搜狗商业平台Java技术实践

发表于2015-05-20 14:56| 1478次阅读| 来源CSDN| 2 条评论| 作者刘建、郭理勇、毛宏

[特别策划](http://www.csdn.net/tag/%E7%89%B9%E5%88%AB%E7%AD%96%E5%88%92/news)[Java20周年](http://www.csdn.net/tag/java20%E5%91%A8%E5%B9%B4/news)[Java](http://www.csdn.net/tag/java/news)[《程序员》电子刊](http://www.csdn.net/tag/%E3%80%8A%E7%A8%8B%E5%BA%8F%E5%91%98%E3%80%8B%E7%94%B5%E5%AD%90%E5%88%8A/news)

**摘要：**搜狗商业平台负责搜狗广告业务，涵盖搜索、网盟、无线、品牌等业务线，其中，基于Java的业务系统在70%以上。从数据库访问、离线大规模数据处理到实时计算，都依托于Java技术。

Java自1995年问世以来，已历经20多年岁月。20年来，IT技术风起云涌，Java始终以其可移植性、跨平台性、生态系统完备性等特点成为最主流的开发语言之一。事实上，Java无处不在，已经渗入到大家的日常生活中，从你的每一次购物到每一笔支付，都有Java技术的身影，国内外的主流网站大部分都是由Java技术支撑。

搜狗商业平台负责搜狗广告业务，涵盖搜索、网盟、无线、品牌等业务线，面向几十万广告主和广告代理商，提供十亿级以上在线广告管理及相关支持，提供近百亿的在线报告。其中，基于Java的业务系统在70%以上。从底层缓存、会话、调度、通信交互，到提供给客户的API接口，从数据库访问、离线大规模数据处理到实时计算，都依托于Java技术。在我们内部长期的实践过程中，Java技术已经逐步自发地形成了一个生态系统。

Java生态圈非常庞大而丰富，我们在长期的实践过程中，自主或基于Java开源组件进行二次开发和优化，构建了搜狗商业平台完整的Java技术框架，如图1所示。

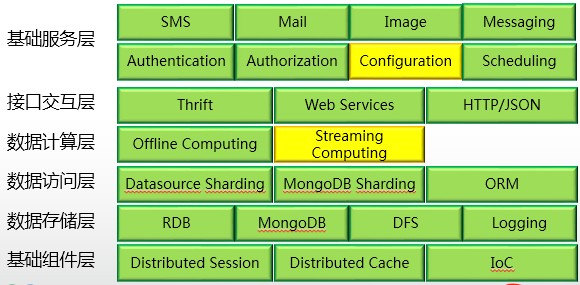


图1搜狗商业平台Java技术栈

在基础组件层，我们直接使用了一些业界著名的框架和类库，比如IoC框架Spring、日志Log4J等；同时也基于一些框架进行了二次开发，比如基于Redis提供了分布式会话和分布式缓存，有效地解决单机内存及I/O瓶颈问题。在数据存储层，我们数据存储主要使用关系数据库MySQL、文档数据库MongoDB、分布式存储HDFS以及自行研发的DFS文件系统。在数据访问层，基于MySQL数据库和MongoDB数据库，分别提供了一套分库分表框架，使得其支持海量数据存储，同时也分别提供了ORM框架，使得能够很容易地完成数据库中的数据到对象的映射。在数据计算层，离线计算方案主要使用了HadoopMapReduce框架，流式计算方案主要使用了Kafka和Storm。在接口交互层，提供了三种框架，分别支持Thrift、WebServices和HTTP/JSON三种交互方式。在基础服务层，提供了认证、授权、配置、分布式任务调度、消息、图片、短信和邮件等多种基础服务。

