阿里云开发者社区
ALIBABA CLOUD DEVELOPER COMMUNITY

Java开发者面试百宝书

集结阿里Java大神一手面试经验诚意出品

- 学习Java还有前景吗？Java趋势报告为你揭秘
- 敲黑板！Java面试常见问题标准答案来了
- 面试不知如何准备？来看看这些一手面经

藏经阁

阿里云开发者电子书系列

《Java 超神季》

等你来战

阿里云开发者“藏经阁”

海量电子书免费下载

目录

看清趋势，找准目标

Java 正青春：现状与技术趋势报告

作者：好好学习

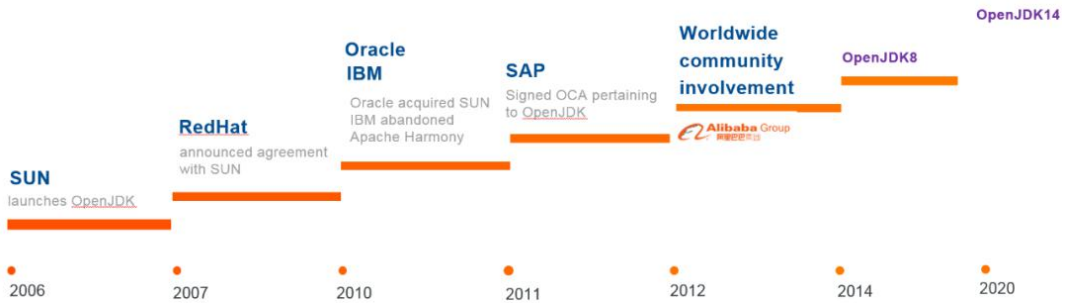
简介：本文将从 JavaSE 开源现状、OpenJDK 版本生态到 OpenJDK 技术趋势三个方面讲述当前基础 Java 技术的现状，进一步讨论在云原生、AI、多语言生态领域支撑 Java 应用的基石——Java Virtual Machine（JVM）技术，面向未来的演进趋势。

一、背景

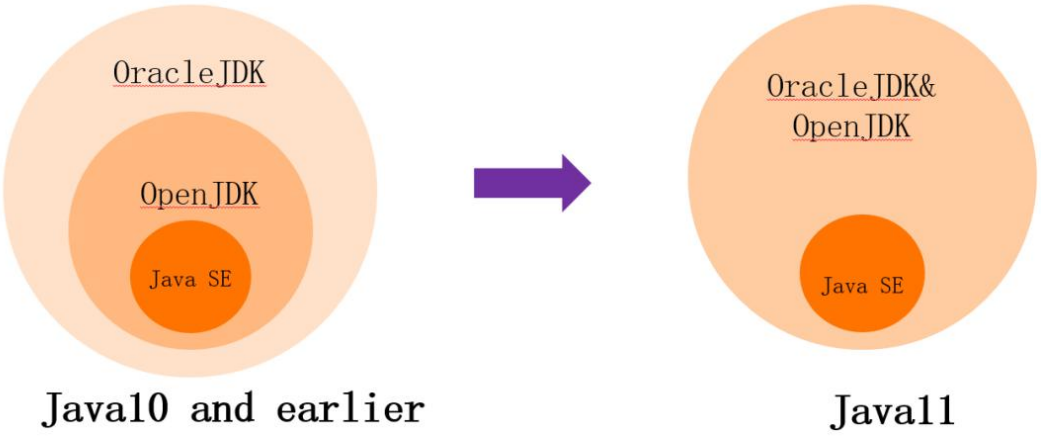
1991 年，James Gosling 带领团队开始了一个叫"Oak"的项目，这个就是 Java 的前身。1995 年，Java1.0 发布。“Write once, run anywhere"这句 Java 口号想必大家耳熟能详。Java 刚开始出现的时候主要面向 Interactive Television 领域，直至后来几年的发展，当时的 SUN（后来在 2010 年被 Oracle 收购）一度想用 Java 来打造桌面的网络操作系统，取代当时如日中天的 Windows。不过 Java 后来的发展，不曾想虽未在桌面领域内取得多大的建树，出乎意料地，却在企业级应用领域开花结果，占据了如今几乎统治的地位。失之東隅，却收之桑榆。

二、JavaSE 开源现状

Sun 在 2006 年的 Java One 大会上，宣布 Java 技术开源，随后 2006 年底在 GPL 协议下发布 HotSpot 以及 javac，这是 Java 发展中的里程碑事件。阿里巴巴最早在 2012 签署 OCA，并参与到了 OpenJDK 的开发。



OpenJDK 是 JavaSE 开源的 Reference Implementation。在 JavaOne 2017 的 Keynote 上 (2018 年 JavaOne 被 Oracle 重命名为 CodeOne)，Oracle 承诺将开源所有的 OracleJDK 里包含的商业实现功能[1]。



在 2018 年发布的 Java11， Oracle 已经让 OpenJDK 和 Oracle JDK 两者的二进制文件在功能上尽可能相互接近，尽管 OpenJDK 与 OracleJDK 两者在一些选项之间仍然存在一些差异[2]。

另外，除了 OpenJDK 这条主线，在最近的几年里，Java 基础技术的开源有愈演愈烈趋势：2017 年，IBM 将内部使用 20 多年之久的 J9 虚拟机开源，并贡献到 Eclipse Foundation， 而随后 2018 年，Oracle 开源 GraalVM 1.0，其核心包含用 Java 写的 Just-in-Time compiler/Graal, SubstrateVM 以及支持多语言解释器的 Truffle 框架。各个企业开源的主要动机，想通过开源构建并受益于一个更为强大的语言生态系统。

云 + 开源结合在一起，使得普通开发者以较低的门槛获得一流工具(链)的使用和体验，任何一家企业都可以像任何大型组织一样，使用的相同技术(democratizing)，这是开发者的黄金时代。

三、Java is Still Free: 你该选择什么样的 JDK?

Java 仍然免费，但随着 OracleJDK License 变化开始转向收费，OpenJDK 会逐渐取代 OracleJDK 成为市场主流，这点也可以从 JVM 2020 生态报告中看出趋势：OracleJDK 从前一年的 70% 的开发者选择使用率降到 2020 年的 34%。

OracleJDK 收费，在客观上也加剧了 OpenJDK 生态的碎片化趋势，出现了包括 Alibaba Dragonwell 在内的多个基于 OpenJDK 的可选实现。

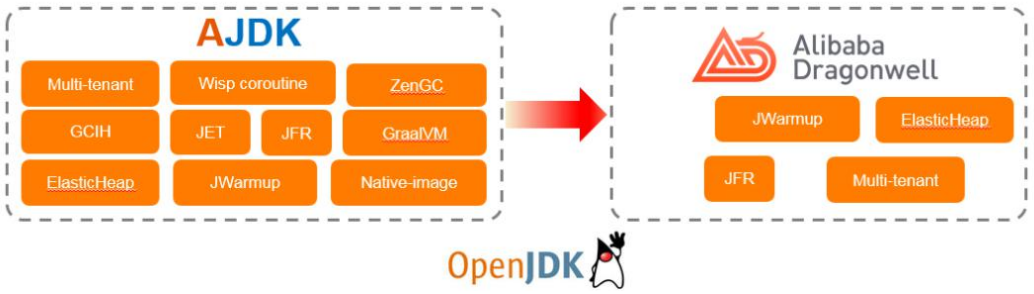
企业在选择使用那个 Java Vendor 的 JDK 版本时，几个方面的考虑因素可以参考：

- 安全与稳定：是否会及时同步上游的最新更新，包括安全补丁，关键的问题修复等。
- JavaSE 标准兼容 ：是否与标准 Java 兼容。
- 性能与效率：是否可以在问题诊断，性能调优方面提供有效的工具支持，帮助一线的开发同学高效地解决 Java 问题。在 JVM，到 JDK (Class library) 层面，是否有面向企业业务场景的优化特性，可以帮助提升资源的利用率，生产系统的稳定性等等。
- 快速的新技术采纳：伴随收费，Oracle 管理 Java 版本生命周期采用了 Long Term Support(LTS) 的概念，Oracle 每三年会指定一个 LTS 的 Java 版本，Java 8/11 都是 LTS 版本。大部分企业，尤其是大中型企业很难跟上 Java 每六个月一发布的节奏，像 Java 12，13 这样的 Feature Release(FR) 版本。那么问题来了，如果你选择 Stay 在 LTS 版本上，比如 Java 11，在新版本 (Java11+) 发布的 JVM/JDK 技术，是否可以在不升级的情况下，提前享受这些技术红利?

这里分享下 Alibaba Dragonwell 在这些方面的计划与思考。

Alibaba Dragonwell 是阿里巴巴内部广泛使用的 AJDK (AlibabaJDK) 的开源版本，Alibaba Dragonwell 作为基石，支撑了阿里经济体内几乎所有的 Java 业务，经过了双 11 等大促的考验。Alibaba Dragonwell 主要针对的场景是数据中心大规模 Java 应用部署情况下，Java 应用稳定性、效率以及性能的优化与提高。

2019 年 3 月阿里开源 Alibaba Dragonwll 8.0.0，我们也一直正在践行开源时候的承诺，AJDK 内部使用的特性在逐步开源。到刚刚发布 Alibaba Dragonwell 8.3.3，我们已经开源了 JWarmup，ElasticHeap，多租户，JFR 等众多功能，协程 Wisp 2.0，GCIH 等也在开源的规划上。



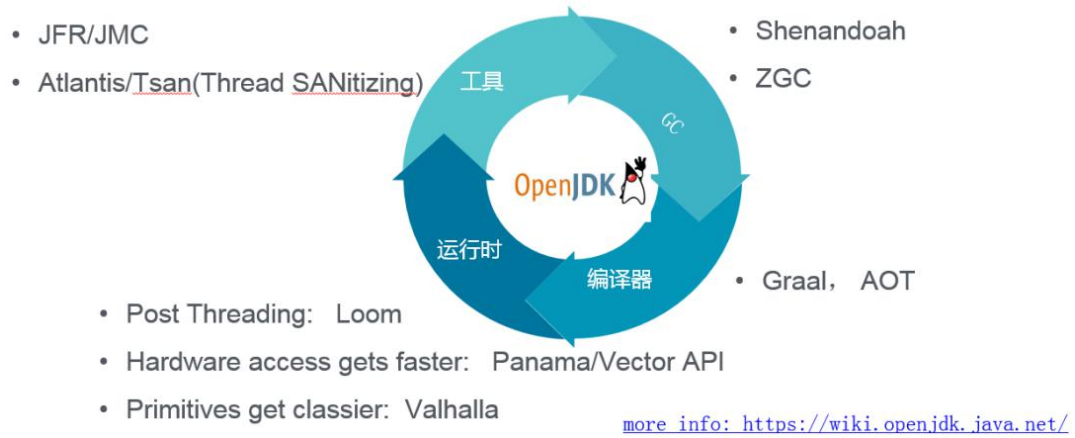
同时，Alibaba Dragonwell 作为 OpenJDK 的下游，每个发行版都会同步上游最新更新，包括安全更新，问题修复等，并经过阿里内部大规模的应用集群测试。

在新技术 Adoption 方面，Alibaba Dragonwell 目前发布和维护了 Java 8,11 两个 LTS 版本，阿里 JVM 团队会根据实际业务状况，移植 Java11+ 的相关功能到 Java 8 和 11 两个版本，这样 Alibaba Dragonwell 用户可以在不跟进 Java 12,13 等这些 FR 版本的情况下，提前享受这些功能带来的技术红利。

四、OpenJDK 技术趋势

纵观 Java 技术 20 多年的发展，始终围绕着两大主题：Productivity 以及 Performance。在很多情况下，Java 在设计上 Productivity 是优于 Performance 考虑的。Java 引入的 Garbage Collector 把程序员从复杂的内存管理中解脱出来，但在另一方面 Java 应用始终困扰于 GC 暂停时间的影响。Java 基于栈式虚拟机的中间字节码设计，很好地抽象了不同平台 (Intel, ARM 等) 的差异性，同时通过 Just-in-Time (JIT) 编译技术，解决的 Java 应用 peak 性能，但在另一方面 JIT 不可避免引入了 Warmup 的代价，正常情况下 Java 程序永远需要先 load class，解释执行，然后再到高度优化的代码执行。

如果从 JVM 视角来总结梳理下目前 OpenJDK 社区正在发生，孵化的相关技术，主要从工具，GC，编译器，以及 Runtime 四个方面进行一个主要概括：



（一）JFR/JMC

Oracle 从 Java 11 开源了其之前一直作为商业功能的 JFR，JFR 是功能强大的 Java 应用问题诊断与性能剖析工具。阿里巴巴也是作为主要的贡献者，与社区包括 RedHat 等，一起将 JFR 移植到了 OpenJDK 8， 预计 2020 年 7 月即将发布的 OpenJDK 8u262 (Java8) 将会默认带有 JFR 功能，这样 Java 8 的用户可以基于这个版本免费使用 JFR 功能。

（二）ZGC/Shandoath

无论是 Oracle 在 Java 11 发布的 ZGC，还是 RedHat 已经做了好几年的 Shandoath，都实现了 concurrent copy GC，解决 Large Heap 情况下的 GC 停机性能。ZGC 最新状态，在 9 月份即将发布的 JDK 15，ZGC 将从 Experimental 功能变为生

