BT

Scala在挖财的应用实践

作者 王宏江 发布于 2015年12月30日 / 注意:QCon全球软件开发大会（北京）2016年4月21-23日，了解更多详情！ 2 讨论

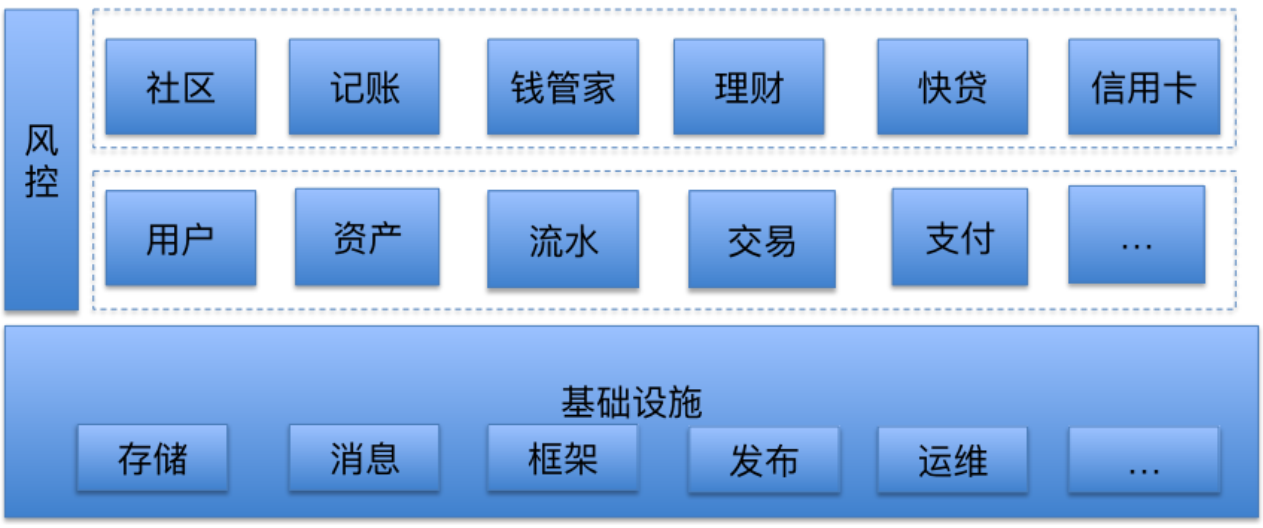
编者按：本文是根据ArchSummit大会上挖财资深架构师王宏江的演讲《Scala在挖财的应用实践》整理而成。

这次分享有三个方面，一是介绍一下挖财当前的开发情况和后端的架构， 二是挖财选择Scala的原因，三是挖财使用Scala相关的技术时碰到的问题以及经验。

第一部分是团队的情况和后端技术的架构。近一年我们的开发团队从50人增长到了现在两百人，公司总人数扩张到600左右，技术人员占的比例跟国内大多互联网创业公司的比例差不多，1/3左右的样子，昨天大会上王天提到Twitter的工程师44%左右在硅谷差不多是平均水平，国内公司比例要低一些。我们是一家典型的业务驱动公司，并不是技术驱动的，业务驱动的一个特点是需要快速尝试各种可能性，需要技术人员能尽快跟上，好的程序员并不是那么好招的，某些阶段在一定程度上先靠人数来堆。所以挖财技术团队中少部分人是有一定经验的，核心的一些架构师主要是从阿里出来的，大部分的程序员相对普通。

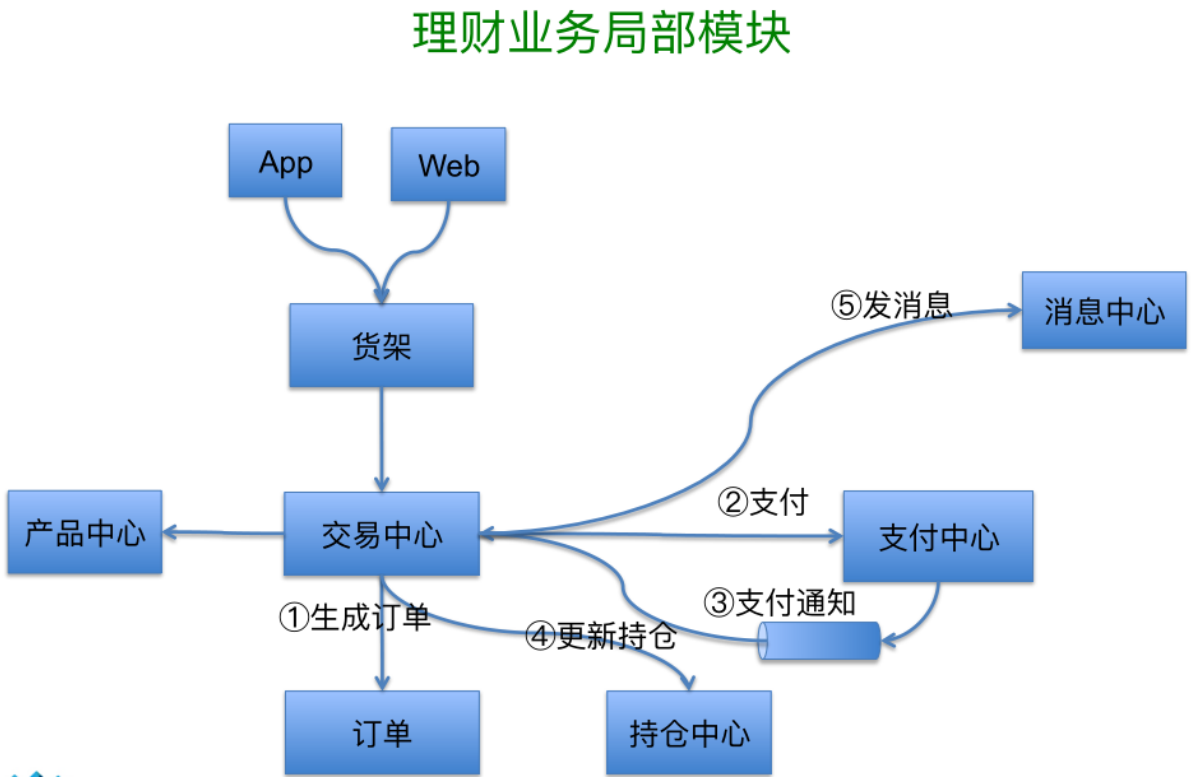
而后端技术用的主要是比较大众的东西，Web容器用Tomcat，框架主要是Spring MVC，也有少量的Play，中间服务层是Dubbo，微容器用Spring Boot，服务注册这一块是用ZooKeeper，核心业务开发方式还是围绕着Spring和Mybatis等；数据的存储这块是MySQL和Hbase，分布存储这块是用阿里巴巴之前开源的一个中间件Cobar。消息和实时计算这块主要是Kafka, Storm，日志以及监控系统则是用典型的ELK和Zabbix。另外我们将要放弃MongoDB和Memcached，并不是这些产品自身的问题，而是人的因素，最早的架构师对MongoDB很熟，后来因为他离开，这块的一些问题缺乏强有力的人把控，就放弃了。其他人对MySQL和Hbase更熟悉，并且这两种组合基本满足我们的需求了。Memcached则是被Redis替代。总的来说，我们裁剪掉了一些之前使用的相似产品，减少为一种，主要是基于自身运维能力考虑的，不在于这些技术是否足够好或者是否符合团队的特性和遇到的问题，而是你能不能把它维护的好，或者说风险控制好。

业务上面我们大概也是比较典型的。基础设施是一块，然后中间是服务，上面是日常的应用。基础设施就包括典型的存储、消息、框架，还有发布、运维、监控等等。中间就是现在典型的SOA服务，用户的资产、服务、流水、交易等等，所有的服务采用Spring Boot打包部署，主要便于运维上的统一和方便做一些切面层的事情。最上面是应用，包括社区、记帐、钱管家、理财。而安全和风控会切入在每一个环节。

