

架构师

ARCHITECT



热点 | Hot

Google发布IPv6的应用情况

推荐文章 | Article

Java 老矣，尚能饭否？

当DevOps遇见AI，智能运维的黄金时代

观点 | Opinion

学会思考，而不只是编程

为什么 AI 工程师要懂一点架构？



卷首语

陈思

有的时候，我们需要慢下来去思考一些事情。

身为技术人的你，在不同的城市、不同的企业，为了各自不同的目标奋斗着。技术圈每时每刻都在发生着变化，瞬息万变这个词语来形容现在的技术发展实在是再合适不过了，于是，每一个技术人都需要比原来更加努力的去学习、接受新的技术，去不断去追赶那些站在技术之巅的大咖。所以，有人每天一多半的时间都在和代码较劲，然而总有新的问题出现，总有 bug 需要修复，总有一些不知道原因的问题在“重启大法”之后得到了莫名其妙的修正……

在被上述各种问题折磨的濒临崩溃而又毫无解决办法的时候，说明你需要停下来思考了。不妨先停一下手里的工作，打开我们给你的这本《架构师》，慢慢品读一下这些为你精心挑选的文章，也许你能得到一些启示，那些令你头疼问题的解决方法也许就在阅读和思考中找到了答案。

代码能够让世界变得更加美好，制造代码的人都在致力于改变世界，但是计算机和程序只是工具，它们是我们通向终点的桥梁，正如本期《架构师》中这篇《学会思考，而不只是编程》里说的那样：我们真正的目标应该是教会人们如何思考。换句话说，我们应该教人们计算机科学，而不仅仅是编程。

如果你暂时没有得到太多的启发，那么就把这段阅读时间当作短暂的休息，起码你紧张的大脑有了一个休息的空当，趁这个时间，多看两篇文章，给大脑充电，再来一杯咖啡或清茶，相信你很快就会满血复活。

技术人们以先行者的身份最先看到世界的变化，在忙着看世界的同时，别忘了给自己留下一些时间思考，《架构师》希望能够给你更多启发，陪你走得更远。

前瞻热点一睹为快

2017年10月17-19日 | 上海·宝华万豪酒店



前瞻热点 实践案例



《How Top Data Science Teams are Driving Risk-based Security》

Hui Wang PayPal Head of Global Risk Modeling



《方圆并济：基于 Spark on Angel 的高性能机器学习》

黄明 腾讯数据平台部 T4 专家



《产品经理 & 技术，如何组成一支特战小队》

快刀青衣 罗辑思维 / 得到 App 联合创始人



《如何提升 Spark 效能并使其完整发挥硬件效能》

陈韦廷 Intel资深软件工程师

技术大咖 重磅加盟



王伟钊

Google
Senior Staff Engineer



Kingsum Chow

Alibaba Infrastructure Services Chief Scientist



王绍翻

阿里巴巴
高级技术专家



张彭善

PayPal 大数据研发架构师
/ 资深数据科学家



孙志岗

网易
教育事业部战略总监



张灿

百度外卖
研发中心总监



Gurinder Grewal
PayPal Chief Architect
Risk and Compliance Management Platform



宗志远
爱奇艺
资深算法工程师



Eric Kim
LinkedIn
数据基础设施团队高级经理
新浪微博 安全高级算法工程师



何为舟

新浪微博
安全高级算法工程师



吴名扬

Movoto
前端工程师



侯栋

美丽联合集团
高级安全工程师

8折 优惠报名中, 每张立减1360元

截至2017年08月20日前 团购享受更多优惠



CONTENTS / 目录

热点 | Hot

Google 发布 IPv6 的应用情况

推荐文章 | Article

Java 老矣，尚能饭否？

当 DevOps 遇见 AI，智能运维的黄金时代

观点 | Opinion

学会思考，而不仅是编程

为什么 AI 工程师要懂一点架构？

理论派 | Theory

整洁代码之道——重构



架构师 2017 年 8 月刊

本期主编 陈思

流程编辑 丁晓昀

发行人 霍泰稳

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

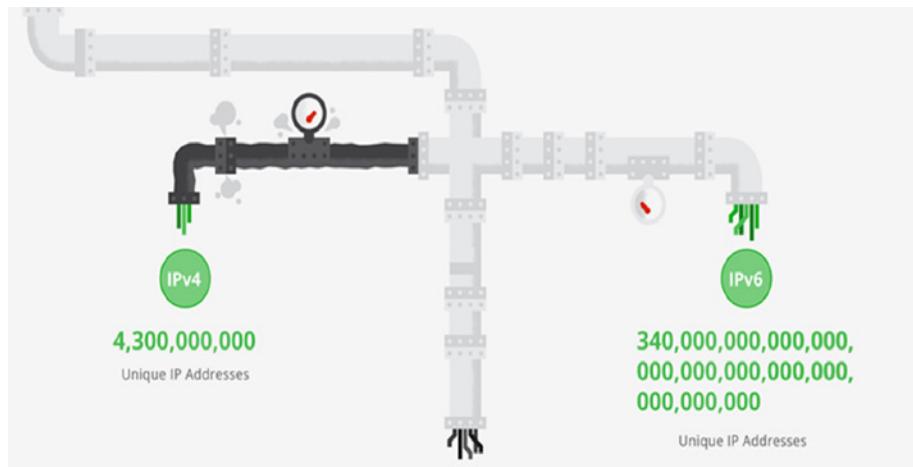
Google 发布 IPv6 的应用情况

作者 薛命灯



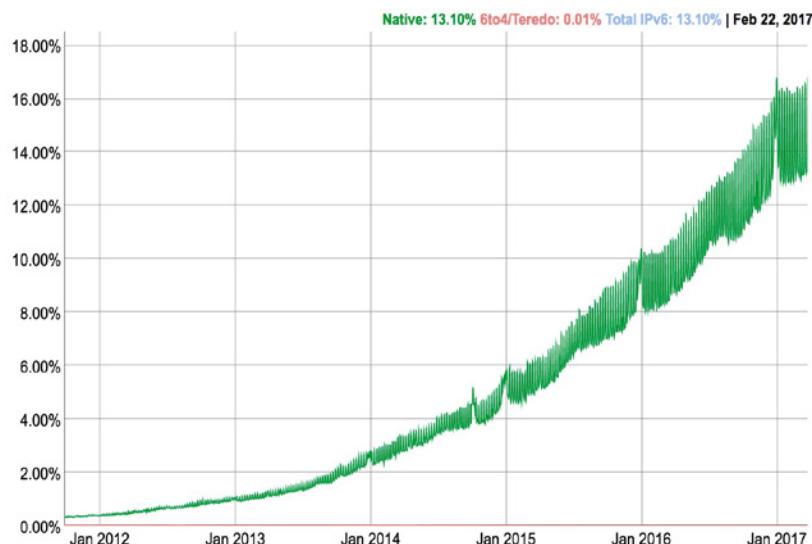
Google 在它的网站上发布了 IPv6 的应用情况，希望借此为网络服务供应商和网站运营者在向 IPv6 迁移时提供一些帮助。

就像每个手机都需要一个号码一样，每一个接入互联网的设备都需要一个 IP 地址。IPv4 只能提供 43 亿个地址，而全球人口超过 60 亿，人均都分不到一个 IP 地址。而随着智能硬件的不断发展，越来越多的硬件需要接入互联网。电脑、手机、电视、穿戴设备、汽车、冰箱、空调，等等，这让原先人均不足一个 IP 地址的 IPv4 已经无法满足需求。而 IPv6 可以提供的地址为 $340,000,000,000,000,000,000,000,000,000,000,000,000,000,000$ (34 后面 37 个 0) 个，这么多地址足够全球用户用上几辈子。



使用 IPv6 代替 IPv4 是必然的，不过这并不是一个一蹴而就的事情，而是一个循序渐进的过程，在相当长的时期内，两种 IP 格式会共同存在，直到不再需要 IPv4 为止。世界 IPv6 于 2012 年 6 月 6 号正式启动，从这一天开始，世界各大网站和网络服务供应商启用了 IPv6，并逐步开始从 IPv4 迁移到 IPv6。那么经过几年的时间，现在进展如何呢？

Google 所展示的数据是基于使用 IPv6 访问 Google 网站而统计出来的。从图中可以看到，从 2012 年到 2017 年，IPv6 的采用率一直呈现出稳步增长得趋势。从 2012 年 6 月份的 0.64% 到 2017 年 3 月份的 16.74%，翻了 25 倍之多。



上面这组数字是全球 IPv6 采用率的一个整体情况，下面再来看看各个国家的 IPv6 采用率情况。下图是一张世界地图，图中的绿色部分表示部署了 IPv6 的国家，颜色越深表示部署的比例越大。而红色部分则表示没有部署 IPv6 或者虽然部署了 IPv6 但存在严重的稳定性和延迟问题。

从图中可以看到，IPv6 采用率最高的国家是比利时，为 51.65%。然后是美国，采用率为 35.2%。接下来是希腊和德国，采用率分别为 32.38% 和 30.85%。印度和巴西也算采用率比较高的国家，分别为 22.42% 和 20.29%。其他国家的采用率大都在 10% 以上，如澳大利亚、法国、英国、日本、新西兰、芬兰、马来西亚等国，大都在 11% 到 19% 之间。还有一些国家在 10% 以下，如越南、泰国、韩国、阿根廷等。波斯瓦纳和哈萨克斯为红色，说明 IPv6 采用率很低，而且存在严重的延迟和稳定性问题。

中国的 IPv6 采用率仅为 1.2%，原因可能是多方面的。一个是因为中国人口基数大，互联网用户数量群也很大，所以迁移速度会比较慢，相应的比率也相应较低。另外，因为网络访问的限制，可能很多中国互联网用户无法直接访问 Google，导致 Google 无法统计到这些数据。

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？

1. 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，一片神鸦社鼓。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。

2. 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 Weblogic。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。

3. 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。GreenTeaJUG 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 Eclipse 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。

4. 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；强大的硬件厂商，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，

Jetbrains 等等。这些公司共同对 Java 的发展起到关键作用。

5. 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 最大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。

6. 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 jschell 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 jschell 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 private method) 等也是非常重要的新特性。

7. 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。

8. 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。如果说要说值得考虑的一些功能，我觉得有以下几点：

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。
6. 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。
 7. 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。

9. 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。

1. 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。
2. 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。

3. 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。
4. 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。
5. 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。

当 DevOps 遇见 AI，智能运维的黄金时代

作者 万金



“弱人工智能”（Narrow AI），是在某些特定领域高效完成任务的专用人工智能，比如识别图片中的内容或是通过搜索大量医学临床案例为医生提出治疗建议的专用人工智能。目前可实现的人工智能的本质是，人来提出目标，由机器分析大量数据人来高效找到答案。

人工智能应用的分类

很多情况下人工智能无法给出 100% 正确的回答（其实人类也是一样的），如何找到人工智能善于解决的问题就成了首要的任务。

人工智能应用可以分成三类：

- 核心业务，失败不可接受。医疗，银行，法律。

- 核心业务，失败率可接受。自动驾驶，自然语言理解。
- 非核心业务，对失败不敏感。用于改善用户体验。

从人工智能发展和应用的过程来看，通过对感知的模拟，帮助人类做决策，直到完全代替人类处理大量重复的数据方面的工作。

另一方面，由巨大商业利益推动的人工智能将很快成为现实，自动驾驶商业应用会带来客观的商业价值比如：

- 人为交通事故减少，保险费降低，无人参与驾驶，用车成本减少到五分之一；
- 按需用车，汽车保有数量会减少到三分之一，导致车商业模式变革；
- 车辆流量变化，大量节省道路和停车场的面积，导致城市规划改变。

人工 + 智能才是最佳的组合方式

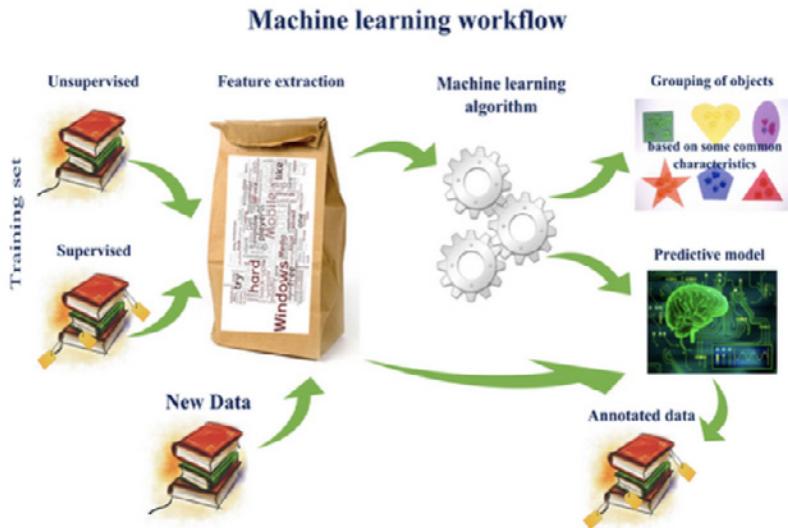
卡斯帕罗夫和李世石真的败给了机器吗？

(IBM 的深蓝和谷歌的 AlphaGo) 在人类选手的对面，是人工智能汇集了所有人类智慧和经验的智能流算法，如果是这样的话人类必败无疑。

但反过来想如果人类也有一个人工智能辅助来比赛呢？那胜负就未可知了。

卡斯帕罗夫在被深蓝击败后，发起了自由式国际象棋比赛，可以使用人工 + 智能（半人马选手）方式参与比赛，由人工智能给出建议，人类来决定是否采纳建议。2014 自由式国际象棋对抗比赛人类赢得了 42 场半人马选手赢得 53 场，当前最优秀的国际象棋团队都是半人马选手由人类和人工智能组成。既然人工智能可以帮助人类成为最优秀的象棋选手，那么可以推测人工智能也能帮助人类成为最优秀的医生、飞行员、法官 和教师甚至是运维和开发人员。

人工智能的工作方式和解决的问题种类



上图为典型的机器学习流程（图来源于 Natalia Konstantinova 博士的博客）

典型的机器学习通过无监督学习和监督学习，抽取特征，再通过机器学习算法；实现基于通用特征分组，得出预测模型，通过预测模型为新数据数据打标签。

机器学习可以解决以数据为基础四类问题：**逻辑推理预测、规划师、沟通者、体验与情感。**

由 Ajit 的一片博客中总结到人工智能善于解决的 12 中问题

1. 领域专家：模拟现场专家给出建议
2. 领域扩展：给出新见解新方法。
3. 复杂规划师：比非AI算法易于优化
4. 更好的沟通者：智能代理，自动语言翻译
5. 新感知能力：机器视觉产生了自主车辆
6. 企业AI：改进业务流程
7. ERP AI：通过认知系统增强ERP
8. 跨界影响预测：比如自主车辆导致司机岗位需求降低；人为交通事故减少，保险费降低；按需用车消费导致车企商业模式变革，车辆流量变化，导致城市规划改变。

9. 目前算法和硬件问题无法很好解决的问题：语音识别达到人的能力。
10. 更好的专家系统：通过资料无监督学习获取知识
11. 超长序列模式识别：时间序列预测模型
12. 情感分析：通过行为预测人类情感的变化

运维发展的历程与人工智能的作用

	手动运维阶段		规模化阶段		生态化阶段	
阶段	初始化	专业化	工具化	平台化	云化	智能化
描述	运维从研发分化出来，负责业务以外的事物。	有细分的分工，稳定，便捷，可靠，快速	随着DevOps概念提出工具大量涌现	运维规模扩大，业务运维SRE产生，保障业务平稳发展	云原生应用将基础运维交给云平台，让SRE更专注于业务可用性	使用云计算大量数据喂养的人工智能使得智能化成为可能
分工	Op	SA, IDC, DBA	DevOps, IDC, DBA	SRE, DevOps, IDC, DBA	SRE, DevOps	DevOps, 云平台运维
运维能力	100os/人	500os/人	2000os/人	5000os/人	20000os/人	？？？
业务规模	小型网站，内部系统	小型公司业务系统	中型公司业务系统	大公司多事业部业务系统 (BAT)	云计算供应商和用户 (AWS, MS, Google)	

OP: 运维; IDC: (IaaS) 数据中心运维; DBA: 数据库管理员; SA: (PaaS) 系统运维; DevOps: 应用研发平台运维; SRE: 站点稳定性工程师

