架构师

ARCHITECT

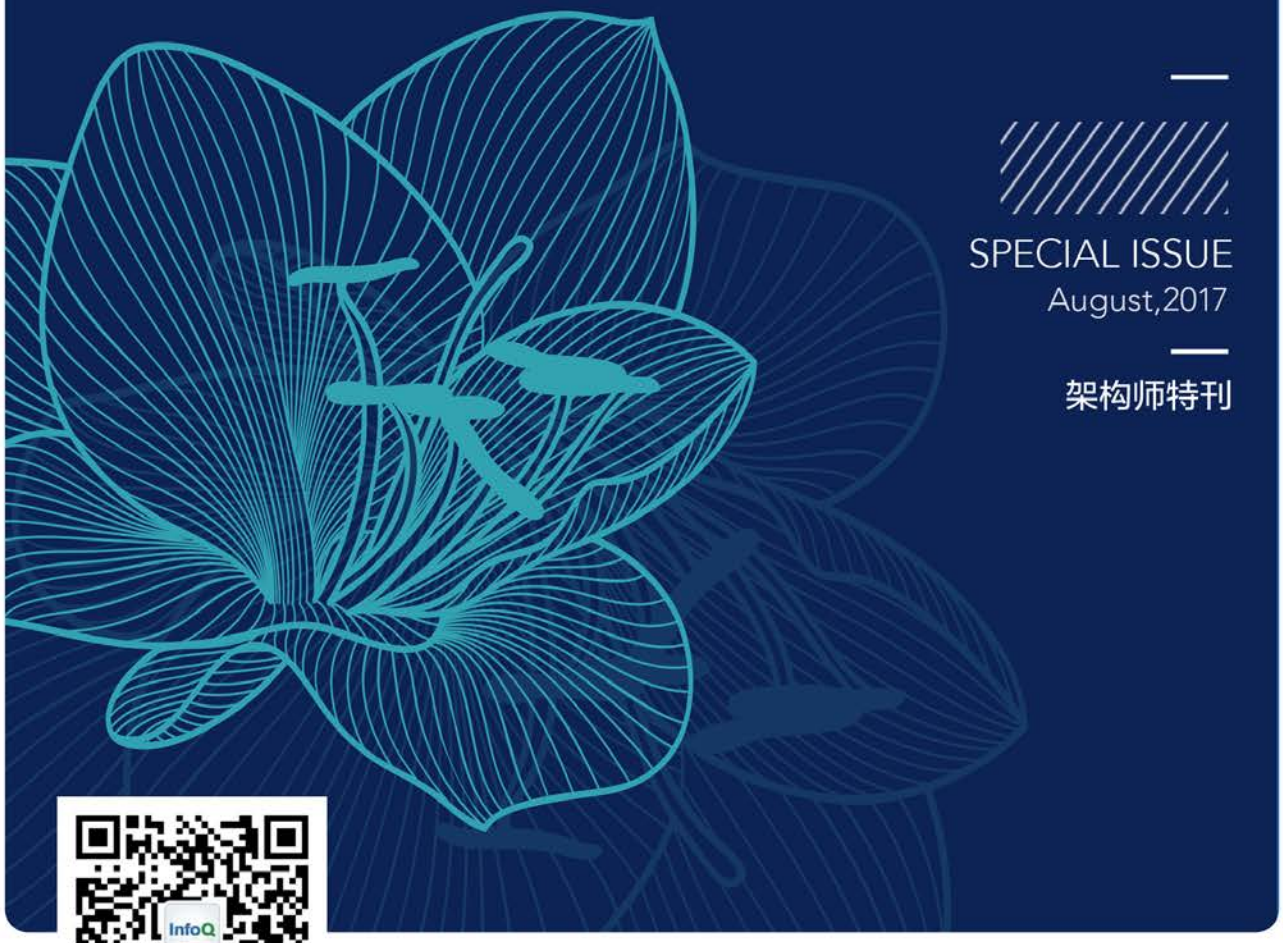
特刊

编程语言

SPECIAL ISSUE

August,2017

架构师特刊



Geekbang. 极客邦科技 | InfoQ

序言

PHP 是最好的编程语言吗?

陈利人

大家都可能听过“PHP 是最好的语言”段子。

编程语言很多，既然存在，就有每个存在的理由。其实没必要评论哪个语言好，哪个语言不好。因为每个编程语言本身都不难，只要学会了一种语言，其他的都是相通的。难的，好坏之分，是理解，记忆，熟悉，和流畅的使用那些每个语言提供的基础库和扩展库，也就是 LIBs，或是 APIs，或是 SDK，或是 Frameworks。

编程语言本身，就最简单的集合来说，就是一堆保留的关键词和一堆的语法，这个大家稍微看看就都会明白，还有其设计思想。就和学一门外语一样，一些词汇加上一些词汇能组合的语法。这些通常包括的编程语言特性有：语句组成，变量定义，算术运算，循环语句，函数定义和调用，面向对象，指针操作，垃圾回收，输入输出等等。基础的东西，大家如果学过或是使用过一种语言，其他的新的语言，花个一两个小时就能基本理解，就能写出一个 hello world 的入门程序。

那么，好坏之分，难的部分，实际上是那些基础库和各种扩展库。这些库，必须在实践中一步一步熟悉和掌握。如果不经常使用，还特别容易忘记，尽管现代的编程环境有各种智能辅助。会不会，熟不熟一门语言，

很大程度上是对他们的持续使用和理解。这也是为什么，没有多个项目，或是几年的实践，很难说是一个语言的高手。

比如同样对于网络的操作，有的语言的库包装的简单高级易用全面，有的语言就基础原始。那么对于那些有高级包装库的语言，大家就会觉得好，语言好用，开发效率高，想要什么都有，socket、tcp/ip、http、async/sync、select、event，甚至是 sever、client、crawler 等等。而对那些比较底层的包装的语言，大家就学觉得开发效率低，难用，比如只有 select 和 socket。是不是大概如此？

学习完了编程语言基础，大家一般要花很多的时间去了解和学习基础库和扩展库，这个才是真正的痛处和难点，学了不用还容易忘记，用到的时候还得想到有这个函数库。那么，**如果有一套对各种语言都适用的 API 或是库函数定义，那么，大家学习和使用一门新的语言的效率会大幅度提高**。这时，哪个语言最好，也就不再是什么问题了。

Protobuf、Thrift、COM、RPC 等等都已经在做这方面尝试，如果有一天，大家不管用什么编程语言，只要记住一套接口函数，那就太美好了。这时，估计没有人会再争论，PHP 是否是最好的语言了。

目录

05 今日头条 Go 建千亿级微服务的实践

20 Java 老矣，尚能饭否?

33 Python 向来以慢著称，为啥 Instagram 却唯独钟爱它?

46 我们为什么要选择小众语言 Rust 来实现 TiKV？

51 Clojure 太灵活，我们能如何驾驭它?

59 JavaScript 成为了一流语言

65 FreeWheel 基于 Go 的实践经验漫谈

今日头条 Go 建千亿级微服务的实践

作者 项超

今日头条当前后端服务超过 80% 的流量是跑在 Go 构建的服务上。微服务数量超过 100 个，高峰 QPS 超过 700 万，日处理请求量超过 3000 亿，是业内最大规模的 Go 应用。

Go 构建微服务的历程

在 2015 年之前，头条的主要编程语言是 Python 以及部分 C++。随着业务和流量的快速增长，服务端的压力越来越大，随之而来问题频出。Python 的解释性语言特性以及其落后的多进程服务模型受到了巨大的挑战。此外，当时的服务端架构是一个典型的单体架构，耦合严重，部分独

立功能也急需从单体架构中拆出来。

为什么选择 Go 语言?

Go 语言相对其它语言具有几点天然的优势:

1. 语法简单，上手快
2. 性能高，编译快，开发效率也不低
3. 原生支持并发，协程模型是非常优秀的服务端模型，同时也适合网络调用
4. 部署方便，编译包小，几乎无依赖

当时 Go 的 1.4 版本已经发布，我曾在 Go 处于 1.1 版本的时候，开始使用 Go 语言开发后端组件，并且使用 Go 构建过超大流量的后端服务，因此对 Go 语言本身的稳定性比较有信心。再加上头条后端整体服务化的架构改造，所以决定使用 Go 语言构建今日头条后端的微服务架构。

2015 年 6 月，今日头条开始使用 Go 语言重构后端的 Feed 流服务，期间一边重构，一边迭代现有业务，同时还进行服务拆分，直到 2016 年 6 月，Feed 流后端服务几乎全部迁移到 Go。由于期间业务增长较快，夹杂服务拆分，因此没有横向对比重构前后的各项指标。但实际上切换到 Go 语言之后，服务整体的稳定性和性能都大幅提高。

微服务架构

对于复杂的服务间调用，我们抽象出五元组的概念: (From, FromCluster, To, ToCluster, Method)。每一个五元组唯一定义了一类的 RPC 调用。以五元组为单元，我们构建了一整套微服务架构。

我们使用 Go 语言研发了内部的微服务框架 kite，协议上完全兼容 Thrift。以五元组为基础单元，我们在 kite 框架上集成了服务注册和发现，分布式负载均衡，超时和熔断管理，服务降级，Method 级别的指标监控，分布式调用链追踪等功能。目前统一使用 kite 框架开发内部 Go 语言的服务，整体架构支持无限制水平扩展。

关于 kite 框架和微服务架构实现细节后续有机会会专门分享，这里主要分享下我们在使用 Go 构建大规模微服务架构中，Go 语言本身给我们带来了哪些便利以及实践过程中我们取得的经验。内容主要包括并发，性能，监控以及对 Go 语言使用的一些体会。

并发

Go 作为一门新兴的编程语言，最大特点就在于它是原生支持并发的。和传统基于 OS 线程和进程实现不同，Go 语言的并发是基于用户态的并发，这种并发方式就变得非常轻量，能够轻松运行几万甚至是几十万的并发逻辑。因此使用 Go 开发的服务端应用采用的就是“协程模型”，每一个请求由独立的协程处理完成。

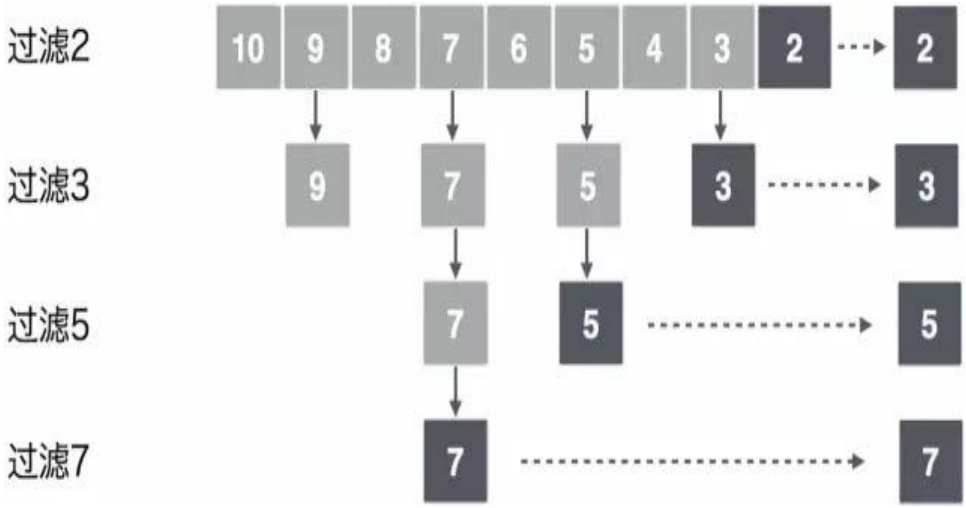
比进程线程模型高出几个数量级的并发能力，而相对基于事件回调的服务端模型，Go 开发思路更加符合人的逻辑处理思维，因此即使使用 Go 开发大型的项目，也很容易维护。

并发模型

Go 的并发属于 CSP 并发模型的一种实现，CSP 并发模型的核心概念是：“不要通过共享内存来通信，而应该通过通信来共享内存”。这在 Go 语言中的实现就是 Goroutine 和 Channel。在 1978 发表的 CSP 论文中有一段使用 CSP 思路解决问题的描述。

“Problem: To print in ascending order all primes less than 10000. Use an array of processes, SIEVE, in which each process inputs a prime from its predecessor and prints it. The process then inputs an ascending stream of numbers from its predecessor and passes them on to its successor, suppressing any that are multiples of the original prime.”

要找出 10000 以内所有的素数，这里使用的方法是筛法，即从 2 开始每找到一个素数就标记所有能被该素数整除的所有数。直到没有可标记的数，剩下的就都是素数。下面以找出 10 以内所有素数为例，借用 CSP 方式解决这个问题。



从上图中可以看出，每一行过滤使用独立的并发处理程序，上下相邻的并发处理程序传递数据实现通信。通过 4 个并发处理程序得出 10 以内的素数表，对应的 Go 实现代码如下：

```
func main() {
    origin, wait := make(chan int), make(chan struct{})
    Processor(origin, wait)
    for num := 2; num < 10000; num++ {
        origin <- num
    }
    close(origin)
    <-wait
}
```

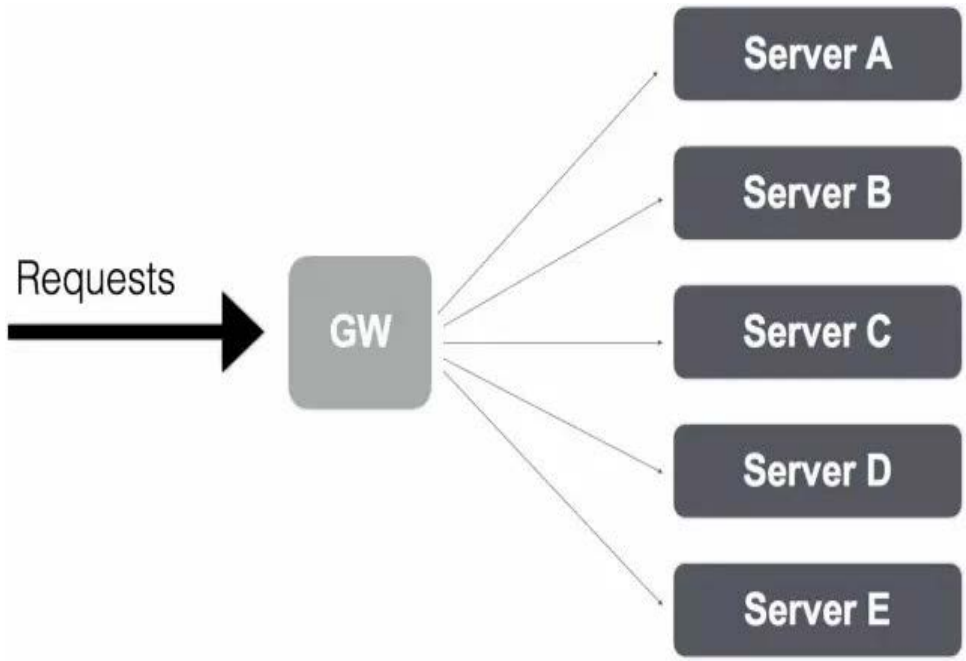
```
func Processor(seq chan int, wait chan struct{}) {
    go func() {
        prime, ok := <-seq
        if !ok {
            close(wait)
            return
        }
        fmt.Println(prime)
        out := make(chan int)
        Processor(out, wait)
        for num := range seq {
            if num%prime != 0 {
                out <- num
            }
        }
        close(out)
    }()
}
```

这个例子体现使用 Go 语言开发的两个特点：

- Go 语言的并发很简单，并且通过提高并发可以提高处理效率。
- 协程之间可以通过通信的方式来共享变量。

并发控制

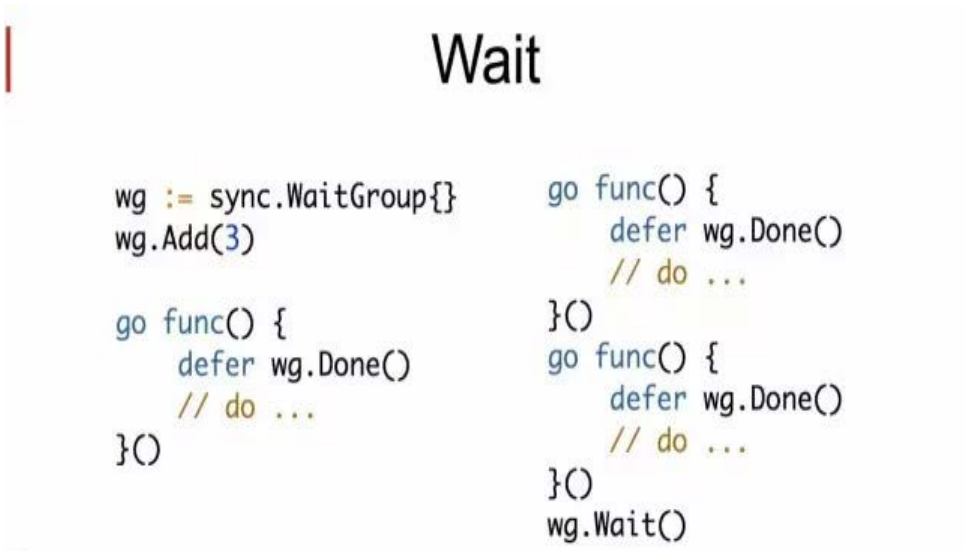
当并发成为语言的原生特性之后，在实践过程中就会频繁地使用并发来处理逻辑问题，尤其是涉及到网络 I/O 的过程，例如 RPC 调用，数据库访问等。下图是一个微服务处理请求的抽象描述：