产可用 [3] 。实际上，在 AJDK 11 上，阿里巴巴团队 JVM 团队已经做了大量 Java 11+ 到 Java 11 的 ZGC 移植工作，以及相关问题修复，2019 年双 11 和阿里数据库团队一起，让数据库应用运行在 ZGC 上，100+ GB Heap 情况下 GC 暂停时间可以保持在 <10ms 以内， 详细讨论参考[4]。

（三）Graal

用 Java 开发的新一代 Just-in-Time 编译技术，用来替代目前 HostSot JVM 的 C1/C2 编译器，OpenJDK 上的 Ahead-of-Time (AOT) 技术也是基于 Graal 编译器开发。

（五）Loom

OpenJDK 社区协程项目，对应于 AJDK 的 Wisp 2.0 实现，详细讨论可以参考[5]。

五、进击的 Java：面向未来演进

2020，站在一个全新的节点上，本文也从三个大的方面 Cloud Native， AI，以及多语言生态三个方面展望下未来的发展，有些讨论本身是超越 Java 本身的。

（一）面向 Cloud Native 的语言进化

云原生时代，软件的交付方式发生的根本性变化。以 Java 为例，在之前 Java 开发者交付的是应用本身，具体体现在以 "jar", "war" 的形式交付，而云原生则是以

Container 为交付单位的：

在运行方面，面向 Cloud Native 的应用要求：

- Reactive
- Always Watching
- Extreme low memory footprint
- Quick boot time

Java 语言作为企业计算，互联网领域的王者，拥有一致性，丰富的构建在 Java 语言之上的生态系统， 丰富的三方库，多样的 Serviceability 支持等，随着云时代应用微服务化，Serverless，这些新的架构逐渐触及到了 Java 程序速度提升的天花板 —— Java 自身的启动运行开销。

在 Cloud Native 这个新的上下文里，我们谈论语言的进化，绝不仅仅限于运行时，编译器层面， 新的计算形态一定伴随着编程模型的变革，这涉及围绕程序语言的 Library，Framework，Tools 等一系列配套的改革。从目前业界来看，也有不少的项目正在发生：配合 GraalVM/SVM（Java 静态编译技术）的下一代编程框架 Quarkus, Micronaut, 以及 Helidon，Quarkus 更是提出了“container first” ，他们提倡的分层的 lightweight uber-jar 的概念正是符合了 container 交付这一趋势。而 Red Hat

的 Java 团队与 OS 团队合作的"Checkpoint Restore Fast Start-up"技术 (AZul 在 JVM 技术峰会 '2019 上也提出过类似的想法) 则是在更加底层的技术栈上解决 Java 快速拉起问题。

在 Java for Cloud Native 方向，我们也开展了相关研发工作。Java 是静态语言，但是包含了大量的动态特性，包括反射，Class Loading，Bytecode Instrument (BCI) 等等，这些动态特性本质上都是违反 GraalVM/SVM 所要求的 Closed-World Assumption (CWA) 原则，这也是导致传统跑在 JVM 的 Java 应用不容易在 SVM 编译运行的主要原因。阿里巴巴 JVM 团队对 AJDK 做了静态化裁剪，务求在 Java 静/动态特性之间找到一个确定的边界，从 JDK 的层面为 Java 静态编译提供可能性。同时向上，与蚂蚁中间团队合作，定义面向静态编译的 Java 编程模型，通过编程框架来约束 - Java 应用的开发是面向静态编译友好的。我们静态编译了基于蚂蚁开源中间件 SOFAStack 构建的服务注册中心 Meta 节点应用，相较于传统 的运行在 JVM 上，性能有量级的提升：服务启动时间降低了 17 倍，可执行文件大小降低了 3.4 倍，运行时内存降低了一半。详见[6]。

（二）AI 的兴起，编程语言异构计算的新挑战

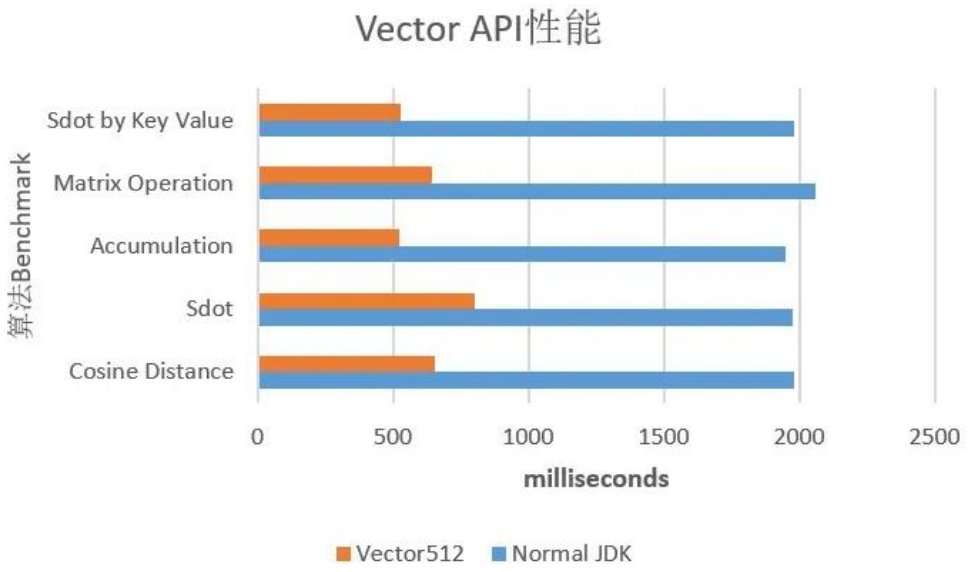
2005 年，时任 Intel CTO 的 Justin Rattner，说过 “We are at the cusp of a transition to multicore, multithreaded architectures”， 在前后的十几年中，编程语言与编译器领域一直在努力面向 parallel architectural paradigm 做优化探索。随着 AI 这些年的兴起， 不同的时间节点，相似的场景，面向 FPGA/GPU 异构计算场景，对编程语言与编译器领域提出了新的挑战。

除了传统 Compiler 诸如 IBM XL Compilers, Intel Compilers 等做的 Automatic Parallelizing 工作，在极致性能探索方面，基于多面体模型 (polytope model) 的编译优化技术作为解决程序并行化、数据局部性优化的一种手段，成为编译优化领域的研究热点。

而在 Parallel Languages 层面，对 C&C++ 开发人员，CUDA 的出现降低了 GPU 的编程门槛，但 GPU 和 CPU 两种硬件模型本质区别，导致过高的开发成本，需要学习和了解更多底层硬件细节，还更不用说更高级语言的开发语言像 Java 等所面临的底层硬件模型与高级语言之间巨大的 GAP。

在 Java 领域，最早在 JVM 技术峰会 '2014，AMD 曾经分享过他们的 Sumatra 项目，尝试实现 JVM 与 Heterogeneous System Architecture 目标硬件交互。而在最近，由 The University of Manchester 发起的 TornadoVM 项目，实现包含：一个 Just-in-Time 编译，支持从 Java bytecode 到 OpenCL 的映射，一个优化的运行时引擎，以及可以保持 Java 堆和异构设备堆内存一致性的内存管理器。TornadoVM 的目标是开发人员不需要了解 GPU 编程语言或者相关的 GPU 体系结构知识就可以编写面向异构的并行程序。TornadoVM 可以透明地运行在 AMD GPUs, NVIDIA GPUs, Intel integrated GPUs 以及 multi-core CPUs 上。

在通用 CPU 领域， OpenJDK 社区的 Vector API 项目 (Panama 的子项目)，依赖 CPU 的 SIMD 指令，获得计算性能的成倍提升，Vector API 在大数据，AI 计算也有非常广的应用场景。阿里 JVM 团队把 Vector API 移植到了 AJDK 11，后续会开源到 Alibaba Dragonwell，分享下我们获得的基础性能数据：



时间（单位：milliseconds）越短，性能越好

（三）Polyglot Programing，链接多语言生态

Polyglot Programming 并不是一个新的概念。在 Managed Runtime 领域，2017 年 IBM 开源 Open Managed Runtime(OMR)，以及 2018 年 Oracle 开源 Truffle/Graal 技术。OMR 和 Graal 技术让开发人员实现一个新的语言成本大幅下降。前者 OMR 以 C、C++ 组件的形式提供了 Garbage Collection (GC)，Just-in-Time（JIT） 以及 Reliability，availability and serviceability (RAS，工具)等，开发人员可以依赖这些组件，通过 'glue' 的方式基于这些组件实现自己的高性能语言。而后者 Truffle/Graal，Truffle 是一个依赖 AST parser 实现新的语言的 Java 框架，本质上是将你的新的语言映射到 JVM 世界。不同于 Scala，JRuby 这些围绕 JVM 生态本身构建的语言，他们本质是还是 Java， 无论是 OMR， 还是 Truffle/Graal，他们都提供了生产级的 GC，JIT，以及 RAS 服务支持，新开发的语言完全不需要再重新实现这些底层技术。

从业界来看，面向特定领域的 Domain Specific Language (DSL) 语言已经有向这些技术迁移的趋势，高盛正在与 Graal 社区合作，把他们的 DSL 迁移到 Graal 上。另外 Ruby/OMR, Python/Graal， JS/Graal，WASM/Graal 等这些真正链接不同语言生态的项目，也正在迅速发展起来。

回到 AJDK， Graal 已经在 AJDK 8 开始支持， JS/Graal 这样成熟的技术，已经在阿里内部业务上线。

六、最后

Java 是一项二十多年前被发明出来的技术，她历经磨难，几易其主，但却历久弥新。这篇报告旨在为 Java 的开发者们梳理下目前的 Java 技术现状，以及讨论在云，AI 等这些重要领域内 Java 技术的演进趋势。在介绍的相关部分，我们也穿插了阿里的一些工程实践。作为世界上最大的 Java 用户之一，我们也一直在探索把前沿的 Java 技术，通过在阿里丰富的业务场景的试验，真正把这些技术应用于真实的生产环境。我们也非常乐于分享和贡献 Java 领域的经验、实践与技术洞见，共同促进 Java 的发展。

七、参考

- https://www.infoq.com/news/2017/10/javaone-opening/
- https://www.oracle.com/technetwork/java/javase/11-relnote-issues-5012449.html#Diffs
- https://openjdk.java.net/jeps/377
- https://mp.weixin.qq.com/s/FQpvT5wIy9xwhX2jHMU7aw
- https://mp.weixin.qq.com/s/K1us6aH-gjHsWGhQ3SulFg
- https://www.infoq.cn/article/uzHpEbpMwiYd85jYslka

必备干货，所向披靡

如何回答性能优化的问题，才能打动阿里面试官？

作者：齐光

简介：阿里妹导读——日常工作中，我们多少都会遇到应用的性能问题。在阿里面试中，性能优化也是常被问到的题目，用来考察是否有实际的线上问题处理经验。面对这类问题，阿里工程师齐光给出了详细流程。来阿里面试前，先看看这篇文章哦。

性能问题和 Bug 不同，后者的分析和解决思路更清晰，很多时候从应用日志（文中的应用指分布式服务下的单个节点）即可直接找到问题根源，而性能问题，其排查思路更为复杂一些。

对应用进行性能优化，是一个系统性的工程，对工程师的技术广度和技术深度都有所要求。一个简单的应用，它不仅包含了应用代码本身，还和容器（虚拟机）、操作系统、存储、网络、文件系统等紧密相关，线上应用一旦出现了性能问题，需要我们从多方面去考虑。

与此同时，除了一些低级的代码逻辑引发的性能问题外，很多性能问题隐藏的较深，排查起来会比较困难，需要我们对应用的各个子模块、应用所使用的框架和组件的原理有所了解，同时掌握一定的性能优化工具和经验。

本文总结了我们在进行性能优化时常用的一些工具及技巧，目的是希望通过一个全面的视角，去感知性能优化的整体脉络。本文主要分为下面三个部分：

一、第一部分会介绍性能优化的一些背景知识。

二、第二部分会介绍性能优化的通用流程以及常见的一些误区。

三、第三部分会从系统层和业务层的角度，介绍高效的性能问题定位工具和高频性能瓶颈点分布。

本文中提到的线程、堆、垃圾回收等名词，如无特别说明，指的是 Java 应用中的相关概念。

一、性能优化的背景

前面提到过，应用出现性能问题和应用存在缺陷是不一样的，后者大多数是由于代码的质量问题导致，会导致应用功能性的缺失或出现风险，一经发现，会被及时修复。而性能问题，可能是由多方面的因素共同作用的结果：代码质量一般、业务发展太快、应用架构设计不合理等，这些问题处理起来一般耗时较长、分析链路复杂，大家都不愿意干，因此可能会被一些临时性的补救手段所掩盖，如：系统水位高或者单机的线程池队列爆炸，那就集群扩容增加机器；内存占用高/高峰时段 OOM，那就重启分分钟解决……

临时性的补救措施只是在给应用埋雷，同时也只能解决部分问题。譬如，在很多场景下，加机器也并不能解决应用的性能问题，如对时延比较敏感的一些应用必须把单机的性能优化到极致，与此同时，加机器这种方式也造成了资源的浪费，长期来看是得不偿失的。对应用进行合理的性能优化，可在应用稳定性、成本核算获得很大的收益。

上面我们阐述了进行性能优化的必要性。假设现在我们的应用已经有了性能问题（eg. CPU 水位比较高），准备开始进行优化工作了，在这个过程中，潜在的痛点会有哪些呢？下面列出一些较为常见的：