运维行业经历了初始、专业化、工具化、平台化、云化和智能化过程。从手动运维阶段基本没有数据，到规模化结构化数据和智能化非结构化数据的趋势。

人工智能发展初期充当辅助人类的助手角色，以增加销售额，提升用户体验，优化生产过程和节省成本为目标。

手动阶运维阶段

运维工作量小运维人员主要工作就是看监控屏幕，随着对运维要求提高，工作分工此阶段产生，产生了稳定，便捷，可靠，快速的工作原则。

人工智能可以做的是：基于人的经验，对结构化销售数据进行商业智能分析找出数据中的知识，从而提升销售额。存在的问题是主要是数据专家基于经验发现业务数据中的知识，对业务了解程度成为 BI 有效性的最大瓶颈。即缺乏即懂业务规则又懂数据发掘的人才阻碍商业智能的发展。

规模化阶段

随着 DevOps 概念的推出，工具大量涌现来协助运维工作运维能力大幅提升，带来问题是很少有一家公司可以生产覆盖所有 DevOps 生命周期的工具，而学习多种不同厂商的工具完成任务带来很高的技术门槛。随着一些创业型公司崛起，运维工作量爆发式增长，为了保证业务的连续性 SRE 也在此时期产生，主要目标是使用软件工程技术实现业务大幅增长而运维工作了保持平稳。

人工智能可以做的是：出现以结构化数据为主工业级解决方案，使用算法为主解决商业通用问题，以提高人员利用率加快创造价值为典型问题。

同时也存应用了工业级智能解决方案有多大的效率提升很难估算和当数据知识变化后很难进行跟踪优化的问题。

生态化阶段

随着互联网规模的发展，少数大公司承担起基础设施的工作，通过高度集中提升数倍的运维效率（在亚马逊购买 1 美元的基础设施，可以带来与传统数据中心 7 美元投资相同的计算力），这种变革让云计算客户专注于业务的发展将基础设施运维交给云计算平台。市场规模继续增长一个公司无法使用一套解决方案覆盖所有细分市场的需求，生态化从而产生。因此大量的数据为人工智能实用化奠定基础。

人工智能可以做的是：出现以非机构化数据为主通用的技术框架，不同的公司负责一部分问题形成生态圈，协助业务人员完成工作，通过新感知能力半自动或自动化完成以前手工的工作。

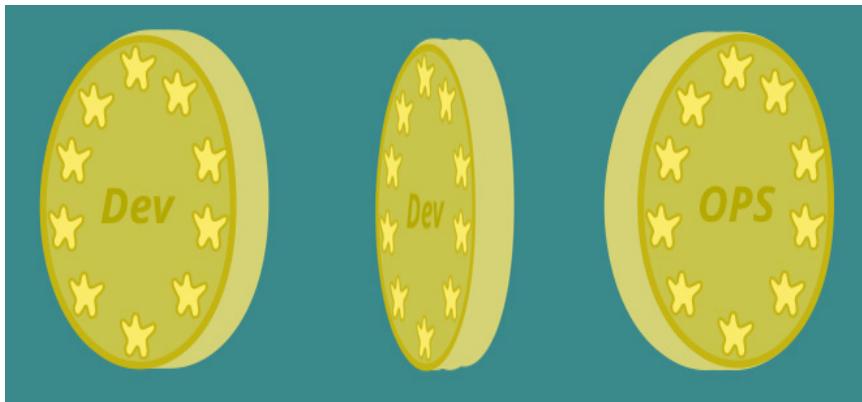
如何结合新的感知能力辅助人类在巨大数据量，变化的规律中做出决策成为新的问题。

用矛盾论的观点分析 DevOps

DevOps 的本质是在解决矛盾的对立与统一的问题

DevOps 存在矛盾的两方面，我们做的事情无外乎一分为二，最终二

合为一 矛盾论。



这是第一次西方的 DevOps 方法论与中国的矛盾论结合，其实所谓的方法论要不就被认为是废话（一般性原则），要不就是不被人理解（太深奥）。不妨我们往下看看，一分为二和二合为一是什么意思。

我们先抛开 DevOps 的定义，假设我们 DevOps 要做什么事情，他就像足球比赛开始时候裁判抛出的硬币正面或反面朝上，来决定由哪一方先发球，先发球就意味着具有很大优势，但是双方认可这枚硬币来作为双方都可以接受的方式来开始一场比赛。这就是 DevOps 在研发和运维工作中起到的低成本的沟通协调的作用。

很有趣的一点就是随着 DevOps 理论的提出各种工具（硬币）大量涌现，这些工具只不过提供了比抛硬币复杂一些的规则而已。而人工智能会给这些工具带来增强效果。

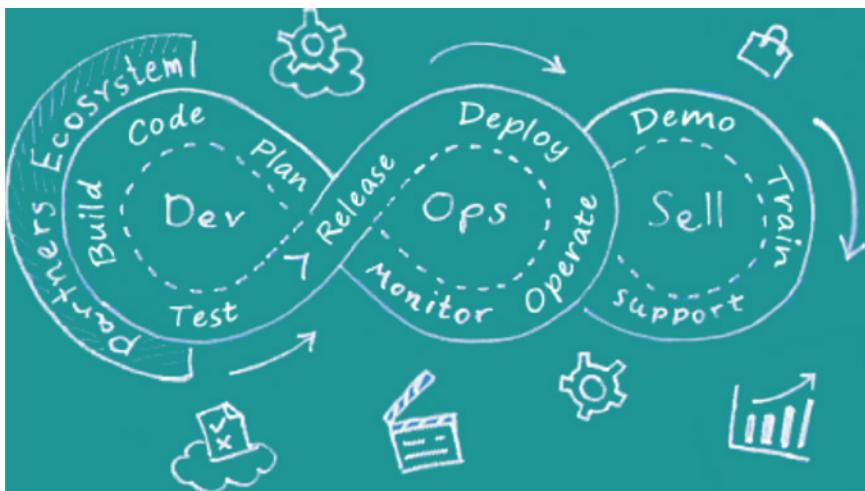
不能一味的追求事物的某个方面而忽略另一方面

我们在回过头来看一分为二是什么意思。

研发追求功能的吞吐量，主要关注需求实现时长，发布频率和部署前置时间。而运维追求稳定性，主要关注部署成功率，应用错误率，事故严重程度和严重 bug。这本来就是一对不可调和的矛盾。

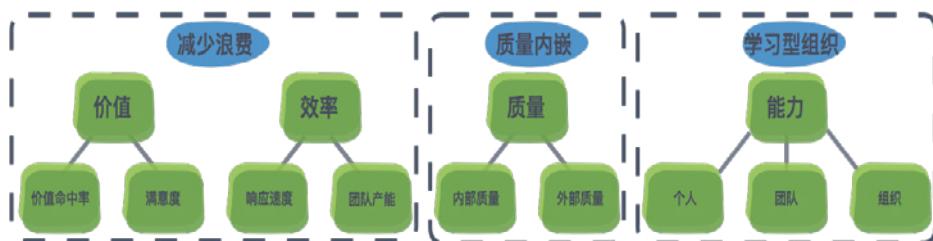
但是从更高的维度看，只做好吞吐量或是稳定性，不能带来性能提升、体验提升和业务成功。当我们确定了运维和研发的共同目标 — 即业务成

功后，问题就变成：为了共同的业务成功，研发和运维在 DevOps 协同过程中，不会一味最求吞吐量或是稳定性。



为什么人工智能在 DevOps 中大有可为？

DevOps 可以获取几乎所有类型的数据。指标体系框架来自《精益软件度量》。



我们了解到人工智能解决的问题都是以数据为基础的，那么有了价值、效率、质量和能力方面有指标和数据就可以在 DevOps 过程中通过人工智能解决问题了。

找到 DevOps 全生命周期中人工智能可以改善问题

在 DevOps 生命周期中还有很多工具无法实现自动化的过程，这些过程往往需要投入大量的人力和沟通成本，也有很多信息不足无法做出很好决定的场景，在这些场景中人工智能可以根据以往大量数据训练的模型，给

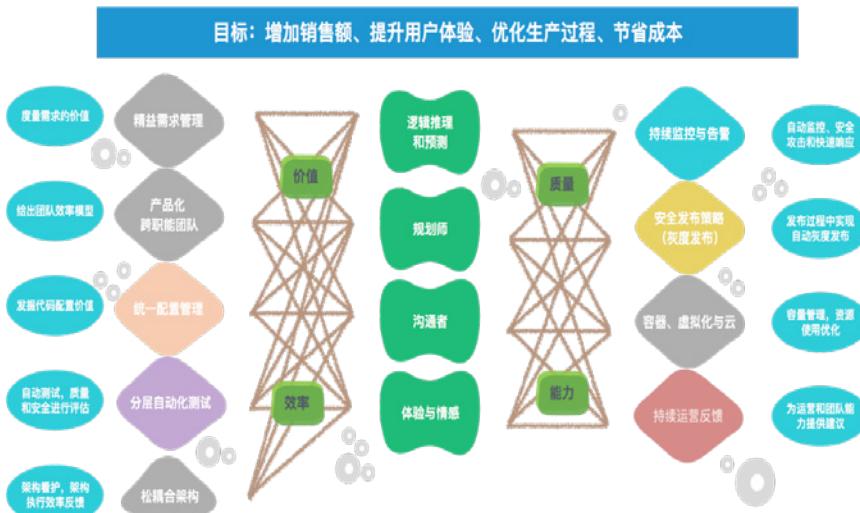
出建议，从而给出研发和运维都能认同的工作方式，提升工作效率提高工作质量。

高度自动化是“精益生产”的核心支柱之一，而自动化的第一步是标准化，智能化是未来的服务界面。



当人工智能牵手DevOps

我们现在有了 DevOps 生命周期中的数据，同时也了解到人工智能易于解决的四类问题。我们可以尝试使用全连接的方式找出 36 (4x9) 个在 DevOps 领域里适合使用人工智能解决的问题。



比如上图中的“精益需求管理”过程中：通过价值和效率数据使用逻辑推理和预测人工智能方法，得出需求的价值命中率和客户满意度的预测。通过这些人工智能得出的标签优化需求的优先级管理。这样从完全靠人工

经验的过程变为人工智能辅助完成的高效过程。

到那时需求人员只需要调节想得到的转化率（运营指标），或是性能（运维指标），就可以通过人工智能方式自动提升改善这些指标的需求的优先级。甚至根据需求改变的特性，分析大量现有代码库中的通过测试的代码而自动为开发人员推荐代码。

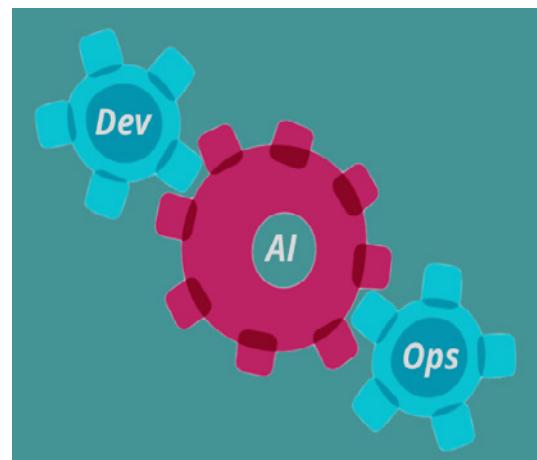
“分层自动测”过程中：使用图片转文字，方式将必须人工完成的测试自动化弯道提高测试效率与准确率。

“持续运营反馈”过程中：通过自动化运维根因分析，提升定位系统问题的效率。

找到成功的第三条路，在吞吐量和稳定性中间建立平衡让两方面都能正常行事。

再说“最终二合为一”的矛盾论下半部分。通过人工智能给出的第三种方式让双方都能向自己的目标前进，从而为一个共同业务成功目标结合为一个整体。

比如，在“安全发布策略（灰度发布）”阶段，使用研发团队的能力数据、内在质量数据和历史中发布后的外部质量数据训练机器学习模型。通过这个模型在发布阶段给出软件发布后外部质量评估。在发布过程中对比前后两个版本的外部质量差距来决定是否进入灰度发布的下一个阶段。回想当初那个正面是 Dev 反面是 Ops 的硬币，这个 AI 的工具是不是先进了不少呢？



人工智能可能的应用

通过 DevOps 过程获得数据，使用人工智能提升服务或产品竞争力需要想象力。唯一限制我的是我们的大脑想不到的伟大的想法，而不是什么

我们做不到。下面想象一下科幻大片里的一些场景。

1. 终结者自动驾驶一样，通过监控系统反馈，进行自动化软件发布过程
2. 安德的游戏一样动态监控互动界面，像一个孩子通过 AR 方式指挥庞大的舰队战胜外星生物那样管理系统软件与硬件的运维工作
 - a. 告警抑制
 - b. 服务自愈
 - c. 主动运维
3. 像钢铁侠超大别墅中智能管家一样的容量规划管理
 - a. 自然语言交流
 - b. 容量预测规划
4. 像黑客帝国杀毒程序一样的风险合规管理
 - a. 攻击特征提取
 - b. 主动防御黑客攻击
5. 像乔布斯一样的先验用户体验
 - a. A/B 测试 (fisher's exact test 费歇尔精准检验)
 - b. 软件质量评估

不得不提的是 1996 年 12 月苹果公司收购了 NeXT，使乔布斯回归苹果，到 2011 年 10 月乔布斯去世，苹果市值在 15 年时间里从 30 亿美金增长到 3470 亿美金，增长 115.7 倍。不得不说乔布斯的用户体验感知能力足够科幻了。

智能运维领域案例

2014-2016 百度通过机器学习实现了被动异常监控和主动的流量调度算法。

腾讯通过人工智能帮助我检测和打击欺诈犯罪

很多公司都在人工智能方面做出了尝试，在目前寻求差异化的时代，人

工智能可以成为产品的一个亮点进行差异化竞争，成为避免价格战的手段。

智能运维行业应用参考：

- [Twitter: Seasonal Hybrid ESD \(S-H-ESD\)](#)
- [Netflix: Robust PCA](#)
- [Linkedin: exponential smoothing](#)
- [Uber: multivariate non-linear model](#)

尾声——在很长一段时间里人工智能不能完全取代人类

人工智能不会导致大范围失业，随着人类的技术发展，生产力的提高，人口在增加，不过失业率没有明显上升反而生活水平在提高，更多的工作产生了，人工智能也不过是一种提高生产力的技术而已。 随着生产力提升各种细分低技术含量工作也会继续细化吸收被释放的低端劳动力，产生比如细分的家政服务换季整理衣服，高层建筑擦玻璃。

再看我们的祖辈从事工业和农业工作，那个年代无法想象会有靠动动手指就能养活自己的程序员的职业。而我们大多从事技术工作，而不是工人或农民。机械化取代了大量农业劳动的同时，生产率的提高了，价格的下降，随着产业发展更多需求来带更多投资。而周边产业获得投资日益发展带来更多高端就业机会，比如生物科技农药化肥，农用机械也随之产生。

目前人工智能没有自我意识，就像人类根据鸟类的启发造出飞机一样，人类和人工智能是不同的智能，无法让人工智能像人类一样自主学习。可以预见在相当长的时期内人工 + 智能的组合方式会成为主流，由人类意识来确定目标（定义模型训练的损失函数），由人工智能高效实现。

所以目前大多数人工智能只应用在非常狭小的领域，这些人工智能虽然高效但“自闭” 在自己的领域里。比如也许人工智能可以与人类高效的沟通，但他并不能像相声演员一样给你带来愉快的沟通氛围。

在可预见的未来的很长一段时间都里，人类把一部分运维工作交给人

工智能，然后去创造新的运维工作，随着新运维工作不断成熟又会把一部分工作交给人工智能不断循环。

作者介绍

万金，Thoughtworks 高级咨询师，10 年 + 工作经验，知名外企与中国企业的 IT 从业经验，包括 IBM、华为、中兴、Thomson。具有 7 年云计算相关经验，多系统的研发和运维经验，熟练掌握敏捷和 DevOps 方法论和实践，具有软件研发生命周期工具与流程改进丰富经验。

听说你是做运维的 想了解运维最新技术趋势和实践?

来CNUTCon学习一定是你的不二选择!