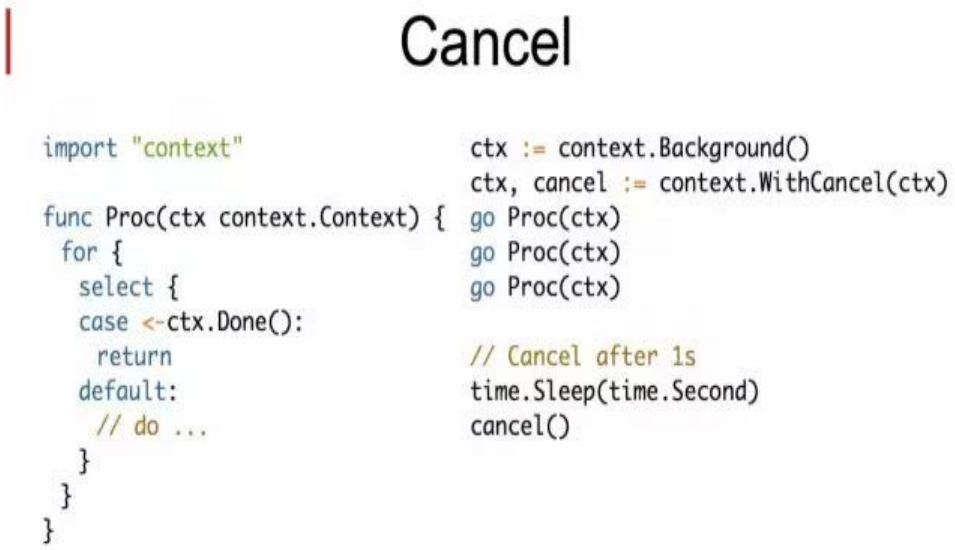


当 Request 到达 GW 之后，GW 需要整合下游 5 个服务的结果来响应本次的请求，假定对下游 5 个服务的调用不存在互相的数据依赖问题。那么这里会同时发起 5 个 RPC 请求，然后等待 5 个请求的返回结果。为避免长时间的等待，这里会引入等待超时的概念。超时事件发生后，为了避免资源泄漏，会发送事件给正在并发处理的请求。在实践过程中，得出两种抽象的模型。

- Wait
- Cancel

Wait 和 Cancel 两种并发控制方式，在使用 Go 开发服务的时候到处



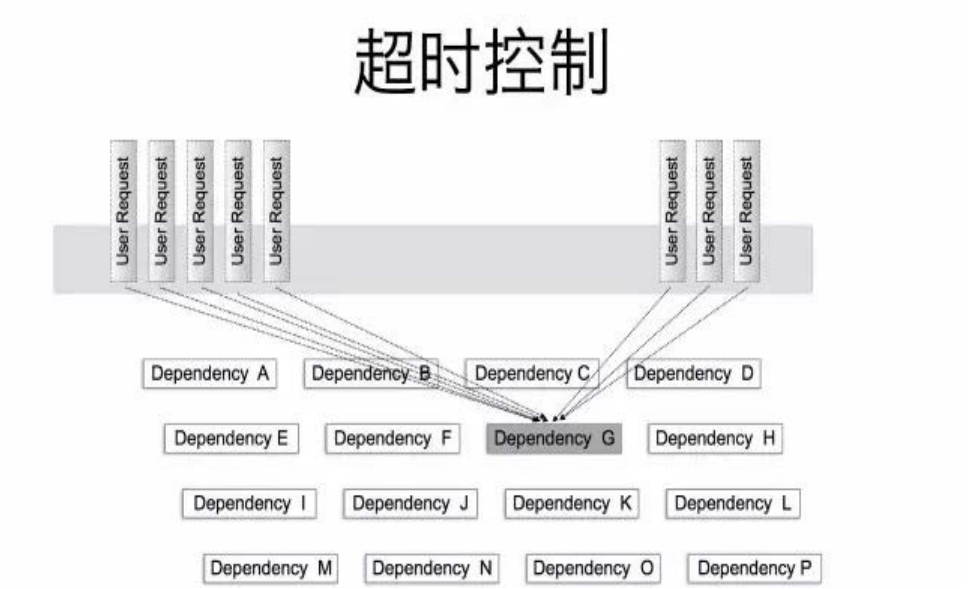


都有体现，只要使用了并发就会用到这两种模式。在上面的例子中，GW 启动 5 个协程发起 5 个并行的 RPC 调用之后，主协程就会进入等待状态，需要等待这 5 次 RPC 调用的返回结果，这就是 Wait 模式。另一中 Cancel 模式，在 5 次 RPC 调用返回之前，已经到达本次请求处理的总超时时间，这时候就需要 Cancel 所有未完成的 RPC 请求，提前结束协程。Wait 模式使用会比较广泛一些，而对于 Cancel 模式主要体现在超时控制和资源回收。

在 Go 语言中，分别有 sync.WaitGroup 和 context.Context 来实现这两种模式。

超时控制

合理的超时控制在构建可靠的大规模微服务架构显得非常重要，不合理的超时设置或者超时设置失效将会引起整个调用链上的服务雪崩。



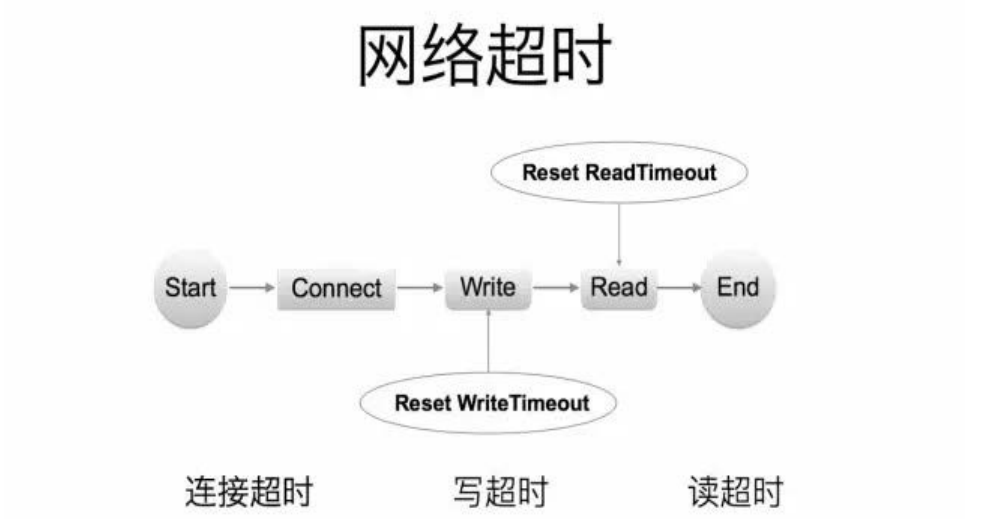
图中被依赖的服务 G 由于某种原因导致响应比较慢，因此上游服务的请求都会阻塞在服务 G 的调用上。如果此时上游服务没有合理的超时控制，导致请求阻塞在服务 G 上无法释放，那么上游服务自身也会受到影响，进一步影响到整个调用链上各个服务。

在 Go 语言中，Server 的模型是“协程模型”，即一个协程处理一个请求。如果当前请求处理过程因为依赖服务响应慢阻塞，那么很容易会在短时间内堆积起大量的协程。每个协程都会因为处理逻辑的不同而占用不同大小的内存，当协程数据激增，服务进程很快就会消耗大量的内存。

协程暴涨和内存使用激增会加剧 Go 调度器和运行时 GC 的负担，进而再次影响服务的处理能力，这种恶性循环会导致整个服务不可用。在使用 Go 开发微服务的过程中，曾多次出现过类似的问题，我们称之为协程暴涨。

有没有好的办法来解决这个问题呢？通常出现这种问题的原因是网络调用阻塞过长。即使在我们合理设置网络超时之后，偶尔还是会出现超时

限制不住的情况，对 Go 语言中如何使用超时控制进行分析，首先我们来看下一次网络调用的过程。



第一步，建立 TCP 连接，通常会设置一个连接超时时间来保证建立连接的过程不会被无限阻塞。

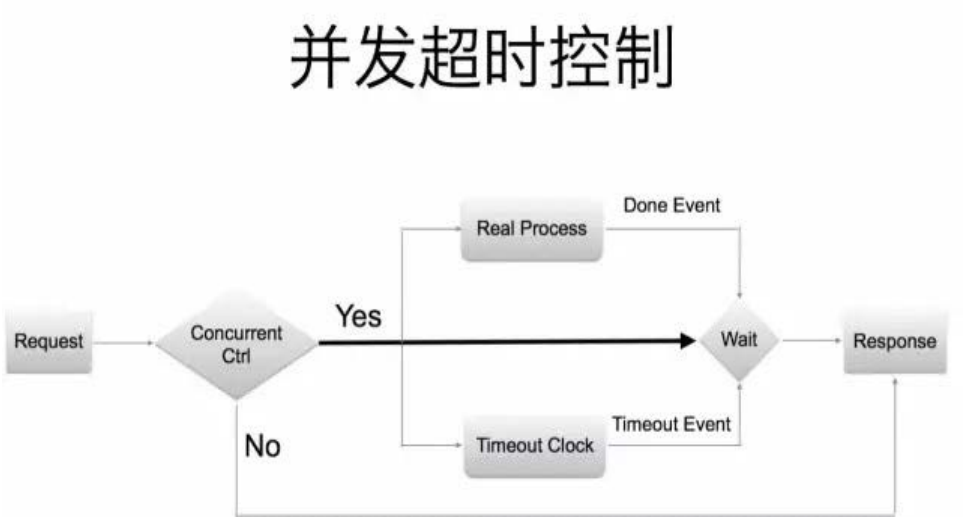
第二步，把序列化后的 Request 数据写入到 Socket 中，为了确保写数据的过程不会一直阻塞，Go 语言提供了 SetWriteDeadline 的方法，控制数据写入 Socket 的超时时间。根据 Request 的数据量大小，可能需要多次写 Socket 的操作，并且为了提高效率会采用边序列化边写入的方式。因此在 Thrift 库的实现中每次写 Socket 之前都会重新 Reset 超时时间。

第三步，从 Socket 中读取返回的结果，和写入一样， Go 语言也提供了 SetReadDeadline 接口，由于读数据也存在读取多次的情况，因此同样会在每次读取数据之前 Reset 超时时间。

分析上面的过程可以发现影响一次 RPC 耗费的总时间的长短由三部分组成：连接超时，写超时，读超时。而且读和写超时可能存在多次，这就导致超时限制不住情况的发生。为了解决这个问题，在 kite 框架中引入了并发超时控制的概念，并将功能集成到 kite 框架的客户端调用库中。

并发超时控制模型如下图所示，在模型中引入了“Concurrent Ctrl”模块，这个模块属于微服务熔断功能的一部分，用于控制客户端能够发起的最大

并发请求数。并发超时控制整体流程是这样的



首先，客户端发起 RPC 请求，经过“Concurrent Ctrl”模块判断是否允许当前请求发起。如果被允许发起 RPC 请求，此时启动一个协程并执行 RPC 调用，同时初始化一个超时定时器。然后在主协程中同时监听 RPC 完成事件信号以及定时器信号。如果 RPC 完成事件先到达，则表示本次 RPC 成功，否则，当定时器事件发生，表明本次 RPC 调用超时。这种模型确保了无论何种情况下，一次 RPC 都不会超过预定义的时间，实现精准控制超时。

```
import (
    "context"
)

func Handler(r *Request) {
    timeout := r.Value("timeout")
    ctx, cancel := context.WithTimeout(context.Background(), timeout)
    defer cancel()
    done := make(chan struct{}, 1)
    go func() {
        RPC(ctx, ...)
        done <- struct{}
    }()
    select{
    case <-done:
        // nice ...
    case <-ctx.Done():
        // timeout ...
    }
}
```

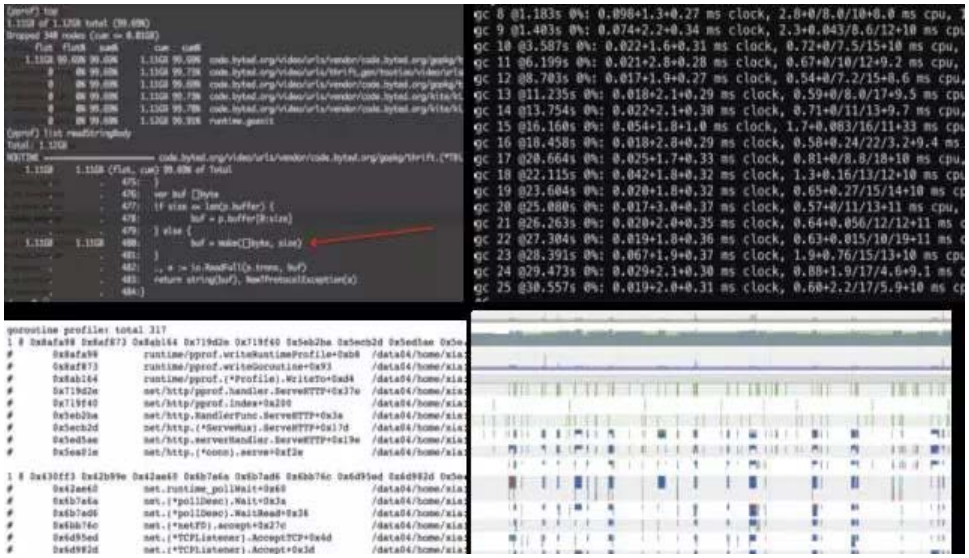
Go 语言在 1.7 版本的标准库引入了“context”，这个库几乎成为了并发控制和超时控制的标准做法，随后 1.8 版本中在多个旧的标准库中增加对“context”的支持，其中包括“database/sql”包。

性能

Go 相对于传统 Web 服务端编程语言已经具备非常大的性能优势。但是很多时候因为使用方式不对，或者服务对延迟要求很高，不得不使用一些性能分析工具去追查问题以及优化服务性能。在 Go 语言工具链中自带了多种性能分析工具，供开发者分析问题。

- CPU 使用分析
- 内部使用分析
- 查看协程栈
- 查看 GC 日志
- Trace 分析工具

下图是各种分析方法截图。



在使用 Go 语言开发的过程中，我们总结了一些写出高性能 Go 服务的方法：

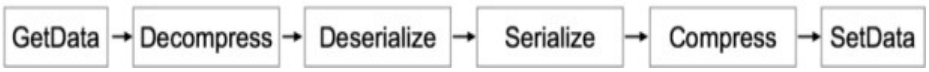
- 注重锁的使用，尽量做到锁变量而不要锁过程；
- 可以使用 CAS，则使用 CAS 操作；

- 针对热点代码要做针对性优化；
- 不要忽略 GC 的影响，尤其是高性能低延迟的服务；
- 合理的对象复用可以取得非常好的优化效果；
- 尽量避免反射，在高性能服务中杜绝反射的使用；
- 有些情况下可以尝试调优“GOGC”参数；
- 新版本稳定的前提下，尽量升级新的 Go 版本，因为旧版本永远不会变得更好。

下面描述一个真实的线上服务性能优化例子。

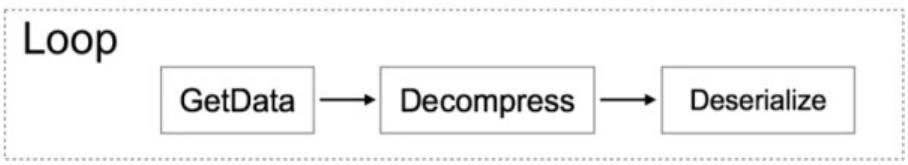
这是一个基础存储服务，提供 SetData 和 GetDataByRange 两个方法，分别实现批量存储数据和按照时间区间批量获取数据的功能。为了提高性能，存储的方式是以用户 ID 和一段时间作为 key，时间区间内的所有数据作为 value 存储到 KV 数据库中。因此，当需要增加新的存储数据时候就需要先从数据库中读取数据，拼接到对应的时间区间内再存到数据库中。

