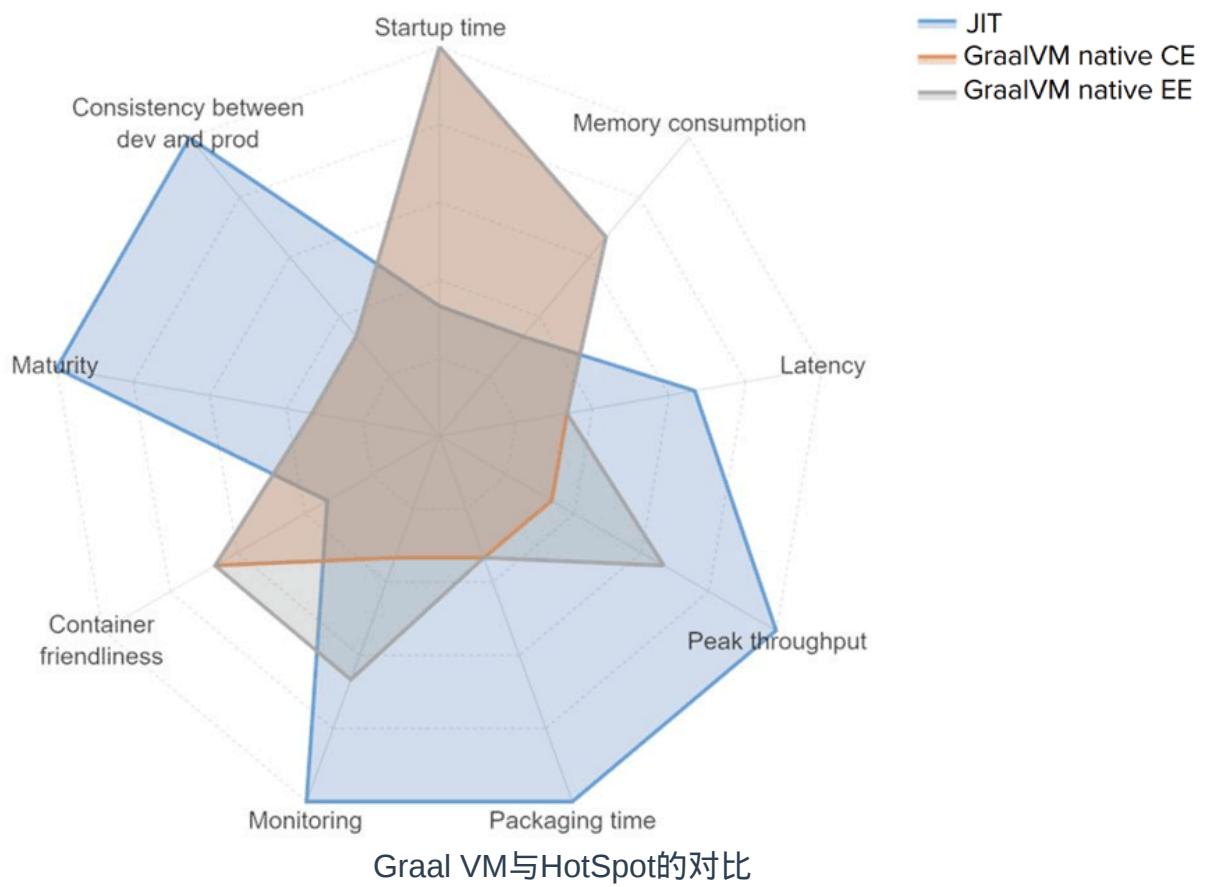
- 因为AOT编译可以放心大胆地进行大量全程序的重负载优化，所以无论是编译时间还是空间占用都非常可观。笔者在intel 9900K、64GB内存的机器上，编译了一个只引用了org.springframework.boot:spring-boot-starter-web的Helloworld类型的工程，大约耗费了两分钟时间。

```
[com.example.exampleapplication:9839]      (typeflow): 22,093.72 ms,  
6.48 GB  
[com.example.exampleapplication:9839]      (objects): 34,528.09 ms,  
6.48 GB  
[com.example.exampleapplication:9839]      (features): 6,488.74 ms,  
6.48 GB  
[com.example.exampleapplication:9839]      analysis: 65,465.65 ms,  
6.48 GB  
[com.example.exampleapplication:9839]      (clinit): 2,135.25 ms,  
6.48 GB  
[com.example.exampleapplication:9839]      universe: 4,449.61 ms,  
6.48 GB  
[com.example.exampleapplication:9839]      (parse): 2,161.78 ms,  
6.32 GB  
[com.example.exampleapplication:9839]      (inline): 3,113.77 ms,  
6.25 GB  
[com.example.exampleapplication:9839]      (compile): 15,892.88 ms,  
6.56 GB  
[com.example.exampleapplication:9839]      compile: 25,044.34 ms,  
6.56 GB  
[com.example.exampleapplication:9839]      image: 6,580.71 ms,  
6.63 GB  
[com.example.exampleapplication:9839]      write: 1,362.73 ms,  
6.63 GB  
[com.example.exampleapplication:9839]      [total]: 120,410.26 ms,  
6.63 GB  
[INFO]  
[INFO] --- spring-boot-maven-plugin:2.3.0.M4:repackage (repackage)  
@ exampleapplication ---  
[INFO] Replacing main artifact with repackaged archive  
[INFO] -----  
-----
```

```
[INFO] BUILD SUCCESS
[INFO] -----
-----
[INFO] Total time: 02:08 min
[INFO] Finished at: 2020-04-25T22:18:14+08:00
[INFO] Final Memory: 38M/599M
[INFO] -----
-----
```