9.10-11 上海 · 光大会展中心大酒店

CNUTCon2017是InfoQ举办的顶级运维技术盛宴，目前已邀请了来自 Google、Uber、eBay、IBM、BAT等公司一线技术大咖前来分享智能化运维、SRE与微服务、运维监控与安全等一系列前沿技术和最新实践，精彩议题抢先看！

部分演讲嘉宾

Uber SRE以及Cache服务在微服务环境下的演进



孟飞

Uber SRE存储部门高级工程师

基于Kubernetes的互联网Ingress实践



孟凡杰

eBay软件工程师

微服务场景下的Serverless架构实践



黄博文

ThoughtWorks高级软件工程师

京东JDOS2.0平台有状态服务编排实践



王华夏

京东商城资深架构师

百度大规模时序指标自动异常检测实战



王博

百度资深软件研发工程师

苏宁大数据平台运维实践



王志强

苏宁云商IT总部技术总监

阿里巴巴国际环境下的SRE体系



周志伟

阿里巴巴AliExpress SRE负责人

全方位的监控与智能透明的自动化运维



邸富杰

IBM CIO DevOps 教练

腾讯游戏容器云平台的演进之路



尹炜

腾讯高级工程师

构建微服务下的性能监控



吴昆

华为性能监控专家

腾讯TB级别的海量日志监控平台



吴树生

腾讯SNG监控负责人

京东物流系统自动化运维平台技术揭密



赵玉开

腾讯SNG监控负责人

9折抢票

9折抢票，团购享受更多优惠 扫码了解智能时代的新运维



学会思考，而不是编程

作者 Yevgeniy Brikman 译者 薛命灯



中国人常说“授之以鱼不如授之以渔”。如果说教授编程是授之以鱼，那么教授计算机科学就是授之以渔。为什么说学习计算机科学比学会编程要重要得多？来听听 Yevgeniy Brikman 的解释。

现如今，似乎每个人都在学习编程：Bill Gates、Mark Zuckerberg 和 Chris Bosh 这些名人在 Code.org 网站上告诉人们每个人都拥有编程的能力；CoderDojo 项目在世界各地大张旗鼓；英国已经把编程作为小学官方课程。

不过，我认为这样有点误入歧途了。但请不要误会——我也确实认为代码能够让世界变得更美好——但编程本身并不是我们的目的。计算机和程序只是工具，它们是我们通向终点的桥梁。

我们真正的目标应该是教会人们如何思考。换句话说，我们应该教人

们计算机科学，而不只是编程。在这篇文章里，我将会解释这两者之间的区别，以及为什么在这两者当中选择正确的一方对于迈向成功来说如此重要。

我们不妨先问自己一个问题：为什么我们要关心编程或计算机科学？

欢迎来到真实的世界

或许你正在使用 Chrome 或 Firefox 阅读这篇文章，这些浏览器可能运行在 Windows 或 macOS 上，而你可能正在使用笔记本或台式机。你今天可能花了一些时间阅读电子邮件、查看朋友圈的状态，或者在视频网站上看了一些视频。我们的生活在很大程度上依赖了计算机：我们的医疗记录保存在数据库里；我们的简历可能放在了 LinkedIn 上；我们使用 Google 或 Facebook 为产品做宣传；我们在 Amazon 上购买这些产品；我们直接在网络上进行报税；我们使用网上电子银行；我们甚至开始涉足数字货币。

现在将你的视线从电脑上移开：在你的口袋里或者桌角的某个地方可能放着你的手机，它装有 GPS、摄像头、触摸屏和大量的应用程序。在你的客厅，可能有 LCD 电视机、DVR、DVD 播放机、Apple TV、Xbox 或 PlayStation。这些设备上的视频、音乐和游戏都是由计算机图形和数字音频组成的。

当你经过你的汽车时，你是否想过，现代汽车是使用软件设计并在满布机器人和计算机的工厂里生产出来的？你开着车，使用 Google 地图导航，在 Yelp 上查找吃饭的地方或在 TripAdvisor 上查找休息的地方。在你的头顶上方有自动驾驶的飞机飞过，飞机里有 Wifi 和娱乐系统，可以与其他飞机、交通指挥中心和飞机厂商联系。再往上，卫星和太空站正围绕着地球绕圈，它们拍照、预报天气、转发电话信号。

软件正在吞噬着这个世界，但这只不过是个开始。在你意识到这一点之前，你可能已经穿上了智能可穿戴设备、使用电脑来锁门、使用机器人来送货或清理房间、开发自己的电子设备、有自己的制造车间、生活在虚拟现实里、乘坐自动驾驶的汽车，甚至飞向太空。

矩阵无处不在

上述的这些科技都是由软件驱动的，我们的生活被代码所包围，而代码的数量在未来只会不断增加。

不过，虽说科技无处不在，但这并不意味着你在学校里就一定要学习这些技术。比方说，我们都需要乘坐飞机，但即使是 K12 也并没有把获得飞行员驾照当作课程的一部分。

相反，学会使用那些能够帮助你理解飞行原理的工具却是课程的一部分：

- 通过学习物理和数学，你了解了重力、作用力、压力、速度、摩擦力和浮力。
- 通过学习生物，你明白了人体在缺氧和寒冷的高空中会发生什么。
- 通过学习历史，你知道了飞机是怎样被发明和制造出来的，以及飞机在旅行、商务和战争中所扮演的角色。

中学毕业之后，你就知道飞机是什么东西，知道飞机是如何飞行的，以及如何安全地搭乘飞机。一般性的课程，如物理、数学、生物和历史，它们教会你如何思考各个领域的问题，包括飞机在内。相反，有些课程只是教会你如何使用一种工具，比如如何驾驶某种型号的飞机。

类似的，我们应该专注于教授计算机科学，而不仅仅是编程：前者能够教会我们一般性的思考方式，而后者只是一种特定的工具。

什么是计算机科学？

计算机科学就是研究计算：如何表示和处理信息。

- 解决问题：你将学会各种算法策略，比如分而治之法、递归、探索法、贪婪搜索和随机算法，它们可以帮你分解和解决任何一种问题。
- 逻辑：你开始使用更准确和正式的方式进行思考，比如抽象、布

尔逻辑、数字理论和集合理论，你因此能够以一种严谨的方式来解决问题。

- 数据：你接触到信息理论，想要了解信息是什么东西，你该如何表示它们，以及如何对这个真实的世界进行建模。
- 系统：你该如何设计和实现复杂的系统来满足一系列的需求？系统工程几乎已经成为各个业务领域的核心议题。
- 思考：了解人类思维的最好途径就是尝试复制它。人工智能、机器学习、计算机视觉和自然语言处理不仅仅是计算机科学的前沿技术，它们也涉及到了生物学、物理学、心理学和数学。

上述的清单并没有提及编程或者程序，因为它们只是计算工具：它们都不是计算机科学。

在计算方面，我们更多地依赖另一个工具：我们的大脑！计算机科学的目的是教会我们的大脑进行创新性、一般性和广泛性的思考。随着科技的日益渗透，新的思考方式变得和物理、数学、生物和历史一样重要。

也就是说，只进行单独的思考是不够的：我们需要知道如何应用我们的思考。在物理学里，我们使用天平、棱镜和磁铁做实验；在生物学里，我们使用试管、植物和有盖培养皿；在计算机科学里，我们学习编程。

什么是编程？

编程，或者说写代码，是指你发出指令让计算机执行一些操作。如果你之前从来没有写过代码，那么你可能习惯了使用已有的应用程序来与计算机发生交互。实际上，这些应用程序是由代码组成的，这些代码告诉计算机如何显示应用、在哪里存储数据、从哪里获取数据，以及如何对用户的鼠标点击做出响应。

编程是基于上述的计算机科学原则进行的。计算机科学的概念——逻辑、算法、数据和系统工程——可以用于构建所有的事物，从 Web 浏览器到飞机的自动驾驶软件。编程涉及到数学和数据结构，同时也是一项具有创造性的活动：每敲出一行代码，你的想法就又向现实迈进了一步。

将编程作为计算机科学的一部分带来了很多好处。

- DIY：如果你会编程，你就可以自己开发软件。你可以从简单的开始：写一个脚本来重命名照片或写一个Excel公式来计算税金。然后更进一步：搭建一个网站；为你的公司开发一个移动应用；开发一款可以与你的朋友们一起玩的游戏。
- 问题诊断：在开发了几款应用之后，就可以轻松地理解其他的应用。在战胜了对计算机的恐惧之后，你将成为技术大神。技术遍布我们的生活，知道如何操纵它们变得与知道如何使用它们一样重要。
- 职业生涯：学习计算机科学的目的并不是为了成为专业的程序员。我们每个人在学校里都学习数学、物理和化学，但并不是每个人都成为专业的数学家、物理学家或化学家。不过，如果你有这方面的热情，你会发现软件工程是一份高评价、高收入、增长快速的工作。

总结

让我们回顾一下：

- 计算机科学代表了一种新的思考方式。在一个被技术渗透的世界里，计算机科学的概念对于每一个人来说都是非常有用的。
- 编程是学习计算机科学的一种途径，但它本身并不具备一般性目的。

混淆了这两者就会让学习编程偏离正确的方向。Slate 写过“[或许不是每个人都要学习编程](#)”的文章，Atlantic 也写过“新闻学院没必要要求学生记者学会编程”的文章，而 Jeff Atwood 在他的文章“[请不要学习编程](#)”中间了一个问题，这个问题就是混淆概念的例子：

如果有一天，Michael Bloomberg 在早上醒来时发现自己变成了一个 Java 编程高手，对于这个领导着美国最大城市的大人物来说，他的日常工作是否会如虎添翼？

当然，这个问题本身就是有问题的。这要归因于人们将学习编程作为终极目标，而不是学习如何思考。即使是 Jeff Atwood 这位经验丰富且倍受尊敬的程序员尚且分不清楚其中的区别，就别指望一般人能够搞清楚问题的实质了。我们应该这么问：

如果 Bloomberg 通过学习新的解决问题策略和掌握更好的逻辑领悟技能来改进他的思考方式，那么他在日常工作中是否会变得更好？

我想答案是显而易见的。随着技术越来越广泛地渗透到我们生活的各个方面，答案会越来越明显。这就是为什么我们要专注于教授计算机科学而不只是教授如何编程的原因。

为什么 AI 工程师要懂一点架构？

作者 王咏刚



AI 时代，我们总说做科研的 AI 科学家、研究员、算法工程师离产业应用太远，这其中的一个含义是说，搞机器学习算法的人，有时候会因为缺乏架构（Infrastructure）方面的知识、能力而难以将一个好的算法落地。

我们招的算法工程师里，也有同学说，我发的顶会 paper 一级棒，或者我做 Kaggle 竞赛一级棒，拿了不少第一名的，不懂架构就不懂呗，我做出一流算法，自然有其他工程师帮我上线、运行、维护的。

鉴于此，我给创新工场暑期深度学习训练营 DeeCamp （ps：这个训练营太火了，只招生 36 名，总共有 1000 多计算机专业同学报名，同学们来自 CMU、北大、清华、交大等最好的大学）设计培训课程时，就刻意把第一节课安排为《AI 基础架构：从大数据到深度学习》，后续才给大

家讲《TensorFlow 实战》、《自然语言处理》、《机器视觉》、《无人驾驶实战》等框架和算法方向的课。

为什么我要说，AI 工程师都要懂一点架构呢？大概有四个原因吧。

原因一：算法实现 ≠ 问题解决

学生、研究员、科学家关心的大多是学术和实验性问题，但进入产业界，工程师关心的就是具体的业务问题。简单来说，AI 工程师扮演的角色是一个问题的解决者，你的最重要任务是在实际环境中、有资源限制的条件下，用最有效的方法解决问题。只给出结果特别好的算法，是远远不够的。

比如一些算法做得特别好，得过 ACM 奖项或者 Kaggle 前几名的学生到了产业界，会惊奇地发现，原来自己的动手能力还差得这么远。做深度学习的，不会装显卡驱动，不会修复 CUDA 安装错误；搞机器视觉的，没能力对网上爬来的大规模训练图片、视频做预处理或者格式转换；精通自然语言处理的，不知道该怎么把自己的语言模型集成在手机聊天 APP 里供大家试用……

当然可以说，做算法的专注做算法，其他做架构、应用的帮算法工程师做封装、发布和维护工作。但这里的问题不仅仅是分工这么简单，如果算法工程师完全不懂架构，其实，他根本上就很难在一个团队里协同工作，很难理解架构、应用层面对自己的算法所提出的需求。

原因二：问题解决 ≠ 现场问题解决

有的算法工程师疏于考虑自己的算法在实际环境中的部署和维护问题，这个是很让人头疼的一件事。面向 C 端用户的解决方案，部署的时候要考虑 serving 系统的架构，考虑自己算法所占用的资源、运行的效率、如何升级等实际问题；面向 B 端用户的解决方案要考虑的因素就更多，因为客户的现场环境，哪怕是客户的私有云环境，都会对你的解决方案有具体的接口、格式、操作系统、依赖关系等需求。

有人用 Python 3 做了算法，没法在客户的 Python 2 的环境中做测试；有人的算法只支持特定格式的数据输入，到了客户现场，还得手忙脚乱地写数据格式转换器、适配器；有人做了支持实时更新、自动迭代的机器学习模型，放到客户现场，却发现实时接收 feature 的接口与逻辑，跟客户内部的大数据流程根本不相容……

部署和维护工程师会负责这些麻烦事，但算法工程师如果完全不懂得或不考虑这些逻辑，那只会让团队内部合作越来越累。

原因三：工程师需要最快、最好、最有可扩展性地解决问题

AI 工程师的首要目的是解决问题，而不是显摆算法有多先进。很多情况下，AI 工程师起码要了解一个算法跑在实际环境中的时候，有哪些可能影响算法效率、可用性、可扩展性的因素。

比如做机器视觉的都应该了解，一个包含大量小图片（比如每个图片 4KB，一共 1000 万张图片）的数据集，用传统文件形式放在硬盘上是个怎样的麻烦事，有哪些更高效的可替代存储方案。做深度学习的有时候也必须了解 CPU 和 GPU 的连接关系，CPU/GPU 缓存和内存的调度方式，等等，否则多半会在系统性能上碰钉子。

扩展性是另一个大问题，用 AI 算法解决一个具体问题是一回事，用 AI 算法实现一个可扩展的解决方案是另一回事。要解决未来可能出现的一大类相似问题，或者把问题的边界扩展到更大的数据量、更多的应用领域，这就要求 AI 工程师具备最基本的架构知识，在设计算法时，照顾到架构方面的需求了。

原因四：架构知识，是工程师进行高效团队协作的共同语言

AI 工程师的确可以在工作时专注于算法，但不能不懂点儿架构，否则，你跟其他工程师该如何协同工作呢？

别人在 Hadoop 里搭好了 MapReduce 流程，你在其中用 AI 算法解决了一个具体步骤的数据处理问题（比如做了一次 entity 抽取），这时

其他工程师里让你在算法内部输出一个他们需要监控的 counter——不懂 MapReduce 的话，你总得先去翻查、理解什么是 counter 吧。这个例子是芝麻大点儿的小事，但小麻烦是会日积月累，慢慢成为团队协作的障碍的。往大一点儿说，系统内部到底该用 protocol buffers 还是该用 JSON 来交换数据，到底该用 RPC 还是该用 message queue 来通信，这些决定，AI 工程师真的都逆来顺受、不发表意见了？

Google 的逆天架构能力是 Google AI 科技强大的重要原因

这个不用多解释，大家都知道。几个现成的例子：

1. 在前 AI 时代，做出 MapReduce 等大神级架构的 Jeff Dean（其实严格说，应该是以 Jeff Dean 为代表的 Google 基础架构团队），也是现在 AI 时代里的大神级架构 TensorFlow 的开发者。
2. 在 Google 做无人驾驶这类前沿 AI 研发，工程师的幸福感要比其他厂的工程师高至少一个数量级。比如做无人驾驶的团队，轻易就可以用已有的大数据架构，管理超海量的 raw data，也可以很简单的在现有架构上用几千台、上万台机器快速完成一个代码更新在所有已收集的路况数据上的回归测试。离开这些基础架构的支持，Google 这几年向 AI 的全面转型哪会有这么快。

课件分享：AI 基础架构——从大数据到深度学习

下面是我给创新工场暑期深度学习训练营 DeeCamp 讲的时长两小时的内部培训课程《AI 基础架构：从大数据到深度学习》的全部课件。全部讲解内容过于细致、冗长，这里就不分享了。对每页课件，我在下面只做一个简单的文字概括。

注：以下这个课件的讲解思路主要是用 Google 的架构发展经验，对大数据到机器学习再到近年来的深度学习相关的典型系统架构，做一个原理和发展方向上的梳理。因为时间关系，这个课件和讲解比较偏重