- SetData



对于读取数据的请求，则会根据请求的时间区间计算对应的 key 列表，然后循环从数据库中读取数据。

- GetDataByRange



这种情况下，高峰期服务的接口响应时间比较高，严重影响服务的整体性能。通过上述性能分析方法对于高峰期服务进行分析之后，得出如下结论：

问题点：

- GC 压力大，占用 CPU 资源高

- 反序列化过程占用 CPU 较高

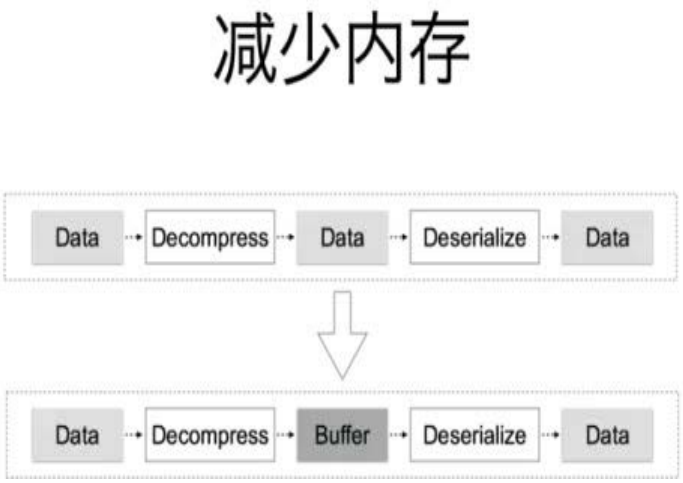
优化思路：

- GC 压力主要是内存的频繁申请和释放，因此决定减少内存和对象的申请；
- 序列化当时使用的是 Thrift 序列化方式，通过 Benchmark，我们找到相对高效的 Msgpack 序列化方式。

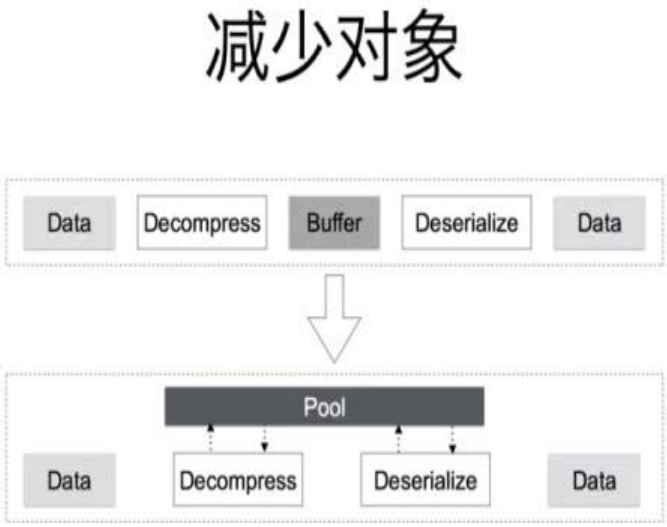
分析服务接口功能可以发现，数据解压缩，反序列化这个过程是最频繁的，这也符合性能分析得出来的结论。仔细分析解压缩和反序列化的过程，发现对于反序列化操作而言，需要一个"io.Reader"的接口，而对于解压缩，其本身就实现了"io.Reader"接口。在 Go 语言中，"io.Reader"的接口定义如下：

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

这个接口定义了 Read 方法，任何实现该接口的对象都可以从中读取一定数量的字节数据。因此只需要一段比较小的内存 Buffer 就可以实现从解压缩到反序列化的过程，而不需要将所有数据解压缩之后再进行反序列化，大量节省了内存的使用。



为了避免频繁的 Buffer 申请和释放，使用sync.Pool实现了一个对象池，达到对象复用的目的。



此外，对于获取历史数据接口，从原先的循环读取多个 key 的数据，优化为从数据库并发读取各个 key 的数据。经过这些优化之后，服务的高峰 PCT99 从 100ms 降低到 15ms。

上述是一个比较典型的 Go 语言服务优化案例。概括为两点：

- 从业务层面上提高并发
- 减少内存和对象的使用

优化的过程中使用了 pprof 工具发现性能瓶颈点，然后发现“io.Reader”接口具备的 Pipeline 的数据处理方式，进而整体优化了整个服务的性能。

服务监控

Go 语言的 runtime 包提供了多个接口供开发者获取当前进程运行的状态。在 kite 框架中集成了协程数量，协程状态，GC 停顿时间，GC 频率，堆栈内存使用量等监控。实时采集每个当前正在运行的服务的这些指标，分别针对各项指标设置报警阈值，例如针对协程数量和 GC 停顿时间。另一方面，我们也在尝试做一些运行时服务的堆栈和运行状态的快照，方便追查一些无法复现的进程重启的情况。

编程思维和工程性

相对于传统 Web 编程语言，Go 在编程思维上的确带来了许多的改变。每一个 Go 开发服务都是一个独立的进程，任何一个请求处理造成 Panic，都会让整个进程退出，因此当启动一个协程的时候需要考虑是否需要使用 recover 方法，避免影响其它协程。对于 Web 服务端开发，往往希望将一个请求处理的整个过程能够串起来，这就非常依赖于 Thread Local 的变量，而在 Go 语言中并没有这个概念，因此需要在函数调用的时候传递 context。

最后，使用 Go 开发的项目中，并发是一种常态，因此就需要格外注意对共享资源的访问，临界区代码逻辑的处理，会增加更多的心智负担。这些编程思维上的差异，对于习惯了传统 Web 后端开发的开发者，需要一个转变的过程。

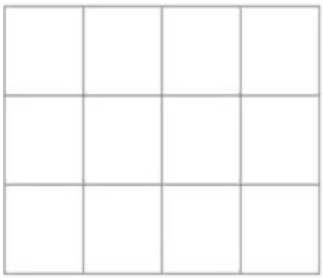
关于工程性，也是 Go 语言不太所被提起的点。实际上在 Go 官方网站关于为什么要开发 Go 语言里面就提到，目前大多数语言当代码量变得巨大之后，对代码本身的管理以及依赖分析变得异常苦难，因此代码本身成为了最麻烦的点，很多庞大的项目到最后都变得不敢去动它。而 Go 语言不同，其本身设计语法简单，类C的风格，做一件事情不会有很多种方法，甚至一些代码风格都被定义到 Go 编译器的要求之内。而且，Go 语言标准库自带了源代码的分析包，可以方便地将一个项目的代码转换成一颗 AST 树。

借助Go的AST包实现自定义检测代码

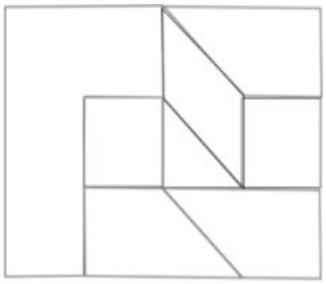
```
import (
    "go/parser"
    "go/token"
)

fset := token.NewFileSet() // positions are relative to fset
f, err := parser.ParseFile(fset, "example.go", nil, parser.AllErrors)
```

下面以一张图形象地表达下 Go 语言的工程性：



Golang



Python

同样是拼成一个正方形，Go 只有一种方式，每个单元都是一致。而 Python 拼接的方式可能可以多种多样。

写在最后

今日头条使用 Go 语言构建了大规模的微服务架构，本文结合 Go 语言特性着重讲解了并发，超时控制，性能等在构建微服务中的实践。事实上，Go 语言不仅在服务性能上表现卓越，而且非常适合容器化部署，我们很大一部分服务已经运行于内部的私有云平台。结合微服务相关组件，我们正朝着 Cloud Native 架构演进。

更多技术实践内容可以关注今日头条技术博客：techblog.toutiao.com。

作者介绍

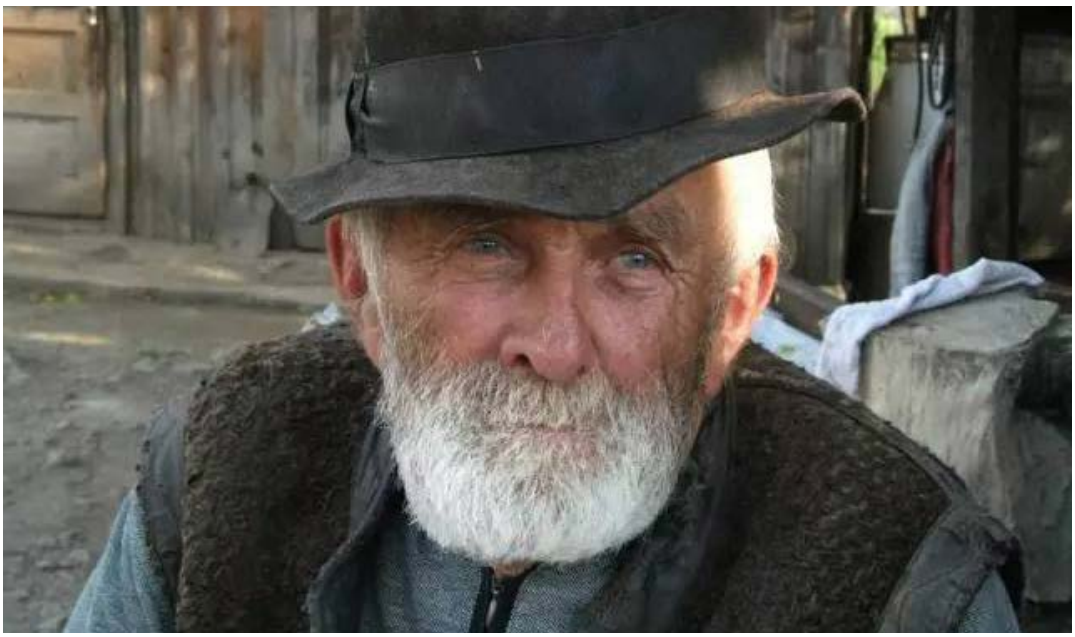
项超，今日头条高级研发工程师。2015 年加入今日头条，负责服务化改造相关工作，在内部推广 Go 语言的使用，研发内部微服务框架 kite，集成服务治理，负载均衡等多种微服务功能，实现了 Go 语言构建大规模微服务架构在头条的落地。曾就职于小米。

推荐文章

我为什么选择使用 Go 语言？

Java 老矣，尚能饭否?

作者 张建锋



22 岁，对于一个技术人来说可谓正当壮年。但对于一门编程语言来说，情况可能又有不同。各类编程语言横空出世，纷战不休，然而 TIOBE 的语言排行榜上，Java 却露出了明显的颓势。这个老牌的语言，未来会是怎样?

写在前面

从 1995 年第一个版本发布到现在，Java 语言已经在跌宕起伏中走过了 22 年，最新的 Java 版本也已经迭代到 Java 9。当年 Java 语言的跨平台

优势如今看来也只不过是家常小菜，Go、Rust 等语言横空出世，进一步拓宽了编程语言的边界。当年发明 Java 语言的 Sun 公司早已被 Oracle 收购，Oracle 现在也正处于水深火热的云计算浪潮当中，甚至连 Java 之父 James Gosling 也加入了当今世界最大的云计算公司 AWS。

Java 语言发展的这 20 年也正是全球互联网迅猛发展的 20 年，Java 语言同时也见证了电商浪潮、移动互联网浪潮、大数据浪潮、云计算浪潮，所以在现今各大互联网公司身上都能看到 Java 的身影。

纵看 Java 语言的发展，不禁让人联想到辛弃疾的一首词：

千古江山，英雄无觅，孙仲谋处。舞榭歌台，风流总被雨打风吹去。斜阳草树，寻常巷陌，人道寄奴曾住。想当年，金戈铁马，气吞万里如虎。元嘉草草，封狼居胥，赢得仓皇北顾。四十三年，望中犹记，烽火扬州路。可堪回首，佛狸祠下，一片神雅社鼓。凭谁问，廉颇老矣，尚能饭否?

TIOBE 的语言排行榜显示，自 2016 年初 Java 语言就出现了明显的下颓趋势，开发者社区也出现了一些唱衰 Java 语言的论调，编者心中也有些许疑问：Java 老矣，尚能『饭』否？基于这样的背景，InfoQ 邀请到了 Java 资深专家张建锋来为大家解读 Java 语言的发展现状以及未来。

Java 语言的发展回顾

Java 语言源于 1991 年 Sun 公司 James Gosling 领导的的 Ork 项目，1995 年 Sun 公司正式起名为 Java，并提出“Write once, Run anywhere" 的口号。

1996 年 1 月 Java 1.0 发布，提供了一个解释执行的 Java 虚拟机，其时恰逢互联网开始兴起，Java 的 Applet 能在 Mozilla 浏览器中运行，被看作是未来的互联网语言。

1997 年 2 月 Java 1.1 发布，Java 语言的基本形态基本确定了，比如反射 (reflection), JavaBean, 接口和类的关系等等，一直到今天都保持一致。然而，Java 最初的一些目标，如在浏览器中执行 Applet，以及跨平台的图形界面 Awt 很快遭遇到负面的评价。

1998 年 12 月，Java 第一个里程碑式的版本，即 Java 1.2 发布了。这个版本使用了 JIT（Just in time）编译器技术，使得语言的可迁移性和执行效率达到最优的平衡，同时 Collections 集合类设计优良，在企业应用开发中迅速得到了广泛使用。Sun 公司把 Java 技术体系分成三个方向，分别是 J2SE（面向桌面和通用应用开发），J2EE（面向企业级应用开发），J2ME（面向移动终端开发）。这个分类影响非常久远，体现出主流语言设计者的思想：针对于不同的应用领域，在形态，API 集合等进行划分。

2000 年 5 月，Java 1.3 发布，这个版本中 Corba 作为语言级别的分布式对象技术，成为 J2EE 的一个技术前提。J2EE 受到 Corba 的设计的影响较大，早期 EJB 的 Home，接口和实现就是 Corba 在 C 语言的实现，被移植到 Java 语言之中。J2EE 中的 Servlet 规范获得了极大的成功，伴随着互联网的兴起，和浏览器直接通过 HTTP 协议交互的 Servlet，和众多的 MVC 框架，成为 Web1.0 的网红。

2002 年 2 月，Java 1.4 发布，Java 语言真正走向成熟，提供了非常完备的语言特性，如 NIO，正则表达式，XML 处理器等。同年微软的 .NET 框架发布，两者开始了为期十几年的暗自竞争。从语言特性上来说，.NET 后发先至，一直处于优势。但 Java 依赖良好的开发者生态，绝大多数大型软件公司的使用者众多和不断贡献，以及对 Linux 操作系统良好的支持，渐渐的在服务器端获得优势地位。

2004 年 9 月，Java 5 发布，Sun 不再采用 J2SE, J2EE 这种命名方式，而使用 Java SE 5, Java EE 5 这样的名称。我认为 Java 5 是第二个里程碑式的版本。Java 语言语法发生很大的变化，如注解 (Annotation)，装箱 (Autoboxing)，泛型 (Generic)，枚举 (Enum)，foreach 等被加入，提供了 java.util.concurrent 并发包。Java 5 对于 Java 语言的推动是巨大的，特别是注解的加入，使得语言定义灵活了很多，程序员可以写出更加符合领域定义的描述性程序。

2006 年 5 月， JavaEE 5 发布，其中最主要是 EJB3.0 的版本升级。在此之前，EJB2.X 版本被广泛质疑，SpringFramework 创建者 Rod Johnson

在经典书籍“J2EE Development without EJB“中，对 EJB2 代表的分布式对象的设计方法予以批驳。EJB3 则重新经过改造，使用注解方式，经过应用服务器对 POJO 对象进行增强来实现分布式服务能力。在某种程度，可以说 EJB3 挽救了 JavaEE 的过早消亡。

2006 年 12 月，Java 6 发布，这个语言语法改进不多，但在虚拟机内部做了大量的改进，成为一个相当成熟稳定的版本，时至今日国内的很多公司依然以 Java6 作为主要 Java 开发版本来使用。同年 Sun 公司做出一个伟大的决定，将 Java 开源。OpenJDK 从 Sun JDK 1.7 版本分支出去，成为今天 OpenJDK 的基础。OpenJDK6 则由 OpenJDK7 裁剪而来，目前由红帽负责维护，来满足 Redhat Enterprise Linux 6.X 用户的需要。

2009 年 12 月，JavaEE 6 发布，这个版本应该说是 JavaEE 到目前为止改进最大影响最深远的一个版本。因为 JavaEE5 只有 EJB3 适应了 Java 注解语法的加入，而 EE6 全面接纳了注解。CDI 和 BeanValidation 规范的加入，在 POJO 之上可以定义完备的语义，由容器来决定如何去做。Servlet 也升级到 3.0 版本，并在接口上加入异步支持，使得系统整体效率可以大幅提高。EE 划分为 Full Profile 和 Web Profile，用户可以根据自己的需要选择不同的功能集。