下面分别介绍我们在数据库分库分表框架、分布式任务调度平台和分布式RPC框架上的实践。

**数据库分库分表框架：Compass**

互联网领域针对大数据存储，基于NoSQL的数据库越来越多，然而，在一致性、事务性、可靠性等方面，特别是在较复杂的业务场景中，关系数据库仍起着不可替代的作用。在关系数据库领域，MySQL数据库基本上主导了市场。然而，互联网面临着用户多、数据量大等的挑战，一组MySQL数据库集群无法支撑如此大的PV及I/O，因此对单表的数据量以及每台机器的数据分布需要做一定的预估。在我们的实践中，对于单表的数据量有一些约束。一般说来，对于定长的记录，单表数据量最好不超过800W，极限不超过1000W；对于不定长记录，单表数据量最好不超过500W，极限不超过800W，否则在较复杂的业务场景中，可能会引起性能下降。

对于大规模数据的存储和访问，一般都采用分库分表的方案解决。主流互联网公司都提供了各自的解决方案，比如淘宝分布式数据层TDDL、百度Dbproxy等。我们也研发了数据库分库分表框架Compass，支持和满足内部的一些需求。

一般来说，数据库分布分表框架的设计有两种方案：独立中间件层方式和嵌入式应用框架方式。独立中间件层方式采用独立的部署，后端数据库对应用程序是透明的，其扩容对应用的可用性不会有明显的影响。嵌入式应用框架方式是独立的类库，应用需要显式地配置后端的单个或者多个数据库。独立中间件层方式的开发和维护成本都较高，且多了一层网络开销；嵌入式应用框架方式数据库配置较为复杂一些，但其开发维护成本低，并且对单元化架构的支持也不错，因此Compass选用了嵌入式应用框架方式。

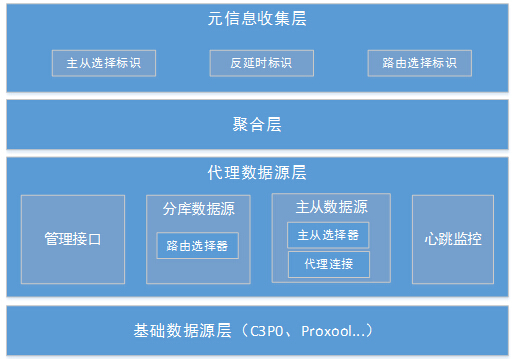


图2分布分表框架Compass架构图

Compass系统架构如图2所示。基础数据源层可以选用开源的数据库连接池方案，比如包括C3P0、Proxool、TomcatJDBC、DBCP等。在我们的实践中主要使用C3P0。代理数据源层主要负责数据库分库分表的处理，包括分库数据源、主从数据源、心跳监控和管理接口四大模块。

分库数据源和主从数据源都实现了Java中标准的javax.sql.DataSource接口，提供基于Spring注解的配置方式，对应用屏蔽底层数据源的差异，从而简化应用配置及应用在不同数据源之间迁移（包括普通数据源迁移至主从数据源、主从数据源迁移至分库数据源），降低数据库扩容成本。由于采用标准的javax.sql.DataSource接口，在JDBC封装层支持方面，我们目前可以兼容业界大部分的ORM框架，比如Hibernate、Mybatis、iBatis和JDBC等。同时由于采用对DataSource层进行分区路由，因此对于依赖javax.sql.DataSource接口的框架或应用均无影响。

分库数据源中包含路由选择器。路由选择器可以通过自定义路由选择策略选择合适的底层数据源，并将获取数据库连接的请求代理到此底层数据源。路由选择策略可通过多种方式实现：1.取模；2.分段；3.Hash；4.特殊路由策略等。在我们的实践中主要是对客户的物料进行管理，路由选择策略是通过客户ID取模实现，同时针对个别大客户使用特殊路由策略的方式，从而保证表中数据的均衡性。