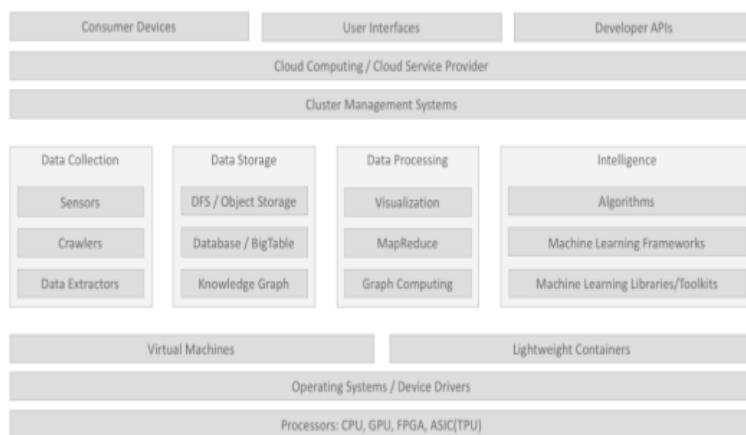
offline 的大数据和机器学习流程，对 online serving 的架构讨论较少——这当然不代表 online serving 不重要，只是必须有所取舍而已。

这个 slides 是最近三四年的时间里，逐渐更新、逐渐补充形成的。最早是英文讲的，所以后续补充的内容就都是英文的（英文水平有限，错漏难免）。

如何认识 AI 基础架构的问题，直到现在，还是一个见仁见智的领域。这里提的，主要是个人的理解和经验，不代表任何学术流派或主流观点。

下面这个图，不是说所有 AI 系统 / 应用都有这样的 full stack，而是说，当我们考虑 AI 基础架构的时候，我们应该考虑哪些因素。而且，更重要的一点，下图，是把大数据架构，和机器学习架构结合在一起讨论的。

AI infrastructures



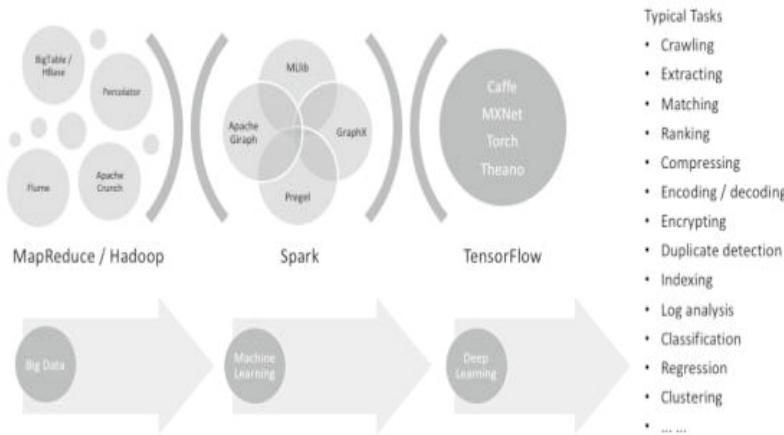
架构图的上层，比较强调云服务的架构，这个主要是因为，目前的 AI 应用有很大一部分是面向 B 端用户的，这里涉及到私有云的部署、企业云的部署等云计算相关方案。

下面这个图把机器学习和深度学习并列，这在概念上不太好，因为深度学习是机器学习的一部分，但从实践上讲，又只好这样，因为深度学习已经枝繁叶茂，不得不单提出来介绍了。

先从虚拟化讲起，这个是大数据、AI 甚至所有架构的基础（当然不

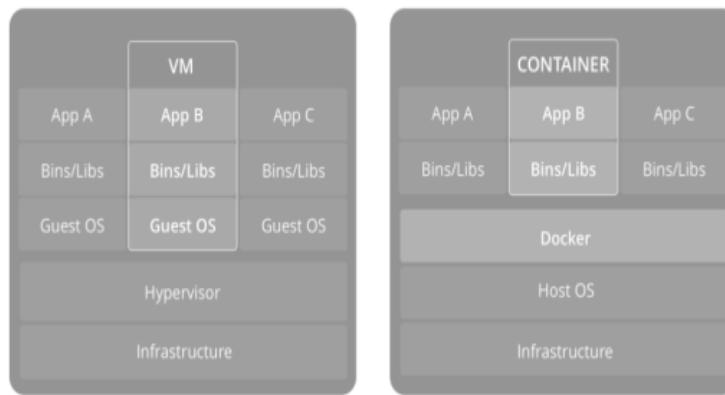
是说所有应用都需要虚拟化，而是说虚拟化目前已经太普遍了）。

From Big Data to Deep Learning



下图是 Docker 自己画的 VM vs. Container 的图。我跟 DeeCamp 学员讲这一页的时候，是先从 Linux 的 chroot 命令开始讲起的，然后才讲到轻量级的 container 和重量级的 VM，讲到应用隔离、接口隔离、系统隔离、资源隔离等概念。

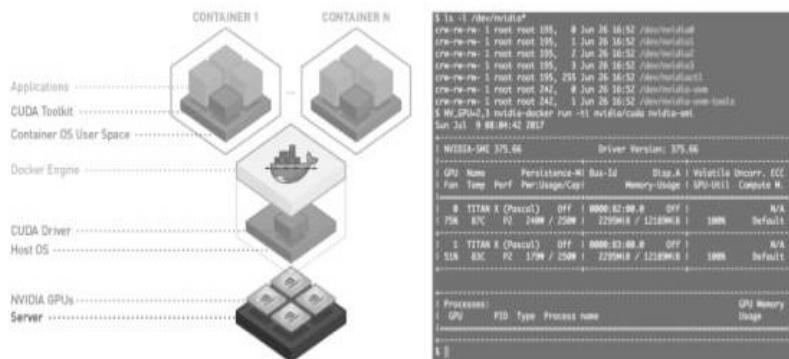
VMs vs. Containers



<https://docs.docker.com/get-started/#containers-vs-virtual-machines>

给 DeeCamp 学员展示了一下 docker（严格说是 nvidia-docker）在管理 GPU 资源上的灵活度，在搭建、运行和维护 TensorFlow 环境时为什么比裸的系统方便。

nvidia-docker: Isolation of GPU devices



<https://github.com/NVIDIA/nvidia-docker>

严格说，Kubernetes 现在的应用远没有 Docker 那么普及，但很多做机器学习、深度学习的公司，包括创业公司，都比较需要类似的 container-management system，需要自动化的集群管理、任务管理和资源调度。Kubernetes 的设计理念其实代表了 Google 在容器管理、集群管理、任务管理方面的整体思路，特别推荐这个讲背景的文章：<http://queue.acm.org/detail.cfm?id=2898444>

讲大数据架构，我基本上会从 Google 的三架马车（MapReduce、GFS、Bigtable）讲起，尽管这三架马车现在看来都是“老”技术了，但理解这三架马车背后的设计理念，是更好理解所有“现代”架构的一个基础。

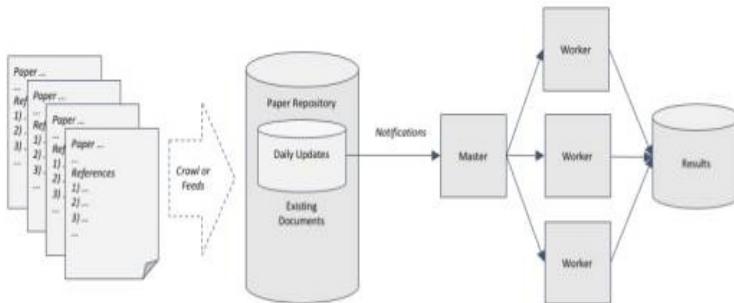
讲 MapReduce 理念特别常用的一个例子，论文引用计数（正向计数和反向计数）问题。

统计一篇论文有多少参考文献，这个超级简单的计算问题在分布式环境中带来两个思考：（1）可以在不用考虑结果一致性的情况下做简单的分布式处理；（2）可以非常快地用增量方式处理数据。

但是，当我们统计一篇文献被多少篇论文引用的时候，这个事情就不那么简单了。这主要带来了一个分布式任务中常见的数据访问一致性问题（我们说的当然不是单线程环境如何解决这个问题啦）。

Count Reference # of Each Paper?

- Incremental Processing
 - Fast (almost real-time)
 - Consistent
 - Distributable



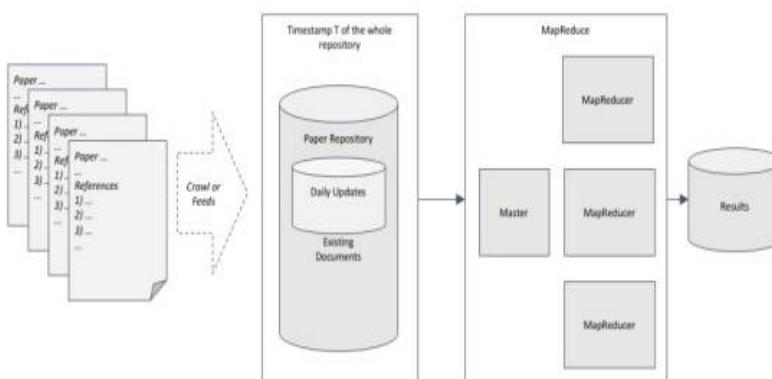
很久以前我们是用关系型数据库来解决数据访问一致性的，关系型数据库提供的 Transaction 机制在分布式环境中，可以很方便地满足 ACID (Atomicity, Consistency, Isolation, Durability) 的要求。但是，关系型数据库明显不适合解决大规模数据的分布式计算问题。

Google 的 MapReduce 解决这个问题的思路非常巧妙，是计算机架构设计历史上绝对的经典案例：MapReduce 把一个可能带来 ACID 困扰的事务计算问题，拆解成 Map 和 Reduce 两个计算阶段，每个单独的计算阶段，都特别适合做分布式处理，而且特别适合做大规模的分布式处理。

MapReduce 解决引用计数问题的基本框架。

MapReduce-Based Batch Processing

- Data processing in the whole repository for a particular timestamp



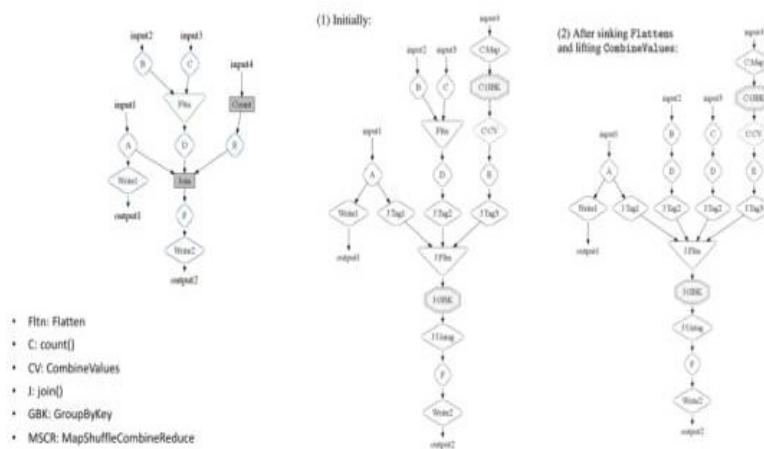
MapReduce 在完美解决分布式计算的同时，其实也带来了一个不大不小的副作用：MapReduce 最适合对数据进行批量处理，而不是那么适合对数据进行增量处理。比如早期 Google 在维护网页索引这件事上，就必须批量处理网页数据，这必然造成一次批量处理的耗时较长。Google 早期的解决方案是把网页按更新频度分成不同的库，每个库使用不同的批量处理周期。

用 MapReduce 带来的另一个问题是，常见的系统性问题，往往是由一大堆 MapReduce 操作链接而成的，这种链接关系往往形成了复杂的工作流，整个工作流的运行周期长，管理维护成本高，关键路径上的一个任务失败就有可能要求整个工作流重新启动。不过这也是 Google 内部大数据处理的典型流程、常见场景。

Flume 是简化 MapReduce 复杂流程开发、管理和维护的一个好东东。

Apache 有开源版本的 Flume 实现。Flume 把复杂的 Mapper、Reducer 等底层操作，抽象成上层的、比较纯粹的数据模型的操作。PCollection、PTable 这种抽象层，还有基于这些抽象层的相关操作，是大数据处理流程进化道路上的重要一步（在这个角度上，Flume 的思想与 TensorFlow 对于 tensor 以及 tensor 数据流的封装，有异曲同工的地方）。

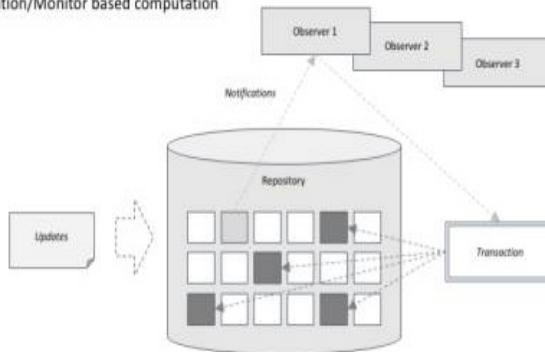
FlumeJava / Apache Crunch – Optimizes the Execution Plan



Flume 更重要的功能是可以对 MapReduce 工作流程进行运行时的优化。

Something between RDBMS and Batch Mode?

- Large-scale Incremental Processing
 - Process large-scale data incrementally, with low latency
 - Strong consistency: ACID distributed transactions
 - Notification/Monitor based computation



更多关于 Flume 运行时优化的解释图。

Flume 并没有改变 MapReduce 最适合于批处理任务的本质。那么，有没有适合大规模数据增量（甚至实时）处理的基础架构呢？

谈到大规模数据增量（甚至实时）处理，我们谈的其实是一个兼具关系型数据库的 transaction 机制，以及 MapReduce 的可扩展性的东西。这样的东西有不同的设计思路，其中一种架构设计思路叫 notification/monitor 模式。

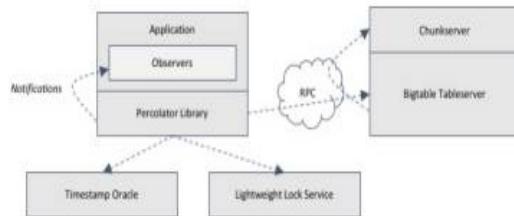
Google percolator 的论文给出了 notification/monitor 模式的一种实现方案。这个方案基于 Bigtable，实际上就是在 Bigtable 超靠谱的可扩展性的基础上，增加了一种非常巧妙实现的跨记录的 transaction 机制。

percolator 支持类似关系型数据库的 transaction，可以保证同时发生的分布式任务在数据访问和结果产出时的一致性。

percolator 实现 transaction 的方法：(1) 使用 timestamp 隔离不同时间点的操作；(2) 使用 write、lock 列实现 transaction 中的锁功能。详细的介绍可以参考 percolator 的 paper。

Percolator – Google's Solution

- Percolator: Bigtable based data processing layer with atomic, multi-row transactions, work distribution and scheduling.
 - Computations on small updates (relative to the size of repository)
 - Computations where the result can't be broken down into small updates are better handled by MapReduce
 - Computations which require strong consistency - otherwise, Bigtable is sufficient
 - Computations which are very large in some dimension - smaller computations can be handled by traditional DBMSs

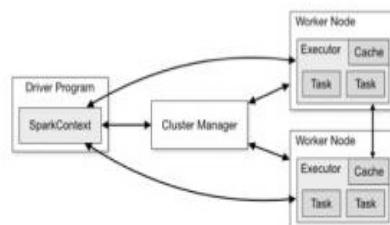


Google 的网页索引流程、Google Knowledge Graph 的创建与更新流程，都已经完成了增量化处理的改造，与以前的批处理系统相比，可以达到非常快（甚至近乎实时）的更新速度。——这个事情发生在几年前，目前 Google 还在持续对这样的大数据流程进行改造，各种新的大数据处理技术还在不停出现。