在此之前，Oracle 已经以 74 亿美金的价格收购了 Sun 公司，获得了 Java 商标和 Java 主导权。也收购了 BEA 公司，获得市场份额最大的应用服务器 Webogic。JavaEE 6 虽然是收购之后发布的版本，但主要的设计工作仍然由原 Sun 公司的 Java 专家完成。

2011 年 7 月，Oracle 发布 Java 7, 其中主要的特性是 NIO2 和 Fork/Join 并发包，尽管语言上没有大的增强，但我个人认为，自从 Oracle JDK（包括 OpenJDK7），Java 虚拟机的稳定性真正做到的工业级，成为一个计算平台而服务于全世界。

2013 年 6 月，Oracle 发布 JavaEE 7，这个版本加入了 Websocket，Batch 的支持，并且引入 Concurrency 来对服务器多线程进行管控。然而所有的子规范，算上可选项 (Optional) 总共有 40 多项，开发者光是阅读

规范文本就很吃力了，更不要说能够全局精通掌握。JavaEE 规范的本质是企业级应用设计的经验凝结，每一个 API 都经过众多丰富经验的专家反复商议并确定。各个版本之间可以做到向后兼容，也就是说，即使是 10 年前写的 Servlet 程序，当前的开发者也可以流畅的阅读源码，经过部分代码调整和配置修改，可以部署在当今的应用服务器上。反过来，今后用 Servlet4 写的程序，浏览器和服务器通信使用全新的 HTTP/2 协议，但程序员在理解上不会有障碍，就是因为 Servlet 规范的 API 非常稳定，基本没有大的变化修改。

2014 年 3 月，Oracle 发布 Java 8，这个版本是我认为的第三个有里程碑意义的 Java 版本。其中最引人注目的便是 Lambda 表达式了，从此 Java 语言原生提供了函数式编程能力。语言方面大的特性增加还有：Streams，Date/Time API, 新的 Javascript 引擎 Nashorn，集合的并行计算支持等，Java8 更加适应海量云计算的需要。

按照原来的计划，Java9 应该在今年 7 月发布，但因为模块化 (JPMS) 投票未通过的原因，推迟到今年 9 月份发布。

JavaEE 8 也会在今年发布，预计的时间在 8-10 月。其中最主要更新是 Servlet 4.0 和 CDI 2.0，后者已经完成最终规范的发布和投票。

Java 社区情况介绍

我们按照两个方面介绍 Java 社区情况。

Java User Group(JUG，Java 用户组）目前全世界范围有 100 多个 JUG 组织，分布在各个大洲各个国家，一般来说以地域命名。目前最有影响力的两个 JUG 分别是伦敦的 LJC(London Java Community) 和巴西的 SouJava，目前都是 JCP 的 EC（执行委员会）成员。国内目前有 GreenTea JUG（北京和杭州），Shanghai JUG，GuangDong JUG, Shenzhen JUG, Nanjing JUG 等。GreanTeaJUG 以阿里巴巴研发部门成员为核心，包括北京和杭州两地各个公司从事 Java 开发的研发人员，过去几年成功举办了很多有业界影响力的活动，特别是邀请到众多国外的 Java 技术专家来分

享知识，目前是国内最大的JUG开发者组织。

Java开源社区 Java是一门开放的语言，其开源社区也是参与者众多。最有名的应当数Apache社区，目前已经拥有近200个顶级项目，其中绝大多数是Java语言项目。在Java生态圈中，具有重要地位的如Ant、Commons、Tomcat、Xerces、Maven、 Struts、Lucene、ActiveMQ、CXF、Camel、Hadoop等等。很多技术时代，一大批Java项目加入，如Web时代的Velocity、Wicket；JavaEE相关的Tomee、OpenJPA、OpenWebBeans、Myfaces；WebService时代的jUDDI、Axis、ServiceMix；Osgi时期的Flex、Karaf；大数据时代的HBase、Hive、ZooKeeper、Cassandra；云时代的Mesos、CloudStack等等。

涉及到软件开发的方方面面，可以说当今几乎所有的中型以上Java应用中，都会有Apache开源项目的身影。国内最早参与Apache社区的以国外软件公司国内研发团队成员为主，如红帽、IONA、Intel、IBM研发中心等。如今国内互联网公司和软件公司也不断的参与，特别是开始主导一些Apache项目，如Kylin等。

JBoss开源社区，包含了50多个Java开源项目，其中有Hibernate、Drools、jBPM等业界知名开源项目，也有Undertow、Byteman、Narayana等名气不算大，但绝对是相应领域业界的顶级优秀项目。当前JBoss开源社区主要以企业应用中间件软件为主，RedHat是主要的技术贡献力量。

Eclipse开源社区，之前主要是包含Eclipse IDE的项目，后来也逐步进行多方面的扩展，比如OSGi，服务器等，目前一些知名Java项目，如Jetty、Vertx等都是Eclipse开源组织成员。此外IOT目前是Eclispe的一个重点方向，在这里可以找到完整的IOT Java开发方案。

Spring开源社区，以SpringFramework为核心，包括SpringBoot、SpringCloud、SpringSecurity、SpringXD等开源项目，在国内有广泛的应用场景。

目前大的玩家

Java 语言和品牌都是 Oracle 公司所有，所以 Oracle 公司是 Java 最主要的厂商。绝大多数 JSR(Java 规范提案）的领导者都是 Oracle 的雇员。

Java 是一个庞大的生态圈，全世界的软件和互联网公司绝大多数都是 Java 用户，同时也可以参与推动 Java 语言的发展。任何组织或者个人都可以加入 JCP（Java Community Process），并提交 JSR 来给 JavaSE，JavaEE，JavaME 等提交新的 API 或者服务定义。Java 拥有当今最完备的语言生态，几乎所有能想到的应用范围，都有软件厂商提出过标准化的构想，其中很多已经被接纳为 JSR 提案。如今 JSR 总数已经都 400 多个。

JCP 是发展 Java 的国际组织，其中的执行委员会（EC）以投票的形式对 JSR 提案进行表决。目前 EC 包括 16 个合约 (Ratified) 席位，6 个选举 (Elected) 席位和 2 个合伙 (Associate) 席位，以及 Oracle 作为所有者的永久席位。非永久席位每两年重新选举一次，每次选举为 24 个席位的一半，即为 12 个。

当前 EC 委员会中，对于 Java 起到最重要作用的，无疑是 Oracle，IBM 和 Redhat 三家公司。Oracle 自然不用说；Redhat 领导着 JavaEE8 中两项 JSR，并且在操作系统，Linux，虚拟化，云计算等基础软件方面是产品领导者；IBM 是软硬件最大的厂商，拥有自己的 Unix 操作系统和 JDK 版本。这三家软件厂商也是中间件厂商的强者，它们对于 Java 的影响是至关重要的。前不久投票被否决的 JSR 376(JPMS) 模块化提案，就是 Redhat 和 IBM 先后表示要投反对票，最后才没有通过的。

另外的几个重要的 Java 参与方分别包括：巨型互联网公司，以 Twitter 为代表；大型金融公司，以高盛，瑞信为代表；强大的硬件产商，Intel，NXP，Gemalto 等；大型系统方案厂商，HP， Fijitsu；当然还有掌握先进 Java 技术的公司，如 Azul，Hazelcast，Tomitribe，Jetrains 等等。这些公司共同对 Java 的发展起到关键作用。

GC 方面的进展

JDK 中主要的 GC 分类有：

- Serial，单线程进行 GC，在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。
- Parallel，相比 Serial 收集器，Parallel 最主要的优势在于使用多线程去完成垃圾清理工作，这样可以充分利用多核的特性，大幅降低 GC 时间。
- CMS(Concurrent Mark-Sweep)，是以牺牲吞吐量为代价来获得最短回收停顿时间的垃圾回收器。实现 GC 线程和应用线程并发工作，不需要暂停所有应用线程。
- G1(Garbage First Garbage Collector)，G 设计初衷是为了尽量缩短处理超大堆（大于 4GB）时产生的停顿。相对于 CMS 的优势而言是内存碎片的产生率大大降低。

目前在 JDK8 中以上 4 种 GC 都可以使用，而在 JDK9 中 G1 GC 会成为默认的垃圾收集器。

在 OpenJDK 方面，Redhat 开源并贡献了 Shenandoah GC。这是一种新的 Java 虚拟机 GC 算法，目标是利用现代多核 CPU 的优势，减少大堆内存在 GC 处理时产生的停顿时间。在使用大内存的应用上使用，如 >20G 堆空间。Fedora24 以后，官方源中的 OpenJDK 即带有 Shenandoah 算法，不过 JDK9 中还不会被加入。

无停顿的高性能 GC 就是 Azul 公司的 C4(Continuously Concurrent Compacting Collector) GC 了，但只提供商业版本使用。

另外 IBM J9 中 Balanced GC，表现也很出色，能够保证相对一致的暂停时间而避免破坏性的长时间停顿。Balanced GC 应用在各类 IBM 中间件产品之中。

Java 9 目前已经可以确认的特性介绍

Java9 中，最受人关注的新特性就是 Jigsaw 项目带来的模块化技术特性。

Java 语言一直缺乏语言级别的模块化能力，目前模块化技术通过 OSGi, JBoss Modules 等项目，已经在服务端程序得到了广泛的应用。Java 在语言级别引入模块化能力，将极大的促进 Java 应用程序组件化，模块化的改变。应用程序通过模块化拆分，可以做到更灵活的引入，加载，移除组件，占用更少的内存，更适合云计算时代的要求。在 JDK9 EA（预览版）中，原有的 rt.jar 已经被划分为若干了 jmod，通过模块内的 module-info.java 文件来声明模块间的引用关系。

然而，模块化改造是个渐进而适度的过程，Java9 为了可兼容 Java8 以前应用程序的运行，做出很多的让步，模块定义严格性没有那么苛刻。各个厂商也有对自己现有系统可无缝运行在 Java9 上的商业诉求。Java 模块化提案还得花更多的时间去讨论和修改。

Java9 中的 jshell 工具实现了 REPL，即读取，求值，打印，循环。这个工具可以使得开发者交互式的使用 Java，方便于系统管理，调试，使用。可以想像到有了 jshell 后，Java 语言更加适合初学者入门学习。

Jlink 工具和 AOT（预先编译技术）。一直以来，Java 运行方式是把程序编译成 class 文件，然后通过 jvm 运行的。这种工作方式可以做到跨平台移植，在互联网时代初期，各种 Unix 繁荣和 Windows 在桌面的一统局面下，对于占据市场起到决定性作用。

然而到了今天，无论是大型互联网公司还是企业内部，x86 平台 64 位服务器已经成为主要的选择。从运行效率考虑，可以把 java 程序编译成可执行的二进制文件，更加适应云计算和容器技术发展的需要。

利用 jlink/jaotc 工具，可以把一个 Java 程序编译成可执行文件，在 Java9 推出时，可能只有 java.base 模块支持 AOT。

安全方面的加强。引入新的摘要算法 SHA-3，内置 ALPN 使得更好的支持 HTTP/2 协议，提供 DTLS（数据包传输层安全性协议），可以保证 UDP 数据传输的安全，PKCS12 格式替代原有的 JKS 成为 keystore 的默认格式。

此外，统一 JVM 日志 (Unified JVM Logging)，多版本共存 jar(Multi-

release jar files)，接口内部的私有方法 (Interface provate method) 等也是非常重要的新特性。

与其他语言的对比，Java 的优势

Java 是最好的语言么？不是，因为在每个领域都有更合适的编程语言。

C 语言无疑是现代计算机软件编程语言的王者，几乎所有的操作系统都是 C 语言写成的。C++ 是面向对象的 C 语言，一直在不断的改进。

JavaScript 是能运行在浏览器中的语言，丰富的前端界面离不开 Javascript 的功劳。近年来的 Node.js 又在后端占有一席之地。Python 用于系统管理，并通过高性能预编译的库，提供 API 来进行科学计算，文本处理等，是 Linux 必选的解释性语言。

Ruby 强于 DSL（领域特定语言），程序员可以定义丰富的语义来充分表达自己的思想。Erlang 就是为分布式计算设计的，能保证在大规模并发访问的情况下，保持强壮和稳定性。Go 语言内置了并发能力，可以编译成本地代码。当前新的网络相关项目，很大比例是由 Go 语言编写的，如 Docker、Kubernetes 等。

编写网页用 PHP，函数式编程有 Lisp，编写 iOS 程序有 Swift/ObjectiveC。

一句话概括，能留在排行榜之上的语言，都是好的语言，在其所在的领域能做到最好。

那么，Java 语言到底有什么优势可以占据排行榜第一的位置呢？

其一，语法比较简单，学过计算机编程的开发者都能快速上手。

其二，在若干了领域都有很强的竞争力，比如服务端编程，高性能网络程序，企业软件事务处理，分布式计算，Android 移动终端应用开发等等。

最重要的一点是符合工程学的需求，我们知道现代软件都是协同开发，那么代码可维护性，编译时检查，较为高效的运行效率，跨平台能力，丰富的 IDE，测试，项目管理工具配合。都使得 Java 成为企业软件公司的

首选，也得到很多互联网公司的青睐。

没有短板，容易从市场上找到 Java 软件工程师，软件公司选择 Java 作为主要开发语言，再在特定的领域使用其他语言协作编程，这样的组合选择，肯定是不会有大的问题。

所以综合而言，Java 语言全能方面是最好的。

Java 未来方向的展望

如今的 Java，已经在功能上相当丰富了，Java 8 加入 Lambda 特性，Java 9 加入模块化特性之后，重要的语言特性似乎已经都纳入进来。如果要说值得考虑的一些功能，我觉得有以下几点：

模块化改造完毕之后，可能会出现更多专业的 JDK 发行软件商，提供在功能方面，比如针对于分布式计算，机器学习，图形计算等，纳入相关的功能库作为文件。这样专业行业客户可以选择经过充分优化后的 JDK 版本。

Java 语义上对“模式匹配”有更强的支持，如今的 switch 语句能力还是比较欠缺，可以向 Erlang， Scala 等语言借鉴。

多线程并发处理，Java 做的已经很好了。不过我个人觉得可以在多进程多线程配合，以及语言级别数据管道表示上，可以进行改造和优化。

JDK9 会有 HTTP/2 client 端的能力，但毫无疑问会有更多更好的三方库出现，JDK 可以和这些三方库通力合作，提供一个更好 API 界面和 SPI 参考实现。

目前 Java 在云计算方面遇到的最大问题还是占用内存过大。我个人认为从两个方面来看：

如果该应用的确是长时间运行的服务，可以考虑结构清晰的单体结构，算下来总的内存消耗并不会比多个微服务进程占用的更多。

微服务应用，未来可以采用编译成本地代码的方式，并使用优化过的三方库，甚至本地 so 文件，减少单个进程的过多内存占用。

安全框架更加清晰，SPI 可以允许三方库提供更强大更高效的安全功

能。

JavaEE 方向则有更多的改进的地方：

EJB 重构目前的 Corba 分布通信基础，参考 gRPC 进行远程系统调用。

分解 EJB 规范，把 JVM 进程相关的特性，如注入 / 加强 / 事务 / 安全都统一到 CDI 规范中；对 EJB 进行裁剪，保留远程访问特性和作为独立执行主体分布式对象能力。

加强 JMS 和 MDB，媲美 Akka 目前的能力。

JaxRS 适度优化，不必要依赖 Servlet，或者适度调整，来提供更大的能力。

JPA 借鉴 JDO，以及融入一部分特性，做到对 NoSQL 更良好的支持。

一些个人的心得和经验分享

软件业有个 Hype Cycle 模型，有很多技术受到市场的追捧而成为明星，也有些身不逢时而备受冷漠。

EJB 是一个广泛被误解的技术，在企业应用分布式计算方面，EJB 给出了非常完备的技术体系。只是目前所有的应用服务器都实现的不够好。对于目前打算转型微服务设计的架构师，EJB 也是一个非常值得学习借鉴的技术。

Java 的慢是相对的，有些是当前实现的不够好。比如原来有人对 Java 的网络 IO 性能提出质疑，然而稳定的 Netty 框架出现后，就没有人再怀疑 Java 处理网络 IO 的能力了，甚至在 JDK8 中自身的 NIO 也相当出色。要知道 Java 为了实现跨平台能力，采用的是各个操作系统的一个公共能力子集，而且其设计哲学就是给出 API 框架，实现是可以自行实现和加载服务的。

Java 在处理界面方面，Swing 和 Swt 表现可圈可点（Idea 和 Eclipse 分别采用的图形基础库），JavaFX 已经运用到很多的行业软件上。在浏览器界面表现上，SpringMVC 在模板渲染页面方面使用者最多；GWT 似乎使用者不多，但基于 GWT 的 Vaddin 在国外企业中用户众多，而且很

多服务器管理软件也用 GWT 写成；JSF 也在企业软件中得到广泛使用，状态信息直接在后端进行管理，配合 js 前端框架，可以充分发挥各种技术的优势。

CDI 规范和 SpringFramework 在服务器程序中作用类似，Spring 是一套设计优良，完备的框架，CDI 具有更强的可扩展性。通过对注解的语义定义，一家公司可以维护一套自己的组件描述语言，来做到产品和项目之间的软件快速复用。CDI 是定义软件组件内部模型的最佳方式，只可惜了解的软件工程师实在太少。

微服务架构在互联网应用，快速开发运维管理方面，配合容器技术使用，有很强的优势。但并不是所有的应用场景都适合微服务：强事务应用系统，采用单体结构的软件体系设计，更容易从整体方面维护，也能获得更优的性能。Java 语言无论在微服务还是单体结构，都有成熟稳定的软件架构供选择使用。

作者介绍

张建锋，永源中间件共同创始人，原红帽公司 JBoss 应用服务器核心开发组成员。毕业于北京邮电大学和清华大学，曾供职于金山软件，IONA 科技公司和红帽软件。对于 JavaEE 的各项规范比较熟悉；开源技术爱好者，喜欢接触各类开源项目，学习优秀之处并加以借鉴，认为阅读好的源码就和阅读一本好书一样让人感到愉悦；在分布式计算，企业应用设计，移动行业应用，Devops 等技术领域有丰富的实战经验和自己的见解；愿意思考软件背后蕴涵的管理思想，认为软件技术是一种高效管理的实现方式，有志于将管理学和软件开发进行结合。

Python 向来以慢著称，为啥 Instagram 却唯独钟爱它?