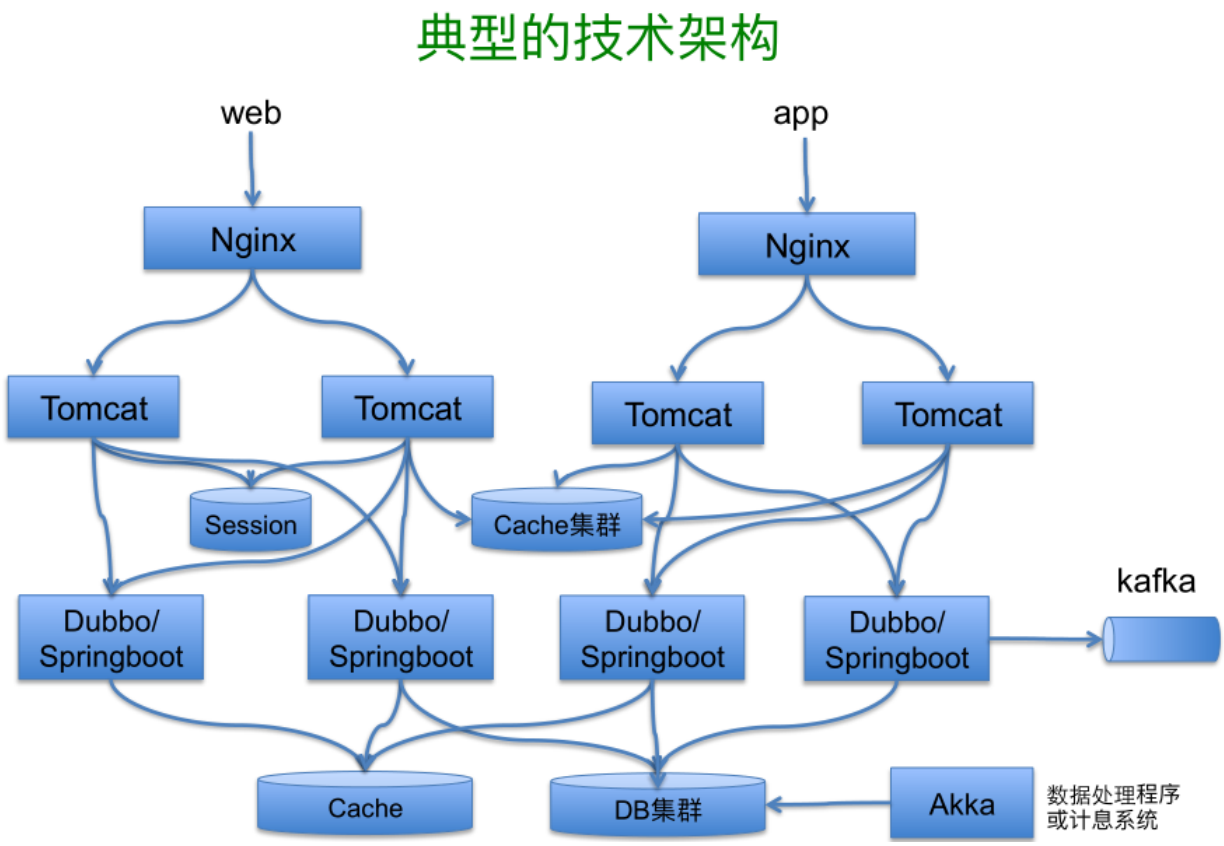


其实作为一个互联网的金融公司，很多时候我们对金融会有一点点陌生，在我们看来做一个理

财产品或者一个理财业务其实和电商业务有很大的相似之处。主要就是理财产品，他也是一个商品，这个商品可能有高的回报，最初形成这个商品的时候背后有一套综合系统或者说有一套逻辑。并且购买这个商品之后，这个商品是有历史的，他的收益是在变化的，所以简化之后，我们发现跟电商系统是相似的。