- 效果评估：

- 笔者使用Graal VM编译一个最简单的Helloworld程序（就只在控制台输出个Helloworld，什么都不依赖），最终输出的结果大约3.6MB，启动时间能低至2ms左右。如果用这个程序去生成Docker镜像（不基于任何基础镜像，即使用FROM scratch打包），产生的镜像还不到3.8MB。而OpenJDK官方提供的Docker镜像，即使是slim版，其大小也在200MB到300MB之间。
- 使用Graal VM编译一个简单的Spring Boot Web应用，仅导入Spring Boot的Web Starter的依赖的话，编译结果有77MB，原始的Fat Jar包大约是16MB，这样打包出来的Docker镜像可以不依赖任何基础镜像，大小仍然是78MB左右（实际使用时最好至少也要基于alpine吧，不差那几MB）。相比起空间上的收益，启动时间上的改进是更主要的，Graal VM的本地镜像启动时间比起基于虚拟机的启动时间有着绝对的优势，一个普通Spring Boot的Web应用启动一般2、3秒之间，而本地镜像只要100毫秒左右即可完成启动，这确实有了数量级的差距。
- 不过，必须客观地说明一点，尽管Graal VM在启动时间、空间占用、内存消耗等容器化环境中比较看重的方面确实比HotSpot有明显的改进，尽管Graal VM可以放心大胆地使用重负载的优化手段，但如果是处于长时间运行这个前提下，至少到目前为止，没有任何迹象表明它能够超越经过充分预热后的HotSpot。在延迟、吞吐量、可监控性等方面，仍然是HotSpot占据较大优势，下图引用了DEVOXX 2019中Graal VM团队自己给出的Graal VM与HotSpot JIT在各个方面的对比评估：



Graal VM团队同时也说了，Graal VM有望在2020年之内，在延迟和吞吐量这些关键指标上追评HotSpot现在的表现。Graal VM毕竟是一个2018年才正式公布的新生事物，我们能看到它这两三年间在可用性、易用性和性能上持续地改进，Graal VM有望成为Java在微服务时代里的最重要的基础设施变革者，这项改进的结果如何，甚至可能与Java的前途命运息息相关。

部署环境

这一部分介绍了Docker容器环境和Kubernetes集群环境的Step-By-Step的安装流程。加入开发与运维的环境依赖这些内容，严格来讲并不符合本文档的主题，但对于刚刚接触这个领域的读者，这些内容确实有一定的复杂性，为避免新人受到不必要的打击，笔者还是在附录中加入这部分的内容，如对这部分已有了解的读者，完全可以略过。

部署Docker CE容器环境

本文为Linux系统安装Docker容器环境的简要说明，主要包括：

1. 安装稳定最新发行版（Stable Release）的命令及含义。
2. 针对国内网络环境的必要镜像加速或者代理设置工作。

若需了解Docker安装其他方面的内容，如安装Nightly/Test版本、Backporting、软件版权和支持等信息，可参考官方的部署指南：<https://docs.docker.com/install/>

文中涉及到的Debian系和Redhat系的包管理工具，主要包括：

- Debian系：Debian、Ubuntu、Deepin、Mint
- Redhat系：RHEL、Fedora、CentOS

如用的其他Linux发行版，如Gentoo、Archlinux、OpenSUSE等，建议自行安装二进制包。

移除旧版本Docker

如果以前已经安装过旧版本的Docker（可能会被称为docker，docker.io 或 docker-engine），需先行卸载。

Debian系：

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc  
docker-ce docker-ce-cli containerd.io
```

sh

RedHat系：

```
$ sudo yum remove docker \  
docker-client \  
docker-client-latest \  
dockerd
```

sh

```
docker-common \
docker-latest \
docker-latest-logrotate \
docker-logrotate \
docker-engine
```

安装Docker依赖工具链及软件源

在Debian上主要是为了apt能够正确使用HTTPS协议，并将Docker官方的GPG Key (GNU Privacy Guard，包的签名机制) 和软件源地址注册到软件源中。

在RHEL上是为了devicemapper获得yum-config-manager、device-mapper-persistent-data、lvm2的支持。

Debian系：

```
$ sudo apt-get install apt-transport-https \
ca-certificates \
curl \
software-properties-common

# 注册Docker官方GPG公钥
$ sudo curl -fsSL https://download.docker.com/linux/debian/gpg | sudo
apt-key add -

# 检查Docker官方GPG公钥指纹是否正确
$ sudo apt-key fingerprint 0EBFCD88

pub    4096R/0EBFCD88 2017-02-22
      Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF
CD88
uid            Docker Release (CE deb) <docker@docker.com>
sub    4096R/F273FCD8 2017-02-22

# 将Docker地址注册到软件源中
# 注意$(lsb_release -cs)是返回当前发行版的版本代号，例如Ubuntu 18.04是bionic，19.10是eoan
# 但在Ubuntu 19.10发布一段时间后，Docker官方并未在源地址中增加eoan目录，导致此命令安装失败，日后在最新的系统上安装Docker，需要注意排查此问题，手动更改版本代号完成安装
$ sudo add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
```

```
$(lsb_release -cs) \
stable"
```

RedHat系：

```
$ sudo yum install -y yum-utils \
device-mapper-persistent-data \
lvm2

# 将Docker地址注册到软件源中
$ sudo yum-config-manager \
--add-repo \
https://download.docker.com/linux/centos/docker-ce.repo
```

sh

更新系统软件仓库

Debian系：

```
$ sudo apt-get update
```

sh

RedHat系：

```
$ sudo yum update
```

sh

安装Docker-Engine Community

Debian系：

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

sh

RedHat系：

```
$ sudo yum install docker-ce docker-ce-cli containerd.io
```

sh

确认Docker安装是否成功

直接运行官方的hello-world镜像测试安装是否成功

```
$ sudo docker run hello-world
```

sh

配置国内镜像库 可选

由于Docker官方镜像在国内访问缓慢，官方提供了在国内的镜像库：<https://registry.docker-cn.com>，以加快访问速度（但其实体验也并不快）。

```
# 该配置文件及目录，在Docker安装后并不会自动创建
$ sudo mkdir -p /etc/docker

# 配置加速地址
$ sudo tee /etc/docker/daemon.json <<- 'EOF'
{
    "registry-mirrors": ["https://registry.docker-cn.com"]
}
EOF

# 重启服务
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
```

sh

注意

以上操作有两点提醒读者重点关注：

1. 必须保证daemon.json文件中完全符合JSON格式，如果错了，Docker不会给提示，直接起来。
2. 如果Docker是作为systemd管理的服务的，daemon.json文件会处于锁定状态，应先关闭后再修改配置；

这两点出了问题都会导致Docker服务直接无法启动，如果出现该情况，可以通过systemd status命令检查，看是否有类似如下的错误提示：