大数据流程建立了之后，很自然地就会出现机器学习的需求，需要适应机器学习的系统架构。

MapReduce 这种适合批处理流程的系统，通常并不适合于许多复杂的机器学习任务，比如用 MapReduce 来做图的算法，特别是需要多次迭代

Spark Cluster

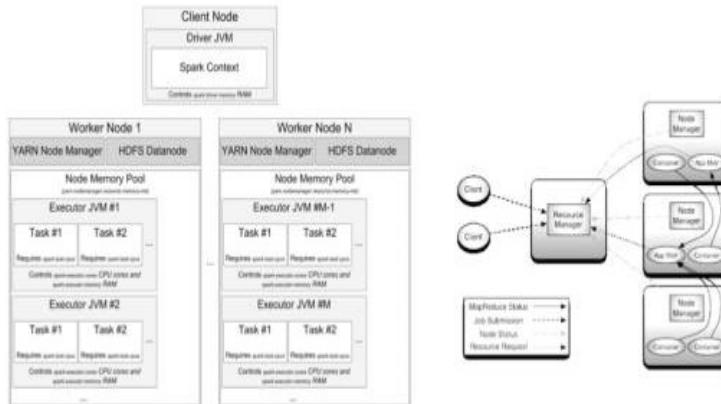


- Driver program: The process running the main() function of the application and creating the SparkContext
- Cluster manager: An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
- Worker node: Any node that can run application code in the cluster
- Executor: A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
- Task: A unit of work that will be sent to one executor

<https://spark.apache.org/docs/latest/cluster-overview.html>

的算法，就特别耗时、费力。

Spark Cluster and YARN – a detailed view



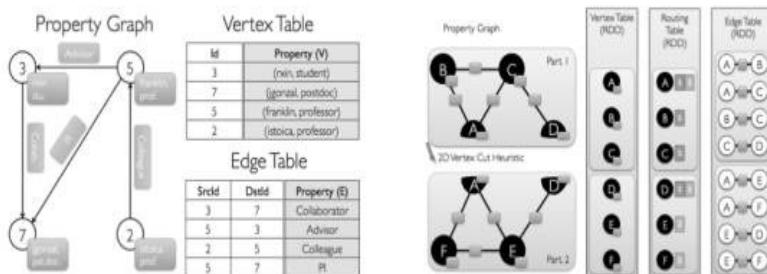
Alexey Grishchenko, <http://0x0fff.com/spark-architecture/>
<http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>

Spark 以及 Spark MLlib 给机器学习提供了更好用的支持框架。Spark 的优势在于支持对 RDD 的高效、多次访问，这几乎是给那些需要多次迭代的机器学习算法量身定做的。

Spark 的集群架构，和 YARN 的集成等。

Google Pregel 的 paper 给出了一种高效完成图计算的思路。

Spark GraphX



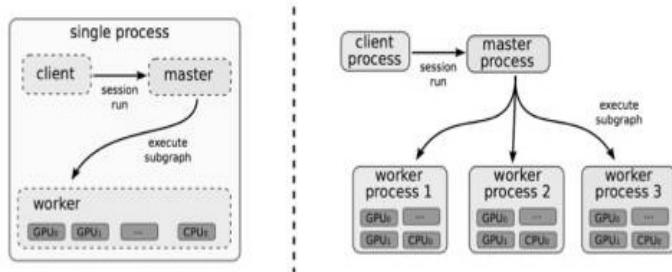
Spark GraphX 也是图计算的好用架构。

深度学习的分布式架构，其实是与大数据的分布式架构一脉相承的。——其实在 Google，Jeff Dean 和他的架构团队，在设计 TensorFlow 的架构时，就在大量使用以往在 MapReduce、Bigtable、

Flume 等的实现中积累的经验。

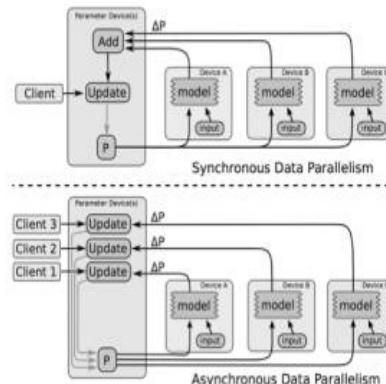
TensorFlow 经典论文中对 TensorFlow 架构的讲解。

TensorFlow: Single Machine and Distributed System



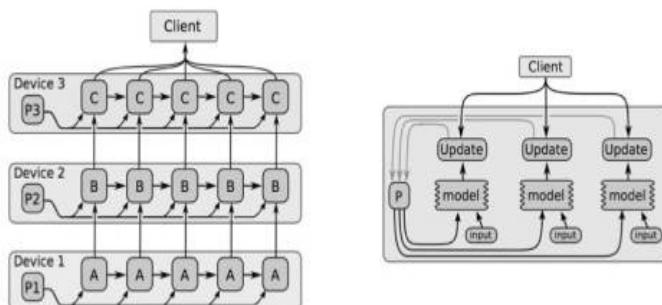
TensorFlow 中的同步训练和异步训练。

TensorFlow: Synchronous and asynchronous data parallel training



<http://download.tensorflow.org/paper/whitepaper2015.pdf>

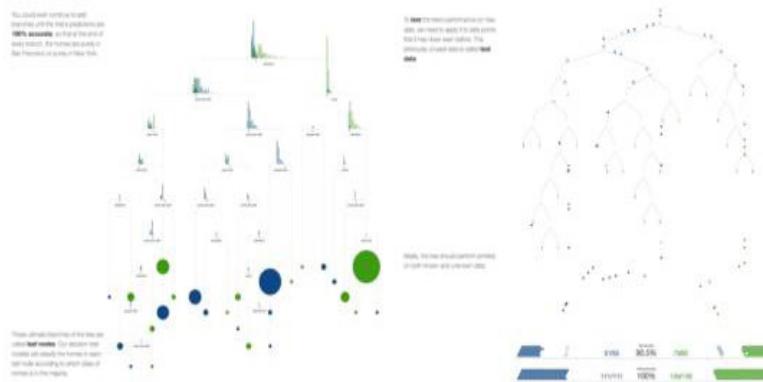
TensorFlow: Model parallel training and concurrent steps



TensorFlow 中的不同的并行策略。

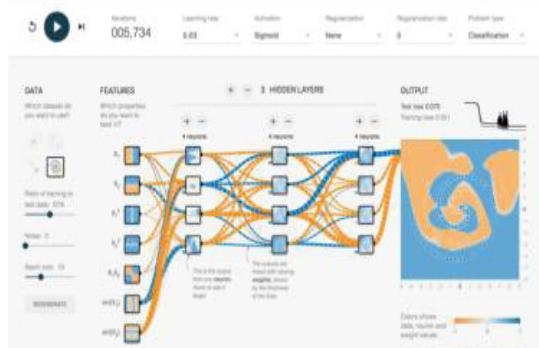
可视化是个与架构有点儿关系，但更像一个 feature 的领域。对机器学习特别是深度学习算法的可视化，未来会变得越来越重要。

Visualize Decision Tree



Visualize Deep Neural Network

- Training: 100%



Papers

- MapReduce: Simplified Data Processing on Large Clusters
 - Jeffrey Dean and Sanjay Ghemawat
 - Google
- FlumeJava: Easy, Efficient Data-Parallel Pipelines
 - Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, Nathan Weizenbaum
 - Google
- Large-scale Incremental Processing Using Distributed Transactions and Notifications
 - Daniel Peng and Frank Dabek
 - Google
- Spark: Cluster Computing with Working Sets
 - Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
 - University of California, Berkeley
- Pregel: A System for Large-Scale Graph Processing
 - Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski
 - Google
- TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems
 - Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, et al.
 - Google Research

这个对决策树算法进行可视化的网站，非常好玩。

TensorFlow 自己提供的可视化工具，也非常有意思（当然，上图应用属于玩具性质，不是真正意义上，将用户自己的模型可视化的工具）。

有关架构的几篇极其经典的 paper 在这里了。

整洁代码之道——重构

作者 南志文



写在前面

现在的软件系统开发难度主要在于其复杂度和规模，客户需求也不再像 Winston Royce 瀑布模型期望那样在系统编码前完成所有的设计满足用户软件需求。在这个信息爆炸技术日新月异的时代，需求总是在不断的变化，随之在 2001 年业界 17 位大牛聚集在美国犹他州的滑雪胜地雪鸟（Snowbird）雪场，提出了“Agile”（敏捷）软件开发价值观，并在他们的努力推动下，开始在业界流行起来。在《代码整洁之道》一书中提出：一种软件质量，可持续开发不仅在于项目架构设计，还与代码质量密切相关，代码的整洁度和质量成正比，一份整洁的代码在质量上是可靠的，为团队开发，后期维护，重构奠定了良好的基础。

接下来笔者将结合自己之前的重构实践经验，来探讨平时实际开发过程中我们注重代码优化实践细节之道，而不是站在纯空洞的理论来谈论代码整洁之道。

在具体探讨如何进行代码优化之前，我们首先需要去探讨和明确下何谓是“代码的坏味道”，何谓是“整洁优秀代码”。因为一切优化的根源都是来自于我们平时开发过程中而且是开发人员自己产生的“代码坏味道”。

代码的坏味道

“如果尿布臭了，就换掉它。”一语出 Beck 奶奶，论抚养小孩的哲学。同样，代码如果有坏味道了，那么我们就需要去重构它使其成为优秀的整洁代码。

谈论到何谓代码的坏味道，重复代码首当其冲。重复在软件系统是万恶的，我们熟悉的分离关注点，面向对象设计原则等都是为了减少重复提高重用，Don't repeat yourself (DRY)。关于 DRY 原则，我们在平时开发过程中必须要严格遵守。

其次还有其他坏味道：过长函数、过大的类、过长参数列表、冗余类、冗余函数无用函数参数、函数圈复杂度超过 10、依恋情结、Switch 过多使用、过度扩展设计、不可读或者可读性差的变量名和函数名、异曲同工类、过度耦合的消息链、令人迷惑的临时字段、过多注释等坏味道。

整洁代码

什么是整洁代码？不同的人会站在不同的角度阐述不同的说法。而我最喜欢的是 Grady Booch（《面向对象分析与设计》作者）阐述：

“整洁的代码简单直接。整洁的代码如同优美的散文。整洁的代码从不隐藏设计者的意图，充满了干净利落的抽象和直截了当的控制语句。”

整洁的代码就是一种简约（简单而不过于太简单）的设计，阅读代码的人能很清晰的明白这里在干什么，而不是隐涩难懂，整洁的代码读起来

让人感觉到就像阅读散文 – 艺术的沉淀，作者是精心在意缔造出来。

整洁代码是相对于代码坏味道的，如何将坏味道代码优化成整洁代码，正是笔者本文所探讨的重点内容：整洁代码之道一重构，接下来笔者将从几个角度重点描述如何对软件进行有效有技巧的重构。

重构 – Why

在软件开发过程中往往开发者不经意间就能产生代码的坏味道，特别是团队人员水平参差不齐每个人的经验和技术能力不同的情况下更容易产生不同阶段的代码坏味道。并且随着需求的迭代和时间推移，代码的坏味道越来越严重，甚至影响到团队的开发效率，那么遇到这个问题该如何去解决。

在软件开发 Coding 之前我们不可能事先了解所有的需求，软件设计肯定会有考虑不周到不全面的地方，而且随着项目需求的 Change，很有可能原来的代码设计结构已经不能满足当前需求。

更何况，我们很少有机会从头到尾参与并且最终完成一个项目，基本上都是接手别人的代码，即使这个项目是从头参与的，也有可能接手团队其他成员的代码。我们都曾有过这样的类似的抱怨经历，看到别人的代码时感觉就像垃圾一样特别差劲，有一种强烈的完全想重写的冲动，但一定要压制住这种冲动，你完全重写，可能比原来的好一点，但浪费时间不说，还有可能引入原来不存在的 Bug，而且，你不一定比原来设计得好，也许原来的设计考虑到了一些你没考虑到的分支或者异常情况。

我们写的代码，终有一天也会被别人接手，很可能到时别人会有和我们现在一样的冲动，所以开发者在看别人代码时候，要怀着一颗学习和敬畏之心，去发现别人的代码之美，在这个过程中挑出写的比较好的优秀代码，吸取精华，去其糟粕，在这个基础上，我们再去谈重构，那么你的重构会是一个好的开端。

总之，我们要做的是重构不是重写，要先从小范围的局部重构开始，然后逐步扩展到整个模块。

重构 — 作用

重构，绝对是软件开发写程序过程中最重要的事之一。那么什么是重构，如何解释重构。名词：对软件内部结构的一种调整，目的是在不改变软件可观察行为的前提下，提高其可理解性，降低其修改成本。动词：使用一系列重构手法，在不改变软件可观察行为的前提下，调整其结构。

重构不仅可以改善既有的设计结构，还可以帮助我们理解原来很难理解的流程。比如一个复杂的条件表达式，我们可能需要很久才能看明白这个表达式的作用，还可能看了好久终于看明白了，过了没多长时间又忘了，现在还要从头看，如果我们把这个表达式运用 Extract Method 抽象出来，并起一个易于理解的名字，如果函数名字起得好，下次当我们再看到这段代码时，不用看逻辑我们就知道这个函数是做什么的。

如果对这个函数内所有难于理解的地方我们做了适当的重构，把每个细小的逻辑抽象成一个小函数并起一个容易理解的名字，当我们看代码时就有可能像看注释一样，不用再像以前一样通过看代码的实现来猜测这段代码到底是做什么的，我一直坚持和秉持这个观点：好的代码胜过注释，毕竟注释还是有可能更新不及时的，不及时最新的注释容易更其他人带来更多的理解上的困惑。

此外重构可以使我们增加对代码和业务逻辑功能的理解，从而帮助我们找到 Bug；重构可以帮助我们提高编程速度，即重构改善了程序结构设计，并且因为重构的可扩展性使添加新功能变得更快更容易。

重构 — 时机

理解了重构的意义和作用，那么我们何时开始重构呢？笔者一直坚持这种观点：重构是一个持续的系统性的工程，它是贯穿于整个软件开发过程中，我们无需专门的挑出时间进行重构，重构应该随时随地的进行，即遵循三次法则：事不过三，三则重构。这个准则表达的意思是：第一次去实现一个功能尽管去做，但是第二次做类似的功能设计时会产生反感，但

是还是会去做，第三次还是实现类似的功能做同样的事情，那你就应该去重构。三次准则比较抽象，那么对应到我们具体的软件开发流程中，一般可以在这三个时机去进行：

1. 当添加新功能时如果不是特别容易，可以通过重构使添加特性和新功能变得更容易。在添加新功能的时候，我们就先清理这个功能所需要的代码。花一点时间，用滴水穿石的方法逐渐清理代码，随着时间的推移，我们的代码就会越来越干净，开发速度也会越来越快。
2. 修改Bug的时候去重构，比如你在查找定位Bug的过程中，发现以前自己的代码或者别人的代码因为设计缺陷比如可扩展性、健壮性比较差造成的，那么此时就是一个比较好的重构时机。可能这个时候很多同学就有疑问了，认为我开发要赶进度，没有时间去重构，或者认为我打过补丁把Bug解决不就行了，不需要去重构。根据笔者之前多年的经验得出的结论：遇到即要解决即那就是每遇到一个问题，就马上解决它，而不是选择绕过它。完善当前正在使用的代码，那些还没有遇到的问题，就先不要理它。在当前前进的道路上，清除所有障碍，以后你肯定还会再一次走这条路，下次来到这里的时候你会发现路上不再有障碍。
 - 软件开发就是这样。或许解决这个问题需要你多花一点时间。但是从长远来看，它会帮你节省下更多的时间。也就是重构是个循序渐进的过程，经过一段时间之后，你会发现之前所有的技术债务会逐步都不见了，所有的坑相继都被填平了。这种循序渐进的代码重构的好处开始显现，编程的速度明显会加快。
 - 3. Code Review时去重构，很多公司研发团队都会有定期的Code Review，这种活动的好处多多，比如有助于在开发团队中传播知识进行技术分享，有助于让较有经验的开发者把知识传递给欠缺经验的人，并帮助更多的人对软件的其他业务模块更加熟悉从而实现跨模块的迭代开发。Code Review可以让更多的人有机会对自己

提出更多优秀好的建议。同时重构可以帮助审查别人的代码，因为在重构前，你需要先阅读代码得到一定程度的理解和熟悉，从而提出一些建议和好的idea，并考虑是否可以通过重构快速实现自己的好想法，最终通过重构实践你会得到更多的成就感满足感。为了使审查代码的工作变得高效有作用，据我以前的经验，我建议一个审查者和一个原作者进行合作，审查者提出修改建议，然后两人共同判断这些修改是否能够通过重构轻松实现，如果修改成本比较低，就在Review的过程中一起着手修改。

如果是比较大型比较复杂的设计复查审核工作，建议原作者使用 UML 类序列图、时间序列图、流程图去向审查者展现设计的具体实现细节，在整个 Code Review 中，审查者可以提出自己的建议或者修改意见。在这种情景下，审查者一般由团队里面比较资深的工程师、架构师、技术专家等成员组成。

关于 Code Review 的形式，还可以采取极限编程中的“结对编程”形式。这种形式可以采取两个人位置坐在一起去审查代码，可以采取两个平台比如 IOS 和 android 的开发人员一起去审查，或者经验资深的和经验不资深的人员一起搭配去审查。

重构的这三个时机要把握好原则，即什么时候不应该重构，比如有时候既有代码实现太混乱啦，重构它还不如重新写一个来得简；此外，如果你的项目已经进入了尾期，此时也应该避免重构，这时机应该尽可能以保持软件的稳定性为主。

理解了重构是做什么，重构的作用，为什么要重构，以及重构的时机，我们对重构有了初步认识，接下来笔者重点篇幅来讲解如何使用重构技巧去优化代码质量达成 Clean Code .