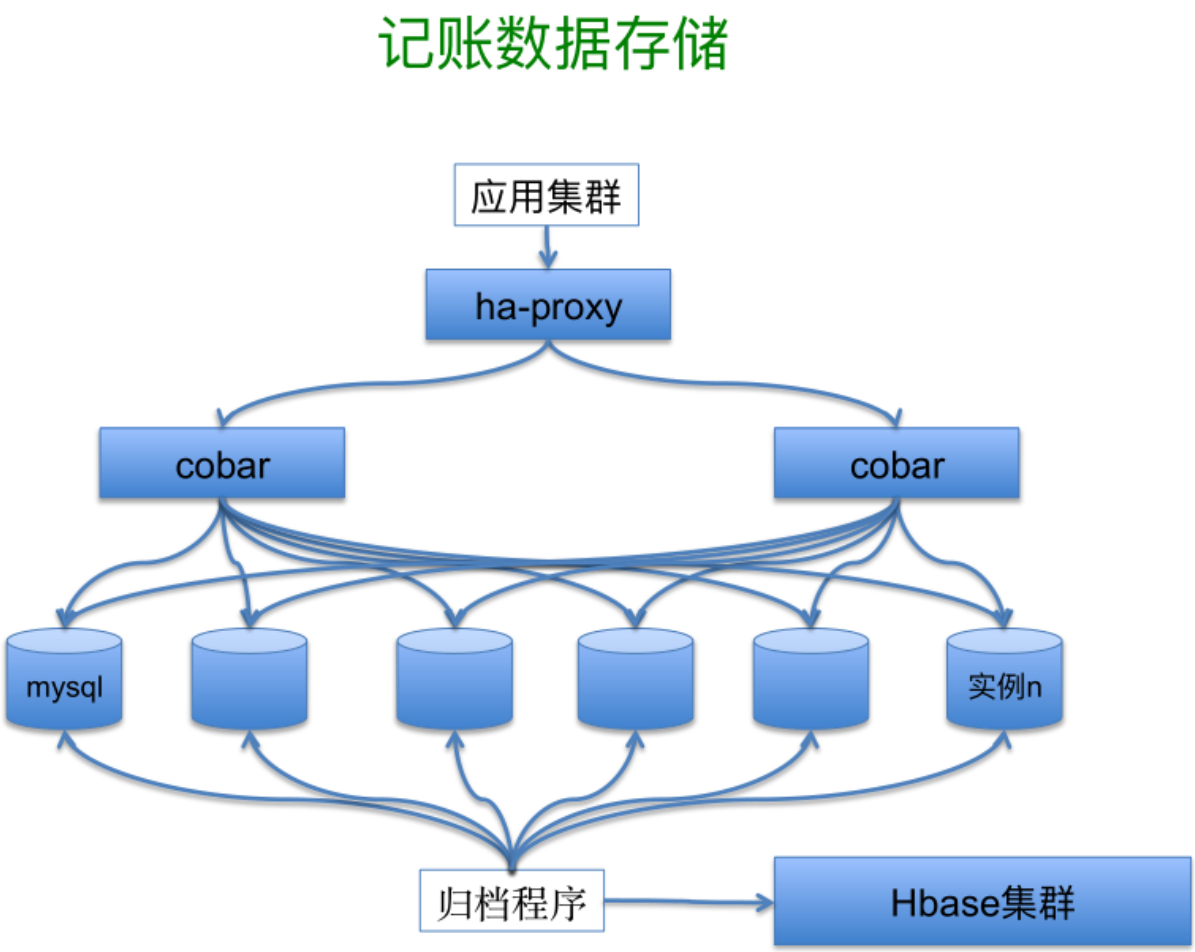


这就是典型的架构，前面是Web或Gateway，中间是Service，中间层协议是Dubbo默认方式或者是HTTP。

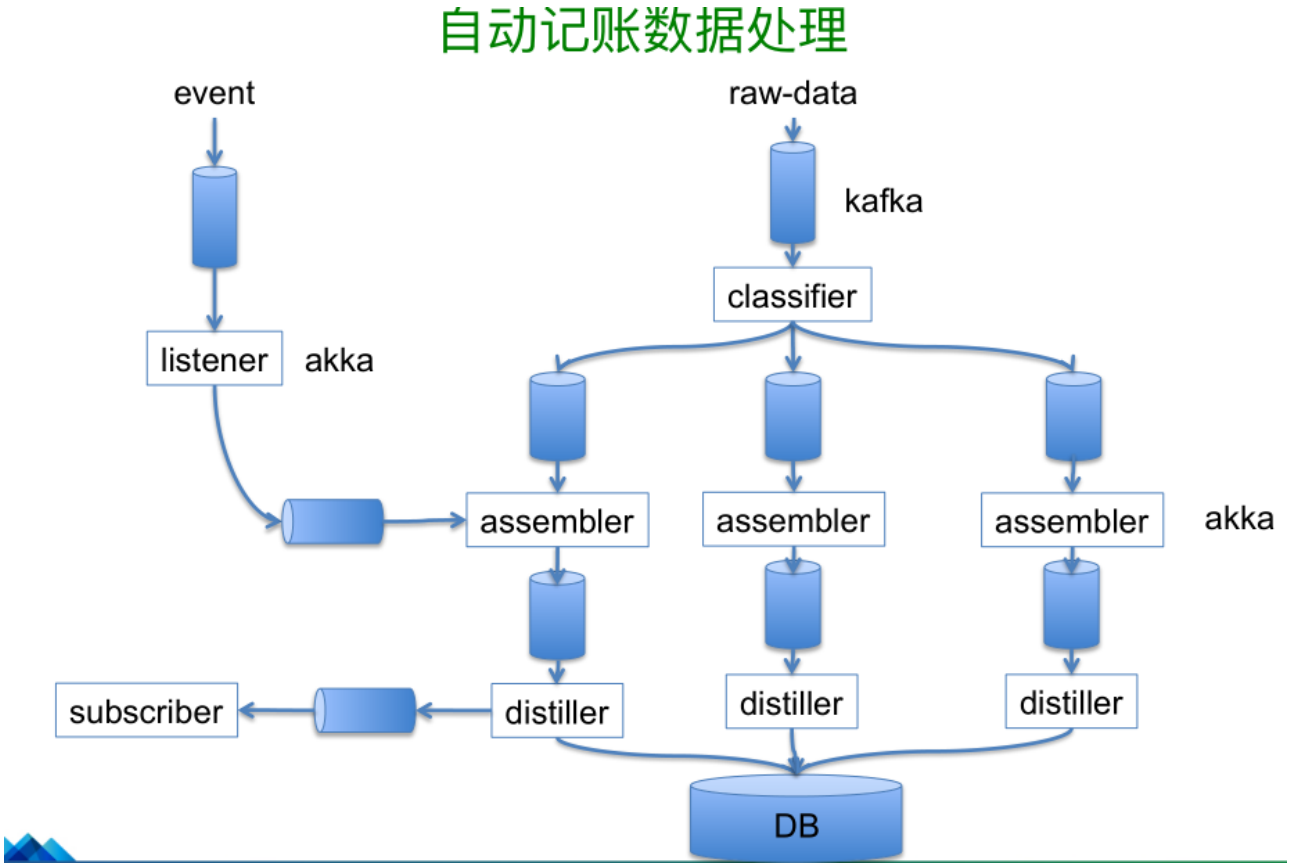


数据存储这一块主要是MySQL和Hbase，当前还有小部分数据使用的Mongo，ORMapping这块我们用了MyBatis。数据的水平扩展主要是基于刚才提到的Cobar，它是阿里比较成功的开源产品之一。Cobar支持数据的二维扩展，能满足我们的场景，比如说几千万用户的流水

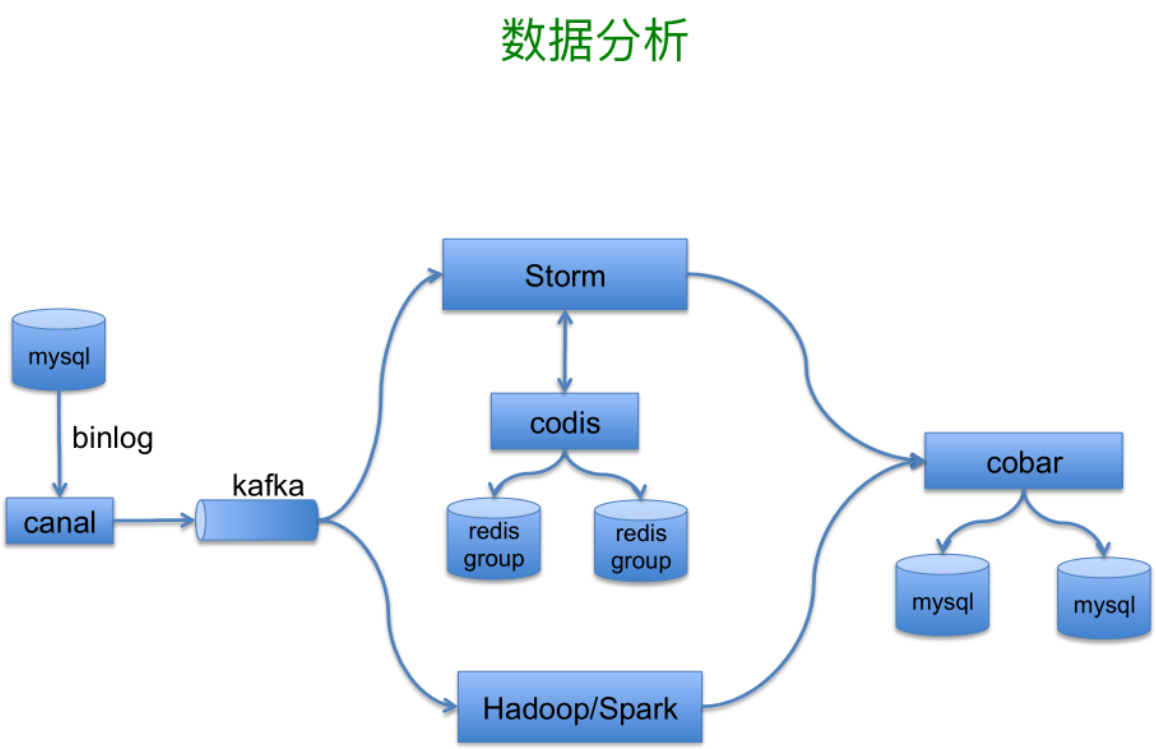
记录可以按用户加时间的维度来划分。如果单纯按时间的维度分区，在查询用户的流水时要散落到各个分区上，对各个分区的压力都较大。而按用户的维度则可能导致这个分区在一定时间后达到我们预设的容量上线，因为一天有几百万甚至上千万的流水进入。二维的方式可以解决这些问题。尽管我们后来发现用户的数据大多在最近一段时间内较热，时间比较早的数据相对较冷，只用了Cobar的一维扩展方式，将冷数据归档到Hbase集群来解决。在数据的在线处理这块，有很多用Akka框架的程序，比如计息系统主要是运算的，所以Akka很适合干这个事情。



自动记帐这块主要是钱管家这款产品，他会根据银行的帐单把这些数据汇总到这个产品里面，主要是从其他系统汇聚出原始的数据，根据这些数据我们会进行分类，进行加工和抽取，最后汇总到database，这个过程的思路就是command+pipeline下面这套组合，只不过是分布式命令和管道，分布式里面命令是Akka所承担的角色。



数据分析这块，Canal会把binlog数据丢到Kafka然后同步到其他系统。在线分析主要通过Storm进行，数据分析过程中缓存用了豌豆荚的Codis。离线分析目前用Hadoop/Spark的方式，数据分析之后，一部分存到Hbase，另一部分在通过Cober存储到MySQL里面。



前面介绍了挖财的架构，第二部分说说我们选择Scala的原因。首先Scala的语言还是有点争议的，不管是大师级的人物还是编程语言爱好者，对编程语言的争议从来没停止过的。但不管别人说Scala怎么样，从工程师的角度，从实用的角度来说，我是比较认可这个语言的。我觉得松本行弘批判保罗·格雷厄姆的话非常代表我的观点，他说的语言会朝着“小而干净”的方向进化，他自己也创造过一个Lisp的方言叫ARC，他崇尚保持简洁和纯粹的设计思路，但这门语言并不算成功。从总体或者说未来的趋势来看，语言已经不可能保持那么小而精的状态。能够真正做到解放生产力这么一个目标才是演化的方向，所以从解放生产力这个角度来看的话，Scala实际上是达到了这个效果的。不管怎么骂，Scala确实为我们解决了很多实际问题，加