```
Drop-In: /etc/systemd/system/docker.service.d
          └─mirror.conf
Active: inactive (dead) (Result: exit-code) since 五 2017-09-15
13:25:28 CST; 7min ago
  Docs: https://docs.docker.com
Main PID: 21151 (code=exited, status=1/FAILURE)
```

如果是，修改daemon.json后重新启动即可。另，关闭systemd服务的方法是：

```
$ sudo systemctl stop docker
$ sudo rm -rf /etc/systemd/system/docker.service.d
```

最后，Docker的官方国内镜像库的速度只能说比起访问国外好了一丢丢，聊胜于无。国内还有一些公开的镜像库，如微软的、网易的等，但要么是不稳定，要么也是慢。比较靠谱的是阿里云的镜像库，但这个服务并不是公开的，需要使用者先到阿里云去申请开发者账户，再使用加速服务，申请后会得一个类似于“<https://yourname.mirror.aliyuncs.com>”的私有地址，把它设置到daemon.json中即可使用。

为Docker设置代理 可选

另外一种解决Docker镜像下载速度慢的方案就是使用代理，Docker的代理可以直接读取系统的全局代理，即系统中的HTTP_PROXY、HTTPS_PROXY两个环境变量。不过，如果设置这两个变量，其他大量Linux下的其他工具也会受到影响，所以建议的方式是给Docker服务设置专有的环境变量，我们使用Systemd来管理Docker服务，那直接给这个服务设置一个额外配置即可，操作如下：

```
sudo mkdir -p /etc/systemd/system/docker.service.d
# 配置代理地址，支持http、https、socks、socks5等协议
$ sudo tee /etc/systemd/system/docker.service.d/http-proxy.conf <<-
'EOF'
[Service]
```

```
Environment="HTTP_PROXY=socks5://192.168.31.125:2012"
EOF

#重启docker
$ sudo systemctl restart docker
```

设置后可以通过systemctl检查一下环境变量，看看是否有设置成功：

```
$ systemctl show --property=Environment docker
```

sh

输出：

```
Environment=HTTP_PROXY=socks5://192.168.31.125:2012
```

sh

开放Docker远程服务

可选

如果需要在其他机器上管理Docker——譬如典型的如在IntelliJ IDEA这类IDE环境中给远程Docker部署镜像，那可以开启Docker的远程管理端口，这步没有设置任何安全访问措施，请不要在生产环境中进行。

具体做法是修改Docker的服务配置：

Debian系：

```
$ sudo vim /lib/systemd/system/docker.service
```

sh

RedHat系：

```
$ sudo vim /usr/lib/systemd/system/docker.service
```

sh

在ExecStart后面增加以下参数（2375端口可以自定义）：

```
-H tcp://0.0.0.0:2375 -H unix://var/run/docker.sock
```

sh

譬如，默认安装完Docker，修改之后完整的ExecStart应当如下所示：

```
sh
ExecStart=/usr/bin/dockerd -H fd:// --
containerd=/run/containerd/containerd.sock -H tcp://0.0.0.0:2375 -H
unix://var/run/docker.sock
```

最后重启Docker服务即可：

```
sh
#重启docker
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
```

启用Docker命令行自动补全功能 可选

在控制台输入docker命令时可以获得自动补全能力，提高效率。

Docker自带了bash的命令行补全，用其他shell，如zsh，则需采用zsh的插件或者自行获取补全信息

bash：

```
sh
$ echo 'source /usr/share/bash-completion/completions/docker' >>
~/.bashrc
```

zsh：

```
sh
$ mkdir -p ~/.zsh/completion
$ curl -L
https://raw.githubusercontent.com/docker/cli/master/contrib/completion/zs
h/_docker > ~/.zsh/completion/_docker

$ echo 'fpath=(~/zsh/completion $fpath)' >> ~/.zshrc
$ echo 'autoload -Uz compinit && compinit -u' >> ~/.zshrc
```

将Docker设置为开机启动

可选

一般使用systemd来管理启动状态

```
# 设置为开机启动  
$ sudo systemctl enable docker  
  
# 立刻启动Docker服务  
$ sudo systemctl start docker
```

sh

安装Docker-Compose

在开发和部署微服务应用时，经常要使用Docker-Compose来组织多个镜像，对于Windows系统它是默认安装的，在Linux下需要另外下载一下，下载后直接扔到bin目录，加上执行权限即可使用

```
# 从GitHub下载  
sudo curl -L  
"https://github.com/docker/compose/releases/download/1.25.5/docker-  
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose  
# 从国内镜像下载  
sudo curl -L  
"https://get.daocloud.io/docker/compose/releases/download/1.25.5/docker-  
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose  
  