对于每次获取数据库连接的请求，主从数据源根据当前的事务属性等上下文相关信息以及自身可用的底层数据库连接池信息，返回底层数据源的连接。它主要包括主从选择和代理连接。主从选择可以支持读写分离、支持多从单主模式，并支持自定义反主从延时策略，更好地保证主从库数据的一致性；在负载均衡方面，我们除了支持内置的随机、权重、轮询、响应时间等负载均衡策略外，还支持自定义扩展负载均衡策略。代理连接主要是提供了一系列的策略对SQL进行拦截，完成分表等功能。在SQL解析与替换方面，目前我们内置了供如MyBatis之类的ORM框架的SQL占位符替换基于SQL语法树解析的SQL结构替换等方式，还支持自定义扩展解析和替换策略。在主从选择择中，反主从延时策略是解决主从延迟可选插件，它是通过将一段访问时间内的读请求路由到主库上来完成的，目前可以基于常见的分布式缓存和内存方式提供反主从延迟标记的管理，并支持自定义扩展。

心跳监控主要对底层数据源的健康性进行监控。一般说来，底层数据源连接池都已经提供了心跳监控，例如C3P0能够监控数据库连接是否有效，检查数据库空闲连接以及数据库是否有效，并提供了一系列的策略来处理上述问题；然而，主从数据源以及分库数据源（特别的）需要对底层的数据源的数据库连接以及数据库是否可用进行监控，并在底层数据源不可用时提供策略进行处理（例如：对主从选择策略中的参数进行调整）。因此，数据源代理层仍需对底层数据源进行监控，并根据底层数据源是否可用进行选库或者负载均衡策略上的调整。同时心跳监控中也支持故障移除、故障恢复等功能，并能针对不同的数据库类型进行自定义监控扩展，并支持一定的JMX监控接口监控数据库连接池（C3P0/DBCP/Proxool等，支持自定义扩展），可以更好地监控其运行状况。

管理接口主要提供访问次数（读写）以及执行时间、数据库连接数（活跃、空闲等状态）的监控管理，并通过JMX、Thrift等手段暴露服务。

数据聚合层提供了数据聚合可选插件，目前支持继承已实现聚合的JdbcTemplate进行全库聚合和操作，并可采用多线程方式提高数据访问效率。

元信息收集层主要提供了一系列的工具收集主从选择的标识、反延时标识、路由选择标识，提供给代理数据源层的路由选择器和主从选择器进行路由选择和主从选择。

当然，Compass中间件在分布式数据库事务（在事务管理方面，我们继续沿用Spring事务管理机制，保证同库事务性）、跨节点join问题上依然没有很好的处理方法，在跨节点count/orderby/groupby等聚合问题上的实现也有待更优雅的支持。特别是针对全库聚合或者查询，其线程资源和内存资源占用都相对较高。这也是任何数据库中间层解决方案都未能完全解决的事情，只能依赖于在业务上进行精巧的设计，尽量避免这些问题。

此外，Compass分库分表框架也大大提升了易用性：

* 最小代码侵入：内置路由标识（Envidence）默认策略，不改变上层ORM框架实现；
* 简化配置：在配置中采用了大量的默认策略和机制，使得配置达到最简单；
* 最小接入成本：支持与目前现有的数据库访问框架并存，支持逐个数据库集群灰度切换。目前，Compass中间件已经接入了所有业务线应用，无论是从功能还是性能，与业界数据库中间件均保持统一水准，在易用性方面更是极大地简化了数据源配置和降低代码侵入，很好地支撑了目前10+亿级的物料存储和管理。

分布式任务调度平台：凌云随着业务的发展，各开发团队经常需要编写大量定时任务来进行相关的业务处理，如导入导出数据、报表计算、汇总统计等。最常见的定时任务可通过Linux系统中的Crontab来在指定时间点触发任务执行。然而，随着任务量逐渐增多，基于Crontab的任务管理方式存在诸多弊端：

1. 任务配置管理成本较高。每增加或修改一个任务，都需要修改Crontab，增加人力管理成本；
2. 无法支持任务依赖。Crontab不支持任务间依赖；
3. 手工配置出错风险高。手动修改Crontab缺少正确性校验机制，有一定出错风险；
4. 不支持任务可视化。Crontab任务只能通过Linux终端查看，没有统一的管理界面查看任务详情及历史运行记录；
5. 存在单点风险。任务每次触发后只能在一台机器上执行，若机器故障，任务将无法执行；
6. 监控不完善。Crontab任务失败后，需要为任务单独增加相应的监控。