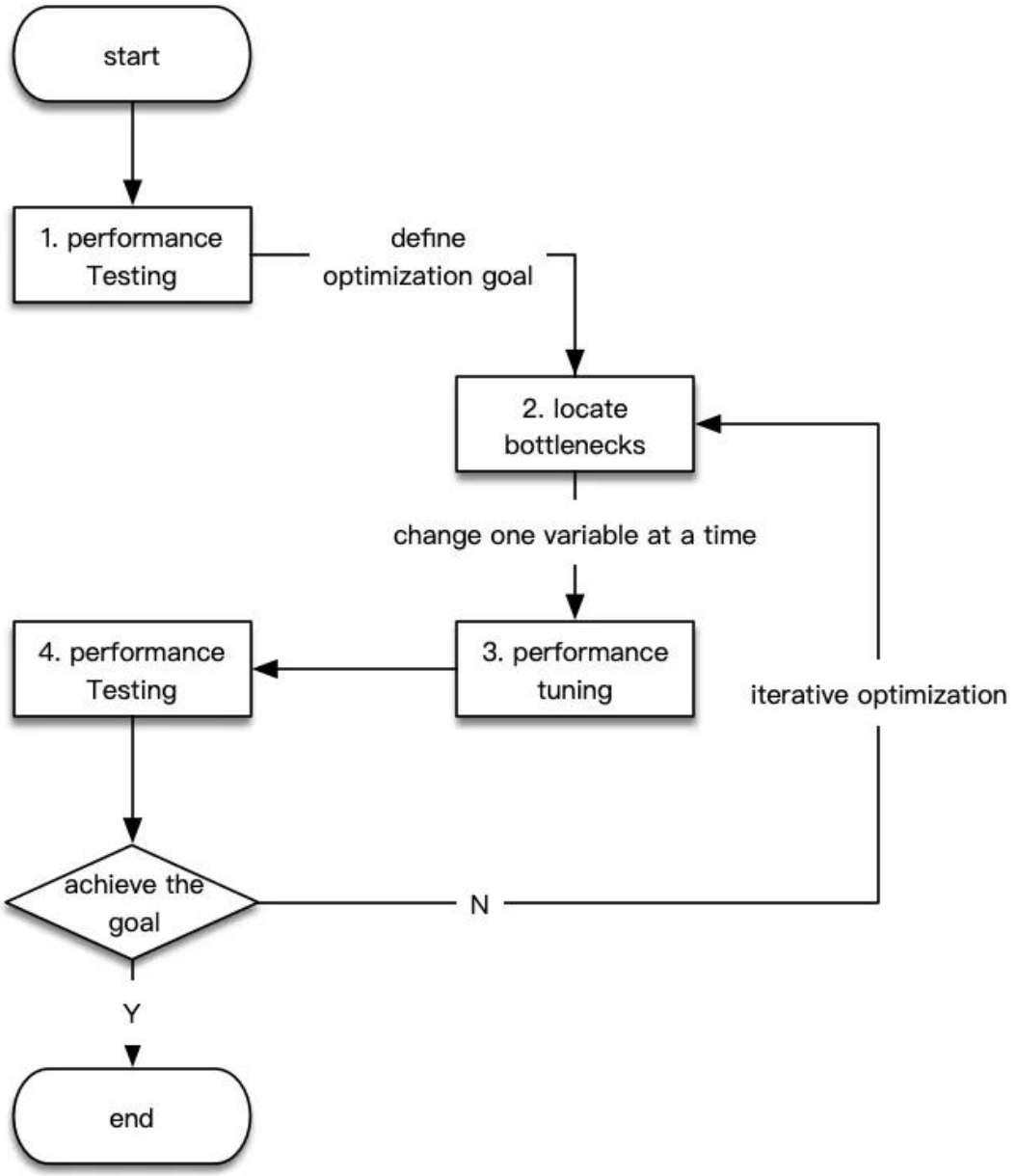
- 对性能优化的流程不是很清晰。初步定为一个疑似瓶颈点后，就兴高采烈地吭哧吭哧开始干，最终解决的问题其实只是一个浅层次的性能瓶颈，真实的问题的根源并未触达；
- 对性能瓶颈点的分析思路不是很清晰。CPU、网络、内存.....这么多的性能指标，我到底该关注什么，应该从哪一块儿开始入手？
- 对性能优化的工具不了解。遇到问题后，不清楚该用哪个工具，不知道通过工具得到的指标代表什么。

二、性能优化的流程

在性能优化这个领域，并没有一个严格的流程定义，但是对于绝大多数的优化场景，我们可以将其过程抽象为下面四个步骤。

- 准备阶段：主要工作是是通过性能测试，了解应用的概况、瓶颈的大概方向，明确优化目标；
- 分析阶段：通过各种工具或手段，初步定位性能瓶颈点；
- 调优阶段：根据定位到的瓶颈点，进行应用性能调优；
- 测试阶段：让调优过的应用进行性能测试，与准备阶段的各项指标进行对比，观测其是否符合预期，如果瓶颈点没有消除或者性能指标不符合预期，则重复步骤 2 和 3。

下图即为上述四个阶段的简要流程。



（一）通用流程详解

在上述通用流程的四个步骤当中，步骤 2 和 3 我们会在接下来两个部分重点进行介绍。首先我们来看一下，在准备阶段和测试阶段，我们需要做一些什么。

准备阶段

准备阶段是非常关键的一步，不能省略。

首先，需要对我们进行调优的对象进行详尽的了解，所谓知己知彼，百战不殆。

- 对性能问题进行粗略评估，过滤一些因为低级的业务逻辑导致的性能问题。譬如，线上应用日志级别不合理，可能会在大流量时导致 CPU 和磁盘的负载飙高，这种情况调整日志级别即可；
- 了解应用的的总体架构，比如应用的外部依赖和核心接口有哪些，使用了哪些组件和框架，哪些接口、模块的使用率较高，上下游的数据链路是怎么样的等；
- 了解应用对应的服务器信息，如服务器所在的集群信息、服务器的 CPU/内存信息、安装的 Linux 版本信息、服务器是容器还是虚拟机、所在宿主机混部后是否对当前应用有干扰等；

其次，我们需要获取基准数据，然后结合基准数据和当前的一些业务指标，确定此次性能优化的最终目标。

使用基准测试工具获取系统细粒度指标。可以使用若干 Linux 基准测试工具（eg. jmeter、ab、loadrunnerwrk、wrk 等），得到文件系统、磁盘 I/O、网络等的性能报告。除此之外，类似 GC、Web 服务器、网卡流量等信息，如有必要也是需要了解记录的；

通过压测工具或者压测平台（如果有的话），对应用进行压力测试，获取当前应用的宏观业务指标，譬如：响应时间、吞吐量、TPS、QPS、消费速率（对于有 MQ 的应用）等。压力测试也可以省略，可以结合当前的实际业务和过往的监控数据，去统计当前的一些核心业务指标，如午高峰的服务 TPS。

测试阶段

进入到这一阶段，说明我们已经初步确定了应用性能瓶颈的所在，而且已经进行初步的调优了。检测我们调优是否有效的方式，就是在仿真的条件下，对应用进行压力测试。注意：由于 Java 有 JIT（just-in-time compilation）过程，因此压力测试时可能需要进行前期预热。

如果压力测试的结果符合了预期的调优目标，或者与基准数据相比，有很大的改善，则我们可以继续通过工具定位下一个瓶颈点，否则，则需要暂时排除这个瓶颈点，继续寻找下一个变量。

（二）注意事项

在进行性能优化时，了解下面这些注意事项可以让我们少走一些弯路。

- 性能瓶颈点通常呈现 2/8 分布，即 80%的性能问题通常是由 20%的性能瓶颈点导致的，2/8 原则也意味着并不是所有的性能问题都值得去优化；
- 性能优化是一个渐进、迭代的过程，需要逐步、动态地进行。记录基准后，每次改变一个变量，引入多个变量会给我们的观测、优化过程造成干扰；
- 不要过度追求应用的单机性能，如果单机表现良好，则应该从系统架构的角度去思考；不要过度追求单一维度上的极致优化，如过度追求 CPU 的性能而忽略了内存方面的瓶颈；

- 选择合适的性能优化工具，可以使得性能优化取得事半功倍的效果；
- 整个应用的优化，应该与线上系统隔离，新的代码上线应该有降级方案。

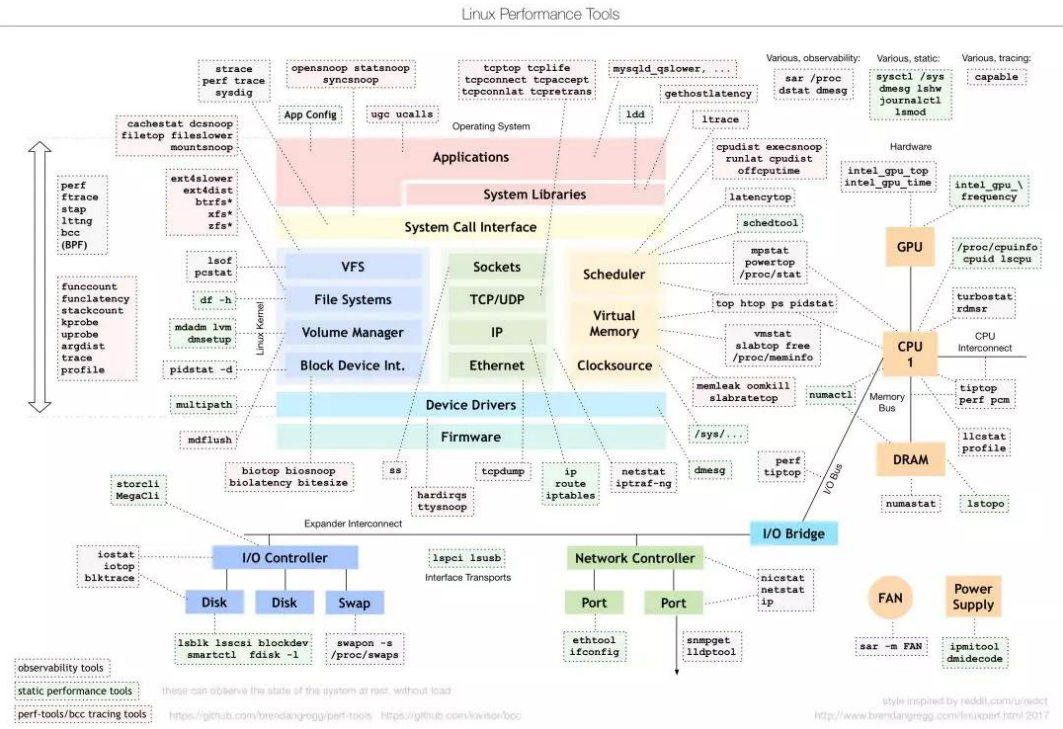
三、瓶颈点分析工具箱

性能优化其实就是找出应用存在性能瓶颈点，然后设法通过一些调优手段去缓解。性能瓶颈点的定位是较困难的，快速、直接地定位到瓶颈点，需要具备下面两个条件：

- 恰到好处的工具；
- 一定的性能优化经验。

工欲善其事，必先利其器，我们该如何选择合适的工具呢？不同的优化场景下，又该选择那些工具呢？

首选，我们来看一下大名鼎鼎的「性能工具(Linux Performance Tools-full)图」，想必很多工程师都知道，它出自系统性能专家 Brendan Gregg。该图从 Linux 内核的各个子系统出发，列出了我们在对各个子系统进行性能分析时，可使用的工具，涵盖了监测、分析、调优等性能优化的方方面面。除了这张全景图之外，Brendan Gregg 还单独提供了基准测试工具(Linux Performance Benchmark Tools)图、性能监测工具(Linux Performance Observability Tools)图等，更详细的内容请参考 Brendan Gregg 的网站说明。



图片来源：http://www.brendangregg.com/linuxperf.html?spm=ata.13261165.0.0.34646b44KX9rGc