sudo chmod +x /usr/local/bin/docker-compose
```

sh

卸载Docker

Debian系：

```
$ sudo apt-get purge docker-ce
```

sh

```
# 清理Docker容器缓存和自定义配置  
$ sudo rm -rf /var/lib/docker
```

RedHat系：

```
$ sudo yum remove docker-ce  
  
# 清理Docker容器缓存和自定义配置  
$ sudo rm -rf /var/lib/docker
```

sh

部署Kubernetes集群

Kubernetes是一个由Google发起的开源自动化部署，缩放，以及容器化管理应用程序的容器编排系统。

部署Kubernetes曾经是一件比较麻烦的事情，kubelet、Api-Server、Etcd、controller-manager等每一个组件都需要自己部署，还要创建自签名证书来保证各个组件之间的网络访问。但程序员大概是最爱偷懒最怕麻烦的群体，随着Kubernetes的后续版本不断改进（如提供了自动生成证书、Api-Server等组件改为默认静态Pod部署方式），使得部署和管理Kubernetes集群正在变得越来越简单。目前主流的方式大致有：

- 使用Kubeadm部署Kubernetes集群
- 使用Rancher部署、管理Kubernetes集群
其他如KubeSphere等在Kubernetes基础上构建的工具均归入此类
- 使用Minikube在本地单节点部署Kubernetes集群
其他如Microk8s等本地环境的工具均归入此类

以上集中部署方式都有很明显的针对性，个人开发环境以Minikube最简单，生产环境以Rancher最简单，在云原生环境中，自然是使用环境提供的相应工具。不过笔者推荐首次接触Kubernetes的同学最好还是选择Kubeadm来部署，毕竟这是官方提供的集群管理工具，是相对更底层、基础的方式，充分熟悉了之后再接触其他简化的方式会快速融会贯通。以上部署方式无需全部阅读，根据自己环境的情况选择其一即可。

使用Kubeadm部署

尽管使用Rancher或者KubeSphere这样更高层次的管理工具，可以更“傻瓜式”地部署和管理Kubernetes集群，但kubeadm作为官方提供的用于快速安装Kubernetes的命令行工具，仍然是应该掌握的基础技能。kubeadm随着新版的Kubernetes同步更新，时效性也会比其他更高层次的管理工具来的更好。

随着Kubernetes不断成熟，kuberadm无论是部署单控制平面（Single Control-Plane，单Master节点）集群还是高可用（High-Availability，多Master节点）集群，都已经有了更优秀的易用性，现在手工部署Kubernetes集群已经不是什么太复杂、困难的事情了。本文以Debian系的Linux为例，介绍通过kuberadm部署集群的全过程。

注意事项

1. 安装Kubernetes集群，需要从谷歌的仓库中拉取镜像，由于国内访问谷歌的网络受阻，需要通过科学上网或者在Docker中预先拉取好所需镜像等方式解决。
2. 集群中每台机器的Hostname不要重复，否则Kubernetes从不同机器收集状态信息时会产生干扰，被认为是同一台机器。
3. 安装Kubernetes最小需要2核CPU、2GB内存，且为x86架构（暂不支持ARM架构）。对于物理机来说，今时今日要找一台不满足以上条件的机器很困难，但对于云主机来说，尤其是购买网站上最低配置的同学，要注意一下是否达到了最低要求，不清楚的话请在/proc/cpuinfo、/proc/meminfo中确认一下。
4. 确保网络通畅的——这听起来像是废话，但确实有相当一部分的云主机默认不对SELinux、iptables、安全组、防火墙进行设置的话，内网各个节点之间、与外网之间会存在访问障碍，导致部署失败。

注册apt软件源

由于Kubernetes并不在主流Debian系统自带的软件源中，所以要手工注册，然后才能使用apt-get安装。

官方的GPG Key地址为：<https://packages.cloud.google.com/apt/doc/apt-key.gpg>，其中包括的软件源的地址为：<https://apt.kubernetes.io/>（该地址最终又会被重定向至：<https://packages.cloud.google.com/apt/>）。如果能访问google.com域名的机器，采用以下方法注册apt软件源是最佳的方式：

```
# 添加GPG Key  
$ sudo curl -fsSL https://packages.cloud.google.com/apt/doc/apt-key.gpg  
| sudo apt-key add -  
  
# 添加K8S软件源  
$ sudo add-apt-repository "deb https://apt.kubernetes.io/ kubernetes-xenial main"
```

sh

对于不能访问google.com的机器，就要借助国内的镜像源来安装了。虽然在这些镜像源中我已遇到过不止一次同步不及时的问题了——就是官方源中已经发布了软件的更新版本，而镜像源中还是旧版的，除了时效性问题外，还出现过其他的一些一致性问题，但是总归比没有的强。国内常见用的apt源有阿里云的、中科大的等，具体为：

阿里云：

- GPG Key：<http://mirrors.aliyun.com/kubernetes/apt/doc/apt-key.gpg>
- 软件源：<http://mirrors.aliyun.com/kubernetes/apt/>

中科大：

- GPG Key：https://raw.githubusercontent.com/EagleChen/kubernetes_init/master/kube_apt_key.gpg
- 软件源：<http://mirrors.ustc.edu.cn/kubernetes/apt/>

它们的使用方式与官方源注册过程是一样的，只需替换里面的GPG Key和软件源的URL地址即可，譬如阿里云：

```
# 添加GPG Key  
$ curl -fsSL http://mirrors.aliyun.com/kubernetes/apt/doc/apt-key.gpg |  
sudo apt-key add -  
  
# 添加K8S软件源
```

sh

```
$ sudo add-apt-repository "deb http://mirrors.aliyun.com/kubernetes/apt kubernetes-xenial main"
```

添加源后记得执行一次更新：

```
$ sudo apt-get update
```

sh

安装kubelet、kubectl、kubeadm

其实并不需要在每个节点都装上kubectl，但是，我缺的是哪点磁盘空间？

下面简要列出了这三个工具/组件的作用，现在看不看得懂都没有关系，以后用到它们的机会多得是，要相信日久总会生情的。

- kubeadm: 引导启动Kubernetes集群的命令行工具。
- kubelet: 在群集中的所有计算机上运行的组件，并用来执行如启动pods和containers等操作。
- kubectl: 用于操作运行中的集群的命令行工具。

```
$ sudo apt-get install kubelet kubeadm kubectl
```

sh

初始化集群前的准备

在使用kubeadm初始化集群之前，还有一些必须的前置工作要妥善处理：

首先，基于安全性（如在文档中承诺的Secret只会在内存中读写）、利于保证节点同步一致性等原因，从1.8版开始，Kubernetes就在它的文档中明确声明了它**默认不支持Swap分区**，在未关闭Swap分区时，集群将直接无法启动。关闭Swap的命令为：

```
$ sudo swapoff -a
```

sh

上面这个命令是一次性的，只在当前这次启动中生效，要彻底关闭Swap分区，需要在文件系统分区表的配置文件中去直接除掉Swap分区。使用vim打开/etc/fstab，注释其中带有sw

ap的行即可，或使用以下命令直接完成修改：

```
# 还是先备份一下  
$ yes | sudo cp /etc/fstab /etc/fstab_bak  
  
# 进行修改  
$ sudo cat /etc/fstab_bak | grep -v swap > /etc/fstab
```

sh

可选操作

当然，在服务器上使用的话，关闭Swap影响还是很大的，如果服务器除了Kubernetes还有其他用途的话（除非实在太穷，否则建议不要这样混用；一定要混用的话，宁可把其他服务搬到Kubernetes上）。关闭Swap有可能会对其他服务产生不良的影响，这时需要修改每个节点的kubelet配置，去掉必须关闭Swap的默认限制，具体操作为：