作者 朱雷

PyCon 是全世界最大的以 Python 编程语言 为主题的技术大会，大会由 Python 社区组织，每年举办一次。在 Python 2017 上，Instagram 的工程师们带来了一个有关 Python 在 Instagram 的主题演讲，同时还分享了 Instagram 如何将整个项目运行环境升级到 Python 3 的故事。

Instagram 是一款移动端的照片与视频分享软件，由 Kevin Systrom 和 Mike Krieger 在 2010 年创办。Instagram 在发布后开始快速流行。于 2012 年被 Facebook 以 10 亿美元的价格收购。而当时 Instagram 的员工仅有区区 13 名。

如今，Instagram 的总注册用户达到 30 亿，月活用户超过 7 亿 （作

为对比，微信最新披露的月活跃用户为 9.38 亿）。而令人吃惊的是，这么高的访问量背后，竟完全是由以速度慢著称的 Python + Django 支撑。

为什么选择 Python 和 Django

Instagram 选择 Django 的原因很简单，Instagram 的两位创始人 (Kevin Systrom and Mike Krieger) 都是产品经理出身。在他们想要创造 Instagram 时，Django 是他们所知道的最稳定和成熟的技术之一。

时至今日，即使已经拥有超过 30 亿的注册用户。Instagram 仍然是 Python 和 Django 的重度使用者。Instagram 的工程师 Hui Ding 说到：“一直到用户 ID 已经超过了 32bit int 的限额（约为 20 亿），Django 本身仍然没有成为我们的瓶颈所在。”

不过，除了使用 Django 的原生功能外，Instagram 还对 Django 做了很多定制化工作：

- 扩展 Django Models 使其支持 Sharding （一种数据库分片技术）。
- 手动关闭 GC（垃圾回收）来提升 Python 内存管理效率，他们同样也写过一篇博客来说明这件事情：Dismissing Python Garbage Collection at Instagram。
- 在位于不同地理位置的多个数据中心部署整套系统。

Python 语言的优势所在

Instagram 的联合创始人 Mike Krieger 说过：『我们的用户根本不关心 Instagram 使用了哪种关系数据库，他们当然也不关心 Instagram 是用什么编程语言开发的。』

所以，Python 这种简单而且实用至上的编程语言最终赢得了 Instagram 的青睐。他们认为，使用 Python 这种简单的语言有助于塑造 Instagram 的工程师文化，那就是：

- 专注于定位问题、解决问题，而不是工具本身的各种花花绿绿的

特性；

- 使用那些经过市场验证过的成熟技术方案，而不用被工具本身的问题所烦扰；
- 用户至上：专注于用户所能看到的新特性，为用户带去价值。

但是，即使使用 Python 语言有这么多好处，它还是很慢，不是吗？

不过，这对于 Instagram 不是问题，因为他们认为：“Instagram 的最大瓶颈在于开发效率，而不是代码的执行效率。”

At Instagram, our bottleneck is development velocity, not pure code execution.

所以，最终的结论是：你完全可以使用 Python 语言来实现一个超过几十亿用户使用的产品，而根本不用担心语言或框架本身的性能瓶颈。

如何提升运行效率

但是，即使是选用了拥有诸多好处的 Python 和 Django。在 Instagram 的用户数迅速增长的过程中，性能问题还是出现了：服务器数量的增长率已经慢慢的超过了用户增长率。Instagram 是怎么应对这个问题的呢？

他们使用了这些手段来缓解性能问题：

- 开发工具来帮助调优：Instagram 开发了很多涵盖各个层面的工具，来帮助他们进行性能调优以及找到性能瓶颈。
- 使用 C/C++ 来重写部分组件：把那些稳定而且对性能最敏感的组件，使用 C 或 C++ 来重写，比如访问 memcache 的 library。
- 使用 Cython：Cython 也是他们用来提升 Python 效率的法宝之一。

除了上面这些手段，他们还在探索异步 IO 以及新的 Python Runtime 所能带来的性能可能性。

为什么要升级到 Python 3

在相当长的一段时间，Instagram 都跑在 Python 2.7 + Django 1.3 的组

合之上。在这个已经落后社区很多年的环境上，他们的工程师们还打了非常非常多的小 patch。难道他们要被永远卡在这个版本上吗?

所以，在经过一系列的讨论后，他们最终做出一个重大的决定：升级到 Python 3 ！！

事实上，Instagram 目前已经完成了将运行环境迁移到 Python 3 的工作 - 他们的整套服务已经在 Python 3 上跑了好几个月了。那么他们是怎么做到的呢？接下来便是由 Instagram 工程师 Lisa guo 带来的 Instagram 如何迁移到 Python 3 的故事。

对于 Instagram 来说，下面这些因素是推动他们将运行环境迁移到 Python 3 的主要原因：

1. 新特性：类型注解 Type Annotations

看看下面这段代码：

```
def compose_from_max_id(max_id):
    '''@param str max_id'''
```

图中函数的 max_id 参数究竟是什么类型呢？ int ？ tuple ？ 或是 list? 等等，函数文档里面说它是 str 类型。

但随着时间推移，万一这个参数的类型发生变化了呢？如果某位粗心的工程师修改代码的同时忘了更新文档，那就会给函数的使用者带来很大麻烦，最终还不如没有注释呢。

2. 性能

Instagram 的整个 Django Stack 都跑在 uwsgi 之上，全部使用了同步的网络 IO。这意味着同一个 uwsgi 进程在同一时间只能接收并处理一个请求。这让如何调优每台机器上应该运行的 uwsgi 进程数成了一个麻烦事:

为了更好利用 CPU，使用更多的进程数？但那样会消耗大量的内存。而过少的进程数量又会导致 CPU 不能被充分利用。

为此，他们决定跳过 Python 2 中哪些蹩脚的异步 IO 实现 （可怜的 gevent、tornado、twisted 众），直接升级到 Python 3，去探索标准库中的

asyncio 模块所能带来的可能性。

3. 社区

因为 Python 社区已经停止了对 Python 2 的支持。如果把整个运行环境升级到 Python 3，Instagram 的工程师们就能和 Python 社区走的更近，可以更好的把他们的工作回馈给社区。

迁移方案

在 Instagram，进行 Python 3 的迁移需要必须满足两个前提条件：

- 不停机，不能有任何的服务因此不可用
- 不能影响产品新特性的开发

但是，在 Instagram 的开发环境中，要满足上面这两点来完成迁移到 Python 3.6 这种庞大的工程是非常困难的。

基于主分支的开发流程

即便使用了以多分支功能著称的 git，Instagram 所有的开发工作都是主要在 master 分支上进行的，Instagram 所奉行的开发哲学是：『不管是多大的新特性或代码重构，都应该拆解成较小的 Commit 来进行。』

那些被合并进 master 分支的代码，都将在一个小时内被发布到线上环境。而这样的发布过程每天将会发生上百次。在这么频繁的发布频率下，如何在满足之前的那两个前提下来完成迁移变得尤其困难。

被弃用的迁移方案

创建一个新分支

很多人在处理这类问题时，第一个蹦进脑子的想法就是：『让我们创建一个分支，当我们开发完后，再把分支合并进来』。但在 Instagram 这么高的迭代频率上，使用一个独立分支并不是好主意：

- Instagram 的 Codebase 每天都在频繁更新，在开发 Python 3 分支的过程中，让新分支与现有 master 分支保持同步开销极大，同时

极易出错；

- 最终将 Python 3 分支这个改动非常多的分支合并回 Master 拥有非常高的风险；
- 只有少数几个工程师在 Python 3 分支上专职负责升级工作，其他想帮助迁移工作的工程师无法参与进来。

挨个替换接口

还有一个方案就是，挨个替换 Instagram 的 API 接口。但是 Instagram 的不同接口共享着很多通用模块。这个方案要实施起来也非常困难。

微服务

还有一个方案就是将 Instagram 改造成微服务架构。通过将那些通用模块重写成 Python 3 版本的微服务来一步步完成迁移工作。

但是这个方案需要重新组织海量的代码。同时，当发生在进程内的函数调用变成 RPC 后，整个站点的延迟会变大。此外，更多的微服务也会引入更高的部署复杂度。

所以，既然 Instagram 的开发哲学是：小步前进，快速迭代。他们最终决定的方案是：一步一步来，最终让 master 分支上的代码同时兼容 Python 2 和 Python 3 。

正式迁移到 Python 3

既然要让整个 codebase 同时兼容 Python 2 和 Python 3，那么首先要符合这点的就是那些被大量使用的第三方 package。针对第三方 package，Instagram 做到了下面几点：

- 拒绝引入所有不兼容 Python 3 的新 package；
- 去掉所有不再使用的 package；
- 替换那些不兼容 Python 3 的 package。

在代码的迁移过程中，他们使用了工具 modernize 来帮助他们。

使用 modernize 时，有一个小技巧：每次修复多个文件的一个兼容

问题，而不是一下修复一个文件中的多个兼容问题。这样可以让 Code Review 过程简单很多，因为 Reviewer 每次只需要关注一个问题。

对于 Python 这种灵活性极强的动态语言来说，除了真正去执行代码外，几乎没有其他比较好的检查代码错误的手段。

前面提到，Instagram 所有被合并到 master 的代码提交会在一个小时内上线到线上环境，但这不是没有前提条件的。在上线前，所有的提交都需要通过成千上万个单元测试。

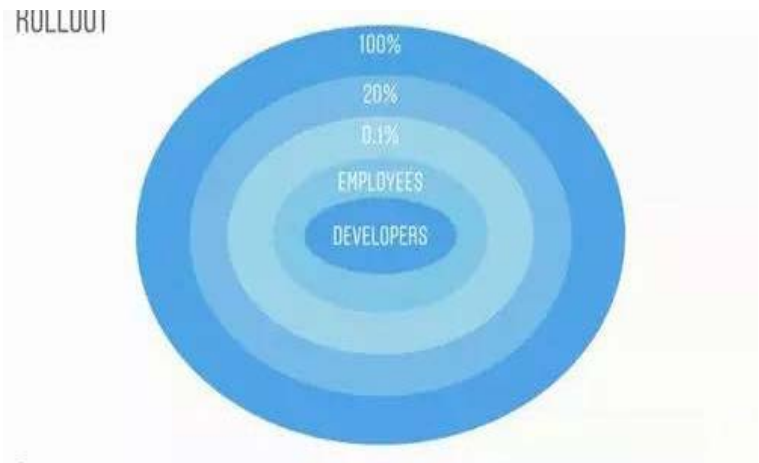
于是，他们开始加入 Python 3 来执行所有的单元测试。一开始，只有极少数的单元测试能够在 Python 3 环境下通过，但随着 Instagram 的工程师们不断的修复那些失败的单元测试，最终所有的单元测试都可以在 Python 3 环境下成功执行。

但是，单元测试也是有局限性的：

- Instagram 的单元测试没有做到 100% 的代码覆盖率；
- 很多第三方模块都使用了 mock 技术，而 mock 的行为与真实的线上服务可能会有所不同。

所以，当所有的单元测试都被修复后，他们开始在线上正式使用 Python 3 来运行服务。

这个过程并不是一蹴而就的。首先，所有的 Instagram 工程师开始访问到这些使用 Python 3 来执行的新服务，然后是 Facebook 的所有雇员，随后是 0.1%、20% 的用户，最终 Python 3 覆盖到了所有的 Instagram 用户。



迁移过程的技术问题

Instagram 在迁移到 Python 3 时碰到很多问题，下面是最典型的几个：

Unicode 相关的字符串问题

Python 3 相比 Python 2 最大的改动之一，就是在语言内部对 unicode 的处理。

在 Python 2 中，文本类型 （也就是 unicode） 和二进制类型 （也就是 str）的边界非常模糊。很多函数的参数既可以是文本，也可以是二进制。但是在 Python 3 中，文本类型和二进制类型的字符串被完全的区分开了。

于是，下面这段在 Python 2 下可以正常运行的代码在 Python 3 下就会报错：

```
mymac = hmac.new('abc')
TypeError: key: expected bytes or bytearray, but got 'str'
```

解决办法其实很简单，只要加上判断：如果 value 是文本类型，就将其转换为二进制。如下所示：

```
value = 'abc'
if isinstance(value, six.text_type):
    value = value.encode(encoding='utf-8')
mymac = hmac.new(value)
```

但是，在整个代码库中，像上面这样的情况非常多。作为开发人员，如果需要在调用每个函数时都要想想： 这里到底是应该编码成二进制，或者是解码成文本呢？ 将会是非常大的负担。

于是 Instagram 封装了一些名为 ensure_str()、ensure_binary()、ensure_text() 的帮助函数，开发人员只需对那些不确定类型的字符串，使用这些帮助函数先做一次转换就好。

```
mymac = hmac.new(ensure_binary('abc'))
```

不同 Python 版本的 pickle 差异

Instagram 的代码中大量使用了 pickle。比如用它序列化某个对象，然后将其存储在 memcache 中。如下面的代码所示：

```
memcache_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
data = pickle.loads(memcache_data)
```

问题在于，Python 2 与 Python 3 的 pickle 模块是有差别的。

如果上文的第一行代码，刚好是由 Python 3 运行的服务进行序列化后存入 memcache。而反序列化的过程却是由 Python 2 进行，那代码运行时就会出现下面的错误：

```
ValueError: unsupported pickle protocol: 4
```

这是由于在 Python 3 中，pickle.HIGHEST_PROTOCOL 的值为 4，而 Python 2 中的的 pickle 最高支持的版本号却是 2。那么如何解决这个问题呢？

Instagram 最终选择让 Python 2 和 Python 3 使用完全不同的 namespace 来访问 memcache。通过将二者的数据读写完全隔开来解决这个问题。

迭代器

在 Python 3 中，很多内置函数被修改成了只返成迭代器 Iterator：

```
map()
filter()
dict.items()
```

迭代器有诸多好处，最大的好处就是，使用迭代器不需要一次性分配大量内存，所以它的内存效率比较高。

但是迭代器有一个天然的特点，当你对某个迭代器做了一次迭代，访问完它的内容后，就没法再次访问那些内容了。迭代器中的所有内容都只能被访问一次。

在 Instagram 的 Python 3 迁移过程中，就因为迭代器的这个特性被坑了一次，看看下面这段代码：

```
CYTHON_SOURCES = [a.pyx, b.pyx, c.pyx]
builds = map(BuildProcess, CYTHON_SOURCES)
while any(not build.done() for build in builds):
    pending = [build for build in builds if not build.started()]
```

```
    <do some work>
```

这段代码的用处是挨个编译 Cython 源文件。当他们把运行环境切换到 Python 3 后，一个奇怪的问题出现了：CYTHON_SOURCES 中的第一个文件永远都被跳过了编译。为什么呢？

这都是迭代器的锅。在 Python 3 中，map() 函数不再返回整个 list，而是返回一个迭代器。

于是，当第二行代码生成 builds 这个迭代器后，第三行代码的 while 循环迭代了 builds，刚好取出了第一个元素。于是之后的 pending 对象便里面永远少了那第一个元素。

这个问题解决起来也挺简单的，你只要手动的吧 builds 转换成 list 就可以了：

```
builds = list(map(BuildProcess, CYTHON_SOURCES))
```

但是这类 bug 非常难定位到。如果用户的 feeds 里面永远少了那最新的第一条，用户很少会注意到。

字典的顺序

看看下面这段代码：

```
>>> testdict = {'a': 1, 'b': 2, 'c': 3}
>>> json.dumps(testdict)
```

它会输出什么结果呢？

```
# Python2
'{"a": 1, "c": 3, "b": 2}'
# Python 3.5.1
'{"c": 3, "b": 2, "a": 1}'    # or
'{"c": 3, "a": 1, "b": 2}'
# Python 3.6
'{"a": 1, "b": 2, "c": 3}'
```

在不同的 Python 版本下，这个 json dumps 的结果是完全不一样的。甚至在 3.5.1 中，它会完全随机的返回两个不同的结果。Instagram 有一段判断配置文件是否发生变动的模块，就是因为这个原因出了问题。

这个问题的解决办法是，在调用 json.dumps 传入 sort_keys=True 参数:

```
>>> json.dumps(testdict, sort_keys=True)
'{"a": 1, "b": 2, "c": 3}'
```

迁移到 Python 3.6 后的性能提升

当 Instagram 解决了这些奇奇怪怪的版本差异问题后，还有一个巨大的谜题困扰着他们：性能问题。

在 Instagram，他们使用两个主要指标来衡量他们的服务性能：

- 每次请求产生的 CPU 指令数（越低越好）
- 每秒能够处理的请求数（越高越好）

所以，当所有的迁移工作完成后，他们非常惊喜的发现：第一个性能指标，每次请求产生的 CPU 指令数居然足足下降了 12% ！！！

但是，按理说第二个指标 - 每秒请求数也应该获得接近 12% 的提升。不过最后的变化却是 0%。究竟是出了什么问题呢?