为此，我们研发了搜狗分布式任务调度平台：凌云，将各种类型的任务统一到该平台进行配置和管理，并支持以下特性：

* 将任务的管理、调度与执行解耦合；
* 任务调度节点和任务执行节点可水平扩展，避免单点；
* 各任务隔离执行，独立平滑上线，互不影响；
* 支持复杂任务依赖，一个任务可有多个前驱和多个后继；
* 任务失败自动报警、任务超时自动报警。

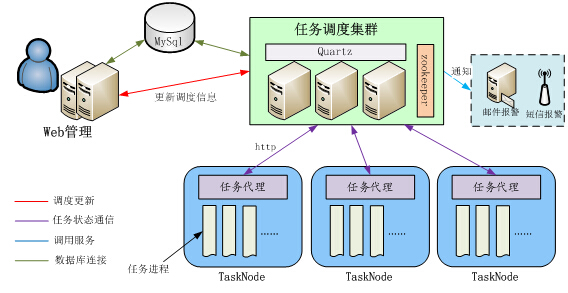


图3凌云分布式任务调度平台

其架构如图3所示。任务执行节点（TaskNode）是执行任务的服务器集合，这些服务器可划分为多个域，域主要保证不同业务间的隔离性。每个任务都需要指定所属的域，任务触发后，将会分配给其所属域中的一台服务器执行。任务执行节点负责具体任务的执行，每个任务执行节点上部署着一个任务代理（基于Thrift的HTTP服务），用于与调度集群交互并启动子进程执行具体任务。任务分为Java任务和Shell脚本任务（其中Hadoop任务属于一种Shell脚本任务），两者都是以子进程的形式由任务代理启动。

任务代理与任务调度集群的交互主要包括接收任务分发请求、处理任务状态轮询请求、任务结束后回调任务调度集群等。任务代理通过获取任务子进程的退出码判断任务是否正常结束，退出码0表示正常完成任务，非0表示出现异常，任务代理会将退出码返回给任务调度集群。

任务调度集群底层基于Quartz集群模式。Quartz是由Java编写的强大的企业级任务调度框架，提供了非常好的伸缩性、高可用性及负载均衡机制。我们在Quartz的基础上进行了扩展，支持任务的定时触发、依赖执行、失败重试及异常报警等。任务调度集群提供基于Thrift的HTTP服务，包括两方面功能：1.对任务管理中心开放任务管理和调度的相关接口，管理员对任务的调度配置会更新到任务调度集群中；2.对任务代理开放任务状态回调接口，任务在执行节点上运行完毕后，任务代理会通过此接口回调调度器告知其任务的运行结果。

任务的依赖执行可以非常方便地实现任务之间的前后顺序衔接，尤其是对于存在跨组数据依赖的情形，通过配置任务之间的依赖关系，即可实现数据获取的及时性和有效性。

用户可在任务管理中心维护任务之间的依赖关系，当任务的所有前驱任务执行完毕后，会自动触发该任务的执行。假设有如图4的任务依赖关系，任务C有两个前驱，分别是任务A和B，任务D的前驱任务是任务B，任务E、F的前驱任务是任务C。则沿时间轴方向，任务B先执行完毕，会自动触发其后继任务D开始执行；当任务A执行完毕后，会自动触发任务C开始执行；任务C执行完毕后，会自动触发E和F执行。

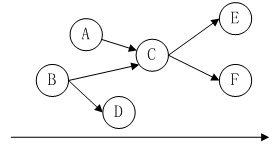


图4任务依赖关系

由于分离了任务调度与执行，调度器需要知道任务状态，并在必要的时候触发重试或者报警。我们采用了任务结果回调和任务状态轮询两种方式。正常情况下，任务执行完毕后，任务代理根据任务子进程的退出状态码确定任务是否执行成功，然后将任务执行情况通过回调的方式返回给调度集群。但考虑到可能存在任务成功完成但任务代理回调失败的情况，我们增加了调度器对任务状态的轮询策略，即对于运行中的任务，调度器轮询处理节点该任务的状态，以确保任务成功与否能及时反馈给调度器。通过轮询，对于耗时超出预期执行时间的任务，调度器会发出任务超时报警。