重构技巧 — 函数重构

重构的源头一切从重构函数开始，掌握函数重构技巧是重构过程中很关键的一步，接下来我们来探讨下函数重构有那些实用技巧。

- 重命名函数（Rename Function Name）：Clean Code要求定义的变量和函数名可读性要强，从名字就可以知道这个变量和函数去做什么事情，所以好的可读性强的函数名称很重要，特别是有助于理解比较复杂的业务逻辑。
- 移除参数（Remove Parameter）：当函数不再需要某个参数时，要果断移除，不要为了某个未知需求预留参数，过多的参数会给使用者带来参数困扰。

将查询函数和修改函数分离：如果某个函数既返回对象值，又修改对象状态。这时候应该建立两个不同的函数，其中一个负责查询，另一个负责修改。如果查询函数只是简单的返回一个值而没有副作用，就可以无限次的调用查询函数。对于复杂的计算也可以缓存结果。

- 令函数携带参数：如果若干函数做了类似的工作，只是少数几个值不同导致行为略有不同，合并这些函数，以参数来表达不同的值。
- 以明确函数取代参数：有一个函数其中的逻辑完全取决于参数值而采取不同行为，针对该参数的每一个可能值建立一个单独的函数。

保持对象完整性：如果你需要从某个对象取若干值，作为函数的多个参数传进去，特别是需要传入较多参数比如 5 个参数或者更多参数时，这种情况建议直接将这个对象直接传入作为函数参数，这样既可以减少参数的个数，增加了对象间的信赖性，而且这样被调用者需要这个对象的其他属性时可以不用人为的再去修改函数参数。

以函数取代参数：对象调用某个函数，并将所得结果作为参数传递给另外一个函数，而那个函数本身也能够调用前一个函数，直接让那个函数调用就行，可以直接去除那个参数，从而减少参数个数。

引入参数对象：某些参数总是同时出现，新建一个对象取代这些参数，不但可以减少参数个数，而且也许还有一些参数可以迁移到新建的参数类中，增加类的参数扩展性。

- 移除设值函数（Setting Method）：如果类中的某个字段应该在对象创建时赋值，此后就不再改变，这种情景下就不需要添加 Setting method。
- 隐藏函数：如果有一个函数从来没有被其他类有用到，或者是本来被用到，但随着类动态添加接口或者需求变更，之后就使用不到了，那么需要隐藏这个函数，也就是减小作用域。

以工厂函数取代构造函数：如果你希望创建对象时候不仅仅做简单的构建动作，最显而易见的动机就是派生子类时根据类型码创建不同的子类，或者控制类的实例个数。

重构技巧一 条件表达式

分解条件表达式：如果有一个复杂的条件语句，if/else 语句的段落逻辑提取成一个函数。

- 合并条件表达式：一系列条件测试，都得到相同的测试结果，可以将这些测试表达式合并成一个，并将合并后的表达式提炼成一个独立函数，如果这些条件测试是相互独立不相关的，就不要合并。
- 合并重复的条件片段：在条件表达式的每个分支上有着相同的一段代码，把这段代码迁移到表达式之外。
- 移除控制标记：不必遵循单一出口的原则，不用通过控制标记来决定是否退出循环或者跳过函数剩下的操作，直接break或者return。

以卫语句替代嵌套条件表达式：条件表达式通常有两种表现形式，一：所有分支都属于正常行为；二：只有一种是正常行为，其他都是不常见的情况。对于一的情况，应该使用 if/else 条件表达式；对于二这种情况，如果某个条件不常见，应该单独检查条件并在该条件为真时立即从函数返回，这样的单独检查常常被称为卫语句。

以多态取代条件表达式：如果有条件表达式根据对象类型的不同选

择而选择不同的行为，将条件表达式的每个分支放进一个子类内的覆写函数中，将原始函数声明为抽象函数。

引入 Null 对象：当执行一些操作时，需要再三检查某对象是否为 NULL，可以专门新建一个 NULL 对象，让相应函数执行原来检查条件为 NULL 时要执行的动作，除 NULL 对象外，对特殊情况还可以有 Special 对象，这类对象一般是 Singleton.

引入断言：程序状态的一种假设

以 MAP 取代条件表达式：通过 HashMap 的 Key-Value 键值对优化条件表达式，条件表达式的判断条件作为 key 值，value 值存储条件表达式的返回值。

通过反射取代条件表达式：通过动态反射原理

重构技巧 — 案例

前面这多章节内容主要都是理论内容，接下来我们来看看具体的重构案例。

Map去除if条件表达式

关于该技巧的实现方法，上章节有讲述，我们直接看代码案例如下代码所示：

原始的条件表达式代码如下图 1 所示：

```
public static int getServiceCode(String str){
    int code = 0;
    if(str.equals("Age")){
        code = 1;
    }else if(str.equals("Address")){
        code = 2;
    }else if(str.equals("Name")){
        code = 3;
    }else if(str.equals("No")){
        code = 4;
    }
}
```

```
    }  
    return code;  
}
```

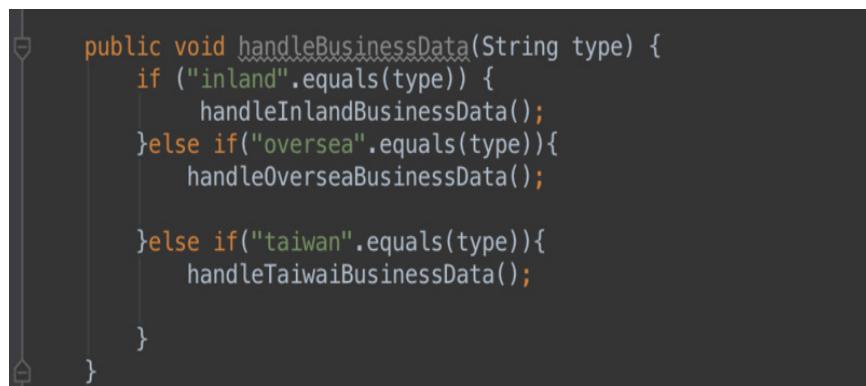
重构后的代码如下所示：

```
public static void initialMap(){  
    map.put("Age",1);  
    map.put("Address",2);  
    map.put("Name",3);  
    map.put("No",4);  
}
```

上述代码是直接通过 Map 结构，将条件表达式分解，Key 是条件变量，Value 是条件表达式返回值。取值很方便，显然高效率 O(1) 时间复杂度取值。这种重构技巧适合于比较简单的条件表达式场景，下面是比较复杂的没有返回值的条件表达式场景，我们去看看如何处理。

反射去除分支结构

原始的条件表达式代码如下图 1 所示：



```
public void handleBusinessData(String type) {  
    if ("inland".equals(type)) {  
        handleInlandBusinessData();  
    } else if ("oversea".equals(type)){  
        handleOverseaBusinessData();  
  
    } else if("taiwan".equals(type)){  
        handleTaiwaiBusinessData();  
    }  
}
```

图 1 条件表达式示范

如图 2 所示，通过 Map 和反射去分解条件表达式，将条件表达式分支的逻辑抽取到子类中的覆写函数中，提取了共同的抽象类，里面包含抽象接口 handleBusinessData，子类继承实现它。

多态取代条件表达式

如图 3 所示，在原始的条件表达式中，有两个条件表达式分支（分支

```

public void handleBusinessData(){
    map.put("inland", "ctrip.android.hotel.order.widget.InlandBusinessData");
    map.put("oversea", "ctrip.android.hotel.order.widget.InlandBusinessData2");
}

public void handleBusinessData(String type){
    if(map.containsKey(type)) {
        HandleData handleData = null;
        try {
            handleData = (HandleData) Class.forName(map.get(type).toString()).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        handleData.handleBusinessdata();
    }
}

```

图2 通过 Map 和反射重构示范

重构—多态（抽象类、简单工厂模式）

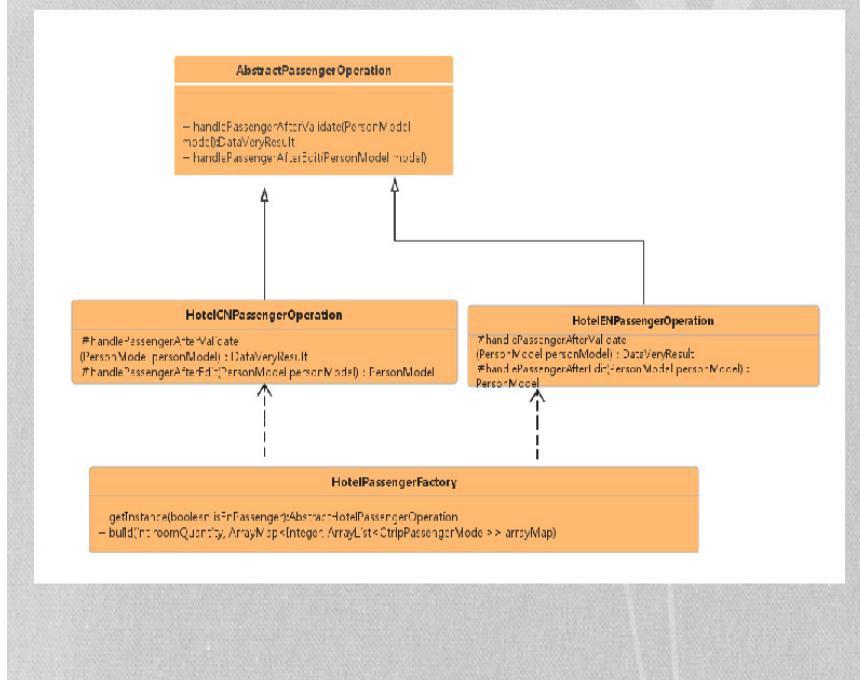


图3 重构 – 抽象类、简单工厂模式思想去实现条件表达式的分解逻辑：

- 中文入住人操作**HotelCNPassengerOperaton**类
- 英文入住人操作**HotelEnPassengerOperation**类

共同抽取了基类抽象类：**AbstractPassengerOperation**，其两个分支子类去继承抽象类。

为了分解条件表达式，笔者采取了多态的重构技巧去实现，具体有两种实现方式，第一种实现方式是采用抽象类去实现多态，代码结构图如图

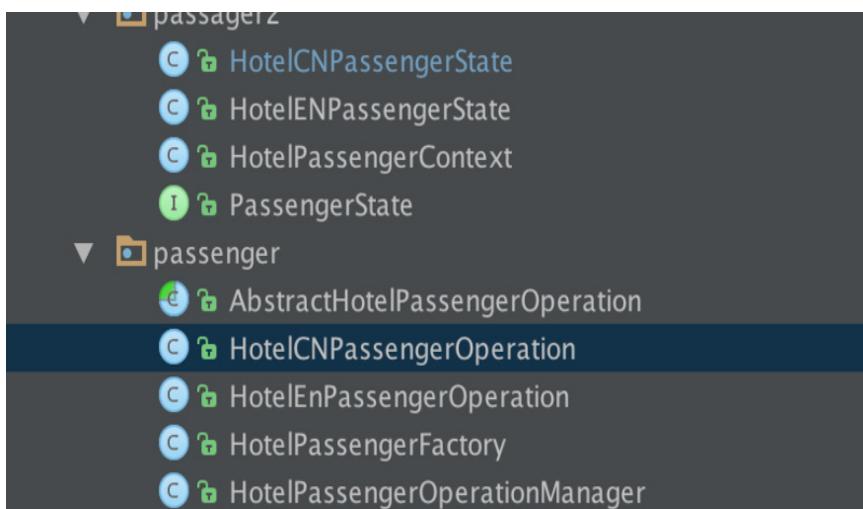


图 4 重构后的代码结构图

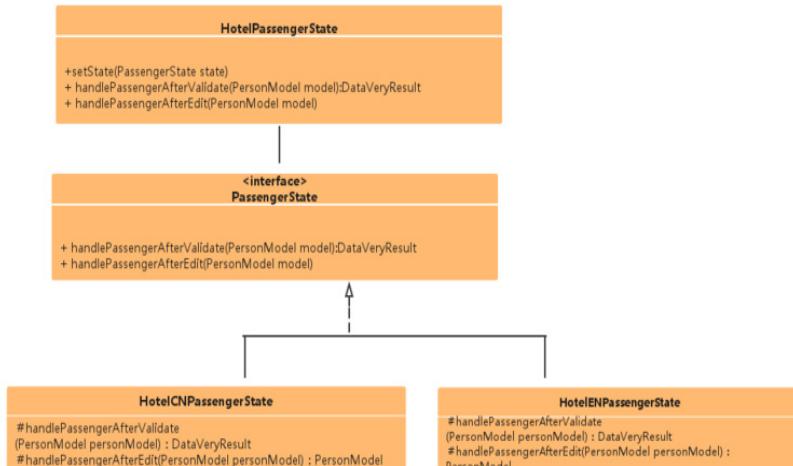


图 5 重构 - 接口状态者模式思想去实现条件表达式的分解

5 passenger 文件夹，UML 类图如上图 3 所示。第二种实现方式是采用接口去实现多态，代码结构如图 4 passenger2 文件夹，UML 类图如图 5 所示。

如上图 5 所示，在原始的条件表达式中，有两个条件表达式分支（分支逻辑），其分支逻辑分别放在了子类 HotelCNPassengerState 和 HotelENPassengerState 中，统一提取了接口类 PassengerState 类，里面包含子类都需要实现的两个基础接口。从图 6，可以看出，是使用了状态者模式。

经过了上述重构之后，我们达成了什么效果：

- 逻辑清晰

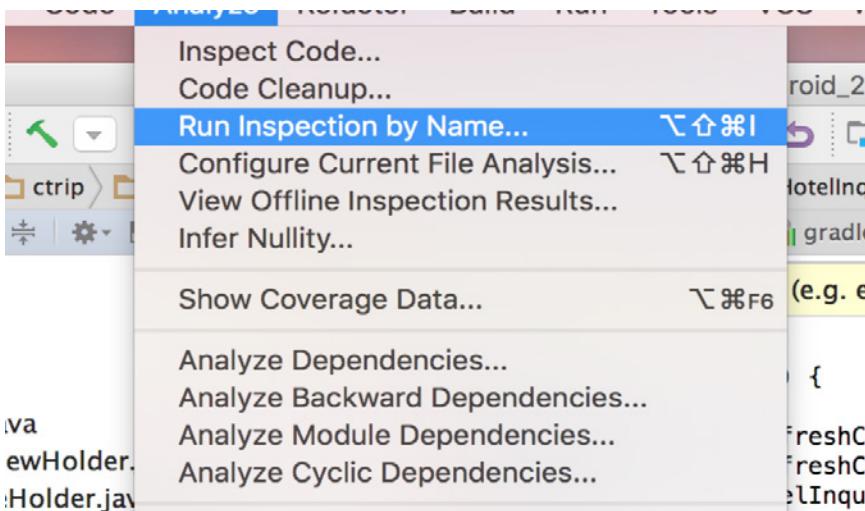


图 6 Analyze 工具示意图

- 主逻辑代码行数减少
- 业务逻辑，更好的封装解耦，无需关注具体的业务细节
- 采用了多态、抽象、状态模式、工厂模式、Build模式的等不同的思想和方法，很多不同的重构技巧去重构一个功能，值得推广和借鉴。

重构技巧 - 移动平台 Android 实战篇

前面笔者从理论和实际案例的角度对重构进行了分析，包括为什么需要重构、重构的作用、重构的时机、如何进行重构等内容，推荐提前阅读。

接下来笔者将从实践的角度去分享，即在平时开发 Android 工程中，我们如何高效去做重构，重构和开发怎么比较好的有效结合起来。

Long Method 实战

Long Method 是笔者前面提到的“代码坏味道”之一，这也是开发者一般经常容易犯的典型错误。

接下来笔者介绍在 Android 平台中如何去解决这个“bad taste”，实际上我们可以通过计算函数的圈复杂度 (cyclomatic complexity) 来判断函数是否过长，一般 cyclomatic complexity > 11，就可以认为函数过长，需要进行重构优化，那么关于函数重构的优化技巧在前面几章我