他们最终定位到，是由于不同 Python 版本下的内存优化配置不同，导致 CPU 指令数下降带来的性能提升被抵消了。那为什么不同 Python 版本下的内存优化配置会不一样呢?

这是他们用来检查 uwsgi 配置的代码:

```
if uwsgi.opt.get('optimize_mem', None) == 'True':
    optimize_mem()
```

注意到那段 ... ... == 'True' 了吗? 在 Python 3 中，这个条件判断总是不会被满足。问题就在于 unicode。在将代码中的 'True' 换成 b'True'（也就是将文本类型换成二进制，这种判断在 Python 2 中完全不区分的）后，问题解决了。

所以，最终因为加上了一个小小的字母 'b'，程序的整体性能提升了 12%。

完美切换

在今年二月份，Instagram 的后端代码的运行环境完全切换到了

Python 3 下：

当所有的代码都都迁移到 Python 3 运行环境后：

- 节约了 12% 的整体 CPU 使用率（Django/uwsgi）
- 节约了 30% 的内存使用（celery）

同时，在整个迁移期间，Instagram 的月活用户经历了从 4 亿到 6 亿的巨大增长。产品也发布了评论过滤、直播等非常多新功能。

那么，那几个最开始驱动他们迁移到 Python 3 的目的呢？

- 类型注解：Instagram 的整个 codebase 里已经有 2% 的代码添加上了类型注解，同时他们还开发了一些工具来辅助开发者添加类型提示。
- asyncio：他们在单个接口中利用 asynio 平行的去做多件事情，最终降低了 20-30% 的请求延迟。
- 社区：他们与 Intel 的工程师联合，帮助他们更好的对 CPU 利用率进行调优。同时还开发了很多新的工具，帮助他们进行性能调优。

Instagram 带给我们的启示

Instagram 的演讲视频时间不长，但是内容很丰富，在编写此文前，我完全没有想到最终的文章会这么长。

那么，Instagram 的视频可以给我们哪些启示呢?

- Python + Django 的组合完全可以负载用户数以 10 亿记的服务，如果你正准备开始一个项目，放心使用 Python 吧！
- 完善的单元测试对于复杂项目是非常有必要的。如果没有那『成千上万的单元测试』。很难想象 Instagram 的迁移项目可以成功进行下去。
- 开发者和同事也是你的产品用户，利用好他们。用他们为你的新特性发布前多一道测试。
- **完全基于主分支的开发流程，可以给你更快的迭代速度。前提是拥有完善的单元测试和持续部署流程。**
- Python 3 是大势所趋，如果你正准备开始一个新项目，无需迟疑，拥抱 Python 3 吧！

好了，就到这儿吧。Happy Hacking ！

推荐文章

是的，Python 比较慢，但我不在乎：牺牲性能以提升工作效率

我们为什么要选择小众语言 Rust 来实现 TiKV？

作者　黄东旭

Rust 是什么?

Rust 是由 Mozilla 研究室主导开发的一门现代系统编程语言，自 2015 年 5 月发布 1.0 之后，一直以每 6 周一个小版本的开发进度稳定向前推进。语言设计上跟 C++ 一样强调零开销抽象和 RAII。拥有极小的运行时和高效的 C 绑定，使其运行效率与 C/C++ 一个级别，非常适合对性能要求较高的系统编程领域。利用强大的类型系统和独特的生命周期管理实现了编译期内存管理，保证内存安全和线程安全的同时使编译后的程序运行速度极快，Rust 还提供函数式编程语言的模式匹配和类型推导，让程序写起来更简洁优雅。宏和基于 trait 的泛型机制让 Rust 的拥有非常强大的抽象能力，在实际工程中尤其是库的编写过程中可以少写很多 boilerplate 代码。

Rust 的生态

Rust 由于有 Cargo 这样一个非常出色的包管理工具，周边的第三方库发展非常迅速，各个领域都有比较成熟的库，比如 HTTP 库有 Hyper，异步 IO 库有 Tokio, mio 等，基本上构建后端应用必须的库 Rust 都已经比较齐备。 总体来说，现阶段 Rust 定位的方向还是高性能服务器端程序开发，另外类型系统和语法层面上的创新也使得其可以作为开发 DSL 的利器。

Rust 的使用情况

Rust 作为一种新锐的语言，具备其独有的优越性，目前在全球落地的项目中比较知名的比如，Dropbox 的后端分布式存储系统（闭源），Firefox 的新的内核 Servo，操作系统 Redox，当让包括 PingCAP 的分布式数据库 TiDB 的存储层 TiKV，TiKV 作为其中的一员，自上线以来非常引人注目，在 GitHub Rust 语言的全球排名项目中，基本上一直徘徊在前几名的状态。

TiKV 是一个事务型分布式 Key-Value 数据库，是作为 TiDB 项目的核心存储组件，也是 Google 著名的分布式数据库 Spanner 的开源实现。对于这样一个大型的分布式存储项目，在 TiKV 的开发语言选择上，我们选择了 Rust 语言从零构建。在今天这个文章里，我想详细聊聊什么驱动我们选择了 Rust。

对于数据库这类基础软件来说，在过去很长的一段时间，我们能选择的编程语言基本只有 C/C++。Java 和 Go 这类编程语言的主要问题还是由于 GC 引起抖动，尤其在读写压力比较大的情况下。另外一方面，对于 Go 来说，一个非常吸引人的特性是轻量级线程 Goroutine，对于开发者来说极大的降低的开发并发程序的复杂度，但是相应的代价是 Goroutine 的 Runtime 会带来额外的上下文切换开销。 对于数据库这样的基础软件，性能显然是很重要的，另一方面，系统需要尽可能的保持 ”确定性”，在性能优化和调试阶段能够更加方便，但是引入 GC 和另一个 Runtime 的问题是

会增加这种不确定性，所以在很长的时间内，C/C++ 是唯一的选择。

TiKV 这个项目起始于 2015 年底，当时也是在 Pure Go / Go + Cgo / C++11 / Rust 几个语言之间纠结，虽然我们的核心团队有大量的 Go 语言开发经验，另外 TiDB 的 SQL 层是完全采用 Go 语言开发，Go 带来的开发效率的极大提升也让我们受益良多，但是在存储层的选型上，我们首先排除的就是 Pure Go 的选项，理由也很简单，在底层，我们已经决定接入 RocksDB，RocksDB 本身就是个 C++ 的项目，而 Go 的 LSM-Tree 的实现大多成熟度不太够没有能和 RocksDB 相提并论的项目，如果选 Go 的话，只能选择用 Cgo 来 bridge，但是当时 Cgo 的问题同样明显，在 2015 年底，在 Go code 里调用 Cgo 的性能损失比较大，并不是在 Goroutine 所在的线程直接 Call cgo 的代码，而且对于数据库来说，调用底层的存储引擎库是很频繁的，如果每次调用 RocksDB 的函数都需要这些额外的开销的话，非常不划算，当然也可以通过一些技巧增大 Cgo 这边的调用的吞吐，比如一段时间内的调用打包成一个 cgo batch call，通过增加单个请求的延迟来增大的整体的吞吐，抹平 cgo 调用本身的开销，但是这样一来，实现就会变得非常复杂，另一方面，GC 的问题仍然没有办法彻底的解决，在存储层我们希望尽可能高效的利用内存，大量使用 syscall.Mmap 或者对象复用这些有些 hacky 的技巧，会让整体的代码可读性降低，我的判断仍然是得不偿失。

所以后来认真在考虑的只剩下 Rust / C++11，先说 C++11 的问题，其实 C++11 也没啥问题，性能上肯定没问题，RocksDB 是 C++11 写的，在纠结了一小段时间后，我们认真评估了一下我们的团队背景和要做的东西，最后还是没有选择 C++，原因主要是：

- 我们核心团队过去都是 C++ 的重度开发者，也基本都有维护过大型 C++ 项目的经历，每个人都有点心里阴影… 悬挂指针、内存泄漏、Data race 在项目越来越大的过程中几乎很难避免，当然你可以说靠老司机带路，严格 Code Review 和编码规范可以将问题发生的概率降低，但是一旦出现问题，Debug 的成本很高，心智负

担很重，而且第三方库不满足规范怎么办。

- C++ 的编程范式太多，而且差异很大，又有很多奇技淫巧，统一风格同样也需要额外的学习成本，特别是团队的成员在不断的增加，不一定所有人都是 C++ 老司机，特别是大家这么多年了都已经习惯了 GC 的帮助，已经很难回到手动管理内存的时代。
- 缺乏包管理，集成构建等现代化的周边工具，虽然这点看上去没那么重要，但是对于一个大型项目这些自动化工具是极其重要的，直接关系到大家的开发效率和项目的迭代的速度。而且 C++ 的第三方库参差不齐，很多轮子得自己造。

Rust 在 2015 年底已经发布了 1.0，Rust 有几点特性非常吸引我们：

- 内存安全性
- 高性能 (得益于 llvm 的优秀能力，运行时实际上和 C++ 几乎没区别)，与 C/C++ 的包的亲缘性
- 强大的包管理和构建工具 Cargo
- 更现代的语法
- 和 C++ 几乎一致的调试调优体验，之前熟悉的工具比如 perf 之类的都可以直接复用
- FFI，可以无损失的链接和调用 RocksDB 的 C API

其中放在第一位的是安全性，之前在 C++ 中提到的内存管理和避免 Data race 的问题，我相信虽然靠老司机是可以解决，但是仍然没有在编译器层面上强约束，把问题扼杀在摇篮之中解决的彻底，Rust 这点非常符合我的口味，对于大型项目来说，永远不要把软件的质量押宝在人身上，人永远会犯错，人是不稳定的。Rust 的做法虽然增加了上手的门槛，但是我认为是值得的。另一方面，Rust 又是一个非常现代化的编程语言，现代的类型系统，模式匹配，功能强大的宏，trait 等在熟悉以后会很大的提升开发效率，其实如果选择 C++ 的话，算上 Debug 的时间，Rust 的开发效率不算低，在我们的实践经验中，我们的工程师从零开始接触 Rust，到能够高效进行开发的时间大概是 1 个月，而且熟练的工程师的 Rust 开发效

率几乎是和写 Go 差不多的。

小结

总体来说，Rust 这门新兴的语言对于国内大多数开发者来说会显得比较陌生，但是并不妨碍 Rust 已经在世界范围内作为公认的 C/C++ 的有希望的挑战者。我认为，从长远来看，在对内存安全性和性能有严苛要求的场景，Rust 将会有广阔空间。

推荐文章

Rust 语言 2017 路线图半年回顾

Clojure 太灵活，我们能如何驾驭它

作者 何婧誉

古话说的好，静若处子，动若脱兔。这个我觉得非常适合形容动静态语言的区别，静态语言因为类型系统的关系，一直给人的是很稳定、很可靠，但是可靠到一定程度就变成了死板，会变成一个牢狱或者困住业务上所需的灵活性，因此常常需要很多层抽象，很多层胶水代码，代码就开始变得非常的晦涩，非常的难懂，而动态语言则完全是相反的。

引言

这个题目非常的生动活泼，但是下面可能有点干（货）。上次来 Qcon 是两年前的事情了，在上海，那次我对 Qcon 是做了一个比较笼统的这样的一个介绍，所以这次主要来讨论一下动静态语言的问题，这个问

题争议非常的大。

Morgan Stanley 公司在国内的技术圈可能不像在英美那边对投行特别有认同感，大摩的技术文化跟团队协作，跟我之前的四个东家相比的话，并没有差到哪里去的，而且他做得东西也非常有意思，大摩现在是全球前三大的 Scala 的公司，公司内部也做了很多编译系统的改进，我们大概有 250 万行 Scala 代码、三四百个 Scala 程序员，内部做得东西也是非常有趣，是做一个很纯函数的系统，是为了解决分布式计算不够简洁的问题，即内部的函数基本上全部都是纯的，基本上没有负作用，这样的话，在什么地方执行都是完全没有关系的，反正结果都是一样的，那么在这样的理念下面，你可以想象 250 万行 Scala，我们做出了双时效数据库，做出了很多支持函数理念的东西，这个平台理论上是非常先进的，所以我在大摩的感觉也是天天可以学到很多东西。

1. 静态语言 VS 动态语言

静态语言因为类型系统的关系，一直给人的是很稳定、很可靠，但是可靠到一定程度就变成了死板，会变成一个牢狱或者困住业务上所需的灵活性，因此常常需要很多层抽象，很多层胶水代码，代码就开始变得非常的晦涩，非常的难懂。动态语言则完全是相反的，所有东西都是从类型上来讲，以函数为例，灵活性已经足够了，但是通常我们写着写着就忘记数据长什么样子了，你可能今天写了一个函数说，输入一个函数的数据，然后过了一个星期之后，我已经完全忘记这个数据是什么东西了，因为生产环境里面，类型系统在没有编译器的帮助下，基本上都是一次性的，这个问题对于用户来说有相当大的困扰。一直以来，这两派之间没有争出特别的高低，静态语言笑动态语言做不出大系统，动态语言笑静态语言写的太慢、废话太多，今天这个主题当然不可能解决这个纷争，但是希望通过 Clojure 这个语言可以给大家一些不太为人知的思路。马上就有人来问了，我写 Clojure 就是为了逃避这样的内容来写系统，这样灵活多好用啊，我想写什么就写什么，快速原型靠的就是这个，我非常同意这一点。

2. 简单示例

```
(defn read-json
  [file-name]
  (json/read (clojure.java.io/reader file-name)))

(read-json "data.json")
```

在 Clojure 里面有一个 json，因为动态语言的关系相当的简单，完全没有废话。这个函数我觉得哪怕是不写 Clojure 的，这个也是应该很能读的懂的。首先有一个 Java 的 Reader，是 FileReader，这个 Reader 被传递到了这个 json 的函数里面，读出来文件内容，读到 Map 里面，但是读完之后，你知道数据长什么样吗？不知道，下次换一个 json 文件，同样的函数可以同样读，但是你不知道读出来是什么东西。讲到这里就已经有一点难度在里面了。现在看一下，我现在读完了要处理，我处理之后，我写任意一个函数，如果说你不看这个函数写的什么东西，你知道它处理完成之后长什么样吗？不知道，你知道他希望这个 json 数据是什么样的形状吗？不知道。我现在看了代码之后，可以给你讲，它里面会有会有 Age、Name、Job、Address。

```
(defn manipulate-json
  [json-data]
  (->> json-data
       (mapv #(select-keys % [:age :name :job :address]))
       (mapv #(update % :age inc))
       (zipmap (range 0 (count json-data)))))
```

看一下 Age，它需要能够使用 Int，那应该是个整数，但是要看代码才知道，再下面还是简单，那你们觉得 Name 的值是什么东西？完全没有使用到，它是一个 String，它是不是姓和名放在一起了？还是放在一个 Vector 里面，可能姓和名是分开的，就是说不知道，要看代码才知道。

你看到代码之后觉得，原来是这样，它应该是一个 Vector，或者是

List，姓和名是分开放，因为它这系，它用空格来 Join 一下，这个是一个很浅显的例子，就已经说明了 Clojure 的动态灵活性非常强，但是也造成对数据的解释性标记不是很清楚。

```
(defn get-names
  [json-data]
  (mapv #(clojure.string/join " " (:name %)) json-data))
```