上面这张图非常经典，是我们做性能优化时非常好的参考资料，但事实上，我们在实际运用的时候，会发现可能它并不是最合适的，原因主要有下面两点：

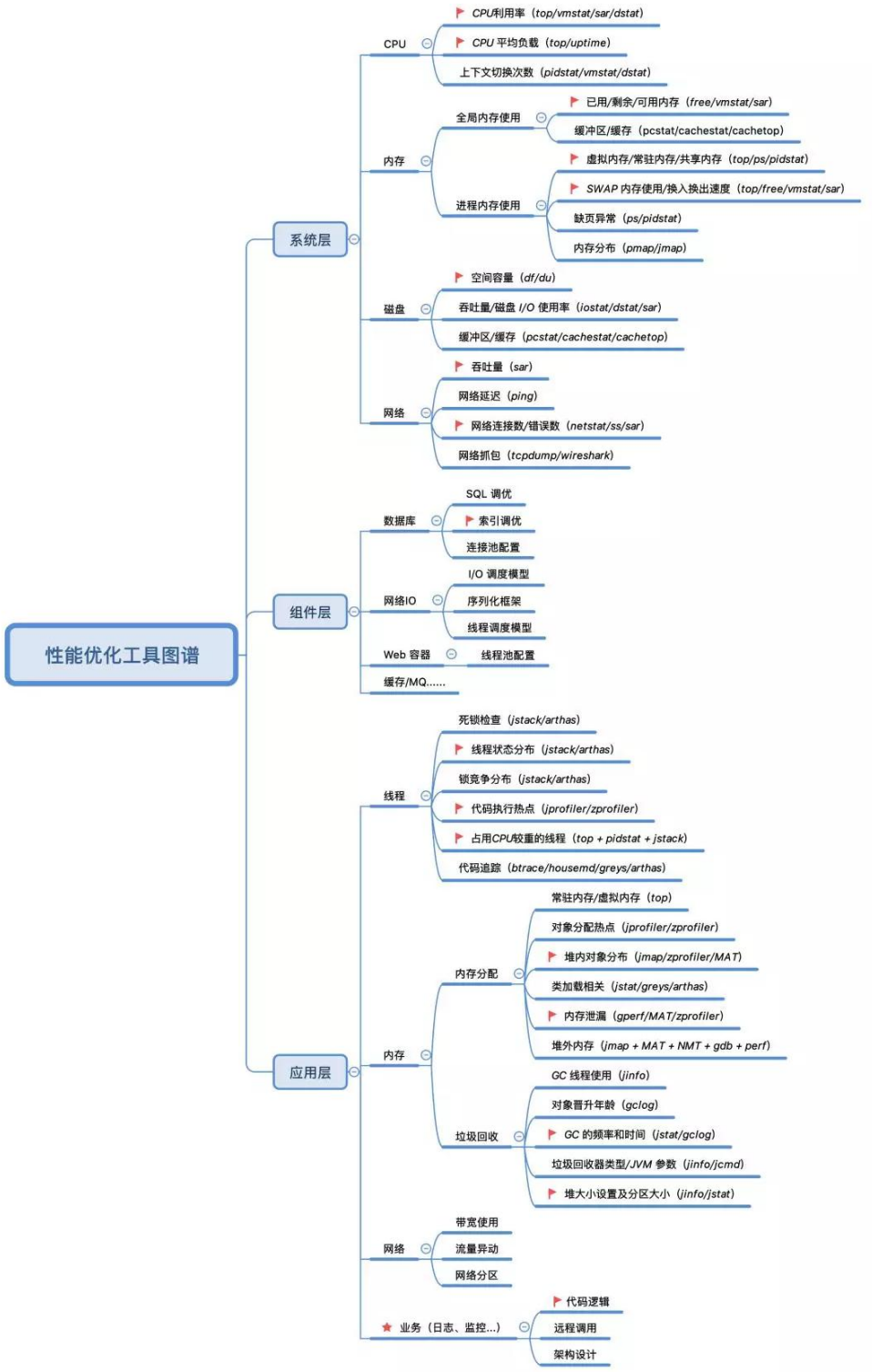
- 对分析经验要求较高。上面这张图其实是从 Linux 系统资源的角度去观测性能指标的，这要求我们对 Linux 各个子系统的功能、原理要有所了解。举例：遇到性能问题了，我们不会拿每个子系统下的工具都去试一遍，大多数情况是：我们怀疑某个子系统有问题，然后根据这张图上列举的工具，去观测或者验证我们的猜想，这无疑拔高了对性能优化经验的要求；

- 适用性和完整性不是很好。我们在分析性能问题时，从系统底层自底向上地分析是较低效的，大多数时候，从应用层面去分析会更加有效。性能工具(Linux Performance Tools-full)图只是从系统层一个角度给出了工具集，如果从应用层开始分析，我们可以使用哪些工具？哪些点是我们首先需要关注的？

鉴于上面若干痛点，下面给出了一张更为实用的「性能优化工具图谱」，该图分别从系统层、应用层（含组件层）的角度出发，列举了我们在分析性能问题时首先需要关注的各项指标（其中?标注的是最需要关注的），这些点是最有可能出现性能瓶颈的地方。需要注意的是，一些低频的指标或工具，在图中并没有列出来，如 CPU 中断、索引节点使用、I/O 事件跟踪等，这些低频点的排查思路较复杂，一般遇到的机会也不多，在这里我们聚焦最常见的一些就可以了。

对比上面的性能工具(Linux Performance Tools-full)图，下图的优势在于：把具体的工具同性能指标结合了起来，同时从不同的层次去描述了性能瓶颈点的分布，实用性和可操作性更强一些。系统层的工具分为 CPU、内存、磁盘（含文件系统）、网络四个部分，工具集同性能工具(Linux Performance Tools-full)图中的工具基本一致。组件层和应用层中的工具构成为：JDK 提供的一些工具 + Trace 工具 + dump 分析工具 + Profiling 工具等。

这里就不具体介绍这些工具的具体用法了，我们可以使用 man 命令得到工具详尽的使用说明，除此之外，还有另外一个查询命令手册的方法：info。info 可以理解为 man 的详细版本，如果 man 的输出不太好理解，可以去参考 info 文档，命令太多，记不住也没必要记住。


性能优化工具图谱
系统层
CPU
CPU利用率（top/vmstat/sar/dstat）
CPU 平均负载（top/uptime）
上下文切换次数（pidstat/vmstat/dstat）
内存
全局内存使用
已用/剩余/可用内存（free/vmstat/sar）
缓冲区/缓存（pcstat/cachestat/cachetop）
进程内存使用
虚拟内存/常驻内存/共享内存（top/ps/pidstat）
SWAP 内存使用/换入换出速度（top/free/vmstat/sar）
缺页异常（ps/pidstat）
内存分布（pmap/jmap）
磁盘
空间容量（df/du）
吞吐量/磁盘 I/O 使用率（iostat/dstat/sar）
缓冲区/缓存（pcstat/cachestat/cachetop）
网络
吞吐量（sar）
网络延迟（ping）
网络连接数/错误数（netstat/ss/sar）
网络抓包（tcpdump/wireshark）
组件层
数据库
SQL 调优
索引调优
连接池配置
网络IO
I/O 调度模型
序列化框架
线程调度模型
Web 容器
线程池配置
缓存/MQ.....
应用层
线程
死锁检查（jstack/arthas）
线程状态分布（jstack/arthas）
锁竞争分布（jstack/arthas）
代码执行热点（jprofiler/zprofiler）
占用CPU较重的线程（top + pidstat + jstack）
代码追踪（btrace/housemd/greys/arthas）
内存
内存分配
常驻内存/虚拟内存（top）
对象分配热点（jprofiler/zprofiler）
堆内对象分布（jmap/zprofiler/MAT）
类加载相关（jstat/greys/arthas）
内存泄漏（gperf/MAT/zprofiler）
堆外内存（jmap + MAT + NMT + gdb + perf）
垃圾回收
GC 线程使用（jinfo）
对象晋升年龄（gclog）
GC 的频率和时间（jstat/gclog）
垃圾回收器类型/JVM 参数（jinfo/jcmd）
堆大小设置及分区大小（jinfo/jstat）
网络
带宽使用
流量异动
网络分区
业务（日志、监控...）
代码逻辑
远程调用
架构设计

上面这张图该如何使用？

首先，虽然从系统、组件、应用两个三个角度去描述瓶颈点的分布，但在实际运行时，这三者往往是相辅相成、相互影响的。系统是为应用提供了运行时环境，性能问题的本质就是系统资源达到了使用的上限，反映在应用层，就是应用/组件的各项指标开始下降；而应用/组件的不合理使用和设计，也会加速系统资源的耗尽。因此，分析瓶颈点时，需要我们结合从不同角度分析出的结果，抽出共性，得到最终的结论。

其次，建议先从应用层入手，分析图中标注的高频指标，抓出最重要的、最可疑的、最有可能导致性能的点，得到初步的结论后，再去系统层进行验证。这样做的好处是：很多性能瓶颈点体现在系统层，会是多变量呈现的，譬如，应用层的垃圾回收（GC）指标出现了异常，通过 JDK 自带的工具很容易观测到，但是体现在系统层上，会发现系统当前的 CPU 利用率、内存指标都不太正常，这就给我们的分析思路带来了困扰。

最后，如果瓶颈点在应用层和系统层均呈现出多变量分布，建议此时使用 ZProfiler、JProfiler 等工具对应用进行 Profiling，获取应用的综合性能信息（注：Profiling 指的是在应用运行时，通过事件（Event-based）、统计抽样（Sampling Statistical）或植入附加指令（Byte-Code instrumentation）等方法，收集应用运行时的信息，来研究应用行为的动态分析方法）。譬如，可以对 CPU 进行抽样统计，结合各种符号表信息，得到一段时间内应用内的代码热点。

下面介绍在不同的分析层次，我们需要关注的核心性能指标，同时，也会介绍如何初步根据这些指标，判断系统或应用是否存在性能瓶颈点，至于瓶颈点的确认、瓶颈点的成因、调优手段，将会在下一部分展开。

（一）CPU&&线程

和 CPU 相关的指标主要有以下几个。常用的工具有 top、 ps、uptime、 vmstat、pidstat 等。

- CPU 利用率（CPU Utilization）
- CPU 平均负载（Load Average）
- 上下文切换次数（Context Switch）

- top - 12:20:57 up 25 days, 20:49, 2 users, load average: 0.93, 0.97, 0.79
  Tasks: 51 total, 1 running, 50 sleeping, 0 stopped, 0 zombie
  %Cpu(s): 1.6 us, 1.8 sy, 0.0 ni, 89.1 id, 0.1 wa, 0.0 hi, 0.1 si, 7.3 st
  KiB Mem : 8388608 total, 476436 free, 5903224 used, 2008948 buff/cache
  KiB Swap: 0 total, 0 free, 0 used. 0 avail Mem

- PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
  119680 admin 20 0 600908 72332 5768 S 2.3 0.9 52:32.61 obproxy
  65877 root 20 0 93528 4936 2328 S 1.3 0.1 449:03.61 alisentry_cli

第一行显示的内容：当前时间、系统运行时间以及正在登录用户数。load average 后的三个数字，依次表示过去 1 分钟、5 分钟、15 分钟的平均负载（Load Average）。平均负载是指单位时间内，系统处于可运行状态（正在使用 CPU 或者正在等待 CPU 的进程，R 状态）和不可中断状态（D 状态）的平均进程数，也就是平均活跃进程数，CPU 平均负载和 CPU 使用率并没有直接关系。

第三行的内容表示 CPU 利用率，每一列的含义可以使用 man 查看。CPU 使用率体现了单位时间内 CPU 使用情况的统计，以百分比的方式展示。计算方式为：CPU 利用率 = 1 - （CPU 空闲时间）/ CPU 总的时间。需要注意的是，通过性能分析工具得到的 CPU 的利用率其实是某个采样时间内的 CPU 平均值。注：top 工具显示的的 CPU 利用率是把所有 CPU 核的数值加起来的，即 8 核 CPU 的利用率最大可以到达 800%（可以用 htop 等更新一些的工具代替 top）。