速了我们的效率。

我们为什么会选择Scala？可以说有一些巧合，在挖财快速发展的过程中，几个核心的架构师对这种语言都是比较喜爱的，并且之前有些经验。如果是一个人，或者说你的公司想做这个事情，但是没有人能够支持你去做这件事情的话，那其实是相当麻烦的。因为你至少要有好几个能力相当不错的人能够一起来做这个事情，才可能在公司把这个事情推广，才能够影响更多的人。这样形成核心的小的影响之后，会吸引来更多的爱好者的加入，这样才能够使这个语言被大家所接受。所以这就必须是一个团队，既使个人能力再强也没用，团队的意愿非常重要，假如他不愿意用这个语言，或者他不接受这个语言，那么用主流的Java也是OK的，培养而不是强迫，如果用这个语言当然更好，如果不使用的话也没有关系。但是我们可能在口味上会偏好这些人，因为我发现相对来讲这些人他会更愿意去思考，或者说在处理一些问题的时候会比普通的程序员想的更多。在这个过程中还有个非常重要的事，就是慎重的选择开发栈。就拿选择Scala来讲，Scala有他自己的生态系统，跟Java还不太一样。比如说Java在Web层有Struts/Spring MVC框架，Scala有Play/Lift，构建工具Java里面是Maven，Scala则以sbt为主。Scala有一套自己的玩法或者说技术栈，你是不是选择整套技术栈是需要非常慎重的。挖财处于一个中等创业公司的规模，对我们来说，Java还是更大众化的。在业务发展过程中，人员加入还是比较快的。而且很多人还是以Java为主，所以我们的做法是不改变这些人员的习惯，还是以Java生态为主，没有强制的用Scala的开发栈。而且Scala的技术栈里有不错的产品，也有维护的很差的产品。你选择一个产品时，先看它的邮件列表，如果邮件列表里都没有什么来往邮件，这个产品的维护性和持续性肯定不够。

Scala的优点体现在，原来Java里的设计模式，现在在Scala的语言层面就提供了。比如说Java里的单例，Scala直接对应的就是object；访问者模式我们可以用更优雅直观的模式去匹配；还有构造器模式；依赖注入在Scala里面有蛋糕模式。

不可变的模式主要是在并发编程里面的，并发编程我们主要面临的就是状态，怎么同步或维护这个状态，保证这个状态不被污染掉。如果这个状态本身就是不可变的，那么就不存在竞争性。Scala本身推崇不可变的思路，当你想要去改变这个状态时，实际是new一个新的状态，原来的数据对象并不会改变本身，但Scala也并不是说完全像一些纯粹的函数式语言偏执于只用一种纯粹的不可变对象，所以Scala也提供了一种像Java的可变的数据变量(用var声明)。还有就是Java里面我们用来在不同的领域（尤其是在网络）之间传输的都是值对象，值对象的描述跟一般的对象一样，表达上略微有点啰嗦。而Scala里使用的case class在表达上则简洁一些，并且这些case class配合模式匹配非常好用。这些都是表达方式的简洁化，或者提升生产力的一种体现。

Scala的优点

更简洁、高效的表达
一些模式，Scala直接在语言层面支持：

- 1）singleton pattern → object
- 2）visitor pattern → pattern matching
- 3）facotry pattern → apply method
- 4）builder pattern → currying
- 5）dependency inject → cake pattern
- 6）immutable pattern → val
- 7）value object → case class
- ……

其他的优点比如动态类型以静态类型的方式去实现动态的效果，我们称之为duck typing，他看起来像duck听起来也像duck那么它就是duck。Scala里面以隐式参数或者隐式转换的方式实现动态语言的等价效果，只不过这种方式在编译器能够更好的检测，所以在安全里更有保障。还有利用函数式特性去自定义流控，比如说C#里的using是一个很好的特性，我们在using的括号里面定义一个resource，不管是一个文件流或者是一个disposable资源，程序处理完会把这个资源自动释放，我们可以在Scala里面通过柯里化和函数特性也可以灵活的定义类似的流控，实现等价效果。另外还有它强大的集合操作，就举两个例子，我们想要从同一组流中进行下图的两个查找，同样的效果的写Java很麻烦，而Scala通过高效的运算以很简洁的方式实现出来。

强大的集合操作：

```
// 从一组流水中找出有转账关系(交易号相同)的
cashflowBuf.groupBy(_.transactionNo).filter(_._2.size == 2).values
// 找出相差不超过1的相邻元素
scala> val timeList = List(1,2,3,5,7,8)
scala> timeList.sliding(2,1).filter(e=>e(1)-e(0)<=1).foreach(println)
  List(1, 2)
  List(2, 3)
  List(7, 8)
```

当然他有不好的地方就是门槛相对高一点，这个门槛就是函数式背景，举一个例子，比如说eta规约，比如说我这个函数里面的参数是接收另外一个函数，简单的说就是高级函数，这个高级函数里面传给他的参数比如说是 arg => method(arg) ，就是一个参数传给另外一个方法，我们可以直接简化一下把这个表达式直接简化成 method，(在Java里这种Lambda的简化表达为class::method还相对好辨识一些) 这些可能会导致很多人在看这些函数式的编程语法的时候会产生一些疑惑和不熟悉，这些疑惑表面上觉得是编程语言的俗语，实际上背后是函

数式的理论或者它的一些特性，不是大部分人都熟悉的。另外一个不好的地方就是在类型上确实做的有点复杂，有点过头了，他的类型系统是从ML系语言以及Hashkell过来的，在类型方面有很多高阶特性，远比Java要复杂多了，可以这么对比，Java的类型系统是一维世界，那么Scala的类型系统是二维世界。甚至他的类型系统本身也是图灵完备的，可以用类型系统解决SKI组合子问题，我推荐大家去搜一下用Scala的类型系统来解决汉诺塔的程序，依赖类型来解决问题思路是这样的：编译即运算。如果编译器编译通过，就说明类型演算证明了这些问题。这样带来的好处就是我只要编译成功，类型演算就完全被编译器执行，就是这些事情完全交给编译器去做。不好的地方还在于他的灵活性必须有一些克制和一些妥协，减少高阶特性，并不是每个人都喜欢高阶特性。我们还要在编码风格上有一些约定。当然在编译过程中也要利用好编译器的参数，这些参数会帮你提前发现一些问题。所以在挖财，我们主要引入Scala语言，但没有用Scala这个生态系统里面的一些框架（除了Akka），没有全面用它提倡的全面异步化或基于事件的处理方式也没有用它Web或RPC之类的框架，我们没有改变传统模式，还是基于Servlet模型，还是以Spring为中心。这种保守的做法使得大家对整个架构接受起来是比较容易的，并且较好把控。

接着讲讲Scala在挖财的使用方式。我们用一种很自然很透明的方式就能将Spring这些框架与Scala集成的很好， Scala与Java只稍微有一些差异，所以Scala与Java的生态系统集成这块是非常顺当自然的。但在生态系统集成这块也会遇到一些问题，就是Akka和Spring都是比较“强势”的托管者。Spring是bean的托管者，bean的创建和生命周期是由context来掌管的，而Akka系统中所有的actor创建都由其父actor掌管，当把actor也当作spring中的一个bean时，Akka需要做一点”妥协”，它提供了Indirect的Producer，可以包装到一个Akka的Extension里来简化actor在spring中的构造。我们选择了Spring，最大的原因就是希望所有的开发人员在接受这些程序时不会感觉到陌生，他所有的编程习惯没有改变，只是语法上的不同而已。Spring与Dubbo集成没有什么问题，只要注意到一点，就是多个系统之间，有的系统不是用Scala的时候，你必须用纯的Java来返回API里定义的模型，否则可能一个底层是Scala实现的对象在对方那里反序列化会找不到类。在不同的系统间，你在API这个层面暴露的只能是Java定义API。

然后要用好REPL，它是交互式的工具，它对于大家快速地验证简短的几行程序是非常有帮助的。编译我们还是没有选择sbt，sbt是基于apache ivy的包管理方式，所以出于统一的考虑我们大多数scala也是采用maven管理。每个人发布他的项目必须保证他的项目是自测的或者他自己用mvn spring-boot:run或mvn tomcat7:run的方式能够跑起来。一些参数的话，编译系统的参数可能也有一些对比，在提前编译的时候，我发现你的问题是有好处的，其中比如第二个"-Yno-adapted-args"这个参数我举一个例子，就是Scala里面有一些类型推导会自动适配，假设有一个Unit类型，就是类似于Java里面Void，println(a:Any)这个方法在Scala的里面定义的其实就是接受Any任何类型的参数，我执行print的时候我可以传给他任何参数，比如print(1,2,3)，这个时候你会发现与你预期不符，明明我传递了三个参数，怎么它最后也成功打印出来了，你就会觉得很诡异，这个到底是编译器搞了什么鬼还是bug，这个叫做类型适配，这个里面就会涉及到Scala的Tuple类型，叫做元组，元组这种类型就是当我用小括号，一般是用小括号去描述它的，比如说(1、2、3)这样的元组，他是三个元素的元组，用小括号来描述一个元组时，你如果没有小括号编译器发现你这个参数与他期望的参数不匹配的时候，他会把这个东西适配成一个元组，然后这里面变成一个Tuple3，就是一个Any具体的实现，导致这个方法可以通过，所以no-adapted-args这个参数告诉编译器我们不要去做适配的事情。