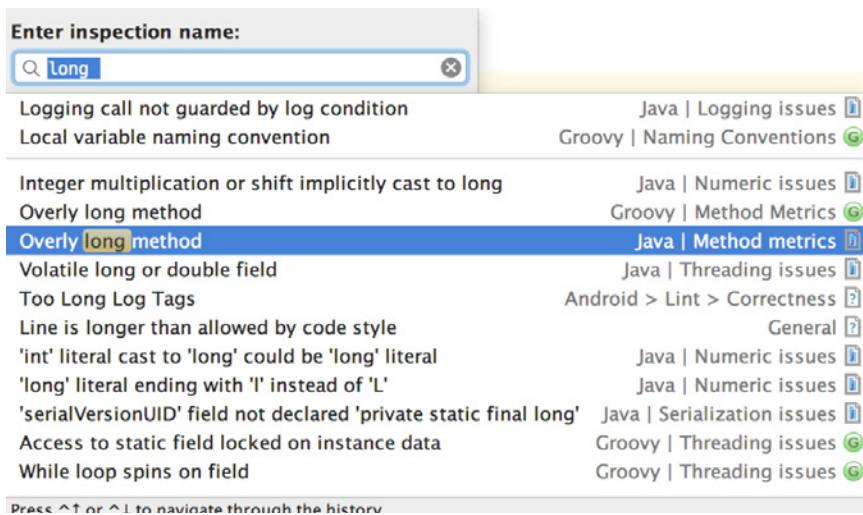


图 7 Overly long method 界面示意图

也有重点提到。

在解决圈复杂度过大这个问题，首先我们要去发现你的工程哪里存在问题，这一步我们可以通过工具或者第三方插件帮我们去解决，比如打开 Android studio 工具栏 Analyze -> Run inspection by name，如图 6 所示。

如图 7 所示，选择 Run inspection by name，打开如图 8 所示界面。

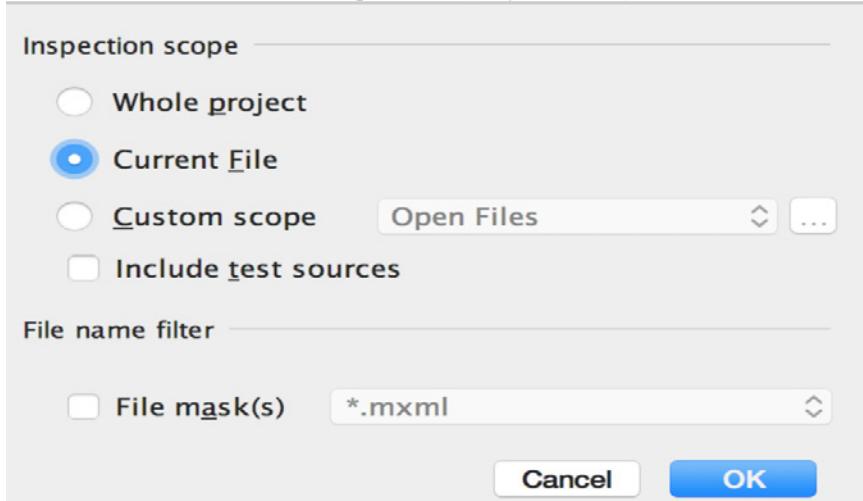


图 8 Run ‘Overly long method’

如图 8 所示，输入 long 出现 Overly long method，选择如图所示，点击会打开一个新的界面如图 9 所示。

如图 9 所示，可以选择对当前工程，当前 File，当前 Module 或者其

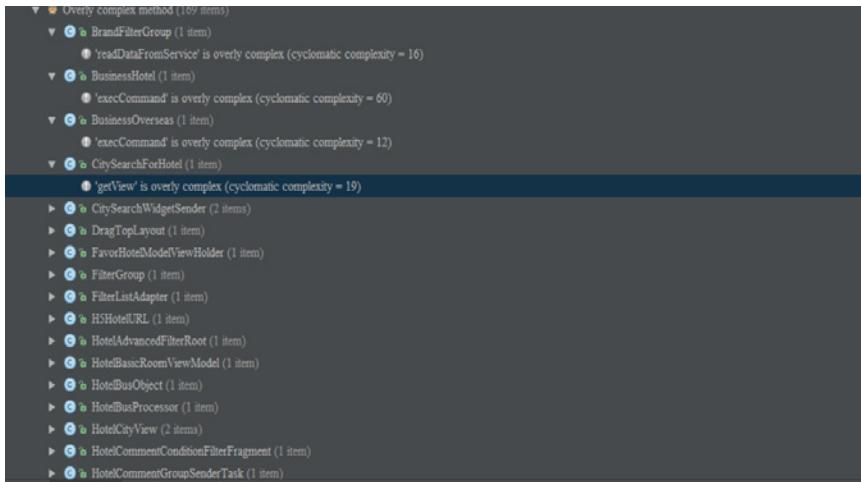


图 9 Run ‘Overly long method’ 结果

对其他 Module 进行分析，等待运行一段时间分析结果如图 9 所示。

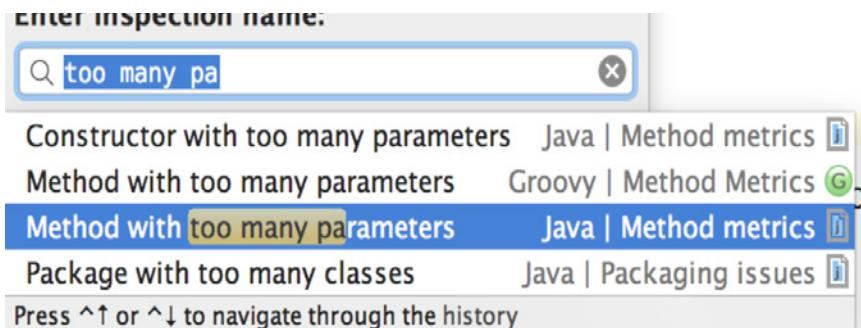


图 10 Analyze too many pa 关键字界面

得到图 10 分析结果之后，我们就可以有针对性的去进行优化重构了，知道哪些类哪些函数需要去优化，具体重构优化是一般可以将过长的函数拆分成几个不同的小函数，拆分原则：一个函数的功能要保持职责单一，查询和修改职责分开；所以可以通过不同类型的功能业务逻辑处理或者查询、修改功能去拆分大函数。

Too many parameters 实战

函数参数过多，也是典型的“代码坏味道”之一，同理打开如图 6 所示的界面，然后输入 too many pa 关键字打开如图 7 所示的界面。

选择图中所示的“Method with too many parameters”，会出现如图 10 所示的界面，然后选择“Whole Project”，运行之后，分析得到的

结果如下图 11 所示。

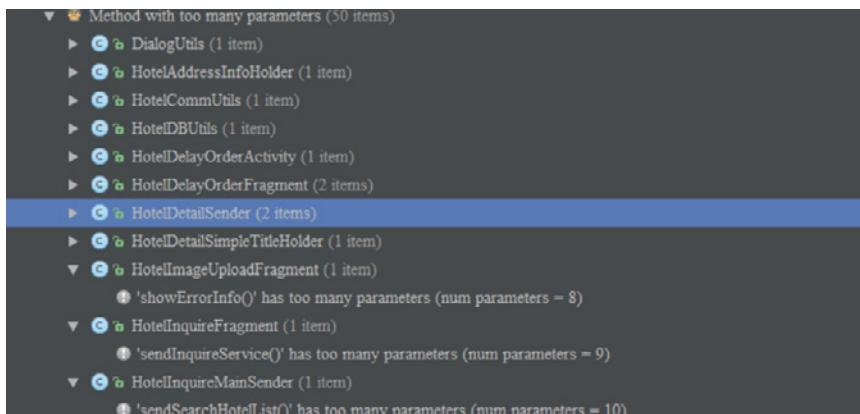


图 11 “Too many parameters” 结果分析图

Redundant local variable 实战

冗余的局部变量，同样是造成代码坏味道的源头，输入“Redudant关键字”，同理执行得出分析结果如图 12 所示，然后我们根据分析后的结果有针对性的去重构优化：

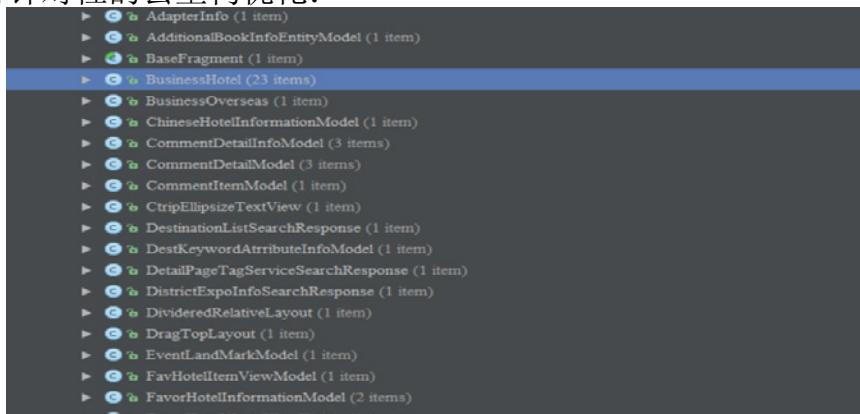


图 12 冗余局部变量分析结果

Unused Declaration - 无用函数实战

无用函数是“代码坏味道”来源之一，很多函数因为历史遗留的原因，需求已经下线了但是代码还在遗留在工程里面，或者因为重构，历史遗留代码没有完全删除或者想暂时留着下个版本使用，这些都是不好的习惯，不用的代码应该立即删除，而不应该保留在工程项目中。

同理打开如上图 6 所示的界面，然后输入 Unused declaration 关键字打开如下图 13 所示的界面：

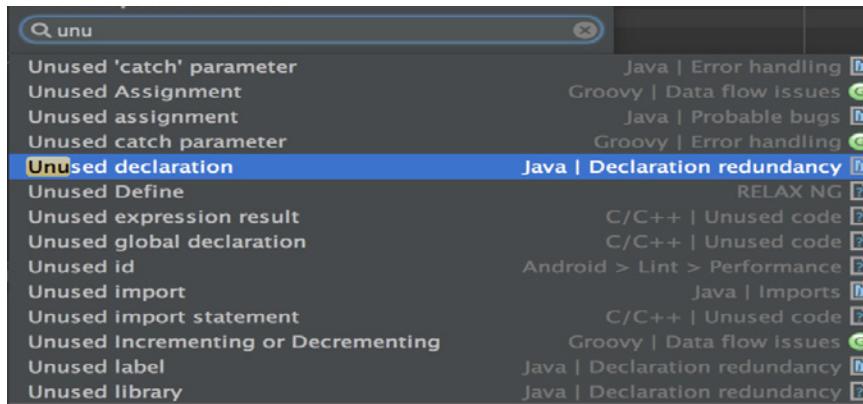


图 13 Unused declaration

分析结果能得出你整个工程中没有被使用的函数，我们都可以删除掉。

无用函数参数-实战

同理，输入关键字 Unused method parameter，如下图 14 所示，执行可以分析出工程中有哪些函数存在无用参数，可以针对性的进行优化。

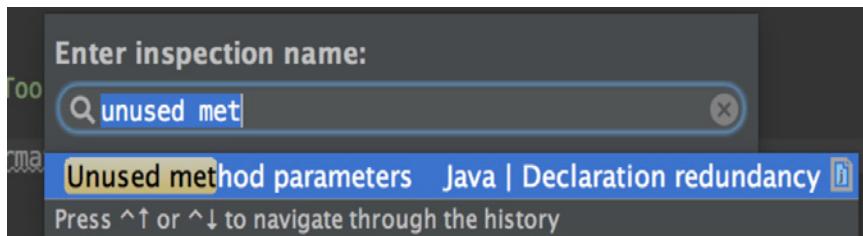


图 14 unused method parameter

infer 实战

Infer 是 Facebook 开源的静态代码检查工具，可检查 Android 和 Java 代码中的 NullPointerException 和 资源泄露。除了以上，Infer 还可发现 iOS 和 C 代码中的内存泄露，内存泄露，内存泄露。

Android studio 已经将 infer 集成到工具栏里面，点击 Analyze->infer Nullity，执行分析得出的界面类似如图 15 所示。

点击图 14 所示的分析结果具体项，可以定位到具体的代码文件，然

后我们去手动判断 或者直接点击“Infer Nullity Annotations”，工具直接帮我去完成改造结果。



图 15 infer Nullity 分析结果图

第三方插件与 Android studio 的集成

FindBugs 集成

FindBugs 是一个开源的静态代码分析工具，基于 LGPL 开源协议，无需运行工程就能对代码进行分析的工具。它不注重 style 及 format，注重检测真正的 bug 及潜在的性能问题，以 bytecode (*.class、*.jar) 为对象进行检查。除了单独运行，还可以用作 Android-studio 和 Eclipse 的 Plug-in，以及嵌入 Ant 或者 Maven 作为 task 之一进行运行。

Findbugs 自带 60 余种 Bad practice，80 余种 Correntness，1 种 Internationalization，12 种 Malicious code vulnerability，27 种 Multithreaded correntness，23 种 Performance，43 种 Dodgy。它可以检测检测 java programming 中容易陷入的 bug pattern，比如 equals() 实现时的一般规约违反 Null pointer 的参照，Method 的返回值的 check 遗漏，初始化前 field 的访问，Multi-thread 的正确性，无条件的 wait，Code 的脆弱性，可以变更的静态 object，内部数列参照的 return 等。

Android Studio 可以通过插件的方式安装，具体是打开 Android

Studio->Preference -> 搜索 plugin 选择 Plugins Tab , 打开界面如下图 16 所示。

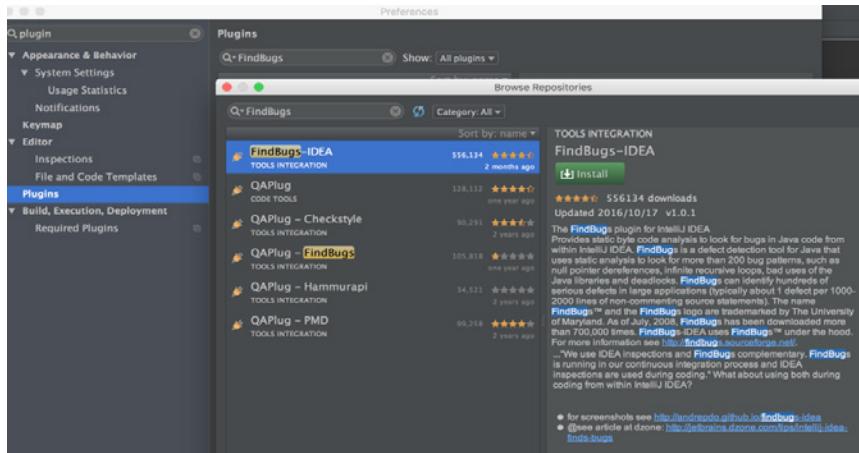


图 16 搜索 FindBugs-IDEA 界面

如图 16 中, 点击 install , downloading plugin install , 然后重启 Android studio , 会有提示界面如下图 17 所示。