使用 vmstat 命令，可以查看到「上下文切换次数」这个指标，如下表所示，每隔 1 秒输出 1 组数据：

- $ vmstat 1

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
0 0 0 504804 0 1967508 0 0 644 33377 0 1 2 2 88 0 9
```

上表的 cs（context switch） 就是每秒上下文切换的次数，按照不同场景，CPU 上下文切换还可以分为中断上下文切换、线程上下文切换和进程上下文切换三种，但是无论是哪一种，过多的上下文切换，都会把 CPU 时间消耗在寄存器、内核栈以及虚拟内存等数据的保存和恢复上，从而缩短进程真正运行的时间，导致系统的整体性能大幅下降。vmstat 的输出中 us、sy 分别用户态和内核态的 CPU 利用率，这两个值也非常具有参考意义。

vmstat 的输只给出了系统总体的上下文切换情况，要想查看每个进程的上下文切换详情（如自愿和非自愿切换），需要使用 pidstat，该命令还可以查看某个进程用户态

和内核态的 CPU 利用率。

CPU 相关指标异常的分析思路是什么？

- CPU 利用率：如果我们观察某段时间系统或应用进程的 CPU 利用率一直很高（单个 core 超过 80%），那么就值得我们警惕了。我们可以多次使用 jstack 命令 dump 应用线程栈查看热点代码，非 Java 应用可以直接使用 perf 进行 CPU 采采样，离线分析采样数据后得到 CPU 执行热点（Java 应用需要符号表进行堆栈信息映射，不能直接使用 perf 得到结果）。

- CPU 平均负载：平均负载高于 CPU 数量 70%，意味着系统存在瓶颈点，造成负载升高的原因有很多，在这里就不展开了。需要注意的是，通过监控系统监测平均负载的变化趋势，更容易定位问题，有时候大文件的加载等，也会导致平均负载瞬时升高。如果 1 分钟/5 分钟/15 分钟的三个值相差不大，那说明系统负载很平稳，则不用关注，如果这三个值逐渐降低，说明负载在渐渐升高，需要关注整体性能；
- CPU 上下文切换：上下文切换这个指标，并没有经验值可推荐（几十到几万都有可能），这个指标值取决于系统本身的 CPU 性能，以及当前应用工作的情况。但是，如果系统或者应用的上下文切换次数出现数量级的增长，就有很大概率说明存在性能问题，如非自愿上下切换大幅度上升，说明有太多的线程在竞争 CPU。

上面这三个指标是密切相关的，如频繁的 CPU 上下文切换，可能会导致平均负载升高。如何根据这三者之间的关系进行应用调优，将在下一部分介绍。

CPU 上的的一些异动，通常也可以从线程上观测到，但需要注意的是，线程问题并不完全和 CPU 相关。与线程相关的指标，主要有下面几个（均都可以通过 JDK 自带的 jstack 工具直接或间接得到）：

- 应用中的总的线程数；
- 应用中各个线程状态的分布；
- 线程锁的使用情况，如死锁、锁分布等；

关于线程，可关注的异常有：

- 线程总数是否过多。过多的线程，体现在 CPU 上就是导致频繁的上下文切换，同时线程过多也会消耗内存，线程总数大小和应用本身和机器配置相关；

- 线程的状态是否异常。观察 WAITING/BLOCKED 线程是否过多（线程数设置过多或锁竞争剧烈），结合应用内部锁使用的情况综合分析；
- 结合 CPU 利用率，观察是否存在大量消耗 CPU 的线程。

（二）内存&&堆

和内存相关的指标主要有以下几个，常用的分析工具有：top、free、vmstat、pidstat 以及 JDK 自带的一些工具。

- 系统内存的使用情况，包括剩余内存、已用内存、可用内存、缓存/缓冲区；
- 进程（含 Java 进程）的虚拟内存、常驻内存、共享内存；
- 进程的缺页异常数，包含主缺页异常和次缺页异常；
- Swap 换入和换出的内存大小、Swap 参数配置；

- JVM 堆的分配，JVM 启动参数；
- JVM 堆的回收，GC 情况。

使用 free 可以查看系统内存的使用情况和 Swap 分区的使用情况，top 工具可以具体到每个进程，如我们可以用使用 top 工具查看 Java 进程的常驻内存大小（RES），这两个工具结合起来，可用覆盖大多数内存指标。下面是使用 free 命令的输出：

- $free -h

total	used	free	shared	buff/cache	available

- Mem: 125G 6.8G 54G 2.5M 64G 118G

Swap: 2.0G 305M 1.7G

上述输出各列的具体含义在这里不在赘述，也比较容易理解。重点介绍下 swap 和 buff/cache 这两个指标。

Swap 的作用是把一个本地文件或者一块磁盘空间作为内存来使用，包括换出和换入两个过程。Swap 需要读写磁盘，所以性能不是很高，事实上，包括 ElasticSearch 、Hadoop 在内绝大部分 Java 应用都建议关掉 Swap，这是因为内存的成本一直在降低，同时这也和 JVM 的垃圾回收过程有关：JVM 在 GC 的时候会遍历所有用到的堆的内存，如果这部分内存被 Swap 出去了，遍历的时候就会有磁盘 I/O 产生。Swap 分区的升高一般和磁盘的使用强相关，具体分析时，需要结合缓存使用情况、swappiness 阈值以及匿名页和文件页的活跃情况综合分析。

buff/cache 是缓存和缓冲区的大小。缓存（cache）：是从磁盘读取的文件的或者向磁盘写文件时的临时存储数据，面向文件。使用 cachestat 可以查看整个系统缓存的读写命中情况，使用 cachetop 可以观察每个进程缓存的读写命中情况。缓冲区（buffer）是写入磁盘数据或从磁盘直接读取的数据的临时存储，面向块设备。free 命令的输出中，这两个指标是加在一起的，使用 vmstat 命令可以区分缓存和缓冲区，还可以看到 Swap 分区换入和换出的内存大小。

了解到常见的内存指标后，常见的内存问题又有哪些？总结如下：

- 系统剩余内存/可用不足（某个进程占用太多、系统本身内存不足），内存溢出；
- 内存回收异常：内存泄漏（进程在一段时间内内存使用持续走高）、GC 频率异常；
- 缓存使用过大（大文件读取或写入）、缓存命中率不高；
- 缺页异常过多（频繁的 I/O 读）；
- Swap 分区使用异常（使用过大）；

内存相关指标异常后，分析思路是怎么样的?

- 使用 free/top 查看内存的全局使用情况，如系统内存的使用、Swap 分区内存使用、缓存/缓冲区占用情况等，初步判断内存问题存在的方向：进程内存、缓存/缓冲区、Swap 分区；
- 观察一段时间内存的使用趋势。如通过 vmstat 观察内存使用是否一直在增长；通过 jmap 定时统计对象内存分布情况，判断是否存在内存泄漏，通过 cachetop 命令，定位缓冲区升高的根源等；
- 根据内存问题的类型，结合应用本身，进行详细分析。

举例：使用 free 发现缓存/缓冲区占用不大，排除缓存/缓冲区对内存的影响后 -> 使用 vmstat 或者 sar 观察一下各个进程内存使用变化趋势 -> 发现某个进程的内存时候用持续走高 -> 如果是 Java 应用，可以使用 jmap / VisualVM / heap dump 分析等工具观察对象内存的分配，或者通过 jstat 观察 GC 后的应用内存变化 -> 结合业务场景，定位为内存泄漏/GC 参数配置不合理/业务代码异常等。

（三）磁盘&&文件

在分析和磁盘相关的问题时，通常是将其和文件系统同时考虑的，下面不再区分。和磁盘/文件系统相关的指标主要有以下几个，常用的观测工具为 iostat 和 pidstat，前者适用于整个系统，后者可观察具体进程的 I/O。

- 磁盘 I/O 利用率：是指磁盘处理 I/O 的时间百分比；
- 磁盘吞吐量：是指每秒的 I/O 请求大小，单位为 KB;
- I/O 响应时间，是指 I/O 请求从发出到收到响应的间隔，包含在队列中的等待时间和实际处理时间；
- IOPS（Input/Output Per Second）：每秒的 I/O 请求数；
- I/O 等待队列大小，指的是平均 I/O 队列长度，队列长度越短越好；

使用 iostat 的输出界面如下：

- $iostat -dx

  Linux 3.10.0-327.ali2010.alios7.x86_64 (loginhost2.alipay.em14) 10/20/2019 x86_64 (32 CPU)

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrq-sz avgqu-sz await r_await w_await svctm %util

sda 0.01 15.49 0.05 8.21 3.10 240.49 58.92 0.04 4.38 2.39 4.39 0.09 0.07

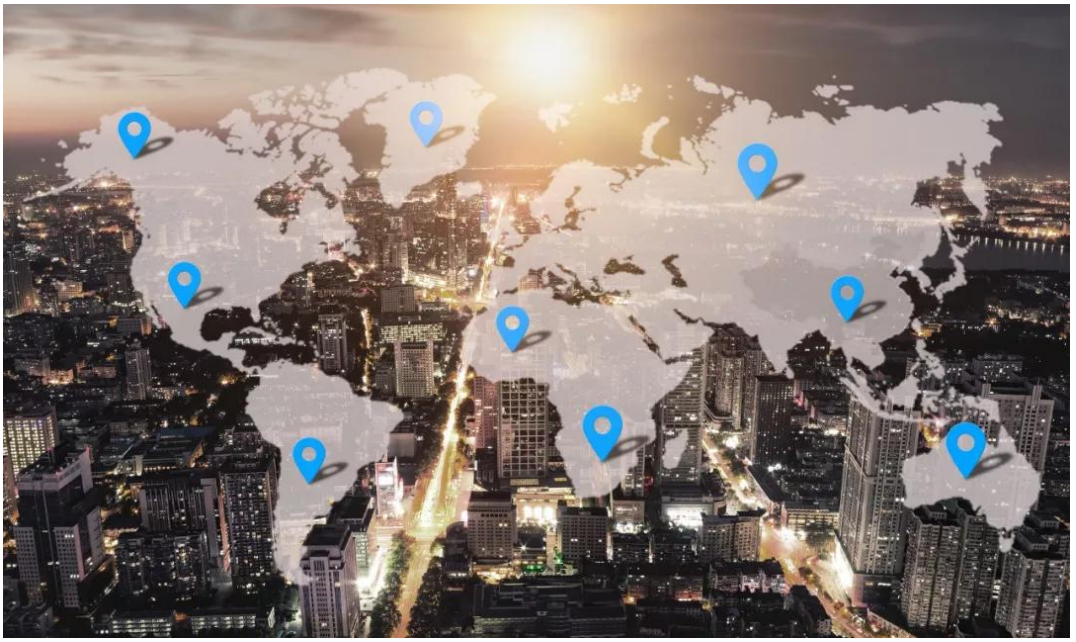
上图中 %util ，即为磁盘 I/O 利用率，同 CPU 利用率一样，这个值也可能超过 100%（存在并行 I/O）；rkB/s 和 wkB/s 分别表示每秒从磁盘读取和写入的数据量，即吞吐量，单位为 KB；磁盘 I/O 处理时间的指标为 r_await 和 w_await 分别表示读/写请求处理完成的响应时间，svctm 表示处理 I/O 所需要的平均时间，该指标已被废弃，无实际意义。r/s + w/s 为 IOPS 指标，分别表示每秒发送给磁盘的读请求数和写请求数；aqu-sz 表示等待队列的长度。

pidstat 的输出大部分和 iostat 类似，区别在于它可以实时查看每个进程的 I/O 情况。

如何判断磁盘的指标出现了异常?

- 当磁盘 I/O 利用率长时间超过 80%，或者响应时间过大（对于 SSD，从 0.0x 毫秒到 1.x 毫秒不等，机械磁盘一般为 5ms~10ms），通常意味着磁盘 I/O 存在性能瓶颈；
- 如果 %util 很大，而 rkB/s 和 wkB/s 很小，一般是因为存在较多的磁盘随机读写，最好把随机读写优化成顺序读写，（可以通过 strace 或者 blktrace 观察 I/O 是否连续判断是否是顺序的读写行为，随机读写应可关注 IOPS 指标，顺序读写可关注吞吐量指标）；

- 如果 avgqu-sz 比较大，说明有很多 I/O 请求在队列中等待。一般来说，如果单块磁盘的队列长度持续超过 2，一般认为该磁盘存在 I/O 性能问题。



（四）网络

网络这个概念涵盖的范围较广，在应用层、传输层、网络层、网络接口层都有不同的指标去衡量。这里我们讨论的「网络」，特指应用层的网络，通常使用的指标如下:

- 网络带宽：表示链路的最大传输速率；
- 网络吞吐：表示单位时间内成功传输的数据量大小；
- 网络延时：表示从网络请求发出后直到收到远端响应，所需要的时间；
- 网络连接数和错误数；

一般来说，应用层的网络瓶颈有如下几类：

- 集群或机器所在的机房的网络带宽饱和，影响应用 QPS/TPS 的提升；
- 网络吞吐出现异常，如接口存在大量的数据传输，造成带宽占用过高；
- 网络连接出现异常或错误；
- 网络出现分区。

带宽和网络吞吐这两个指标，一般我们会关注整个应用的，通过监控系统可直接得到，如果一段时间内出现了明显的指标上升，说明存在网络性能瓶颈。对于单机，可以使用 sar 得到网络接口、进程的网络吞吐。

使用 ping 或者 hping3 可以得到是否出现网络分区、网络具体时延。对于应用，我们更关注整个链路的时延，可以通过中间件埋点后输出的 trace 日志得到链路上各个环节的时延信息。

使用 netstat、ss 和 sar 可以获取网络连接数或网络错误数。过多网络链接造成的开销是很大的，一是会占用文件描述符，二是会占用缓存，因此系统可以支撑的网络链接数是有限的。

（五）工具总结

可以看到的是，在分析 CPU、内存、磁盘等的性能指标时，有几种工具是高频出现的，如 top、vmstat、pidstat，这里稍微总结一下:

- CPU：top、vmstat、pidstat、sar、perf、jstack、jstat；
- 内存：top、free、vmstat、cachetop、cachestat、sar、jmap；
- 磁盘：top、iostat、vmstat、pidstat、du/df；
- 网络：netstat、sar、dstat、tcpdump；
- 应用：profiler、dump 分析。

上述的很多工具，大部分是用于查看系统层指标的，在应用层，除了有 JDK 提供的一系列工具，一些商用的产品如 gceasy.io（分析 GC 日志）、fastthread.io（分析线程 dump 日志）也是不错的。

排查 Java 应用的线上异常或者分析应用代码瓶颈，可以使用阿里开源的 Arthas ，这个工具非常强大，下面简单介绍下。

Arthas 主要面向线上应用实时诊断，解决的是类似「线上应用异常了，需要在线进行分析和定位」的问题，当然，Arthas 提供的一些方法调用追踪工具，对我们排查诸如「慢查询」等问题，也是非常有帮助的。Arthas 提供的主要功能有：

- 获取线程统计，如线程持有的锁统计、CPU 利用率统计等；
- 类加载信息、动态类加载、方法加载信息；
- 调用栈追踪，调用耗时统计；
- 方法调用参数、结果检测；
- 系统配置、应用配置信息；
- 反编译加载类；
- ....

需要注意的是，性能工具只是解决性能问题的手段，我们了解常用工具的一般用法即可，不要在工具学习上投入过多精力。

在通过工具得到异常指标，初步定位瓶颈点后，如何进一步进行确认和调优？这里将给出常见的一些调优分析思路，内容会按照 CPU、内存、网络、磁盘等进行组织。详情见：https://developer.aliyun.com/article/727625?spm=5176.8068049.0.0.7f0d6d19WJXuiS

10 问 10 答：你真的了解线程池吗？

作者：风楼

简介：《Java 开发手册》中强调，线程资源必须通过线程池提供，而创建线程池必须使用 ThreadPoolExecutor。手册主要强调利用线程池避免两个问题，一是线程过渡切换，二是避免请求过多时造成 OOM。但是如果参数配置错误，还是会引发上面的两个问题。所以本节我们主要是讨论 ThreadPoolExecutor 的一些技术细节，并且给出几个常用的最佳实践建议。

我在查找资料的过程中，发现有些问题存在争议。后面发现，一部分原因是因为不同 JDK 版本的现实是有差异的。因此，下面的分析是基于当下最常用的版本 JDK1.8，并且对于存在争议的问题，我们分析源码，源码才是最准确的。

一、corePoolSize=0 会怎么样

这是一个争议点。我发现大部分博文，不论是国内的还是国外的，都是这样回答这个问题的：

- 提交任务后，先判断当前池中线程数是否小于 corePoolSize，如果小于，则创建新线程执行这个任务。
- 否者，判断等待队列是否已满，如果没有满，则添加到等待队列。
- 否者，判断当前池中线程数是否大于 maximumPoolSize，如果大于则拒绝。
- 否者，创建一个新的线程执行这个任务。

按照上面的描述，如果 corePoolSize=0，则会判断等待队列的容量，如果还有容量，则排队，并且不会创建新的线程。

但其实，这是老版本的实现方式，从 1.6 之后，实现方式就变了。我们直接看 execute 的源码（submit 也依赖它），我备注出了关键一行：