第三部分我会介绍一下我们在使用的一些中间件产品如Cobar、Kafka、Akka的问题和经验。挖财使用Cobar主要是因为之前我们的首席架构师王福强曾参与过Cobar的开发。相对来讲在我们的团队里面，它还是比较稳定的。对于个别的小问题我们是可以规避的，或者我们知道了以后是可以改的。我们现在采用的分区策略有两种，一种是by range的方式还有一种是by hash。第一种比如说每天你记帐进来，按照这个时间的纬度当一个库承载不了时，他会按照时间的纬度自动的到下一个库，所以是比较简单的方式。第二种就是你开始就明确的建好，比如说我们分成十个，当后面发现不够的时候，可以再把它变成20个。所以你只要开始预留好，当你翻倍的时候你再做一些事情就可以了。Cobar本身也是支持二维的，但是这个方式稍微有点高阶，尽管我们有些场景需要这个。就是说你按照时间维度去划分时，很多时候用户的请求，也就是人的流水会分布在所有的时间，这个会造成一些麻烦，他可能会碰到多个区间。但是我们可以用一些方法来解决这些问题。Kafka在挖财是比较重要的中间件，我们现在的规模并不大，业务消息是几千万级的情况，日志相对多一些有几个亿，Kafka在我们这就是万金油，它不仅仅是消息，并且当数据在处理时，把它当成分布式管道来运用的情况是很多的，还有一些数据我现在先把它放在Kafka里，然后又交给其他的程序做后续处理。所以它是被我们广泛使用的中间件。我们了解到一些金融公司，他们是采用那种EventSourcing的设计思路，也就是事件驱动。事件驱动这块他会面临一些问题，就是event要怎么样去journal好，

不管是审计还是其他的一些需要。也看到有一些系统利用它做了应用级的journal。

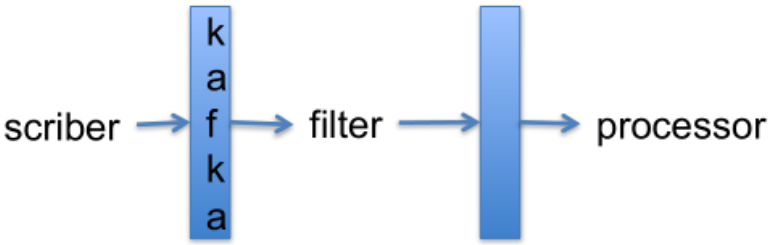
Kafka的使用经验

1) 作为分布式管道

单机版：

```
cat  file | grep content | awk '{print $1}'
```

分布式版：



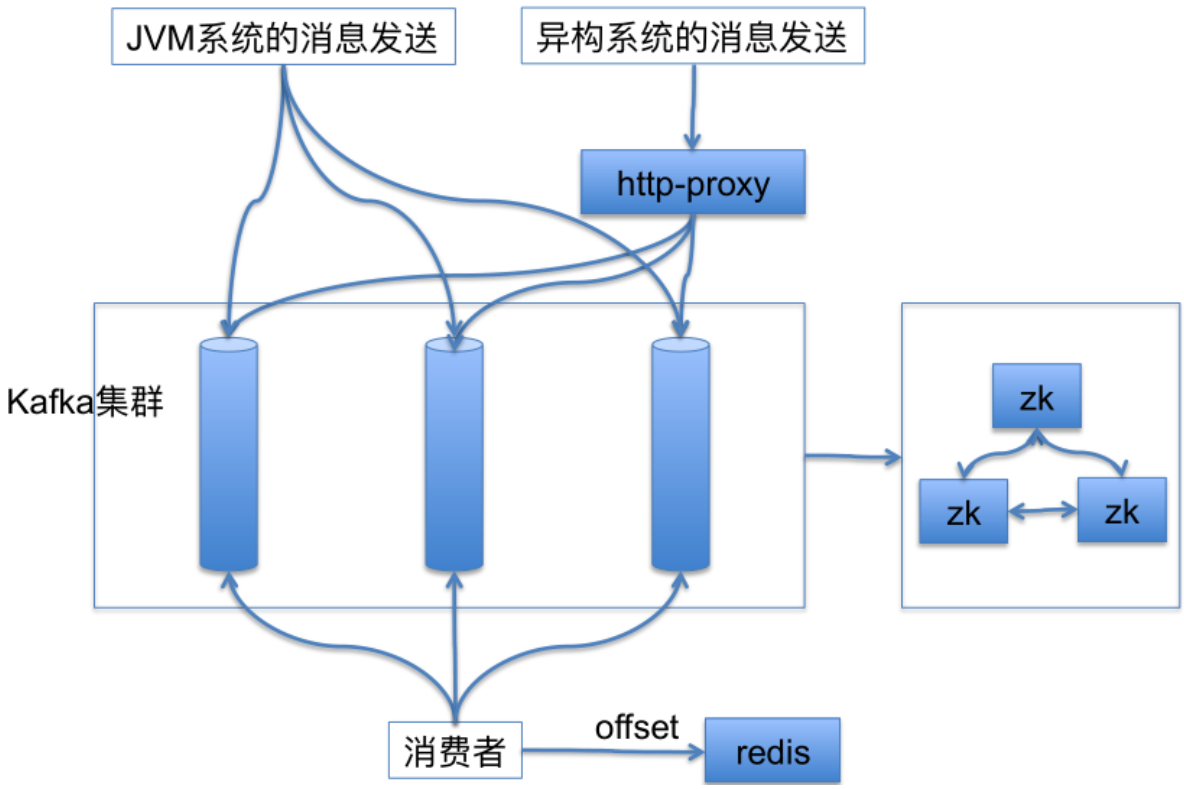
2) 作为存储器和Journal System

充分利用了磁盘和文件系统特性

《日志：每个软件工程师都应该知道的有关实时数据的统一抽象》

在我们这边的话，Kafka的一种方式，就是基于JVM的业务系统直接去连接这个集群，而另外一些异构的系统，通过HTTP-proxy的方式可以简单地处理。我们稍微有一点不同的是，我们是把offset全部存在Redis里，而ZooKeeper里没有存offset。