```
$ echo "KUBELET_EXTRA_ARGS=--fail-swap-on=false" >>  
/etc/sysconfig/kubelet
```

sh

其次，由于Kubernetes与Docker默认的cgroup（root控制组）驱动程序并不一致，Kubernetes默认为systemd，而Docker默认为cgroupfs。

更新信息

从1.18开始，Kubernetes默认的cgroup驱动已经默认修改成cgroupfs了，这时候再进行改动反而会不一致

在这里我们要修改Docker或者Kubernetes其中一个的cgroup驱动，以便两者统一。根据官方文档《[CRI installation](#)》中的建议，对于使用systemd作为引导系统的Linux的发行版，使用systemd作为Docker的cgroup驱动程序可以服务器节点在资源紧张的情况下表现得更为稳定。

这里选择修改各个节点上Docker的cgroup驱动为systemd，具体操作为编辑（无则新增）/etc/docker/daemon.json文件，加入以下内容即可：

```
{  
  "exec-opts": ["native.cgroupdriver=systemd"]}
```

{}

然后重新启动Docker容器：

```
$ systemctl daemon-reload  
$ systemctl restart docker
```

sh

预拉取镜像

可选

预拉取镜像并不是必须的，本来初始化集群的时候系统就会自动拉取Kubernetes中要使用到的Docker镜像组件，也提供了一个“kubeadm config images pull”命令来一次性的完成拉取，这都是因为如果要手工来进行这项工作，实在非常非常非常的繁琐。

但对于许多人来说这项工作往往又是无可奈何的，Kubernetes的镜像都存储在k8s.gcr.io上，如果您的机器无法直接或通过代理访问到gcr.io（Google Container Registry，敲黑板：这是属于谷歌的网址）的话，初始化集群时自动拉取就无法顺利进行，所以就不得不手工预拉取。

预拉取的意思是，由于Docker只要查询到本地有相同（名称和tag完全相同、哈希相同）的镜像，就不会访问远程仓库，那只要从GitHub上拉取到所需的镜像，再将tag修改成官方的一致，就可以跳过网络访问阶段。

首先使用以下命令查询当前版本需要哪些镜像：

```
$ kubeadm config images list --kubernetes-version v1.17.3
```



```
k8s.gcr.io/kube-apiserver:v1.17.3  
k8s.gcr.io/kube-controller-manager:v1.17.3  
k8s.gcr.io/kube-scheduler:v1.17.3  
k8s.gcr.io/kube-proxy:v1.17.3  
k8s.gcr.io/pause:3.1  
k8s.gcr.io/etcd:3.4.3-0  
k8s.gcr.io/coredns:1.6.5
```

sh

.....

这里必须使用“--kubernetes-version”参数指定具体版本，因为尽管每个版本需要的镜像信息在本地是有存储的，但如果不去加的话，Kubernetes将向远程GCR仓库查询最新的版本号，会因网络无法访问而导致问题。但加版本号的时候切记不能照抄上面的命令中的“v1.17.3”，应该与你安装的kubelet版本保持一致，否则在初始化集群控制平面的时候会提示控制平面版本与kubectl版本不符。

得到这些镜像名称和tag后，可以从DockerHub上找存有相同镜像的仓库来拉取，至于具体哪些公开仓库有，考虑到以后阅读本文时Kubernetes的版本应该会有所差别，所以需要自行到网站上查询一下。笔者比较常用的是一个名为“anjia0532”的仓库，有机器人自动跟官方同步，相对比较及时。

```
#以k8s.gcr.io/coredns:1.6.5为例，每个镜像都要这样处理一次  
$ docker pull anjia0532/google-containers.coredns:1.6.5  
  
#修改tag  
$ docker tag anjia0532/google-containers.coredns:1.6.5  
k8s.gcr.io/coredns:1.6.5  
  
#修改完tag后就可以删除掉旧镜像了  
$ docker rmi anjia0532/google-containers.coredns:1.6.5
```

初始化集群控制平面

到了这里，终于可以开始Master节点的部署了，先确保kubelet是开机启动的：

```
$ sudo systemctl start kubelet  
$ sudo systemctl enable kubelet
```

接下来使用su直接切换到root用户（而不是使用sudo），然后使用以下命令开始部署：

```
$ kubeadm init --kubernetes-version v1.17.3 --pod-network-cidr=10.244.0.0/16
```

这里使用“--kubernetes-version”参数（要注意版本号与kubelet一致）的原因与前面预拉取是一样的，避免额外的网络访问；另外一个参数“--pod-network-cidr”着在稍后介绍完CNI网

络插件时会去说明。

当看到下面信息之后，说明集群主节点已经安装完毕了。

```
Your Kubernetes control-plane has initialized successfully! Kubeadm can't pull the image, the following log applies.

To start using your cluster, you need to run the following as a regular user:
  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:
  kubeadm join 10.3.7.5:6443 --token ejg4tt.y08moym055dn9i32 \
    --discovery-token-ca-cert-hash sha256:9d2079d2844fa2953d33cc0da57ab15f571e974aa40ccb50edde12c5e906d513
```

这信息先恭喜你已经把控制平面安装成功了，但还有三行“you need.....”、“you shoul d.....”、“you can.....”开头的内容，这是三项后续的“可选”工作，下面继续介绍。

为当前用户生成kubeconfig

使用Kubernetes前需要为当前用户先配置好admin.conf文件。切换至需配置的用户后，进行如下操作：

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

安装CNI插件 可选

CNI即“容器网络接口”，在2016年，CoreOS发布了CNI规范。2017年5月，CNI被CNCF技术监督委员会投票决定接受为托管项目，从此成为不同容器编排工具（Kubernetes、Mesos、OpenShift）可以共同使用的、解决容器之间网络通讯的统一接口规范。

部署Kubernetes时，我们可以有两种网络方案使得以后受管理的容器之间进行网络通讯：

- 使用Kubernetes的默认网络
- 使用CNI及其插件

第一种方案，尤其不在GCP或者AWS的云主机上，没有它们的命令行管理工具时，需要大量的手工配置，基本上是反人类的。实际通常都会采用第二种方案，使用CNI插件来处理容器之间的网络通讯，所以本节所标识的“[可选]”其实也并没什么选择不安装CNI插件的余地。

Kubernetes目前支持的CNI插件有：Calico、Cilium、Contiv-VPP、Flannel、Kube-router、Weave Net等六种，每种网络提供了不同的管理特性（如MTU自动检测）、安全特性（如是否支持加密通讯）、网络策略（如Ingress、Egress规则）、传输性能（甚至对TCP、UDP、HTTP、FTP、SCP等不同协议来说也有不同的性能表现）以及主机的性能消耗。后续我们将专门对不同CNI插件进行测试对比，在环境部署这部分，对于初学者来说，使用Flannel是较为合适的，它是最精简的CNI，没有安全特性的支持，主机压力小，安装便捷，效率也不错，使用以下命令安装Flannel网络：