```
int c = ctl.get();
            if (workerCountOf(c) < corePoolSize) {
                if (addWorker(command, true))
                    return;
                c = ctl.get();
            }
            if (isRunning(c) && workQueue.offer(command)) {
                int recheck = ctl.get();
                if (! isRunning(recheck) && remove(command))
                    reject(command);
                // 注意这一行代码，添加到等待队列成功后，判断当前池内线程数是否为 0，如果是则创建一个 firstTask 为 null 的 worker，这个 worker 会从等待队列中获取任务并执行。
                else if (workerCountOf(recheck) == 0)
                    addWorker(null, false);
            }
            else if (!addWorker(command, false))
                reject(command);
```

- 线程池提交任务后，首先判断当前池中线程数是否小于 corePoolSize。
- 如果小于则尝试创建新的线程执行该任务；否则尝试添加到等待队列。
- 如果添加队列成功，判断当前池内线程数是否为 0，如果是则创建一个 firstTask 为 null 的 worker，这个 worker 会从等待队列中获取任务并执行。
- 如果添加到等待队列失败，一般是队列已满，才会再尝试创建新的线程。
- 但在创建之前需要与 maximumPoolSize 比较，如果小于则创建成功。
- 否则执行拒绝策略。

答：

上述问题需区分 JDK 版本。在 1.6 版本之后，如果 corePoolSize=0，提交任务时如果线程池为空，则会立即创建一个线程来执行任务（先排队再获取）；如果提交任务的时候，线程池不为空，则先在等待队列中排队，只有队列满了才会创建新线程。

所以，优化在于，在队列没有满的这段时间内，会有一个线程在消费提交的任务；1.6 之前的实现是，必须等队列满了之后，才开始消费。

二、线程池创建之后，会立即创建核心线程么

之前有人问过我这个问题，因为他发现应用中有些 Bean 创建了线程池，但是这个 Bean 一般情况下用不到，所以咨询我是否需要把这个线程池注释掉，以减少应用运行时的线程数（该应用运行时线程过多。）

答：

不会。从上面的源码可以看出，在刚刚创建 ThreadPoolExecutor 的时候，线程并不会立即启动，而是要等到有任务提交时才会启动，除非调用了 prestartCoreThread/prestartAllCoreThreads 事先启动核心线程。

- prestartCoreThread：Starts a core thread, causing it to idly wait for work. This overrides the default policy of starting core threads only when new tasks are executed.

- prestartAllCoreThreads：Starts all core threads.

三、核心线程永远不会销毁么

这个问题有点 tricky。首先我们要明确一下概念，虽然在 JavaDoc 中也使用了“core/non-core threads”这样的描述，但其实这是一个动态的概念，JDK 并没有给一部分线程打上“core”的标记，做什么特殊化的处理。这个问题我认为想要探讨的是闲置线程终结策略的问题。

在 JDK1.6 之前，线程池会尽量保持 corePoolSize 个核心线程，即使这些线程闲置了很长时间。这一点曾被开发者诟病，所以从 JDK1.6 开始，提供了方法 allowsCoreThreadTimeOut，如果传参为 true，则允许闲置的核心线程被终止。

请注意这种策略和 corePoolSize=0 的区别。我总结的区别是：

- corePoolSize=0：在一般情况下只使用一个线程消费任务，只有当并发请求特别多、等待队列都满了之后，才开始用多线程。
- allowsCoreThreadTimeOut=true && corePoolSize>1：在一般情况下就开始使用多线程（corePoolSize 个），当并发请求特别多，等待队列都满了之后，继续加大线程数。但是当请求没有的时候，允许核心线程也终止。

所以 corePoolSize=0 的效果，基本等同于 allowsCoreThreadTimeOut=true && corePoolSize=1，但实现细节其实不同。

答：

在 JDK1.6 之后，如果 allowsCoreThreadTimeOut=true，核心线程也可以被终止。

四、如何保证线程不被销毁

首先我们要明确一下线程池模型。线程池有个内部类 Worker，它实现了 Runnable 接口，首先，它自己要 run 起来。然后它会在合适的时候获取我们提交的 Runnable 任务，然后调用任务的 run()接口。一个 Worker 不终止的话可以不断执行任务。

我们前面说的“线程池中的线程”，其实就是 Worker；等待队列中的元素，是我们提交的 Runnable 任务。

每一个 Worker 在创建出来的时候，会调用它本身的 run()方法，实现是 runWorker(this)，这个实现的核心是一个 while 循环，这个循环不结束，Worker 线程就不会终止，就是这个基本逻辑。

- 在这个 while 条件中，有个 getTask()方法是核心中的核心，它所做的事情就是从等待队列中取出任务来执行：
- 如果没有达到 corePoolSize，则创建的 Worker 在执行完它承接的任务后，会用 workQueue.take()取任务、注意，这个接口是阻塞接口，如果取不到任务，Worker 线程一直阻塞。

- 如果超过了 corePoolSize，或者 allowCoreThreadTimeOut，一个 Worker 在空闲了之后，会用 workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)取任务。注意，这个接口只阻塞等待 keepAliveTime 时间，超过这个时间返回 null，则 Worker 的 while 循环执行结束，则被终止了。

```
final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        Runnable task = w.firstTask;
        w.firstTask = null;
        w.unlock(); // allow interrupts
        boolean completedAbruptly = true;
        try {
            // 看这里，核心逻辑在这里
            while (task != null || (task = getTask()) != null) {
                w.lock();
                // If pool is stopping, ensure thread is interrupted;
                // if not, ensure thread is not interrupted.  This
                // requires a recheck in second case to deal with
                // shutdownNow race while clearing interrupt
                if ((runStateAtLeast(ctl.get(), STOP) ||
                        (Thread.interrupted() &&
                         runStateAtLeast(ctl.get(), STOP))) &&
                    !wt.isInterrupted())
                    wt.interrupt();
                try {
                    beforeExecute(wt, task);
                    Throwable thrown = null;
                    try {
```

```
                    task.run();
                } catch (RuntimeException x) {
                    thrown = x; throw x;
                } catch (Error x) {
                    thrown = x; throw x;
                } catch (Throwable x) {
                    thrown = x; throw new Error(x);
                } finally {
                    afterExecute(task, thrown);
                }
            } finally {
                task = null;
                w.completedTasks++;
                w.unlock();
            }
        }
        completedAbruptly = false;
    } finally {
        processWorkerExit(w, completedAbruptly);
    }
}
```

```
private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);
```

```
            // Check if queue empty only if necessary.
            if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
                decrementWorkerCount();
                return null;
            }

            int wc = workerCountOf(c);

            // Are workers subject to culling?
            boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

            if ((wc > maximumPoolSize || (timed && timedOut))
                && (wc > 1 || workQueue.isEmpty())) {
                if (compareAndDecrementWorkerCount(c))
                    return null;
                continue;
            }

            try {
                // 注意，核心中的核心在这里
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();
                if (r != null)
                    return r;
                timedOut = true;
            } catch (InterruptedException retry) {
                timedOut = false;
            }
        }
    }
```

答：

实现方式非常巧妙，核心线程（Worker）即使一直空闲也不终止，是通过 workQueue.take()实现的，它会一直阻塞到从等待队列中取到新的任务。非核心线程空闲指定时间后终止是通过 workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) 实现的，一个空闲的 Worker 只等待 keepAliveTime，如果还没有取到任务则循环终止，线程也就运行结束了。

引申思考

Worker 本身就是个线程，它再调用我们传入的 Runnable.run()，会启动一个子线程么？如果你还没有答案，再回想一下 Runnable 和 Thread 的关系。

五、空闲线程过多会有什么问题

笼统地回答是会占用内存，我们分析一下占用了哪些内存。首先，比较普通的一部分，一个线程的内存模型：

- 虚拟机栈
- 本地方法栈
- 程序计数器

我想额外强调是下面这几个内存占用，需要小心：

- ThreadLocal：业务代码是否使用了 ThreadLocal？就算没有，Spring 框架中也大量使用了 ThreadLocal，你所在公司的框架可能也是一样。
- 局部变量：线程处于阻塞状态，肯定还有栈帧没有出栈，栈帧中有局部变量表，凡是被局部变量表引用的内存都不能回收。所以如果这个线程创建了比较大的局部变量，那么这一部分内存无法 GC。
- TLAB 机制：如果你的应用线程数处于高位，那么新的线程初始化可能因为 Eden 没有足够的空间分配 TLAB 而触发 YoungGC。

答：

线程池保持空闲的核心线程是它的默认配置，一般来讲是没有问题的，因为它占用的内存一般不大。怕的就是业务代码中使用 ThreadLocal 缓存的数据过大又不清理。

如果你的应用线程数处于高位，那么需要观察一下 YoungGC 的情况，估算一下 Eden 大小是否足够。如果不够的话，可能要谨慎地创建新线程，并且让空闲的线程终止；必要的时候，可能需要对 JVM 进行调参。

六、keepAliveTime=0 会怎么样

这也是个争议点。有的博文说等于 0 表示空闲线程永远不会终止，有的说表示执行完立刻终止。还有的说等于-1 表示空闲线程永远不会终止。其实稍微看一下源码知道了，这里我直接抛出答案。

答：

在 JDK1.8 中，keepAliveTime=0 表示非核心线程执行完立刻终止。

默认情况下，keepAliveTime 小于 0，初始化的时候才会报错；但如果 allowsCoreThreadTimeOut，keepAliveTime 必须大于 0，不然初始化报错。

七、怎么进行异常处理

很多代码的写法，我们都习惯按照常见范式去编写，而没有去思考为什么。比如：

- 如果我们使用 execute()提交任务，我们一般要在 Runable 任务的代码加上 try-catch 进行异常处理。
- 如果我们使用 submit()提交任务，我们一般要在主线程中，对 Future.get()进行 try-catch 进行异常处理。

但是在上面，我提到过，submit()底层实现依赖 execute()，两者应该统一呀，为什么有差异呢？下面再扒一扒 submit()的源码，它的实现蛮有意思。

首先，ThreadPoolExecutor 中没有 submit 的代码，而是在它的父类 AbstractExecutorService 中，有三个 submit 的重载方法，代码非常简单，关键代码就两行：

```
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}
public <T> Future<T> submit(Runnable task, T result) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task, result);
    execute(ftask);
    return ftask;
}
public <T> Future<T> submit(Callable<T> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task);
    execute(ftask);
    return ftask;
}
```

正是因为这三个重载方法，都调用了 execute，所以我才说 submit 底层依赖 execute。通过查看这里 execute 的实现，我们不难发现，它就是 ThreadPoolExecutor 中的实现，所以，造成 submit 和 execute 的差异化的代码，不在这。那么造成差异的一定在 newTaskFor 方法中。这个方法也就 new 了一个 FutureTask 而已，FutureTask 实现 RunnableFuture 接口，RunnableFuture 接口继承 Runnable 接口和 Future 接口。而 Callable 只是 FutureTask 的一个成员变量。

所以讲到这里，就有另一个 Java 基础知识点：Callable 和 Future 的关系。我们一般用 Callable 编写任务代码，Future 是异步返回对象，通过它的 get 方法，阻塞式地获取结果。FutureTask 的核心代码就是实现了 Future 接口，也就是 get 方法的实现：

```
public V get() throws InterruptedException, ExecutionException {
    int s = state;
    if (s <= COMPLETING)
        // 核心代码
        s = awaitDone(false, 0L);
    return report(s);
}

private int awaitDone(boolean timed, long nanos)
    throws InterruptedException {
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    WaitNode q = null;
    boolean queued = false;
    // 死循环
    for (;;) {
        if (Thread.interrupted()) {
            removeWaiter(q);
            throw new InterruptedException();
        }

        int s = state;
        // 只有任务的状态是’已完成‘，才会跳出死循环
        if (s > COMPLETING) {
            if (q != null)
```