任务管理中心是一个对任务可视化管理平台，用户可登录此系统查看并管理自己的任务。具体包括任务管理、触发器管理、参数管理、依赖管理，任务配置项包括任务的基本属性、执行时间的触发器、任务的参数等。也可以直接在管理中心立即启动执行指定任务。系统按角色划分权限，不同角色的用户拥有不同的权限。此外，任务管理中心采用分布式会话，可以进行集群部署，提高了容灾能力。

目前，搜狗分布式任务调度平台承载着绝大多数任务调度工作。通过将任务的管理、调度与执行解耦合，减少了三者之间的互相侵入；通过提供丰富的任务配置属性和任务可视化展现，极大提高了任务管理效率；通过提供任务远程调度、任务隔离执行、任务平滑上线、任务多依赖等特性，很好地支撑了各产品线业务的发展。

**分布式RPC框架：Polaris**

在长期的业务发展过程中，对于系统间的交互，我们使用了Socket、RMI、Hessian、JSON等技术，针对每种技术，都需要维护一套相应的故障转移、故障恢复、追踪框架，对于商业平台多条业务线的大量系统交互来说，经常面临着硬件故障问题，此种交互方式引起的维护成本及故障迁移成本都是巨大的。另外，多种不同的接口技术也面临着接口兼容性的问题，例如对于RMI来说，我们就遇到了Spring从2.5.6升级到3.1.0所带来的RMI接口不兼容导致的接口调用错误问题。最后，我们内部也有一些跨语言的调用需求，比如用C++查询广告物料信息、Python脚本做统计，这些都会涉及到接口调用，而使用不能跨语言的技术（例如RMI和Hessian）成本会非常高。因此，我们需要统一内部系统接口交互方式，并降低接口的管理和维护成本。

在实践过程中，我们比较了WebService、ProtocolBuffer、HTTP/JSON、Dubbo以及Thrift等技术。其中，WebService基于XML，兼容性良好，但其序列化/反序列化性能较差；ProtocolBuffer和HTTP/JSON无服务接口/地址描述；Dubbo的框架过于庞大；Thrift在接口定义文档、数据类型支持、跨语言等方面存在优势，然而对服务地址和服务管理方面支持力度不够，并且存在类型侵入问题。在分析比较之后，我们选定了Thrift，并在此基础上开发了服务注册中心，解决服务地址和服务管理方面的问题。对于类型侵入问题，从描述语言来看，长远来看它生成的POJO代码应该可以像WebService一样，消除类型侵入（例如，Facebook提供了开源的Swift框架，已经朝这个方向迈进了一步）。我们目前是提供了一系列的方法和转换类，自主进行类型转换适配工作。

我们基于Thrift构建的分布式RPC框架Polaris如图5所示。

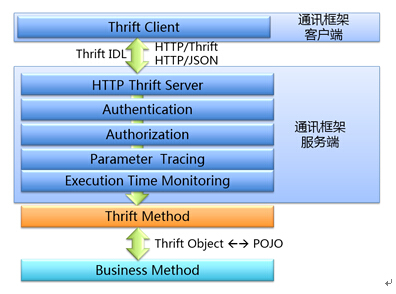


图 5 分布式RPC框架Polaris

我们对Polaris的服务端和客户端都进行了增强。在服务器端，我们提供了标准的HTTP服务器，能够很轻易地将一个Thrift服务发布到一个URL上。同时，也集成了标准的认证和授权框架，有效地保证了业务的安全性；另外，我们在运维监控方面做了很多增强，能够监控每个方法的执行时间以及执行参数，这样的话可以面向整个商业平台，建立标准的监控统计架构。因为我们提供的服务是基于HTTP的，因此部分统计是可以直接沿用线上已有的监控统计软件，也有效减少了重新构建监控框架的成本。