```
$ curl --insecure -sfL https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml | kubectl apply -f -
```

使用Flannel的话，要注意要在创建集群时加入“--pod-network-cidr”参数，指明网段划分。

移除Master节点上的污点

可选

污点（Taint）是Kubernetes Pod调度中的概念，在这里通俗地理解就是Kubernetes决定在集群中的哪一个节点建立新的容器时，要先排除掉带有特定污点的节点，以避免容器在Kubernetes不希望运行的节点中创建、运行。默认情况下，集群的Master节点是会带有污点的，以避免容器分配到Master中创建。但对于许多学习Kubernetes的同学来说，并没有多宽裕的机器数量，往往是建立单节点集群或者最多只有两、三个节点，这样Master节点不能运行容器就显得十分浪费了。需要移除掉Master节点上所有的污点，在Master节点上执行以下命令即可：

```
$ kubectl taint nodes --all node-role.kubernetes.io/master-
```

做到这步，如果你只有一台机器的话，那Kubernetes的安装已经宣告结束了，可以使用此环境来完成后续所有的部署。你还可以通过cluster-info和get nodes子命令来查看一下集群

的状态，类似如下所示：

```
# ubuntu @ linux in ~ [15:37:45]
$ kubectl cluster-info
Kubernetes master is running at https://10.3.7.5:6443
KubeDNS is running at https://10.3.7.5:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

# ubuntu @ linux in ~ [15:37:53]
$ kubectl get nodes
NAME      STATUS    ROLES     AGE   VERSION
lab.server Ready     master   29m   v1.17.3
```

调整NodePort范围 可选

Kubernetes默认的NodePort范围为30000-32767，为了方便使用低端口，可能需要修改此范围，这需要调整Api-Server的启动参数，具体操作如下（如过是高可用部署，需要对每一个Master节点进行修改）：

- 修改 `/etc/kubernetes/manifests/kube-apiserver.yaml` 文件，添加一个参数在 `spec.containers.command` 中增加一个参数 `--service-node-port-range=1-32767`
- 重启Api-Server，现在Kubernetes基本都是以静态Pods模式部署，Api-Server是一个直接由kubelet控制的静态Pod，删除后它会自动重启：

```
# 获得 apiserver 的 pod 名字
export apiserver_pods=$(kubectl get pods --selector=component=kube-
apiserver -n kube-system --output=jsonpath={.items..metadata.name})
# 删除 apiserver 的 pod
kubectl delete pod $apiserver_pods -n kube-system
```

- 验证修改结果：可以在pod中看到该参数即可

```
kubectl describe pod $apiserver_pods -n kube-system
```

启用kubectl命令自动补全功能 可选

由于kubectl命令在后面十分常用，而且Kubernetes许多资源名称都带有随机字符，要手工照着敲很容易出错，强烈推荐启用命令自动补全的功能，这里仅以bash和笔者常用的zsh为例，如果您使用其他shell，需自行调整：

bash：

```
$ echo 'source <(kubectl completion bash)' >> ~/.bashrc  
$ echo 'source /usr/share/bash-completion/bash_completion' >> ~/.bashrc
```

sh

zsh：

```
$ echo 'source <(kubectl completion zsh)' >> ~/.zshrc
```

sh

将其他Node节点加入到Kubernetes集群中

在安装Master节点时候，输出的最后一部分内容会类似如下所示：

```
Then you can join any number of worker nodes by running the following  
on each as root:
```

```
kubeadm join 10.3.7.5:6443 --token ejg4tt.y08moym055dn9i32 \  
--discovery-token-ca-cert-hash  
sha256:9d2079d2844fa2953d33cc0da57ab15f571e974aa40ccb50edde12c5e906d513
```

这部分内容是告诉用户，集群的Master节点已经建立完毕，其他节点的机器可以使用“kubeadm join”命令加入集群。这些机器只要完成kubeadm、kubelet、kubectl的安装即可、其他的所有步骤，如拉取镜像、初始化集群等等都不需要去做，就是可以使用该命令加入集群了。需要注意的是，该Token的有效时间为24小时，如果超时，使用以下命令重新获取：

```
$ kubeadm token create --print-join-command
```

sh

使用Rancher部署

Rancher是在Kubernetes更上层的管理框架，Rancher是图形化的，有着比较傻瓜式的操作，只有少量一两处地方（如导入集群）需要用到Kubernetes命令行。也由于它提供了一系列容器模版、应用商店等的高层功能，使得要在Kubernetes上部署一个新应用，简化到甚至只需要点几下鼠标即可，因此用户们都爱使用它。

Rancher还推出了RancherOS（极致精简专为容器定制的Linux，尤其适合边缘计算环境）、K3S（Kubernetes as a Service，5 Less Than K8S，一个大约只有40MB，可以运行在x86和ARM架构上的极小型Kubernetes发行版）这样的定制产品，用以在用户心中暗示、强化比K8S更小、更简单、更易用的主观印象。

不过也由于Rancher入门容易，基础性的应用需求解决起来很方便，也导致了不少人一开始使用它之后，就陷入了先入为主的印象，后期再接触Kubernetes时，便觉得学习曲线特别陡峭，反而限制了某些用户对底层问题的进一步深入。

在本文中，笔者以截图为主，展示如何使用Rancher来导入或者创建Kubernetes集群的过程。

安装Rancher

前置条件：已经安装好Docker。

使用Docker执行Rancher镜像，执行以下命令即可：