```
                    q.thread = null;
                return s;
            }
            else if (s == COMPLETING) // cannot time out yet
                Thread.yield();
            else if (q == null)
                q = new WaitNode();
            else if (!queued)
                queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                     q.next = waiters, q);
            else if (timed) {
                nanos = deadline - System.nanoTime();
                if (nanos <= 0L) {
                    removeWaiter(q);
                    return state;
                }
                LockSupport.parkNanos(this, nanos);
            }
            else
                LockSupport.park(this);
        }
    }
```

get 的核心实现是有个 awaitDone 方法，这是一个死循环，只有任务的状态是“已完成”，才会跳出死循环；否则会依赖 UNSAFE 包下的 LockSupport.park 原语进行阻塞，等待 LockSupport.unpark 信号量。而这个信号量只有当运行结束获得结果、或者出现异常的情况下，才会发出来。分别对应方法 set 和 setException。这就是异步执行、阻塞获取的原理，扯得有点远了。

回到最初我们的疑问，为什么 submit 之后，通过 get 方法可以获取到异常？原因是 FutureTask 有一个 Object 类型的 outcome 成员变量，用来记录执行结果。这个结果可以是传入的泛型，也可以是 Throwable 异常：

```
public void run() {
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                     null, Thread.currentThread()))
        return;
    try {
        Callable<V> c = callable;
        if (c != null && state == NEW) {
            V result;
            boolean ran;
            try {
                result = c.call();
                ran = true;
            } catch (Throwable ex) {
                result = null;
                ran = false;
                setException(ex);
            }
            if (ran)
                set(result);
        }
    } finally {
        // runner must be non-null until state is settled to
        // prevent concurrent calls to run()
```

```
                runner = null;
                // state must be re-read after nulling runner to prevent
                // leaked interrupts
                int s = state;
                if (s >= INTERRUPTING)
                    handlePossibleCancellationInterrupt(s);
            }
        }

    // get 方法中依赖的，报告执行结果
        private V report(int s) throws ExecutionException {
            Object x = outcome;
            if (s == NORMAL)
                return (V)x;
            if (s >= CANCELLED)
                throw new CancellationException();
            throw new ExecutionException((Throwable)x);
        }
```

FutureTask 的另一个巧妙的地方就是借用 RunnableAdapter 内部类，将 submit 的 Runnable 封装成 Callable。所以就算你 submit 的是 Runnable，一样可以用 get 获取到异常。

答：

- 不论是用 execute 还是 submit，都可以自己在业务代码上加 try-catch 进行异常处理。我一般喜欢使用这种方式，因为我喜欢对不同业务场景的异常进行差异化处理，

至少打不一样的日志吧。

- 如果是 execute，还可以自定义线程池，继承 ThreadPoolExecutor 并复写其 afterExecute(Runnable r, Throwable t)方法。
- 或者实现 Thread.UncaughtExceptionHandler 接口，实现 void uncaughtException(Thread t, Throwable e);方法，并将该 handler 传递给线程池的 ThreadFactory。
- 但是注意，afterExecute 和 UncaughtExceptionHandler 都不适用 submit。因为通过上面的 FutureTask.run()不难发现，它自己对 Throwable 进行了 try-catch，封装到了 outcome 属性，所以底层方法 execute 的 Worker 是拿不到异常信息的。

八、线程池需不需要关闭

答：

一般来讲，线程池的生命周期跟随服务的生命周期。如果一个服务（Service）停止服务了，那么需要调用 shutdown 方法进行关闭。所以 ExecutorService.shutdown 在 Java 以及一些中间件的源码中，是封装在 Service 的 shutdown 方法内的。

如果是 Server 端不重启就不停止提供服务，我认为是不需要特殊处理的。

九、shutdown 和 shutdownNow 的区别

答：

- shutdown => 平缓关闭，等待所有已添加到线程池中的任务执行完再关闭。
- shutdownNow => 立刻关闭，停止正在执行的任务，并返回队列中未执行的任务。

本来想分析一下两者的源码的，但是发现本文的篇幅已经过长了，源码也贴了不少。感兴趣的朋友自己看一下即可。

十、Spring 中有哪些和 ThreadPoolExecutor 类似的工具

答：

SimpleAsyncTaskExecutor	每次请求新开线程，没有最大线程数设置.不是真的线程池，这个类不重用线程，每次调用都会创建一个新的线程。
SyncTaskExecutor	不是异步的线程。同步可以用SyncTaskExecutor，但这个可以说不算一个线程池，因为还在原线程执行。这个类没有实现异步调用，只是一个同步操作。
ConcurrentTaskExecutor	Executor的适配类，不推荐使用。如果ThreadPoolTaskExecutor不满足要求时，才用考虑使用这个类。
SimpleThreadPoolTaskExecutor	监听Spring's lifecycle callbacks，并且可以和Quartz的Component兼容.是Quartz的SimpleThreadPool的类。线程池同时被quartz和非quartz使用，才需要使用此类。

这里我想着重强调的就是 SimpleAsyncTaskExecutor，Spring 中使用的@Async 注解，底层就是基于 SimpleAsyncTaskExecutor 去执行任务，只不过它不是线程池，而是

每次都新开一个线程。

另外想要强调的是 Executor 接口。Java 初学者容易想当然的以为 Executor 结尾的类就是一个线程池，而上面的都是反例。我们可以在 JDK 的 execute 方法上看到这个注释：

```
/**
* Executes the given command at some time in the future.  The command
* may execute in a new thread, in a pooled thread, or in the calling
* thread, at the discretion of the {@code Executor} implementation.
*/
```

所以，它的职责并不是提供一个线程池的接口，而是提供一个“将来执行命令”的接口。真正能代表线程池意义的，是 ThreadPoolExecutor 类，而不是 Executor 接口。

十一、最佳实践总结

- 【强制】使用 ThreadPoolExecutor 的构造函数声明线程池，避免使用 Executors 类的 newFixedThreadPool 和 newCachedThreadPool。
- 【强制】 创建线程或线程池时请指定有意义的线程名称，方便出错时回溯。即 thr eadFactory 参数要构造好。
- 【建议】建议不同类别的业务用不同的线程池。
- 【建议】CPU 密集型任务(N+1)：这种任务消耗的主要是 CPU 资源，可以将线程数设置为 N（CPU 核心数）+1，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原因导致的任务暂停而带来的影响。一旦任务暂停，CPU

就会处于空闲状态，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。

- 【建议】I/O 密集型任务(2N)：这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交出给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程，具体的计算方法是 2N。
- 【建议】workQueue 不要使用无界队列，尽量使用有界队列。避免大量任务等待，造成 OOM。
- 【建议】如果是资源紧张的应用，使用 allowsCoreThreadTimeOut 可以提高资源利用率。
- 【建议】虽然使用线程池有多种异常处理的方式，但在任务代码中，使用 try-catch 最通用，也能给不同任务的异常处理做精细化。
- 【建议】对于资源紧张的应用，如果担心线程池资源使用不当，可以利用 ThreadPoolExecutor 的 API 实现简单的监控，然后进行分析和优化。

线程池初始化示例：

```
        private static final ThreadPoolExecutor pool;
        static {
            ThreadFactory threadFactory = new ThreadFactoryBuilder().setNameFormat("po-detail-pool-%d").build();
            pool = new ThreadPoolExecutor(4, 8, 60L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<>(512),
                    threadFactory, new ThreadPoolExecutor.AbortPolicy());
            pool.allowCoreThreadTimeOut(true);
        }
```

- threadFactory：给出带业务语义的线程命名。
- corePoolSize：快速启动 4 个线程处理该业务，是足够的。
- maximumPoolSize：IO 密集型业务，我的服务器是 4C8G 的，所以 4*2=8。
- keepAliveTime：服务器资源紧张，让空闲的线程快速释放。
- pool.allowCoreThreadTimeOut(true)：也是为了在可以的时候，让线程释放，释放资源。
- workQueue：一个任务的执行时长在 100~300ms，业务高峰期 8 个线程，按照 10 s 超时（已经很高了）。10s 钟，8 个线程，可以处理 10 1000ms / 200ms 8 = 400 个任务左右，往上再取一点，512 已经很多了。
- handler：极端情况下，一些任务只能丢弃，保护服务端。

那些你不知道的 TCP 冷门知识！

作者：韩述

简介：最近在做数据库相关的事情，碰到了很多 TCP 相关的问题，新的场景新的挑战，有很多之前并没有掌握透彻的点，大大开了一把眼界，选了几个案例分享一下。

案例一：TCP 中并不是所有的 RST 都有效

（一）背景知识

在 TCP 协议中，包含 RST 标识位的包，用来异常的关闭连接。在 TCP 的设计中它是不可或缺的，发送 RST 段关闭连接时，不必等缓冲区的数据都发送出去，直接丢弃缓冲区中的数据。而接收端收到 RST 段后，也不必发送 ACK 来确认。

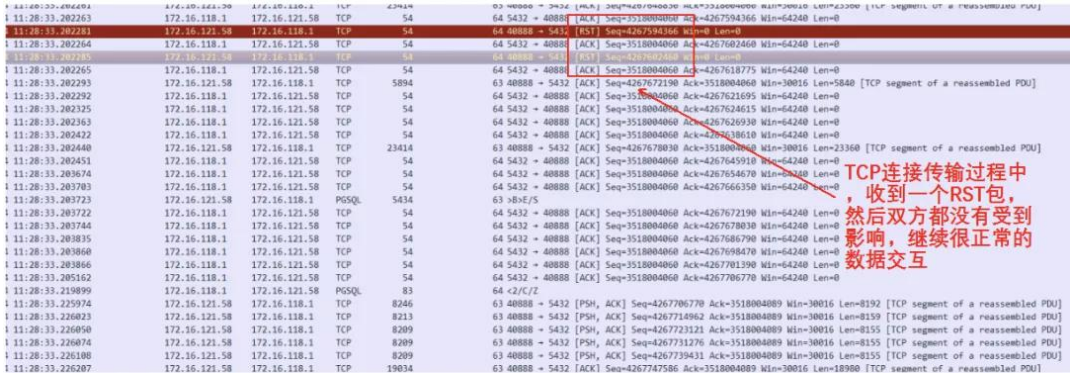
（二）问题现象

某客户连接数据库经常出现连接中断，但是经过反复排查，后端数据库实例排查没有执行异常或者 Crash 等问题，客户端 Connection reset 的堆栈如下图：

```
[2020-12-21 14:17:12.418] [ERROR] [DruidDataSource:handleFatalError:1844] [qtp362311125-5020] [-[discard connection]-]
org.postgresql.util.PSQLException: An I/O error occurred while sending to the backend.
        at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:333)
        at org.postgresql.jdbc.PgStatement.executeInternal(PgStatement.java:441)
        at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:365)
        at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags$original$z5DqnUcf(PgPreparedStatement.java:155)
        at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags$original$z5DqnUcf$accessor$dTivC8uv(PgPreparedStatement.java)
        at org.postgresql.jdbc.PgPreparedStatement$auxiliary$JNxxv8Y3.call(Unknown Source)
        at org.apache.skywalking.apm.agent.core.plugin.interceptor.enhance.InstMethodsInter.intercept(InstMethodsInter.java:93)
        at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags(PgPreparedStatement.java)
        at org.postgresql.jdbc.PgPreparedStatement.executeUpdate$original$z5DqnUcf(PgPreparedStatement.java:132)
        at org.postgresql.jdbc.PgPreparedStatement.executeUpdate$original$z5DqnUcf$accessor$dTivC8uv(PgPreparedStatement.java)
        at org.postgresql.jdbc.PgPreparedStatement$auxiliary$b2PgRAO1.call(Unknown Source)
        at org.apache.skywalking.apm.agent.core.plugin.interceptor.enhance.InstMethodsInter.intercept(InstMethodsInter.java:93)
        at org.postgresql.jdbc.PgPreparedStatement.executeUpdate(PgPreparedStatement.java)
        at com.alibaba.druid.filter.FilterChainImpl.preparedStatement_executeUpdate(FilterChainImpl.java:3253)
......#这里省去了无关紧要的线程栈
Caused by: java.net.SocketException: Connection reset
        at java.net.SocketInputStream.read(SocketInputStream.java:210)
        at java.net.SocketInputStream.read(SocketInputStream.java:141)
        at org.postgresql.core.VisibleBufferedInputStream.readMore(VisibleBufferedInputStream.java:140)
        at org.postgresql.core.VisibleBufferedInputStream.ensureBytes(VisibleBufferedInputStream.java:109)
        at org.postgresql.core.VisibleBufferedInputStream.read(VisibleBufferedInputStream.java:67)
        at org.postgresql.core.PGStream.receiveChar(PGStream.java:293)
        at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:1947)
        at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:306)
        ... 106 common frames omitted
```

经过复现及双端抓包的初步定位，找到了一个可疑点，TCP 交互的过程中客户端发了一个 RST（后经查明是客户端本地的一些安全相关 iptables 规则导致），但是神奇的

是，这个 RST 并没有影响 TCP 数据的交互，双方很愉快的无视了这个 RST，很开心的继续数据交互，然而 10s 钟之后，连接突然中断，参看如下抓包：



（三）关键点分析

从抓包现象看，在客户端发了一个 RST 之后，双方的 TCP 数据交互似乎没有受到任何影响，无论是数据传输还是 ACK 都很正常，在本轮数据交互结束后，TCP 连接又正常的空闲了一会，10s 之后连接突然被 RST 掉，这里就有两个有意思的问题了：

- TCP 数据交互过程中，在一方发了 RST 以后，连接一定会终止么？
- 连接会立即终止么，还是会等 10s？

查看一下 RFC 的官方解释：

```
Reset Processing

In all states except SYN-SENT, all reset (RST) segments are validated
by checking their SEQ-fields.  A reset is valid if its sequence number
is in the window.  In the SYN-SENT state (a RST received in response
to an initial SYN), the RST is acceptable if the ACK field
acknowledges the SYN.

