从分布式开始

分布式更多的一个概念，**是为了解决单个物理服务器容量和性能瓶颈问题而采用的优化手段。**该领域需要解决的问题极多，在不同的技术层面上，又包括：分布式文件系统、分布式缓存、分布式数据库、分布式计算、分布式失误等，一些名词如Hadoop、zookeeper、MQ等都跟分布式有关。从理念上讲，分布式的实现有两种形式：

水平扩展：当一台机器扛不住流量时，就通过添加机器的方式，将流量平分到所有服务器上，所有机器都可以提供相当的服务；

垂直拆分：前端有多种查询需求时，一台机器扛不住，可以将不同的需求分发到不同的机器上，比如A机器处理余票查询的请求，B机器处理支付的请求。

有一个概念和分布式比较相似，那就是**微服务。**

微服务是一种架构风格，一个大型复杂软件应用由一个或多个微服务组成。系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表着一个小的业务能力。

去饭店吃饭就是一个完整的业务，饭店的厨师、配菜师、传菜员、服务员就是分布式；厨师、配菜师、传菜员和服务员都不止一个人，这就是集群；分布式就是微服务的一种表现形式，分布式是部署层面，微服务是设计层面

单机结构、集群结构、分布式

集群是个物理形态，分布式是个工作方式。

单机结构

大家最熟悉的就是单机结构，一个系统业务量很小的时候所有的代码都放在一个项目中就好了，然后这个项目部署在一台服务器上就好了。整个项目所有的服务都由这台服务器提供。这就是单机结构。

那么，单机结构有什么缺点呢？显而易见的，单机的处理能力毕竟是有限的，当你的业务增长到一定程度的时候，单机的硬件资源将无法满足你的业务需求。此时便出现了集群模式。

集群结构

单机处理到达瓶颈的时候，你就把单机复制几份，这样就构成了一个“集群”。集群中每台服务器就叫做这个集群的一个“节点”，所有节点构成了一个集群。每个节点都提供相同的服务，那么这样系统的处理能力就相当于提升了好几倍

> 集群的瓶颈：磁盘io、网络（稀缺资源）

但问题是用户的请求究竟由哪个节点来处理呢？最好能够让此时此刻负载较小的节点来处理，这样使得每个节点的压力都比较平均。要实现这个功能，就需要在所有节点之前增加一个“调度者”的角色，用户的所有请求都先交给它，然后它根据当前所有节点的负载情况，决定将这个请求交给哪个节点处理。这个“调度者”有个高大上的名字——负载均衡服务器。

集群结构的好处就是系统扩展非常容易。如果随着你们系统业务的发展，当前的系统又支撑不住了，那么给这个集群再增加节点就行了。但是，当你的业务发展到一定程度的时候，你会发现一个问题——无论怎么增加节点，貌似整个集群性能的提升效果并不明显了。这时候，你就需要使用微服务结构了。

分布式结构

先来对前面的知识点做个总结。

从单机结构到集群结构，你的代码基本无需要作任何修改，你要做的仅仅是多部署几台服务器，每台服务器上运行相同的代码就行了。但是，当你要从集群结构演进到微服务结构的时候，之前的那套代码就需要发生较大的改动了。所以对于新系统我们建议，系统设计之初就采用微服务架构，这样后期运维的成本更低。但如果一套老系统需要升级成微服务结构的话，那就得对代码大动干戈了。所以，对于老系统而言，究竟是继续保持集群模式，还是升级成微服务架构，这需要你们的架构师深思熟虑、权衡投入产出比。

OK，下面开始介绍所谓的分布式结构。

分布式结构就是将一个完整的系统，按照业务功能，拆分成一个个独立的子系统，在分布式结构中，每个子系统就被称为“服务”。这些子系统能够独立运行在web容器中，它们之间通过RPC方式通信。

举个例子，假设需要开发一个在线商城。按照微服务的思想，我们需要按照功能模块拆分成多个独立的服务，如：用户服务、产品服务、订单服务、后台管理服务、数据分析服务等等。这一个个服务都是一个个独立的项目，可以独立运行。如果服务之间有依赖关系，那么通过RPC方式调用。

这样的好处有很多：

1. 系统之间的耦合度大大降低，可以独立开发、独立部署、独立测试，系统与系统之间的边界非常明确，排错也变得相当容易，开发效率大大提升。
2. 系统之间的耦合度降低，从而系统更易于扩展。我们可以针对性地扩展某些服务。假设这个商城要搞一次大促，下单量可能会大大提升，因此我们可以针对性地提升订单系统、产品系统的节点数量，而对于后台管理系统、数据分析系统而言，节点数量维持原有水平即可。
3. 服务的复用性更高。比如，当我们将用户系统作为单独的服务后，该公司所有的产品都可以使用该系统作为用户系统，无需重复开发。

RPC

> 远程过程调用(Remote Procedure Call)