```
$ sudo docker run -d --restart=unless-stopped -p 8080:80 -p 8443:443  
rancher/rancher
```

sh

使用Rancher管理现有Kubernetes集群

前置条件：已经安装好了Kubernetes集群。

使用Rancher的导入功能将已部署的Kubernetes集群纳入其管理。登陆Rancher主界面（首次登陆会要求设置admin密码和Rancher在集群中可访问的路径，后者尤其不能乱设，否则Kubernetes无法访问到Rancher会一直处于Pending等待状态）之后，点击右上角的Add Cluster，然后有下面几个添加集群的选择：

Add Cluster - Select Cluster Type

From existing nodes (Custom)
Create a new Kubernetes cluster using RKE, out of existing bare-metal servers or virtual machines.

Import an existing cluster
Import an existing Kubernetes cluster. The provider that created it will continue to manage the provisioning and configuration of the cluster.

With RKE and new nodes in an infrastructure provider

- Amazon EC2
- Azure
- DigitalOcean
- Linode
- vSphere

With a hosted Kubernetes provider

- Amazon EKS
- Azure AKS
- Google GKE

- 要从某台机器中新安装Kubernetes集群选择“From existing nodes (Custom)”
- 要导入某个已经安装好的Kubernetes集群选择“Import an existing cluster”
- 要从各种云服务商的RKE (Rancher Kubernetes Engine) 环境中创建，就选择下面那排厂商的按钮，没有的话（譬如国内的阿里云之类的），请先到Tools->Driver中安装对应云服务厂商的驱动。

这里选择“Import an existing cluster”，然后给集群起个名字以便区分（由于Rancher支持多集群管理，所以集群得有个名字以示区别），之后就看见这个界面：

Note: If you want to import a Google Kubernetes Engine (GKE) cluster (or any cluster that does not supply you with a `kubectl` configuration file with the ClusterRole `cluster-admin` bound to it), you need to bind the ClusterRole `cluster-admin` using the command below.

Replace `[USER_ACCOUNT]` with your Google account address (you can retrieve this using `gcloud config get-value account`). If you are not importing a Google Kubernetes Engine cluster, replace `[USER_ACCOUNT]` with the executing user configured in your `kubectl` configuration file.

```
kubectl create clusterrolebinding cluster-admin-binding --clusterrole cluster-admin --user [USER_ACCOUNT]
```

Run the `kubectl` command below on an existing Kubernetes cluster running a supported Kubernetes version to import it into Rancher:

```
kubectl apply -f https://localhost:8443/v3/import/vgkj5tzphj9vzg6157krdc9gfc4b4zsfp419prrf6sb7z9d2wvbbh5.yaml
```

If you get an error about 'certificate signed by unknown authority' because your Rancher installation is running with an untrusted/self-signed SSL certificate, run the command below instead to bypass the certificate check:

```
curl --insecure -sfL https://localhost:8443/v3/import/vgkj5tzphj9vzg6157krdc9gfc4b4zsfp419prrf6sb7z9d2wvbbh5.yaml | kubectl apply -f -
```

Rancher自动生成了加入集群的命令，这行命令其实就是部署一个运行在Kubernetes中的代理（Agent），在Kubernetes的命令行中执行以上自动生成的命令。

最后那条命令意思是怕由于部署的Rancher服务没有申请SSL证书，导致HTTPS域名验证过不去，`kubectl`下载不下来yaml。如果你的Rancher部署在已经申请了证书的HTTPS地址上那可以用前面的，否则还是直接用`curl --insecure`命令来绕过HTTPS证书查验吧，譬如以下命令所示：

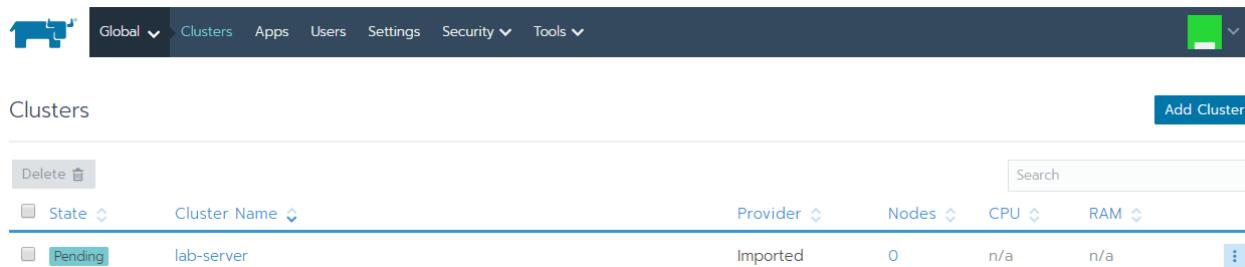
```
$ curl --insecure -sfL
https://localhost:8443/v3/import/vgkj5tzphj9vzg6157krdc9gfc4b4zsfp419pr
rf6sb7z9d2wvbbh5.yaml | kubectl apply -f -
```

多说一句，用哪条命令安装的Agent只决定了yaml文件是如何下载获得的，对后续其他事情是毫无影响的，所以怎么简单怎么来，别折腾。

执行结果类似如下所示，一堆secret、deployment、daementset创建成功，就代表顺利完成了：