刚才是一个很浅显的例子吗，现在来看一个更具体的。为了这个主题想了好几天，觉得还是写一个很小的项目来展示一下我今天要讲的东西，那写什么东西呢？我又想了好几天，最后的结果是，先谢谢链家，因为是这样的，这个既然要来北京，就要关注一下房价，这个大家都在很关注房价。那我就到网上去看一看二手房，然后一页页翻过去很累的，我不可能就是这样手写一个总结，那我就写点程序把它抓一下，当然这个不是真的写了一个爬虫，只是抓几个页面做做样子，没有让链家服务受到伤害，请鸟哥放心，我不知道鸟哥在不在，可能不在。主要是这个命名空间，它做的事情基本上就是通过一个库把 html 读进来之后，它会进行一些简单的操作，把这个整理好的这个数据写到一个 EDN 文件里边，比如说第一条你可以看到这个小区，然后 1150 万，三卧室两个客厅，一个厨房两个卫生间这样，面积之类的东西，那么我们看到这个数据转换的这个函数，它收到一个参数是 Page，但这个 Page 长什么样完全不知道，我是通过库读进来的，读进来之后，我不知道它长什么样子，我现在看这个代码也非常难知道，它到底会返回一个什么样的类型，什么样的数据，如果将来需要扩展的话，或者将来我要给另外一个人用，或者帮助另外的一个人去做一些扩展，做一些维护是很难搞定，这就是我刚才说的 Clojure 作为一个动态语言的弊端，就是太灵活，导致经常会忘记这个函数的参数是长什么样子，而且这个是小项目，项目一大，那就更麻烦，那我知道你们有人会说的，说这个文档不就是做这个事情的吗？文档跟测试，但是文档它本身的代码是剥离的，它没有紧密的联合在一起，而相对代码本身是没有限制的，所以完全有可能你们自己有经验对吧？比如说很多代码上面会写，但是其实代码里

面并没有，它可能起到的效果某种程度上也是挺有限的。

3. Core.typed

Core.typed 是一个类型系统。它和其他语言的类型系统还是有点不一样的地方，不同点在于它不是语言的一部分，而是一个即查即用的一个库，就是说 Lisp 灵活性导致它能够作为一个库直接插进去，而不是要作为一个语言核心。因为它有宏，通过宏可以把一个很大的类型系统直接插进去，而且这个类型系统比一般的系统要灵活很多，主要体现在这几个方面：

第一，它可以给已经写好的，没有标注过的，或者说是用的库里面没有标注过的函数直接加上类型。

第二，不需要把所有函数全部加上类型，你不想要的话，就不需要。

第三，你即使加上了也不需要一定要进行类型检查，所以它是一个非常选择性是非常强的一个东西，因为它是为了能够和 Clojure 这样的语言进行协作，那么最后一个为什么要这样，为什么要加类型？然后不进行类型检查呢？有时你用库的时候，你给一些要用到的函数加了类型之后，不想要这个东西进行类型检查，因为一旦检查就要把这个库全部都标上去，库面里面所有函数全部都加上类型是很累人的。那我们现在看一下它支持什么东西，OptionType，现在很流行，这个流行的语言现在都有这个结构。Ordered Intersection Type 这个我不多讲了，这个就是说一个函数，比如有两种参数形式，这两种参数类型可能又不一样，你再进行类型检查的时候，它会把这个参数从上到下有序的来进行一个匹配。unionType，写过 Haskell 人都知道，这个很简单，比如说整数，或者说是字符串，把它 union 一下，那就表示这个类型里面的东西可以是字符串，也可以是函数。Identity 是很简单的函数，它会给你一模一样的东西，那它的类型是什么呢？它这个函数的类型是什么东西呢？它这个函数的类型是，可以看一下，让 Core.type 来帮我看一下。这个基本上可以看到前面有个 all，在这里对所有的 X 能取得的类型它返回的是一个 X，就是 Polymorphism 最简单的一个体现了。Occurrence Typing 这个东西见到的比较少，它是什么呢？它

是通过检查代码里面写的控制流，比如像 if，或者像 switch，它能够进行类型推导，我可以给你们举个例子，那首先呢，把这个 Form 绑到 A 这个名字上面，值就是 1，但是我把它标注成了 any，就是说这个 A，就算只是 1，然后再返回 A，这里大家觉得会返回什么东西？如果是检查一个类型，它最后返回的是 A，它是什么类型？ Any，因为我已经标过了，我说 A 是 Any，所以它就相信 A 是 Any，但是如果我这么写，这个会返回什么东西？你可以看到它现在还是返回的是 A，这个 A 或者这个也是 A，那其他情况返回的是 Nil，那他现在觉得这个东西是什么呢？还是不是 Any，因为你现在有了控制流在这边，代码里已经写过了，所以它知道你只可能是 number，或者是 string，要不然就是 nil，所以最后 A 是 union string/number/nil。这个东西功能上是非常强大的，这个也是我强推的一个东西，这个你真正用起来就知道方便。

最后一个就是宏也会被展开之后再推导类型，宏跟大家刚刚知道的 switch 有点像，就是已经很直接了当的，告诉大家这个宏是可以展开之后判断类型，然后给大家看一下我做的这个 Demo 的这个 types Demo，就是把刚刚链家那个小项目加了类型系统，就是说我现在是把它所制造的结果定义类型，然后它其实是什么呢？是一个 Map。

Core.spec 总结：

- 通过一个库给动态语言加上类型系统——即插即用
- 可以给已经写好的函数或者是用的无类型库标注类型
- 可以选择性地加上类型
- 加上了类型也并非一定要type check
- 支持Option Type，Ordered Intersection Types, Union Types
- 支持Heterogenous Maps和Sequentials
- 支持Polymorphism （All, Context Bounds），Higher-Kinds
- 支持Occurrence Typing！（通过检查control flow进行类型推导）
- 宏也会被展开后再推导类型

4. Core.spec

我本人是很喜欢写这个东西，我觉得给函数加上类型是非常过瘾，但是有问题，那有别的办法吗？有的，Core.spec，现在这个东西是 Clojure 核心，在很尽力地推广。在方法上或者在函数上，加上先限条件，它会稍微更加，功能要强大一点，因为它强大在什么地方呢？

第一，生产环境，Runtime 不会受到影响，它的性能不会受到影响，因为如果你一天到晚在检验，它的性能上是会受到影响的，所以缺省验证是关闭掉的，那如果你觉得某些东西可能重要性比较大，你要加上去也是可以的。spec 非常灵活，它可以把那种正则方式的 rule 给写起来，就是比如某个 list，我觉得里面开头至少有一个字符串，然后后面跟着的至少是 0 个的整数等等，你就可以用正则里面的加号，星号直接定义这个 rule。并且所有只有一个参数的 predicate 的函数统统可以跟它进行无缝对接，不需要另外语法把它转换成 spec。这里面有很多种的验证方式，那么多的验证的方式可能现在没有时间讲，就不讲了，总体来说就是可以把数据套在一个很灵活的模子里。

Core.spec 总结

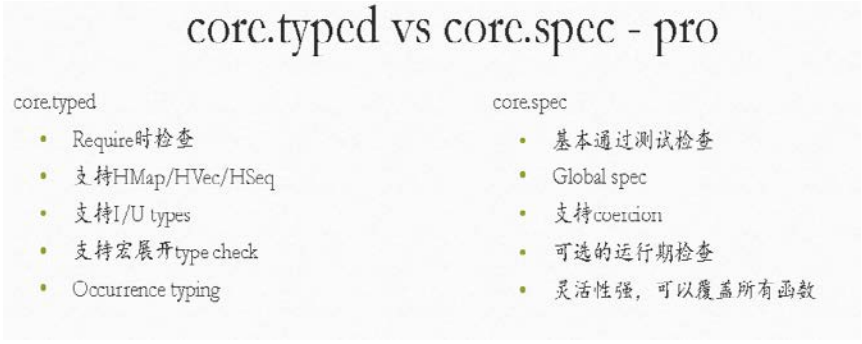
- Runtime性能基本不会受到影响（缺省spec验证关闭）
- Map的类型应该就是key及其对应的值的类型！(keys)
- Sequence可以多方面限制（cat, alt, regex style matching, coll-of）
- 只有一个参数的返回boolean值的函数通通都自动成为predicate
- 各种验证方式，满足你的需求(conform, explain, valid?)
- multi-spec支持更复杂的数据结构

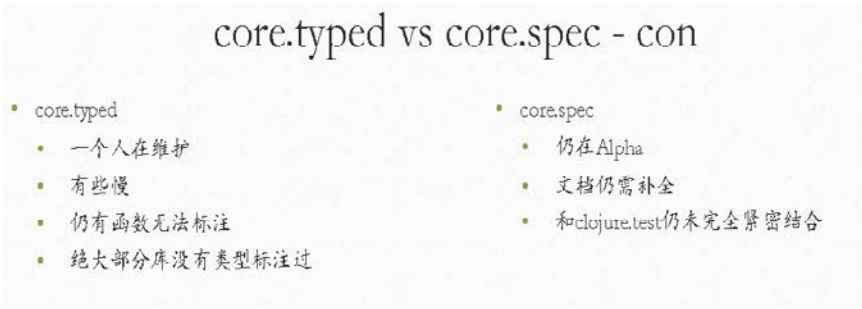
5. Core.type vs Core.spec（图见下页）

总结

core.typed 和 core.spec 你推荐哪个？

我的脑子喜欢 core.spec，因为有前景。我的内心喜欢 core.typed，因为给东西加类型写起来真得很过瘾。





作者简介

何婧誉（Loretta），Morgan Stanley VP。我是一枚剑桥大学计算机科学系毕业的妹子，平时爱好各种新鲜事物，不会的都想学一点看一点。兴趣范围从技术、数学、金融到桌游、国标、英文书法、语言学、哲学、钢琴等范围极广，属于样样都知道一些的典型 jack of all trades。有收集德式桌游及大型乐高模型的癖好。

技术上主要擅长 JVM 语言，有几年 Java 经验，2010 年遇见 Clojure 之后顿时被其简洁的语法、简单的写法及极具表达力的特性深深吸引，2011 年得以开始专业 Clojure 5 年多，现于大摩写 Scala。主要用 Clojure 做数据流处理，但也曾用其做过网络应用乃至安卓应用。JVM 之外亦与 Python、Perl 等主流语言，以及 ML 等非主流函数语言打过交道。

约四年前开始与国内的 Clojure 社区有所接触，业余时间致力于解答 Clojure 相关问题，并希望能将 Clojure 的影响范围继续扩大。

最终，JavaScript 成为了一流语言

作者 Tom Goldenberg 译者 刘振涛

2003 年，保罗 · 格雷厄姆（Paul Graham） 曾撰文提到，他的公司决定使用 Lisp（一门编程语言）。在该 文章 中他将 Lisp 描绘成计算机语言界的法语，它独特、深邃，能够表达难以描述的事物（亦即法语 _je ne sais quoi_ 所指）。他指出自己公司相比竞争对手的优势在于 Lisp。

如果 Lisp 像法语，那么现如今的 JavaScript 就像英语一般。尽管二者的语法不一致，但英语是世界上最广泛使用的语言，JavaScript 是最广泛应用的计算语言。

然而，JavaScript 仍未得到与其他语言同等的尊重。尽管它的使用率在创业公司和大型公司中持续增长，但若非必要，人们不会认为它是一门有用的语言。大公司的高级工程师声称它不是一门“真正的”编程语言，许

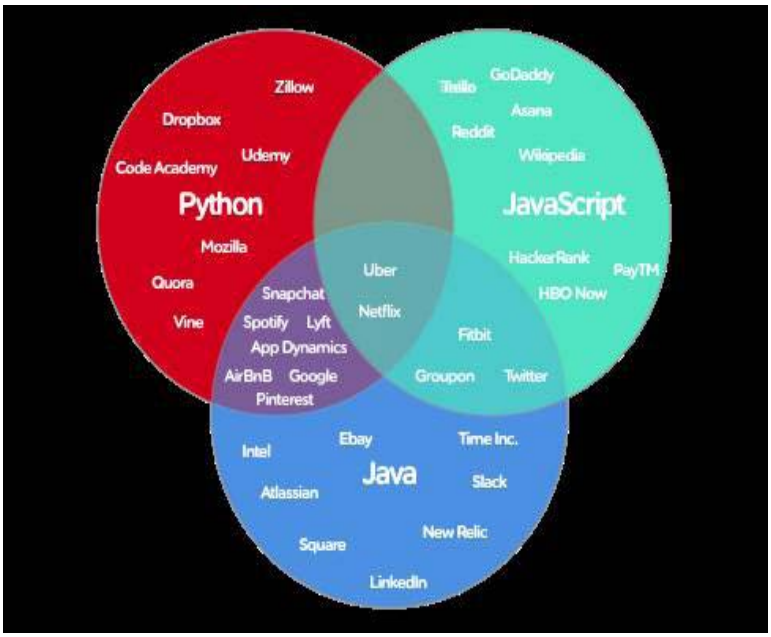
多人并不知道除了操作像素外它还能被用于何处。

作为一名 JavaScript 工程师，我希望更深入地了解公众对这门语言的看法，并观察这些观点在现实中到底有多牢不可破。我发现，一些批评有失水准，但更多的批评则是有意义的。

不断增长的生态系统

除了样式效果外，JavaScript 也被越来越多地应用于软件开发，例如后端任务、Web 服务器以及数据处理。Zeit 首席执行官 Guillermo Rauch 指出，JavaScript“不是人为设计出来的，它是在进化过程中得到的结果。它成型很快，起初只关注一个很小的目标，后来都是市场的力量改造了这门语言。

Rauch 的公司提供一个仅在浏览器和服务器中使用 JavaScript 的开源 Web 框架，事实证明，许多公司都在做同样的事情。

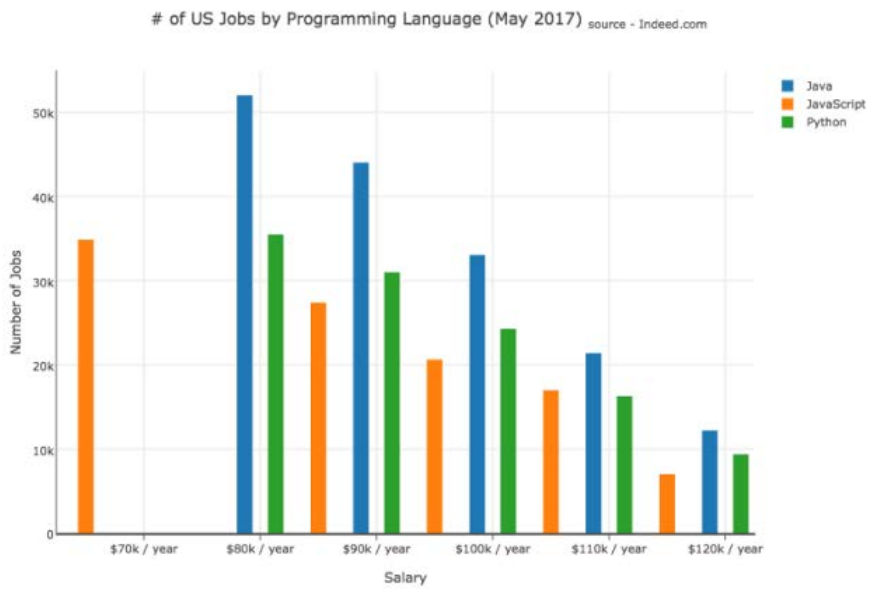


技术公司后端服务编程语言剖面图（市场占有率估值）

Rauch 的公司提供了一个仅在浏览器和服务器上使用 JavaScript 的开源 Web 框架。 事实证明，许多公司都在做同样的事情。根据展示公司技

术栈信息的网站 StackShare.io 的数据，在后端语言的选取上，相比 Python（4000）或 Java（3900），更多公司使用 JavaScript（6000）。这个网站面向的更多是创业公司，但它从侧面反映出 JavaScript 是一个不断增长的生态系统。以下是展示不同公司技术栈极各自市场份额的维恩图（数据来自 StackShare.io）。

再来看看编程工作的数据吧，Indeed.com 上的一个看法告诉我们，在美国，大多数编程工作都使用 Java，但 JavaScript 并没有落后，如下图所示：



对 JavaScript 有正面影响的其他统计数据：

- 在 Github 上 JavaScript 开源项目的数量最多（比 Java 多出 50%）。
- NodeJS 被评为 StackOverflow 2017 年开发者调查中最受欢迎的框架。
- JavaScript 是 Stack Overflow 中最流行的编程语言。