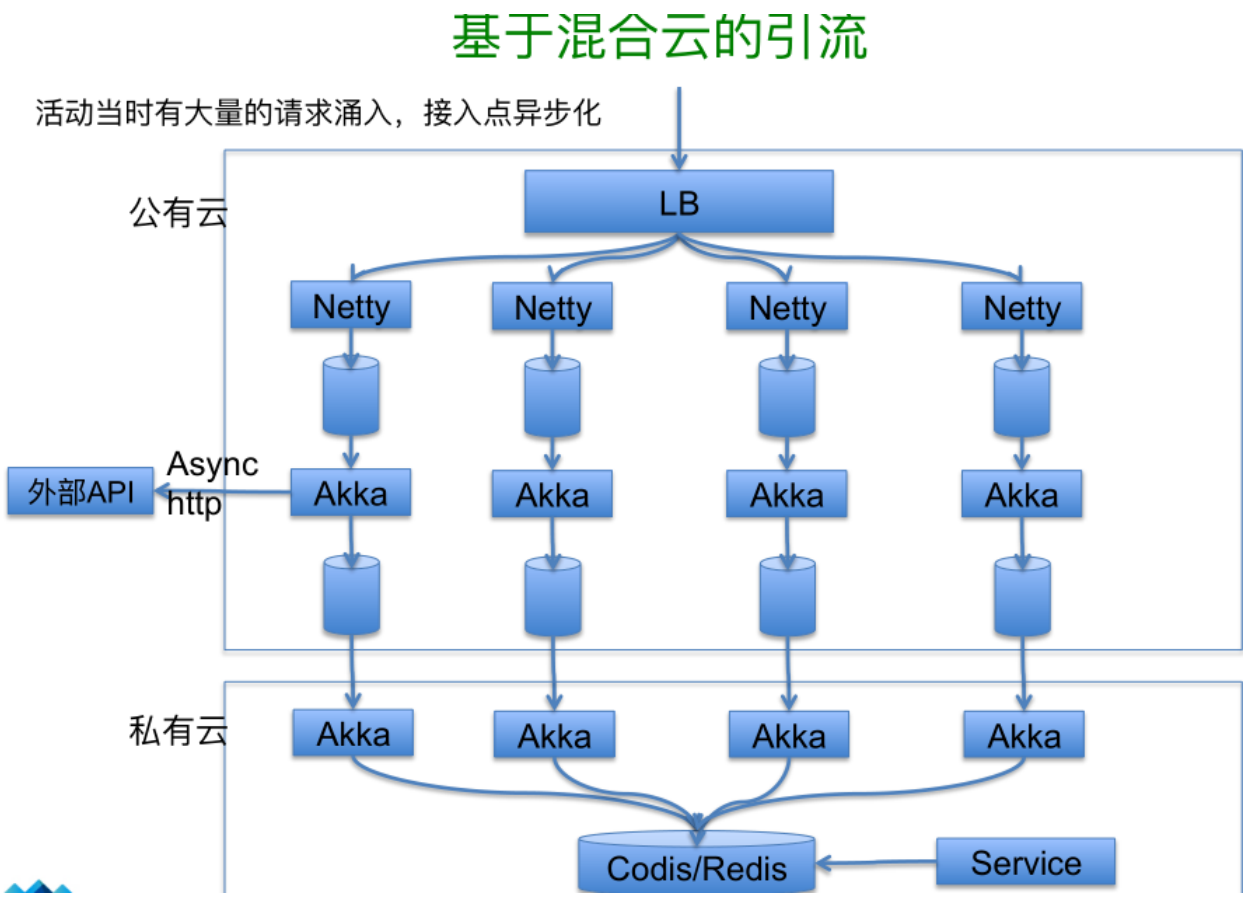
Kafka的使用经验



没有选择ZooKeeper存储是因为刚搭建的时候，当时对ZooKeeper的运维能力相对弱一些，所以我们会更偏好使用相对有把握的一些东西，所以我们基于SimpleConsumer做了一些封装，把offset的存储为本地文件或Redis。我们在运维上做了一个简单的约束，当前业务量不是很大的情况下我们尽量减少分区数，大多情况一个分区就够了，必要的话可以多个topic。

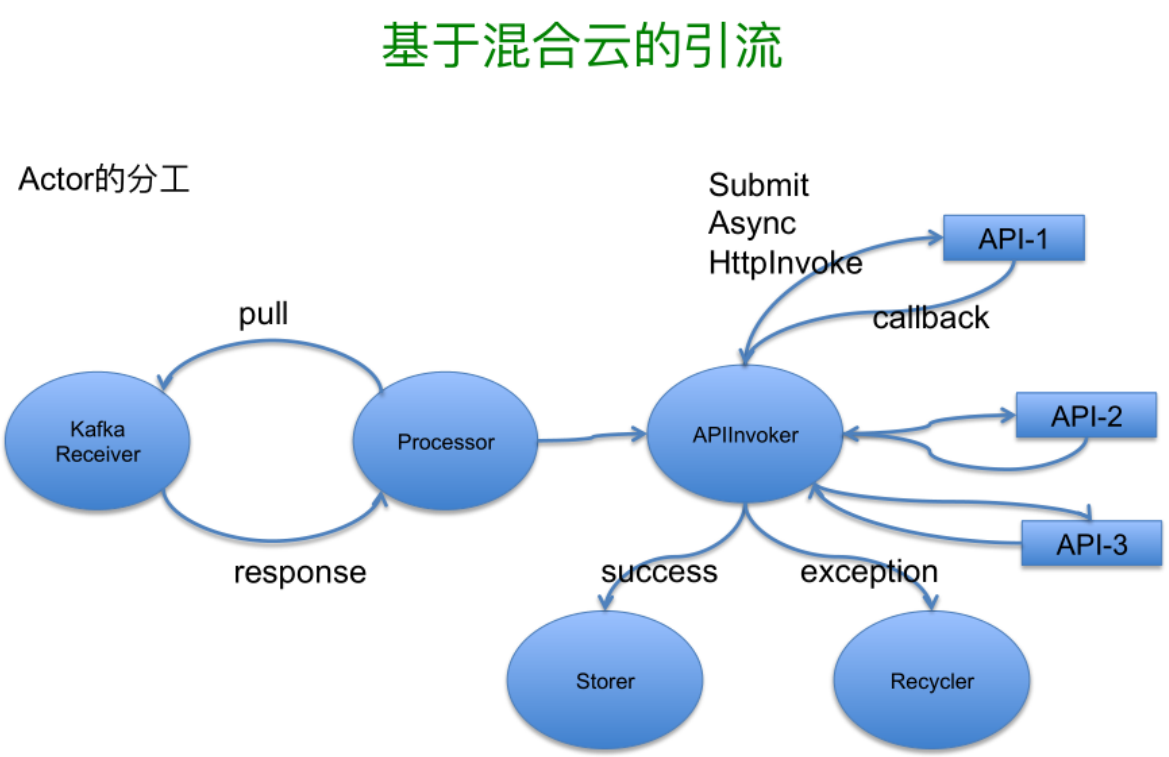
有一个案例就是Kafka，不管同步还是异步发送都会出现消息重复的情况。还有一个我们当时的测试环境， 某个topic发现消息较大发送不成功，调整了一下msgeSize和客户端的fetchSize，当时leader这边可以接收这个大的消息，但follow出了问题，副本是使用的replicaSize的，结果发现它是不可修改（至少在我们遇到的例子时的版本里并没有地方可以修改这个参数）而导致了两个follow从leader去取数据的时候就取不了这个大的消息，因为这个msgSize大于它的replicaSize，所以造成follow这边不断的去连，但是又消费不了，整个网络流量都异常了。

Akka在我们这边也是广泛使用的系统，在业务里面在用它做运算，其他的场景主要是在数据处理，它天生具有扩展性，所以我们开始做一个事情的时候，可能一个Akka就可以处理这个东西，等到将来它量变大时，因为一个Akaa和多个Akka之间的扩展性是有天然的，所以扩展起来还是非常方便的。我们在使用它的时候也因为人员的情况，并没有使用一些很复杂的情况，比如说持久化那块曾经去尝试过，但是发现还是有一些问题，所以后来就没有再用持久化，还有它的集群也还没有用。下面我介绍一个简单的场景。假设你在类似于BAT这样的流量入口做一些广告或者做一些引流的时候，可能它会瞬间给你带来巨大的流量，而如果你自己的私有云或者你自己的机房没有立刻准备好几百个机器应对时，可能会宕机。这时就有了公有云和私有云结合的方案，我们就在公有云上面申请上百个甚至几百个实例，公有云是活动的入口，你这个广告可能会面向几千万的用户，但是假设这几千万的用户就像一个漏斗层层在损失，最后能用的时候就变成了几十万，取决于你的业务特性，在最前头也就是入口这块量是最大的，你怎么解决入口这块呢？可以简单地在公有云上购买一大堆服务器，然后把流量引入过来之后，把这些请求全部丢到Kafka里，然后用Akka异步做一些处理，这些处理需要你去调你做的这个活动的接口，调用完之后再把这些东西存到Akka，由你自己的集团去搭那些处理过的东西，这样就极大的利用公有云构建了这些设备，解决了你活动的需求。