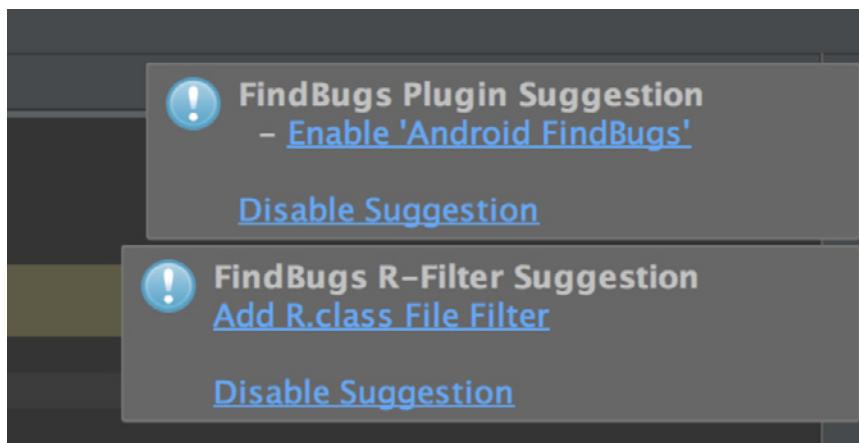


图 17 Android FindBugs Enable

点击 “Enable Android FindBugs” , 会打开界面, 在见面中添加 Plugin For Android FindBugs 即可。

然后在 Android Studio 工具栏上, 打开如图 18 所示的界面。

如图 18 所示, 可以分析对前的文件, 可以分析一个 Module files , 也可以分析一个工程文件, 选择一项会得出分析结果如下图 19 所示。

根据图 19 所示的结果, 我们可以查看具体的 Bug details , 存在什

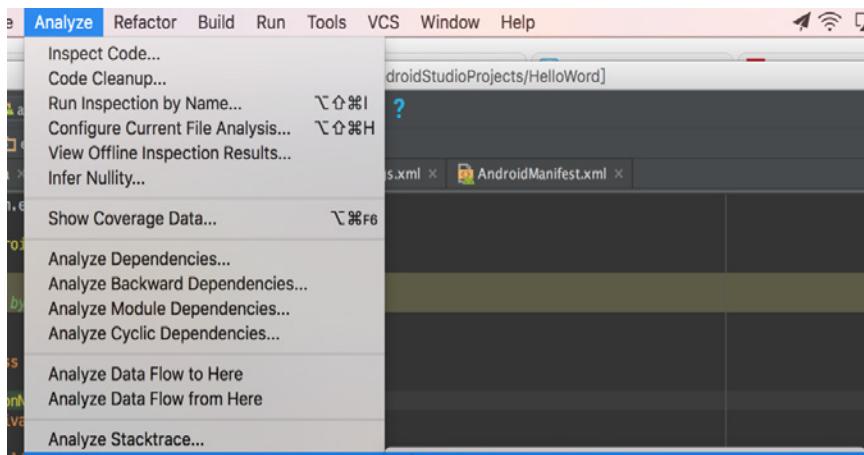


图 18 FindBugs 入口界面

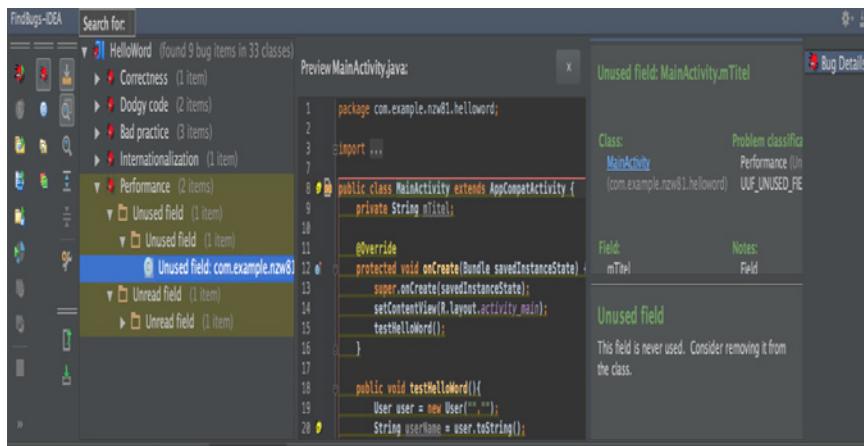


图 19 FindBugs 分析结果图

么问题，然后具体跟踪到对应的代码，根据对应的建议去修改。

MetricsReloaded 集成

MetricsReloaded 是一个计算代码复杂度即圈复杂度的 Jetbrain 开源开发的第三方插件。关于代码复杂度，有个维度的衡量，在这里需要普及下软件复杂度的相关知识：基本复杂度（Essential Complexity ($ev(G)$)）、模块设计复杂度（Module Design Complexity ($iv(G)$)）、Cyclomatic Complexity ($v(G)$) 圈复杂度。

$ev(G)$ 基本复杂度是用来衡量程序非结构化程度的，非结构成分降低了程序的质量，增加了代码的维护难度，使程序难于理解。因此，基本复杂度高意味着非结构化程度高，难以模块化和维护。实际上，消除了一个

错误有时会引起其他的错误。

Iv(G) 模块设计复杂度是用来衡量模块判定结构，即模块和其他模块的调用关系。软件模块设计复杂度高意味模块耦合度高，这将导致模块难于隔离、维护和复用。模块设计复杂度是从模块流程图中移去那些不包含调用子模块的判定和循环结构后得出的圈复杂度，因此模块设计复杂度不能大于圈复杂度，通常是远小于圈复杂度。

v(G) 是用来衡量一个模块判定结构的复杂程度，数量上表现为独立路径的条数，即合理的预防错误所需测试的最少路径条数，圈复杂度大说明程序代码可能质量低且难于测试和维护，经验表明，程序的可能错误和高的圈复杂度有着很大关系。

同理，如上图 15 所示一样去安装 MetricsReloaded 插件，安装成功后执行 Analyze->Calculate Metrics，打开如下图 20 所示的界面。

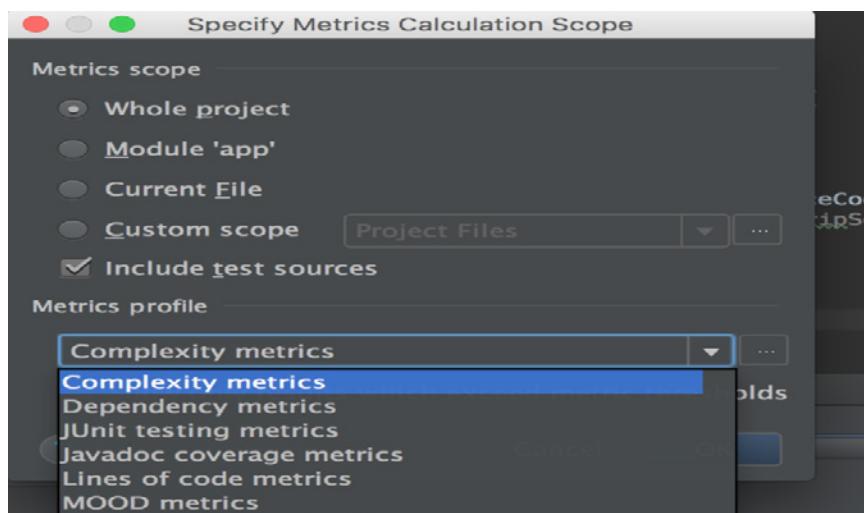


图 20 Calculate Metrics 界面

如图 20 所示，选择 Complexity metrics，执行分析结果如图 21 所示。图 21 所示界面中的红颜色部分，代表需要去重构优化的，点击当前行，会定位到源代码，然后我们针对性去优化函数。上图中，可以分析出方法的圈复杂度、类的圈复杂度、包的圈复杂度、模块的圈复杂度、工程的圈复杂度。

Sonar 集成

Metrics Complexity metrics for Project 'HelloWord' from 星期日, 25 十二月 2016 19:34:04 CST					
	Method metrics	Class metrics	Package metrics	Module metrics	Project metrics
method		ev(G)	iv(G)	v(G)	
com.example.nzw81.helloworld.HotelJsonUtils.conve		2	1	2	
com.example.nzw81.helloworld.HotelJsonUtils.conV		13	23	34	
com.example.nzw81.helloworld.HotelJsonUtils.deSe		1	2	2	
com.example.nzw81.helloworld.HotelJsonUtils.dum		1	2	2	
com.example.nzw81.helloworld.HotelJsonUtils.getEr		4	3	5	
com.example.nzw81.helloworld.HotelJsonUtils.getLi		3	2	4	
com.example.nzw81.helloworld.HotelJsonUtils.toJo		4	5	10	
com.example.nzw81.helloworld.HotelJsonUtils.write		2	9	10	
com.example.nzw81.helloworld.MainActivity.onCrea		1	1	1	

图 21 Calculate Metrics 分析结果图

对于 Android (Java) 工程，Sonar 官方提供了 Java Plugin 和 Java-specific Plugins，这些插件可以实现大部分 Findbugs、PMD、Checkstyle、Android Lint 等的检查规则。主要可以从以下几个方面检测代码质量：

1. 复杂度：项目中方法、类、文件的复杂度分布情况；
2. 重复：展示代码中重复严重的地方；
3. 单元测试覆盖率：统计并展示单元测试覆盖率（主要用于java工程）；
4. 代码标准：通过PMD、CheckStyle等代码规则检测工具规范代码编写；
5. 代码注释：没有注释或者过多的注释都不是一个良好的编程习惯；
6. 潜在的bug：通过PMD、Findbugs等代码检测工具检测出潜在的 bug；
7. 架构设计：可以检测耦合、依赖关系、架构规则、管理第三方的 jar包等。

集成 Sonar 之后，我们需要着手解决的就是代码重复率问题，这也是“代码坏味道”最典型的问题，开发者最容易犯这个问题，特别是不少开

发者喜欢偷懒，容易拷贝来拷贝去，造成工程代码的重复率比较高。一次构建运行之后，我们可以得出分析结果，类似如图 22 所示。



图 22 sonar 构建运行结果

点击重复率，我们可以看出哪些文件之间的代码是重复的，然后针对性使用抽取工具类、合并类、合并分解函数等技术重构手段去优化。

SonarLint集成

前面我们所讲到的 Sonar 之前的提供的本地工具是需要依赖 SonarQube 服务器的，这样导致其运行速度缓慢。 新出的 SonarLint 的扫描引擎直接安装在本地，速度超快，实时探测代码技术债务，给程序员最快速的反馈，排除代码异味的绝佳利器，帮助程序员获得 Clean code。 新版 SonarLint 也能链接 SonarQube 服务器，但这并不必要。本地安装 SonarLint 来做代码本地扫描，本地发现本地修改，而且能快速看到修改结果，快速处理代码臭味，有效控制技术债务。

按照如上图 17 所示一样去安装 SonarLint 插件，安装之后重启 Android Studio，即可动态扫描出结果如图 23 所示。

重构技巧实战 - 小结

本文我们讲述了在 Android 程序开发过程中如何结合工具去帮助我们做重构优化的各种技能包括 Android Studio 自己已经集成的插件 Code Inspection 、infer Nullity 以及 FindBugs、MetricsReloaded、

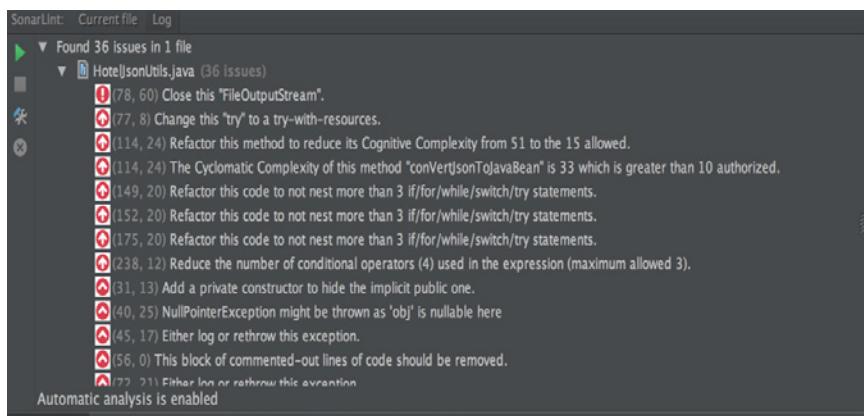


图 23 SonarLint 扫描结果图

Sonar、SonarLint 等第三方插件工具，其实还有很多类似著名的插件比如 QAPlugin、PMD、Hammurapi 、Lint4j 等工具，大家可以自行尝试使用，在这里我不一一说明。

所谓工欲善其事必先利就是这个道理，我们如果需要去做重构优化，首先我们要知道我们做的不好的地方 – 代码的坏味道在哪里，这种工作如果靠人为手动的去发现，那么效率和产出将会及其低下，所以我们需要借助于集成插件工具帮我们自动去扫描发现问题，然后再去针对性的重构优化，产出 Clean code 。

写在最后

重构是一门比较大而深的话题和课题，笔者这次主要探讨了如何通过有效的重构技巧去写成优秀的整洁代码，代码整洁之道就是要将重构始终贯穿在整个开发过程中，不断的持续的渐进重构，从而将以前的技术债全部还完。

重构是个技术活，需要很资深的人士去整体架构把控技术方案和产品质量，才能使重构做的更加有效并且不会引入新的问题，但是无论我们最终采取什么手段去重构，最终我们都需要尽量符合 Solid 设计相关原则。

- **类的单一职责：**体现了类只应该做一件事，良好的软件设计中系统是由一组大量的短小的类组成，以及需要他们之间功能协作完成，而不是几个上帝类。如果类的职责超过一个，这些职责之间

就会产生耦合。改变一个职责，可能会影响和妨碍类为其他人服务的功能。这种类型的耦合将会导致脆弱的设计，在修改的时候可能会引入不少未知的问题。

- **开闭原则：**其定义是说一个软件实体如类，模块和函数应该对扩展开放，而对修改关闭，具体来说就是你应该通过扩展来实现变化，而不是通过修改原有的代码来实现变化，该原则是面相对象设计最基本的原则。其指导思想就是，（1）抽象出相对稳定的接口，这部分应该不动或者很少改动；（2）封装变化；不过在软件开发过程中，要一开始就完全按照开闭原则来可能比较困难，更多的情况是在不断的迭代重构过程中去改进，在可预见的变化范围内去做设计。
- **里氏替换原则：**子类可以扩展父类的功能，但不能改变父类原有的功能。简单来说，所有使用基类代码的地方，如果换成子类对象的时候还能够正常运行，则满足这个原则，否则就是继承关系有问题，应该废除两者的继承关系，这个原则可以用来判断我们的对象继承关系是否合理。通常在设计的时候，我们都会优先采用组合而不是继承，因为继承虽然减少了代码，提高了代码的重用性，但是父类跟子类会有很强的耦合性，破坏了封装。
- **接口隔离原则：**不能强迫用户去依赖那些他们不使用的接口。简单来说就是客户端需要什么接口，就提供给它什么样的接口，其它多余的接口就不要提供，不要让接口变得臃肿，否则当对象一个没有使用的方法被改变了，这个对象也将会受到影响。接口的设计应该遵循最小接口原则，其实这也是高内聚的一种表现，换句话说，使用多个功能单一、高内聚的接口总比使用一个庞大的接口要好。
- **依赖倒置（DIP）：**高层模块不应该依赖低层模块，两者都应该是依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。其实这就是我们经常说的“针对接口编程”，这里的接口就是抽象，我们

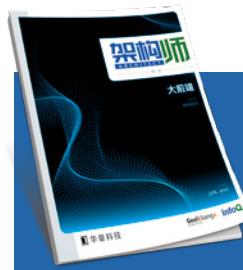
应该依赖接口，而不是依赖具体的实现来编程。DIP描述组件之间高层组件不应该依赖于底层组件。依赖倒置是指实现和接口倒置，采用自顶向下的方式关注所需的底层组件接口，而不是其实现。DI模式很好例子的就是应用IOC（控制反转）框架，构造方式分为分构造注入，函数注入，属性注入。

当我们在做重构优化的时候应该充分考虑上面这几个原则，一开始可能设计并不完美，不过可以在重构的过程中不断完善。但其实很多人都跳过了设计这个环节，拿到一个模块直接动手编写代码，更不用说去思考设计了，项目中也有很多这样的例子。当然对于简单的模块或许不用什么设计，不过假如模块相对复杂的话，能够在动手写代码之前好好设计思考一下，养成这个习惯，肯定会对编写出可读性、稳定性、健壮性、灵活性、可服用性、可扩展性较高的代码有帮助。



架构师 月刊 2017年7月

本期主要内容：Oracle 的 Java 模块化系统保卫战；Apple 发布 Core ML，为 Apple 设备提供了机器学习功能；WebAssembly，火狐赢了？Python 向来以慢著称，为啥 Instagram 却唯独钟爱它？高负载微服务系统的诞生过程



架构师特刊 大前端

本期主要内容：当我们在谈大前端的时候，我们谈的是什么；如何落地和管理一个“大前端”团队？



顶尖技术团队访谈录 第九季

本次的《中国顶尖技术团队访谈录》第八季挑选的六个团队虽然都来自互联网企业，却是风格各异。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱与坚持。



架构师特刊 用户画像实践

本电子书中几个作者介绍一个公司如何从无到有的搭建用户画像系统，以及其中的技术难点与实际操作中的注意事项，实为用户画像的实操精华之选，推荐各位收藏阅读。