先看什么是进程间通信

进程间通信（IPC，Inter-Process Communication），指至少两个进程或线程间传送数据或信号的一些技术或方法。进程是计算机系统分配资源的最小单位。每个进程都有自己的一部分独立的系统资源，彼此是隔离的。为了能使不同的进程互相访问资源并进行协调工作，才有了进程间通信。这些进程可以运行在同一计算机上或网络连接的不同计算机上。 进程间通信技术包括消息传递、同步、共享内存和远程过程调用。 IPC是一种标准的Unix通信机

制。

有两种类型的进程间通信(IPC)。

- 本地过程调用(LPC)LPC用在多任务操作系统中，使得同时运行的任务能互相会话。这些任务共享内存空间使任务同步和互相发送信息。
- 远程过程调用(RPC)RPC类似于LPC，只是在网上工作。简单地说，让调用远程计算机上的服务，就像调用本地服务一样。

为什么RPC呢？就是无法在一个进程内，甚至一个计算机内通过本地调用的方式完成的需求，比如比如不同的系统间的通讯，甚至不同的组织间的通讯。由于计算能力需要横向扩展，需要在多台机器组成的集群上部署应用

RPC的核心并不在于使用什么协议。RPC的目的是让你在本地调用远程的方法，而对你来说这个调用是透明的，你并不知道这个调用的方法是部署哪里。通过RPC能解耦服务，这才是使用RPC的真正目的。RPC的原理主要用到了动态代理模式，至于http协议，只是传输协议而已。

简单的说，

- RPC就是从一台机器（客户端）上通过参数传递的方式调用另一台机器（服务器）上的一个函数或方法（可以统称为服务）并得到返回的结果。
- RPC 会隐藏底层的通讯细节（不需要直接处理Socket通讯或Http通讯） RPC 是一个请求响应模型。
- 客户端发起请求，服务器返回响应（类似于Http的工作方式） RPC 在使用形式上像调用本地函数（或方法）一样去调用远程的函数（或方法）。

目前流行的开源 RPC 框架还是比较多的，有阿里巴巴的 Dubbo、Facebook 的 Thrift、Google 的 gRPC、Twitter 的 Finagle 等。

RPC 和 Restful API 对比

面对对象不同：

- RPC 更侧重于动作。
- REST 的主体是资源。

RESTful 是面向资源的设计架构，但在系统中有很多对象不能抽象成资源，比如登录，修改密码等而 RPC 可以通过动作去操作资源。所以在操作的全面性上 RPC 大于 RESTful。

传输效率：

- RPC 效率更高。RPC，使用自定义的 TCP 协议，可以让请求报文体积更小，或者使用 HTTP2 协议，也可以很好的减少报文的体积，提高传输效率。

复杂度：

- RPC 实现复杂，流程繁琐。
- REST 调用及测试都很方便。

RPC 实现(参见***节)需要实现编码，序列化，网络传输等。而 RESTful 不要关注这些，RESTful 实现更简单。

灵活性：

- HTTP 相对更规范，更标准，更通用，无论哪种语言都支持 HTTP 协议。
- RPC 可以实现跨语言调用，但整体灵活性不如 RESTful。

总结

RPC 主要用于公司内部的服务调用，性能消耗低，传输效率高，实现复杂。

HTTP 主要用于对外的异构环境，浏览器接口调用，App 接口调用，第三方接口调用等。

RPC 使用场景(大型的网站，内部子系统较多、接口非常多的情况下适合使用 RPC)：

- 长链接。不必每次通信都要像 HTTP 一样去 3 次握手，减少了网络开销。
- 注册发布机制。RPC 框架一般都有注册中心，有丰富的监控管理;发布、下线接口、动态扩展等，对调用方来说是无感知、统一化的操作。
- 安全性，没有暴露资源操作。
- 微服务支持。就是最近流行的服务化架构、服务化治理，RPC 框架是一个强力的支撑。

go rpc

客户端

```
package main

import (
   "net/rpc"
   "log"
   "fmt"
)

type Args struct {
   A, B int
}

type Quotient struct {
   Quo, Rem int
}

func main()  {
   client, err := rpc.DialHTTP("tcp", "127.0.0.1:1234")
   if err != nil {
      log.Fatal("dialing:", err)
   }

   // Synchronous call
   args := &Args{7,8}
   var reply int
   err = client.Call("Arith.Multiply", args, &reply)
   if err != nil {
      log.Fatal("arith error:", err)
   }
```

```
    fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

    var reply1 int
    err = client.Call("Arith.Plus", args, &reply1)
    if err != nil {
        log.Fatal("arith error:", err)
    }
    fmt.Printf("Arith: %d+%d=%d\n", args.A, args.B, reply1)

    // Asynchronous call
    quotient := new(Quotient)
    divCall := client.Go("Arith.Divide", args, quotient, nil)
    replyCall := <-divCall.Done    // will be equal to divCall
    if replyCall.Error != nil {
        log.Fatal("arith error:", replyCall.Error)
    }
    fmt.Printf("Arith: %d/%d=%d...%d", args.A, args.B, quotient.Quo, quotient.Rem)
    // check errors, print, etc.
}
```