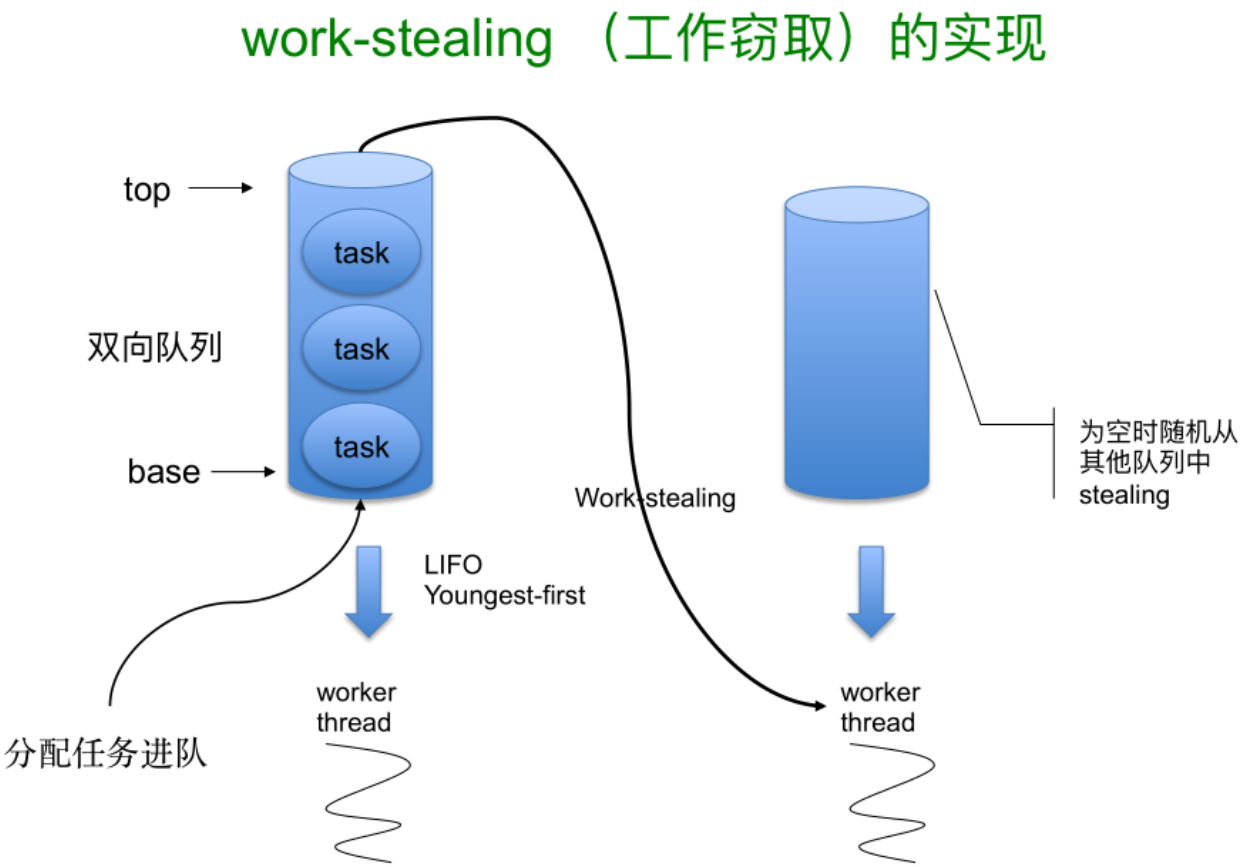


还是刚才这个例子里面我们看一下在Akka里的几个actor角色，这前面我们通过Netty把这些请求丢掉Kafka之后，后面每个Akka都会处理这些请求，第一个就是KafkaReceiver，他负责从Kafka去拉这些消息，这块采用pull而不是采用push，一个原因是我们的Kafka-client是有自己独立的线程，并非Akka里的dispatch线程，因为最早实现Kafka-client的时候，它并非为Akka所设计和使用。KafkaReceiver角色里通过一个BlockingQueue来平衡receiver与

后续actor之间处理的能力的不匹配问题。

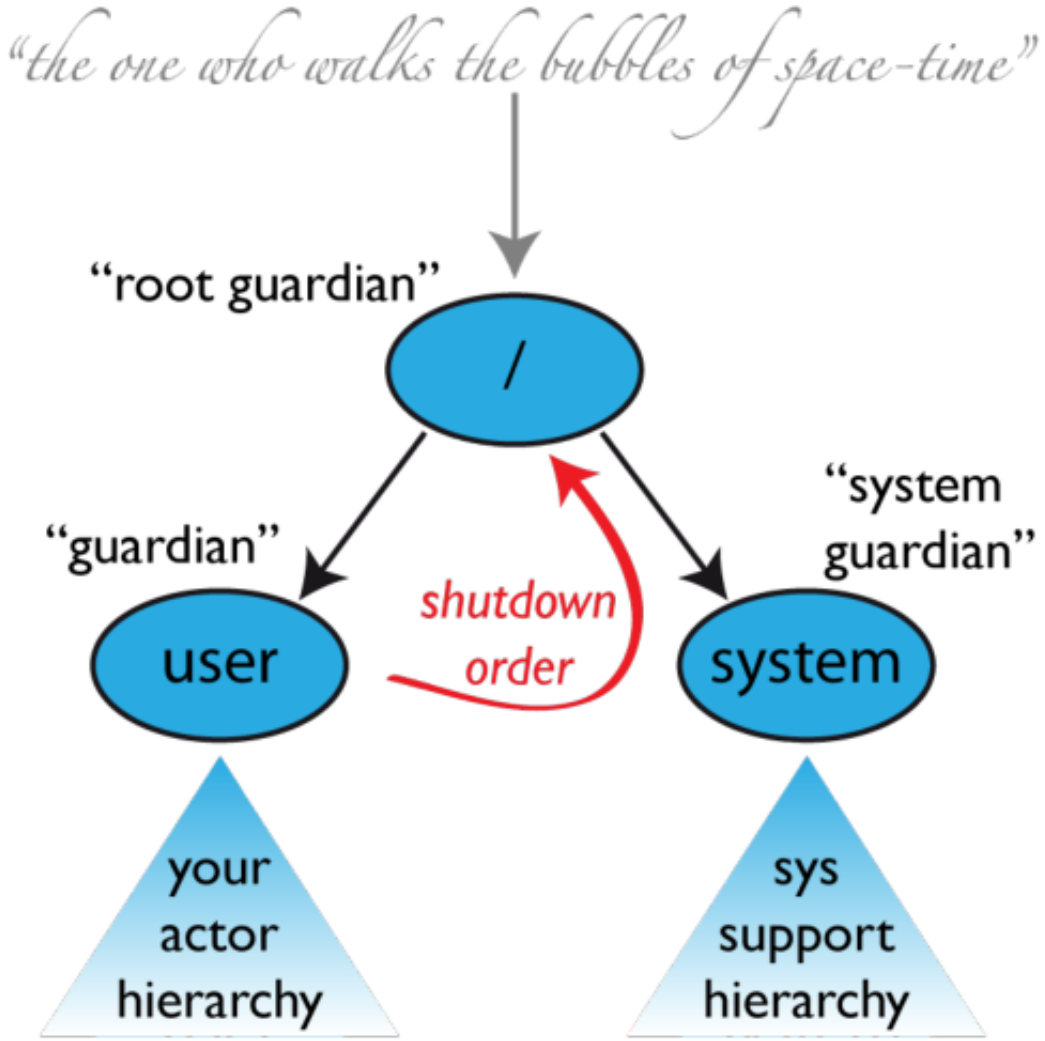


最重要的一点是在actor的世界里面千万避免同步阻塞，所有的actor底层的线程池是复用的，如果你的HTTP调用在里面堵塞住了，那么可能整个actor都可能被堵塞了，mailbox里后续的消息无法被处理，甚至整个系统都会被阻塞，所以同步在这里面是一定要避免的，当这些调用成功的时候我们交给后面一组actor，异常的时候我们会交给另外一个组，他可能会丢到另外一边，由另外一边来处理，这样就避免了你自己在一套逻辑里容错的时候要考虑一下这个问题。Akka的感觉比Scala原本自己的actor是要做的更好一些，要不然TypeSafe也不会收购Akka这个团队，在actor的模型里面，他有很天然的扩展能力，但是在JVM里，他的底层实现还是要通过线程调度，他的线程实现这块是利用非常高效的fork/join，最大的特点就是双向队列，当一个队列里面被处理完以后，线程会从其他队列里偷取别人的任务，这样会解决不均的情况，有的处理慢有的处理快，处理快的就帮助处理慢的这些，所以他在实际执行过程中Akka还是很容易把CPU跑满，所以利用率还是非常高效。