对 JavaScript 的批评

我咨询过一位 Oracle 的朋友，他们的工程师对 JavaScript 有什么顾虑。

他说“由于 JavaScript 是一门解释型无类型语言，对于系统编程来说它不是一门理想的编程语言”，这种针对 JavaScript 的投诉非常普遍。JavaScript 函数接受任意类型的参数，但在 Java 中，如果参数不是特定类型[1]就会引发错误。

```
`function doSomething(literallyAnything) { return; }`
```

我又咨询另外一位在谷歌工作的朋友，他向我指出 NodeJS 的一些公开的 问题，他说，一些错误虽然微乎其微，但他会认为这个框架不够成熟。

Rauch 指出，JavaScript 的垃圾回收不是最理想的。另一个批评是 Java 和 Python 更适合数据科学类的项目，如机器学习和自然语言处理。这可能与这些语言可用的库有关，而非批判 JavaScript 的内在缺陷。学术界对 Java 和 Python 的依赖也助长了这种论调。

上述几位工程师都曾提到，每当讨论编程语言时，经常听到其他工程师贬低 JavaScript。大家对于 JavaScript 用于后端任务依然心存疑虑，但是大部分敌意似乎又与语言和生态系统的现状无关。

JavaScript 艺术的现状

JavaScript 在过去 5 年中已经走过很长一段路，早期 JavaScript 用例一般像 Facebook 的“Like”按钮这样的功能，每当用户点击“Like”图标，页面不会刷新，但会改变页面状态，这种特性只能通过 JavaScript 在网络上实现。

开发者几年前开始通过 JavaScript 来制作单页面应用程序（SPA）。术语“单页”是指在浏览器中这些应用程序只加载一次代码，所有后续视图都是通过 JavaScript 生成的。反对者认为，用户需要花很长时间才能完成初始下载，在手机上更是长达 20-30 秒！

在过去的两年中，向浏览器发送 JavaScript 代码的技术已得到显著提高（参见：webpack）。这可以解决 JavaScript Web 应用缓慢的加载速度，提升性能并创建更好的用户交互体验。这是目前 Web 开发领域最先进的技术。

伴随着技术进步，出现了新的 JavaScript 范式。状态管理库将计算机科学原理应用于用户交互，JavaScript 工程师的门槛变得更高。

在这些变化的背景下，对于早期阶段的公司来说，使用 JavaScript 作为后端语言非常有意义，如果您已拥有优秀的前端 JS 攻城师，此举可以让它们更轻松地协作，审核和共享代码。

尽管 JavaScript 最初是一门浏览器中的语言，但在计算机科学的各个方面 -Web、移动端、物联网和后端服务中，它都变得更加普遍。工程师是明智的，不要因为他们对语言过时的看法而解雇他们。其实 JavaScript 一直是一门“真正的”编程语言，只是此时此刻，这种声明会比其他任何事情更容易被误解。

总结

从这些观察结果可以看出，JavaScript 已经达到以下这些成为一流编程语言的标准：

- 被创业公司和大型私营公司用作后端服务框架（NodeJS）
- 有一个蓬勃发展的开源社区（在 Github 上最活跃）
- 作为一门专业技能，有大量的招聘需求要求掌握 JavaScript 知识（Indee.com）

最后，一家公司决定贯彻某种技术方案都是需要妥协的。我们在 Commandiv 这款产品中就同时使用 JavaScript 作为前端和后端服务语言，但这并不适合搜有人，我们这么做决定一部分原因是因为我么熟悉 JavaScript 这门语言。为了在创业初期快速启动，请您使用最趁手的工具。

也就是说，我认为质疑 JavaScript 是否是一种“真正的”编程语言的时代已经过去，它的旅程远没有结束，但是其应用率和改进速度使我对其前进道路充满信心。

感谢 Alim S. Gafar 对本文进行审核。汤姆·金伯格（Tom Goldenberg）是 Commandiv 的首席技术官兼联合创始人，Commandiv 是一个人投资平台。

附注

尽管该语言不支持静态类型，但像 Facebook 和微软这样的公司已经发布了添加此功能的库（Facebook 发布了 Flow，微软发布了 TypeScript）。

推荐文章

Kotlin 成为正式的 Android 编程语言

FreeWheel 基于 Go 的实践经验漫谈

作者 FreeWheel

FreeWheel 从 2014—2015 年间开始使用 Go 语言进行开发，截至目前，整个体系内有超过 30 个项目是基于 Go 编写，占其总体应用数的四分之一以上，这几年 FreeWheel 内部所有团队都在或多或少使用 Go。对于新业务的开发，各工程师团队会首先评估 Go 语言是否适合完成这样的工作。

FreeWheel 在 Go 上的进化

作为美国最大的综合性传媒集团 Comcast 的子公司，FreeWheel 服务的对象大多为欧美地区大型的媒体公司，包括电视媒体和运营商等。这些公司拥有大量的视频资源和广告资源，因此 FreeWheel 主要为他们提供从广告部署、管理、投放、计费到预测的一系列完整解决方案。

正因为系列流程的复杂和业务场景各不相同，FreeWheel 各个产品子系统所使用的编程语言也会根据实际的业务场景有不同选择，其前期主要使用的语言包括 Ruby、C++、Erlang 等。

Go 语言在 FreeWheel 最早的一个尝试是 IVI(Instant Video Ingestion) 系统的应用（该项实践在后文进行详解），用 Go 重写 IVI 服务以后，在保持同样 20 毫秒相应时间的前提下，整体性能可以从每天几十万条数据扩展到每天几百万条数据。

从 IVI 项目中，一个 Go Common Library 的共用代码衍生而出，FreeWheel 中许多基本功能如日志解析等，都会被封装到这个库里供其他团队使用。此外，随着该项目的成功，工程师团队将其中积累的一些较好的经验进行应用和传承，Go 语言在 FreeWheel 的使用范围也逐渐扩展到其他更多的团队。比如由前端 UI 团队、广告决策和投放团队所开发运维的几大主要系统也得到了 Go 的很好的支持。

- 广告部署、管理系统的开发和工作流处理：之前的广告部署和管理系统主要是基于 Ruby on Rails 框架的 Web 应用程序，开发过程还结合了更前端的 JavaScript、CSS 等语言。Ruby 面向对象、贴近英语的主要特性，以及 Rails 的速度和灵活性，可以使广告部署和管理系统的开发运维过程更加快捷高效，易于维护，另外欧美地区客户也能更顺畅地在该系统上登录使用，进而管理他们的视频和广告。此后，因为 Go 语言应用在这种轻量级、高性能服务端的适应性很强，因此该团队就用 Go 重写了所有的 API，包括对 API 网关，以及网关相应功能（如用户的验证、流量控制等）的实现。
- 广告决策和投放系统：该系统对实施性要求较高，因此基于性能的考虑，FreeWheel 用运行时效率更高的 C++ 语言直接编写广告服务器，使其具备支持高并发、低延迟、高吞吐等性能。Go 的使用得以广泛延伸后，该系统主干部分还是用 C++ 实现，但为了满足一些客户的新需求，比如在线视频转码服务就是用 Go 语言写

的独立的服务。最近该系统的技术负责团队欲实现的一个新功能也是使用 Go 来开发，因其在性能和开发效率上的优势，该团队可以很方便的调用 Common Library 中已有的方法去完成大部分的基础工作。

- 广告预测系统：该系统与上面所说的广告决策系统关系较紧密，最早实现是一个基于 Erlang 语言开发的调度系统，经过演变后的最新版本是使用 Go 语言进行的编写。演变的原因有三点：第一是性能的考虑，Go 提供了较好的库和工具链去帮助工程师团队找到其自身的瓶颈、分析并提高其性能；第二是团队适应性，FreeWheel 拥有上百位工程师，分布在全球亚、欧、美洲的一些国家，因此一致的工业化生产方式相比程序的美观优雅，会有更优先级需求，在让所有工程师共享一致的、可控的、容易分享的公共编程框架和规范上，Go 语言也更具优势；第三是相比之下 Go 在部署简易性上的特性。

Go 语言是 FreeWheel 公司目前主要力推的一个方向，在其看来，面向服务的架构的大环境中，Go 非常适合做一些功能相对独立、功能比较明确的微服务的语言。在结合已有的各种编程语言，计算框架（如 Hadoop、Java、Ruby、C++）的基础上，FreeWheel 把 Go 语言定位成用来实现轻量级服务或 API 的缺省编程语言，将之与用来完成更小粒度工作的 Python 结合在一起，就构成了 FreeWheel 的整个技术语言栈 。

FreeWheel 在 Go 上所经历的“坑”

虽然从 2012 年 Go 1.0 发布到团队相继采用 Go 来编写项目，这中间经历了大致三年左右的时间，但由于在 GC 等许多问题的克服上需要 Go 本身去做一部分迭代，FreeWheel 也需要把技术对客户的影响控制在一个可控的范围内，因此作为一家 B-to-B 企业，其采用了更为渐进的方式将 Go 语言应用到自身的生产平台上。

在这个过程中，FreeWheel 也经历过两个较为重要的“坑”。

GC 的问题

如多数人所知道的一样，Go 语言垃圾回收器存在一定的缺陷，特别是容易导致整个进程不可预知的间歇性停顿。像某些大型后台服务程序，如游戏服务器、APP 容器等，由于占用内存巨大，其内存对象数量极多，GC 完成一次回收周期，可能需要数秒甚至更长时间，这段时间内，整个服务进程是阻塞的、停顿的，在外界看来就是服务中断、无响应。FreeWheel 在使用 Go 1.4 版本时也遇到过类似问题：广告预测团队用 Go 来实现调度器，平常运行的时候没有问题，但一旦触发 1.4 版本下 GC 的时候，该系统的 downgrade 非常厉害，导致任务的堆积非常严重，触发报警，同时其处理性会下降很多，也会影响其他上下游系统的正常运转。

于是，FreeWheel 在当时主要采取了三种对策：一、并不把 Go 用在非常关键的、对服务进程稳定性要求较高的系统里；二、引入 Kafka 之类的能够持久化的消息队列，能够缓存和重释这样的方式去解决这个问题，使系统能扛住冲击，并在后面把它消化掉；三、尽量复用已经创建的对象，防止 Go 频繁的创建了回收对象。

Go 1.5 到 1.7 版本相继出来后，GC 的系统性能得到不断改进和持续提升（从秒级到毫秒级）。对于目前 FreeWheel 内生存环境不太关键的系统来说，Go 1.7 之后的 GC 已经可以达到可接受的范围和程度以内。

内置数据结构的变化

很多人都知道，Go 语言提供的字典类型并不是并发安全的，此外由于 Go 语言发展较快，有些内置的数据结构如 Map 的行为也发生了变化。因为 Map 为引用类型，所以即使函数传值调用，参数副本依然指向映射 m，所以多个 goroutine 并发写同一个映射 m。例如，如果 map 由多协程同时读和写就会出现 fatal error:concurrent map read and map write 的错误。

Go 1.6 版本之前 Map 可以支持并发读写，但 FreeWheel 开发的程序在升级到 1.6 之后也就发现 Map 产生了读写竞争的问题。

对于这一问题，常用的有两种解决方案，一是如上所说的加锁（包括

通用锁和读写锁），二是利用 channel 串行化处理。FreeWheel 的做法也主要靠两方面，其一是将锁粒度设计的更细，使得并发的依赖更少；另外是在不同的数据结构中，选择性能更高的一方。比如 array 和 slice 中，前者就是更优选择。

FreeWheel 首席架构师刘昊植认为，并发的时候不能假设 Go 能完美处理所有的工作，工程师需要结合并借鉴传统（成熟）的编程语言，比如 Java 或者 C 对并发的经验，在动手之前就想清楚并发的规模、锁的粒度等，并对系统会如何运行有非常明确的设计和理解。

同时，刘昊植也坦言，随着 Go 的快速发展阶段，FreeWheel 内部也有多种不同的声音，例如运维团队就会觉得快速发展阶段的语言稳定性不够，它的特性和数据结构会因为版本升级等原因产生很大变化，并且部分变化不能完全保持向前兼容。在这个过程中 FreeWheel 的总结是：不管 Go 怎样实现，都要对系统并发做很好的支持，在应用层面做保护和控制，这样才保证这个系统能够正常的运行。

基于 Go 创建微服务的例子

如上文所说，Go 在 FreeWheel 的第一个试水项目是 IVI(Instant Video Ingestion) 系统。这个系统的主要功能是为了能够接受客户的海量视频资源元信息的插入，并完成后续的一些处理任务。于是，FreeWheel 主要以 Service API 提供相应的服务。该服务的第一版使用 Ruby 语言实现，但因为 Ruby 对并发的控制相对复杂，因此后续的性能、响应时间和吞吐率都不足以支撑整个公司不断发展的业务量。此后，FreeWheel 将 Ruby 替换为 Go，用 Go 的 gRPC 去实现了新的 API（当然也支持 RESTful API）。

正因为 API 架构需要足够灵活地支持未来的业务集成，所以它也成为其自身开发微服务的一个很好的例子。在设计和实现过程中，FreeWheel 把该架构分解成很多基础模块，比如流量控制、用户验证等，可以很灵活的把这些模块根据定制化的需求拼装在一起，提供适应市场和客户发展需求的真正价值。通过清晰的接口定义，Go 语言可以很好地把整个系统串

联在一起。

此外，因为 Go 对容器及虚拟化技术有一些天然的支持，FreeWheel 的 API 团队也正在加速采用这种架构，他们准备将 API 都封装在 Docker 的 image 镜像里，用 Kubernetes 把所有的系统都管理起来，用 ETCD 或其他类似软件来辅助服务的发现和管理。

为什么 FreeWheel 没有全部用系统重写 Go

从编程范式的角度来说， Go 语言是变革派，而不是改良派。对于 C++、Java 和 C# 等语言为代表的面向对象的思想体系，总体来说全球范围内许多公司对 Go 语言的态度更为保守，多数持有限吸收的观念（这可从下图中 Go 的热度分布情况看出）。

即使 FreeWheel 在实践中发现 Go 比其他类编程语言具有许多更为明显的优势，如在写并行上，相比 Python 这样的解释语言要高一个数量级；如前期由 Python 开发的很多轻量级 API，因为全局解释锁 GIL 的关系而面临着进程间通信带来额外开销，所以就把轻量级 API 迁移到 Go 上；又比如 Go 在并发上的优势，适合面向多用户同时上传、同时调用 API 的场景……但其内部团队也并没有用 Go 来重写全部系统。原因主要有两点：

第一，FreeWheel 有很多已有的算法实现，想全部切入到 Go 上会面临巨大的开销和成本；第二，相比 C 或者 C++，Go 在高性能方面还没有完全的证明自己。在 Web 服务器端，它目前也没有一个特别好的像 RoR、Django、或者 PHP 的流行框架。对于 FreeWheel 来说，整个广告服

务器是不允许出现明显的 downgrade 情况（尤其是当 GC 时），所以对这种非常关键的系统，目前还不能完全用 Go 去写。所以刘昊植也认为，Go 在扩展使用场景层面可能还需要做一些较大变革。

此外，FreeWheel 基于 Go 的 Web 程序，目前使用的是 Gorilla 框架。但从 Martini、Revel、Gocraft/web 等几款主流框架的使用和评价上看（可以下图 github 上的数据做个参考），Go 社区还没有一款处于统治地位的 Web 框架。如果 Go 想把它的触角伸得更长，这可能是其未来发力的一个方向。

name	Watch	Star	Fork
https://github.com/gorilla	126	2,149	342
https://github.com/go-martini/martini	501	8,290	863
https://github.com/gocraft/web	58	1,005	72
https://github.com/lunny/tango	43	378	51
https://github.com/gin-gonic/gin	378	5,992	631
https://github.com/revel/revel	487	6,493	936
https://github.com/hoisie/web	168	2,615	379
https://github.com/kataras/iris	34	549	30
https://github.com/astaxie/beego	631	6,553	1,630
https://github.com/go-macaron/macaron	94	1,026	105
https://github.com/labstack/echo	184	3,130	252
https://github.com/codegangsta/negroni	185	3,492	242
https://github.com/zenazn/goji	142	3,006	202

但总的来说，Go 在高并发、开发效率等特性上的优势，决定了 Go 在 FreeWheel 内的采用程度会越来越深。刘昊植说：“除了一些已有业务依赖于 Hadoop、Spark 这样的基础设施，对于新增的业务和功能，Go 语言会是我们的首选。”

另外，FreeWheel 希望其使用的编程语言是能够得到跨大陆、跨时区、

受所有工程师共同认可的，所以 Go 或许会是其最好的选择。这个过程中，FreeWheel 也评估过很多其他语言如 Scala、Rust 等，但最终因为 Go 在学习成本、统一实践、社区规模等方面的优势而胜出。

版权声明

InfoQ 中文站出品

出版：北京极客邦科技有限公司
北京市朝阳区洛娃大厦 C 座 1607
欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。
网 址：www.infoq.com.cn