在客户端，主要添加了失败重试、提供了类RPC的调用方式，并且提供了一种抽象的机制简化了测试成本。通过Spring的依赖注入，可以支持在仅调整配置，不改变客户端代码的情况下进行本地调用和远程HTTP调用的切换。使用本地调用时可以更快地进行单元测试，在完成单元测试之后，可以直接通过调整配置切换成远程HTTP调用，在此过程中无需任何代码的调整。此方案还对接口迁移，比如从RMI接口迁移至Thrift接口提供了很大的帮助。一般情况下，大型复杂依赖系统内部接口间的依赖关系都会特别复杂，在进行接口梳理和迁移时成本和风险都非常高，特别是在系统的服务化改造过程中需要将内部接口提升为API接口的时候。在这种情况下，可以通过引入一个Thrift模块，将此模块依赖于需要提升为API接口的模块，而其他模块则仅依赖于此Thrift模块。在此调整过程中，可以随着业务版本的开发同步进行，而且可以重用大部分单元测试稍作调整即可使用，这样的话能够将成本和风险都降至最低。

事实上，Thrift有标准的定义语言，能够对服务接口、异常、接口参数和返回值进行描述，同时支持Set、Map之类的数据结构。然而，它没有描述服务地址以及服务依赖关系等。因此，我们也构建了服务中心Aura，它的概念架构图6所示。

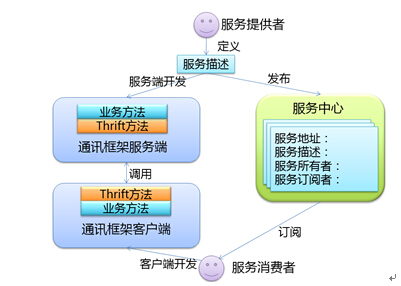


图6 服务中心Aura概念架构图

图6中的通信框架服务端和通信框架客户端已经在前面介绍了。服务中心的主要功能是对Thrift接口描述语言文件进行管理，管理其发布/订阅关系。实际上，它定义了一种团队间互相协作的方法，此协作遵循面向服务体系结构。在开发实践中，我们不鼓励依赖通过接口描述语言直接生成的代码，鼓励依赖于接口描述语言生成代码，这样能够保证耦合性，减少代码升级所带来的成本。另外，通过管理服务的所有者和服务的订阅者，能够迅速地了解到服务间的依赖关系以及服务演化时所带来的相关影响。

在商业平台内部有多个业务系统使用分布式RPC框架Polaris，包括资金、计费、客户、财务、合同、物料、报告等，也覆盖了Java、C++、Python等语言，目前已经完成大部分接口的迁移，利用服务中心Aura，构建了完善的服务发布、发现和使用的流程，也针对其安全性、易用性和可维护性做了很多工作，因为有了一致的技术集，连怎么去使用都有了最佳实践，迁移后的服务接口显著减少了重复开发成本，有效地提升了沟通效率，降低了风险。

**总结**

在长期的实践中，我们一直基于Java技术，致力于解决分布式、高并发、大数据量、强一致性等带来的各种技术难题及挑战。同时，我们也致力于基础组件和框架的统一化，构建和持续优化基础架构、基础服务，保证高可靠、高性能、高可扩展性、低成本的快速支撑各项新业务，保证业务的高速发展。在此过程中，Java及其生态系统有效地进行了支撑，成为了搜狗商业平台Java生态系统的基石，显著降低了开发和维护成本，提高了可维护性和系统健壮性。

**作者简介：**

刘建，搜狗架构师，商业平台基础平台负责人，十年Java相关研发经验，在互联网软件体系结构、分布式计算、面向服务体系结构、用户身份安全等方面有浓厚的兴趣及实践经验。目前主要关注分布式高可用软件架构和大数据基础架构。

郭理勇，搜狗高级研发工程师，商业平台网盟广告技术负责人，主要关注Java\J2EE及其相关技术、分布式高可用软件架构。

毛宏，搜狗高级研发工程师，商业平台基础服务技术负责人，主要关注Java开发、Hadoop及其相关技术。