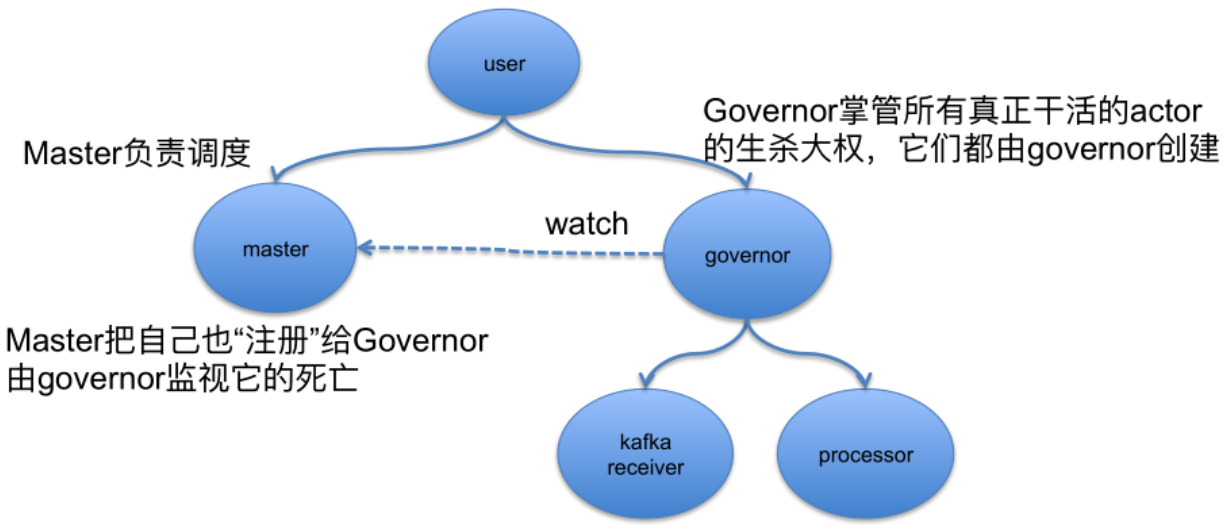
我们大部分的数据处理，是用Akka和Kafka的简单模式，就是Unix上的管道模式，现在是用分布式的方式，我们用它也还是要和业务结合，就比如说你要聚焦这个服务，而这些现有的服务都是Dubbo的一种，所以Akka与Spring整合起来还是很方便的。最后我们总结一下对Akka使用的一些小结。第一是尽可能保证Actor职责尽可能单一。第二避免阻塞，阻塞在Akka的模式下一定要避免这个方式。第三是Supervisor和错误处理也是要有自己的策略，到底是收到以后重启它还是立刻把这个上报，这个都要根据自己的业务情况确定，在消息处理的时候要保持平衡，这边生产出的那边要消费了，你的能力强和弱的时候你要自己去考虑。有些地方我们可以通过加一些简单的监控，比如对mailbox的计数，或者访问统计，将待处理的任务用WaterMark标记出来，就能够知道当前Actor的处理能力，让我们更好的了解各个Actor状况。还有常用的模式，类似于替身模式，或者短路模式等等。因为时间的关系我们就不去展开的讲。

最后分享一个小技巧，就是关于Actor的优雅关闭。其实优雅关闭的话题在每个系统里都要有的，比如你发布后，你怎么样让这个系统优雅地下线，还有你后面关闭服务的时候，Actor或Akka如何优雅关闭。你应该先关闭什么后关闭什么从而让这个顺序有所保障，这就是让你尽可能的做到优雅。



有一种模式就是我们根据Terminator模式设计的一个变种Governor模式，所有干活的Actor都由Governor来掌管，Governor是他们的父Actor，他们在消亡中的时候所有都是由Governor来掌控他们的生杀大权的。另外一个角色是Master，他不是真正干活的，他只是负责这个任务要开始真正执行了，Master跟Governor之间是协作，Master把自己注册到Governor，由Governor去watch它，当Master死亡的时候会发送消息给Governor。实现上先提供一个Governor的模板，在这里负责创建那些干活的Actor，以及顺序停止的时候先停止谁后停止谁这么一个过程。把他的顺序制订好，比如说KafkaReceiver，他相比Processor的顺序是1，然后Processor的顺序是2，大家在关闭的时候都会按照这个顺序先把KafkaReceiver给关闭了，然后再关闭Processor，最后在关闭Storer。当Master发给自己一个毒药丸说我要退出了时，因为Governor会关注Master，所以他会按顺序杀死子Actor。

Governor：Terminator模式的变种



优雅(顺序)关闭的问题

提供一个 Governor的模板，实现顺序停止子Actor

```
abstract class ContextManager(signal: Semaphore) extends Actor {

 // 顺序的停止所有子actor
  protected def stopAll(kids: List[OrderedActorRef]): Future[Any] = {
    kids match {
      case first :: Nil =>
        gracefulStop(first.actor, stopTimeout).flatMap { _ => Future { AllDead } }
      case first :: rest =>
        gracefulStop(first.actor, stopTimeout).flatMap { _ => stopAll(rest) }
      case Nil =>
        Future { AllDead }
    }
  }
...
```

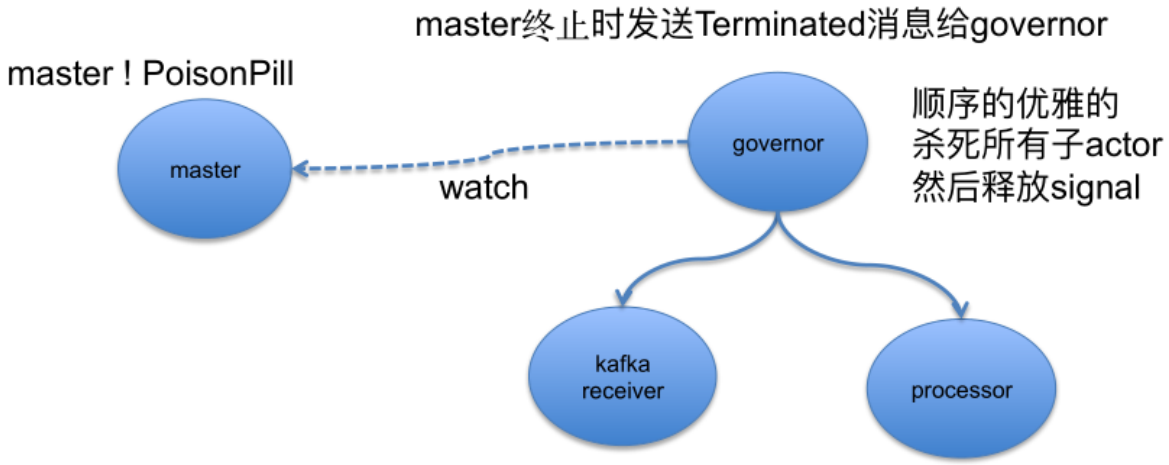
优雅(顺序)关闭的问题

实现 Governor 时，指定每个子Actor的关闭顺序，
比如下面在shutdown时先停止receiver再停止processor最后停止storer

```
class Governor(signal: Semaphore) extends ContextManager(signal) {
  override def createActors() {
    val ext = SpringExtension(context.system)
    // 1) receiver
    val receiver = context.actorOf(props(ext, "serialConsumer"), "kafkaReceiver")
    setOrderNo(receiver, 0)
    // 2) processor
    val processor = context.actorOf(props(ext, "processor").withRouter(getProcessorRouter), "processor")
    setOrderNo(processor, 1)
    // 3) storer
    val storer = context.actorOf(props(ext, "storer").withRouter(getStorerRouter), "storer")
    setOrderNo(storer, 2)
  }
....
}
```

在系统关闭的逻辑里(shutdownhook)，master发给自己毒药丸，并等待 signal，signal满足后整个akka系统退出

优雅(顺序)关闭的问题



最后推荐一些Scala相关的书。第一本是《Programming in Scala》，是Scala比较重量级的一本书，Scala之父他们几个人联合写的一本书，里面写的很含蓄但是信息量是非常大的，所以Scala真的要去消化很多遍。第二本是《Scala for the impatient》，第三本是《Scala in depth》这个是相对高级一点的书，最后一本是《Functional Programming In Scala》，这本书里面讲到了很多类型性的和函数式的比较高阶的比如 monad等一些特性。

BT