服务端

```
package main

import (
    "errors"
    "net"
    "net/rpc"
    "log"
    "net/http"
)

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Plus(args *Args, reply *int) error {
    *reply = args.A + args.B
    return nil
```

```
}


func (t *Arith) Divide(args *Args, quo *Quotient) error {
   if args.B == 0 {
      return errors.New("divide by zero")
   }
   quo.Quo = args.A / args.B
   quo.Rem = args.A % args.B
   return nil
}

func main() {
   arith := new(Arith)
   rpc.Register(arith)
   rpc.HandleHTTP()
   l, e := net.Listen("tcp", ":1234")
   if e != nil {
      log.Fatal("listen error:", e)
   }
   http.Serve(l, nil)
}
```

分布式id生成器

> https://books.studygolang.com/advanced-go-programming-book/ch6-cloud/ch6-01-dist-id.html

有时我们需要能够生成类似MySQL自增ID这样不断增大，同时又不会重复的id。以支持业务中的高并发场景。比较典型的，电商促销时，短时间内会有大量的订单涌入到系统，比如每秒10w+。明星出轨时，会有大量热情的粉丝发微博以表心意，同样会在短时间内产生大量的消息。

在插入数据库之前，我们需要给这些消息、订单先打上一个ID，然后再插入到我们的数据库。对这个id的要求是希望其中能带有一些时间信息，这样即使我们后端的系统对消息进行了分库分表，也能够以时间顺序对这些消息进行排序。

Twitter的snowflake算法是这种场景下的一个典型解法。先来看看snowflake是怎么一回事，见*图6-1*：

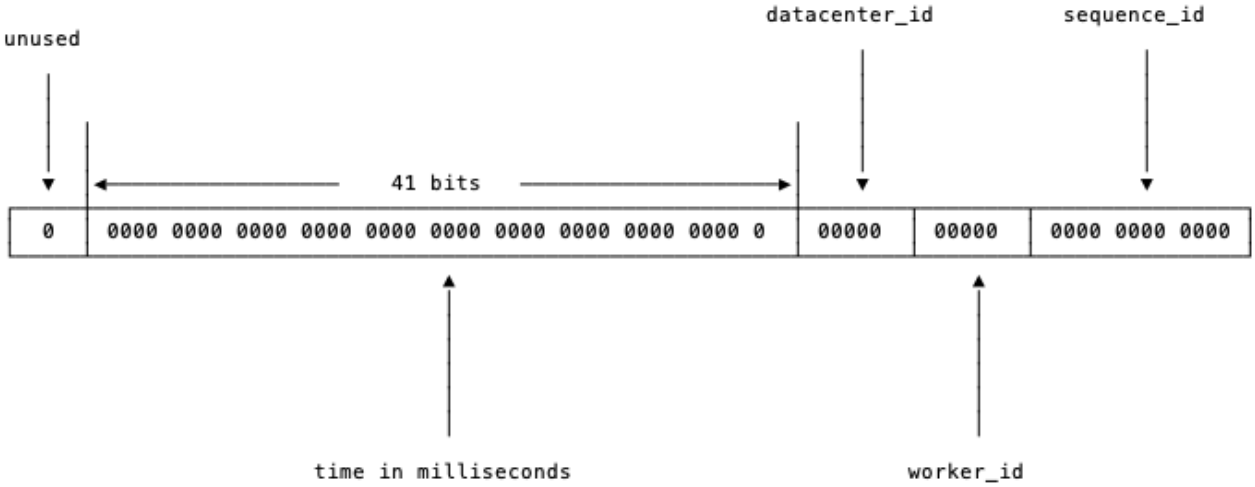

图 6-1 snowflake 中的比特位分布

首先确定我们的数值是64位，int64类型，被划分为四部分，不含开头的第一个bit，因为这个bit是符号位。用41位来表示收到请求时的时间戳，单位为毫秒，然后五位来表示数据中心的id，然后再五位来表示机器的实例id，最后是12位的循环自增id（到达1111,1111,1111后会归0）。

这样的机制可以支持我们在同一台机器上，同一毫秒内产生`2 ^ 12 = 4096`条消息。一秒共409.6万条消息。从值域上来讲完全够用了。

数据中心加上实例id共有10位，可以支持我们每数据中心部署32台机器，所有数据中心共1024台实例。

表示`timestamp`的41位，可以支持我们使用69年。当然，我们的时间毫秒计数不会真的从1970年开始记，那样我们的系统跑到`2039/9/7 23:47:35`就不能用了，所以这里的`timestamp`实际上只是相对于某个时间的增量，比如我们的系统上线是2018-08-01，那么我们可以把这个timestamp当作是从`2018-08-01 00:00:00.000`的偏移量