```
# ubuntu @ linux in ~ [16:41:24]
$ curl --insecure -sfL https://localhost:8443/v3/import/vgkj5tzphj9vzg6157krdc9gfc4b4zsfp419prrf6sb7z9d2wvbbh5.yaml | kubectl apply -f -
clusterrole.rbac.authorization.k8s.io/proxy-clusterrole-kubeapiserver created
clusterrolebinding.rbac.authorization.k8s.io/proxy-role-binding-kubernetes-master created
namespace/cattle-system created
serviceaccount/cattle created
clusterrolebinding.rbac.authorization.k8s.io/cattle-admin-binding created
secret/cattle-credentials-5d6cb35 created
clusterrole.rbac.authorization.k8s.io/cattle-admin created
deployment.apps/cattle-cluster-agent created
daemonset.apps/cattle-node-agent created
```

然后回到Rancher网页，点击界面上的“Done”按钮。可以看到集群正处于Pending状态：



Cluster Name	Provider	Nodes	CPU	RAM
lab-server	Imported	0	n/a	n/a

如果Agent成功到达Running状态的话，这里也会很快就变成Waiting状态，然后再变为Active状态，导入工作即宣告胜利结束。

而如果一直持续Pending状态，说明安装的Agent运行失败。典型的原因是无法访问到Rancher的服务器，这时可以通过kubectl logs命令查看一下cattle-cluster-agent-xxx的日志，通常会看见"XXX is not accessible"，其中的XXX是Rancher第一次进入时跟你确认过的访问地址，假如你乱填了，或者该地址被防火墙挡掉，又或者因为证书限制等其他原因导致Agent无法访问，Rancher就会一直Pending。

最后再提一句，Rancher与Kubernetes集群之间是被动链接的，即由Kubernetes去主动找Rancher，这意味着部署在外网的Rancher，可以无障碍地管理处于内网（譬如NAT后）的Kubernetes集群，这对于大量没有公网IP的集群来说是很方便的事情。

使用Rancher创建Kubernetes集群

也可以直接使用Rancher直接在裸金属服务器上创建Kubernetes集群，此时在添加集群中选择From existing nodes (Custom)，在自定义界面中，设置要安装的集群名称、Kubernetes版本、CNI网络驱动、私有镜像库以及其他一些集群的参数选项。

添加集群 - Custom

集群名称 *

hk

添加描述

成员角色

控制哪些用户可以访问集群，以及他们拥有的对其进行更改的权限。

标签/注释

为集群配置标签和注释。

无

集群选项

编辑 YAML

全部展开

Kubernetes选项

自定义集群功能

Kubernetes版本

v117.2-rancher1-2

网络驱动

Flannel

Windows支持

启用

禁用

项目网络隔离

启用

禁用

网络 MTU

0

Only applied if the value is non-zero. When applied, the MTU value is explicitly configured for the chosen network provider (disabling auto-discovery). The override must be calculated from the host's MTU minus the CNI plugin's required overhead.

下一步确认该主机在Kubernetes中扮演的角色，每台主机可以扮演多个角色。但至少要保证每个集群都有一个Etcd角色、一个Control角色、一个Worker角色。

添加集群 - Custom

集群选项

添加主机命令

选择主机角色,端口映射请参考: <https://rancher.com/docs/rancher/v2.x/en/installation/references/>

角色选择 (每台主机可以运行多个角色。每个集群至少需要一个Etcd角色、一个Control角色、一个Worker角色)

Etcd Control Worker

主机地址

为主机配置公网地址和内网地址, 如果为VPC网络的云服务器, 如果不指定公网地址节点将无法获取到对应公网IP。

公网地址: 例如: 12.3.4 内网地址: 例如: 12.3.4

节点名称

(可选) 自定义节点显示的名称, 不显示实际的主机名

例如: My-worker-node

主机标签

(可选) 添加到节点的标签

+ 添加标签

节点污点 (Taints)

(可选) 添加到节点的污点 (taints)

+ 添加污点 (Taint)

2 复制以下命令在主机的SSH终端运行。

```
sudo docker run -d --privileged --restart=unless-stopped --net=host -v /etc/kubernetes:/etc/kubernetes -v /var/run:/var/run
rancher/rancher-agent:v2.3.5 --server https://k8s.icyfenix.cn:444 --token 8snnt4mmj4hfwld892cl8f7knwj1v8824hwtm05grqm7gg7ftzkz2 --ca-
checksum 825812c06dea6cf75008f91df2ccabe23b177b7d3cbd522585af025cdbe5ec4b --etcd --controlplane --worker
```

复制生成的命令，在要安装集群的每一台主机的SSH中执行。此时Docker会下载运行Rancher的Agent镜像，当执行成功后，Rancher界面会有提示新主机注册成功。



点击完成，将会在集群列表中看见正在Provisioning的新集群，稍后将变为Active状态。

状态	集群名称	供应商	主机数	处理器	内存
Provisioning	hk	自定义	0	n/a	n/a

Waiting for etcd and controlplane nodes to be registered

安装完成后你就可以在Rancher的图形界面管理Kubernetes集群了，如果还需要在命令行中工作，kubectl、kubeadm等工具是没有安装的，可参考“[使用Kubeadm部署Kubernetes集群](#)”的内容安装使用。

使用Minikube部署

Minikube是Kubernetes官方提供的专门针对本地单节点集群的Kubernetes集群管理工具。针对本地环境对Kubernetes使用有一定的简化和针对性的补强。这里简要介绍其安装过程

安装Minikube

Minikube是一个单文件的二进制包，安装十分简单，在已经完成Docker安装的前提下，使用以下命令可以下载并安装最新版的Minikube。

```
$ curl -Lo minikube  
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-  
amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/  
sh
```

安装Kubectl工具

Minikube中很多提供了许多子命令以代替Kubectl的功能，安装Minikube时并不会一并安装Kubectl。但是Kubectl作为集群管理的命令行，要了解Kubernetes是无论如何绕不过去的，通过以下命令可以独立安装Kubectl工具。

```
$ curl -LO https://storage.googleapis.com/kubernetes-  
release/release/$(curl -s https://storage.googleapis.com/kubernetes-  
release/release/stable.txt)/bin/linux/amd64/kubectl && chmod +x kubectl  
&& sudo mv kubectl /usr/local/bin/  
sh
```

启动Kubernetes集群

有了Minikube，通过start子命令就可以一键部署和启动Kubernetes集群了，具体命令如下：

```
$ minikube start --iso-url=https://kubernetes.oss-cn-hangzhou.aliyuncs.com/minikube/iso/minikube-v1.6.0.iso
          --registry-mirror=https://registry.docker-cn.com
          --image-mirror-country=cn
          --image-repository=registry.cn-hangzhou.aliyuncs.com/google_containers
          --vm-driver=none
          --memory=4096
```

sh

以上命令中，明确要求Minikube从指定的地址下载虚拟机镜像、Kubernetes各个服务Pod的Docker镜像，并指定了使用Docker官方节点作为国内的镜像加速服务。

“vm-drvier”参数是指Minikube所采用的虚拟机，根据不同操作系统，不同的虚拟机可以有以下选项：

操作系统	支持虚拟机	参数值
Windows	Hyper-V	hyperv
Windows	VirtualBox	virtualbox
Linux	KVM	kvm2
Linux	VirtualBox	virtualbox
MacOS	HyperKit	hyperkit
MacOS	VirtualBox	virtualbox
MacOS	Parallels Desktop	parallels
MacOS	VMware Fusion	vmware

特别需要提一下的是如果读者使用的并非物理机器，而是云主机环境——现在流行将其成为“裸金属”（Bare Metal）服务器，那在上面很可能是无法再部署虚拟机环境的，这时候应该将vm-drvier参数设为none。也可以使用以下命令设置虚拟机驱动的默认值：

```
$ minikube config set vm-driver none
```

sh

至此，整个Kubernetes就一键启动完毕了。其他工作，如命令行的自动补全，可参考使用Kubeadm安装Kubernetes集群中相关内容。

部署Istio

部署Elastic Stack

部署Prometheus