The receiver of a RST first validates it, then changes state.  If the
receiver was in the LISTEN state, it ignores it.  If the receiver was
in SYN-RECEIVED state and had previously been in the LISTEN state,
then the receiver returns to the LISTEN state, otherwise the receiver
aborts the connection and goes to the CLOSED state.  If the receiver
was in any other state, it aborts the connection and advises the user
and goes to the CLOSED state.
```

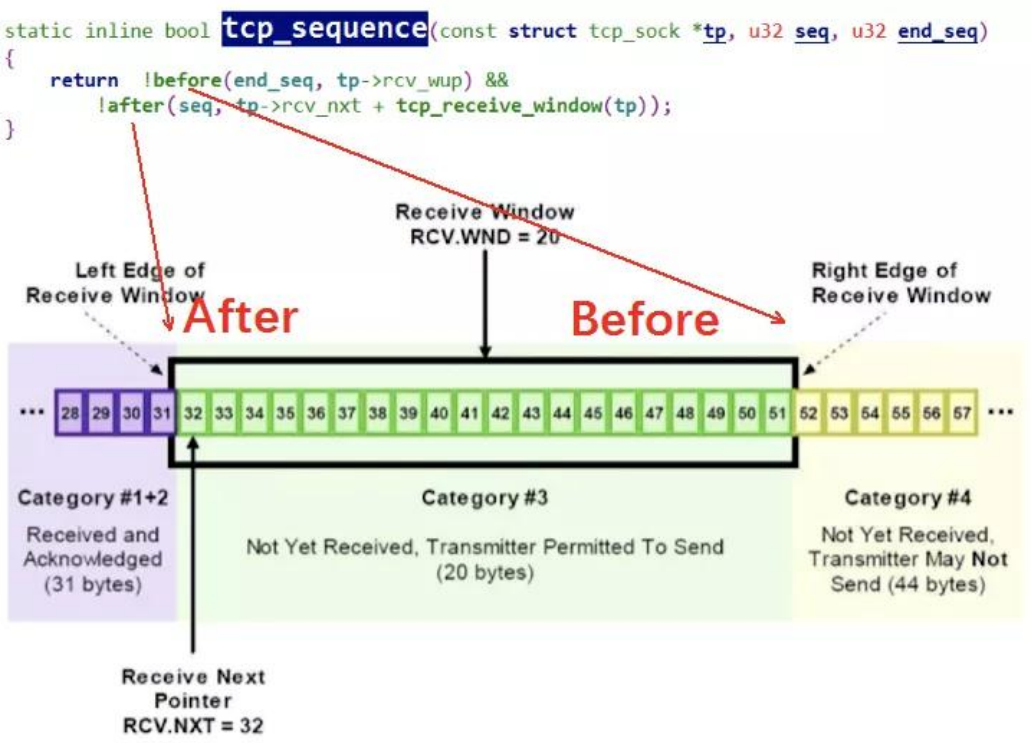
简单来说，就是 RST 包并不是一定有效的，除了在 TCP 握手阶段，其他情况下，RST 包的 Seq 号，都必须 in the window，这个 in the window 其实很难从字面理解，经过对 Linux 内核代码的辅助分析，确定了其含义实际就是指 TCP 的 —— 滑动窗口，准确说是滑动窗口中的接收窗口。

我们直接检查 Linux 内核源码，内核在收到一个 TCP 报文后进入如下处理逻辑：

```
static bool tcp_validate_incoming(struct sock *sk, struct sk_buff *skb,
                  const struct tcphdr *th, int syn_inerr)
{
    struct tcp_sock *tp = tcp_sk(sk);

    /* RFC1323: H1. Apply PAWS check first. */
    if (tcp_fast_parse_options(skb, th, tp) && tp->rx_opt.saw_tstamp &&
        tcp_paws_discard(sk, skb)) {
        if (!th->rst) {
            NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSESTABREJECTED);
            tcp_send_dupack(sk, skb);
            goto ↓discard;
        }
        /* Reset is accepted even if it did not pass PAWS. */
    }

    /* Step 1: check sequence number */
    if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq)) {
        /* RFC793, page 37: "In all states except SYN-SENT, all reset
         * (RST) segments are validated by checking their SEQ-fields."
         * And page 69: "If an incoming segment is not acceptable,
         * an acknowledgment should be sent in reply (unless the RST
         * bit is set, if so drop the segment and return)".
         */
        if (!th->rst) {
            if (th->syn)
                goto ↓syn_challenge;
            tcp_send_dupack(sk, skb);
        }
        goto ↓discard;
    }

    /* Step 2: check RST bit */
    if (th->rst) {
        /* RFC 5961 3.2 :
         * If sequence number exactly matches RCV.NXT, then
         *     RESET the connection
```

先判Seq合法性，再处理RST

下面是内核中关于如何确定 Seq 合法性的部分：



总结

Q：TCP 数据交互过程中，在一方发了 RST 以后，连接一定会终止么？

A：不一定会终止，需要看这个 RST 的 Seq 是否在接收方的接收窗口之内，如上例中就因为 Seq 号较小，所以不是一个合法的 RST 被 Linux 内核无视了。

Q：连接会立即终止么，还是会等 10s？

A：连接会立即终止，上面的例子中过了 10s 终止，正是因为，linux 内核对 RFC 严

格实现，无视了 RST 报文，但是客户端和数据库之间经过的 SLB（云负载均衡设备），却处理了 RST 报文，导致 10s（SLB 10s 后清理 session）之后关闭了 TCP 连接。

这个案例告诉我们，透彻的掌握底层知识，其实是很有用的，否则一旦遇到问题，（自证清白并指向 root cause）都不知道往哪个方向排查。

案例二：Linux 内核究竟有多少 TCP 端口可用

（一）背景知识

我们平时有一个常识，Linux 内核一共只有 65535 个端口号可用，也就意味着一台机器在不考虑多网卡的情况下最多只能开放 65535 个 TCP 端口。

但是经常看到有单机百万 TCP 连接，是如何做到的呢，这是因为，TCP 是采用四元组（Client 端 IP + Client 端 Port + Server 端 IP + Server 端 Port）作为 TCP 连接的唯一标识的。如果作为 TCP 的 Server 端，无论有多少 Client 端连接过来，本地只需要占用同一个端口号。而如果作为 TCP 的 Client 端，当连接的对端是同一个 IP + Port，那确实每一个连接需要占用一个本地端口，但如果连接的对端不是同一个 IP + Port，那么其实本地是可以复用端口的，所以实际上 Linux 中有效可用的端口是很多的（只要四元组不重复即可）。

（二）问题现象

作为一个分布式数据库，其中每个节点都是需要和其他每一个节点都建立一个 TCP

连接，用于数据的交换，那么假设有 100 个数据库节点，在每一个节点上就会需要 100 个 TCP 连接。当然由于是多进程模型，所以实际上是每个并发需要 100 个 TCP 连接。假如有 100 个并发，那就需要 1W 个 TCP 连接。但事实上 1W 个 TCP 连接也不算多，由之前介绍的背景知识我们可以得知，这远远不会达到 Linux 内核的瓶颈。

但是我们却经常遇到端口不够用的情况， 也就是“bind:Address already in use”：

```
18:39:46,800 [Thread-70] ERROR  jTPCCTData : ERROR: failed to acquire resources on one or more segments
  Detail: FATAL:  interconnect Error: Could not set up tcp listener socket
DETAIL:  bind: Address already in use
 (seg8 11.200.14.21:3025)
org.postgresql.util.PSQLException: ERROR: failed to acquire resources on one or more segments
  Detail: FATAL:  interconnect Error: Could not set up tcp listener socket
DETAIL:  bind: Address already in use
 (seg8 11.200.14.21:3025)
        at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2455)
        at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:2155)
        at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:288)
        at org.postgresql.jdbc.PgStatement.executeInternal(PgStatement.java:430)
        at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:356)
        at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags(PgPreparedStatement.java:168)
        at org.postgresql.jdbc.PgPreparedStatement.executeQuery(PgPreparedStatement.java:116)
        at jTPCCTData.executeStockLevel(jTPCCTData.java:1379)
        at jTPCCTData.execute(jTPCCTData.java:107)
        at jTPCCTerminal.executeTransactions(jTPCCTerminal.java:182)
        at jTPCCTerminal.run(jTPCCTerminal.java:88)
        at java.lang.Thread.run(Thread.java:756)
18:39:46,822 [Thread-181] ERROR  jTPCCTData : Unexpected SQLException in STOCK_LEVEL
18:39:46,822 [Thread-181] ERROR  jTPCCTData : ERROR: failed to acquire resources on one or more segments
  Detail: FATAL:  interconnect Error: Could not set up tcp listener socket
DETAIL:  bind: Address already in use
 (seg6 11.196.60.97:3029)
org.postgresql.util.PSQLException: ERROR: failed to acquire resources on one or more segments
  Detail: FATAL:  interconnect Error: Could not set up tcp listener socket
DETAIL:  bind: Address already in use
```

其实看到这里，很多同学已经在猜测问题的关键点了，经典的 TCP time_wait 问题呗，关于 TCP 的 time_wait 的背景介绍以及应对方法不是本文的重点就不赘述了，可以自行了解。乍一看，系统中有 50W 的 time_wait 连接，才 65535 的端口号，必然不可用：

```
#ss -ta |wc -l ; ss -tan state time-wait | wc -l
502420
500021
```

但是这个猜测是错误的！因为系统参数 net.ipv4.tcp_tw_reuse 早就已经被打开了，所以不会由于 time_wait 问题导致上述现象发生，理论上说在开启 net.ipv4.cp_tw_reuse 的情况下，只要对端 IP + Port 不重复，可用的端口是很多的，因为每一个对端 IP + Port 都有 65535 个可用端口：

```
#cat /proc/sys/net/ipv4/tcp_tw_reuse
1
```

（三）问题分析

- Linux 中究竟有多少个端口是可以被使用？
- 为什么在 tcp_tw_reuse 情况下，端口依然不够用？

Linux 有多少端口可以被有效使用？

理论来说，端口号是 16 位整型，一共有 65535 个端口可以被使用，但是 Linux 操作系统有一个系统参数，用来控制端口号的分配：

```
net.ipv4.ip_local_port_range
```

我们知道，在写网络应用程序的时候，有两种使用端口的方式：

- 方式一：显式指定端口号 —— 通过 bind() 系统调用，显式的指定 bind 一个端口号，比如 bind(8080) 然后再执行 listen() 或者 connect() 等系统调用时，会使用应用程序在 bind() 中指定的端口号。
- 方式二：系统自动分配 —— bind() 系统调用参数传 0 即 bind(0) 然后执行 listen()。或者不调用 bind()，直接 connect()，此时是由 Linux 内核随机分配一个端口号，Linux 内核会在 net.ipv4.ip_local_port_range 系统参数指定的范围内，随机分配一个没有被占用的端口。

例如如下情况，相当于 1-20000 是系统保留端口号（除非按方法一显式指定端口号），自动分配的时候，只会从 20000 - 65535 之间随机选择一个端口，而不会使用小于 20000 的端口：

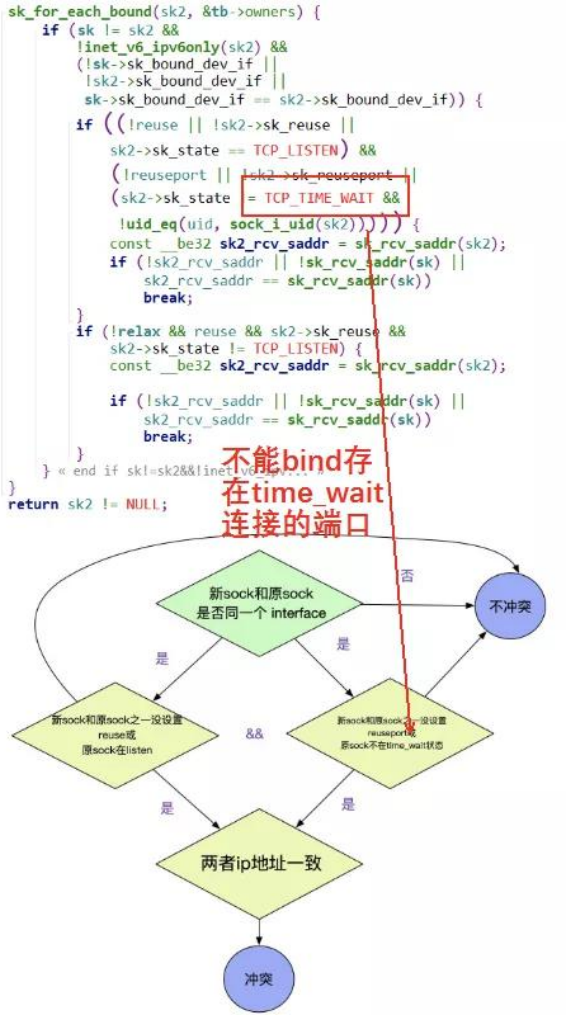
```
$cat /proc/sys/net/ipv4/ip_local_port_range
20000   65535
```

为什么在 tcp_tw_reuse=1 情况下，端口依然不够用？

细心的同学可能已经发现了，报错信息全部都是 bind() 这个系统调用失败，而没有一个是 connect() 失败。在我们的数据库分布式节点中，所有 connect() 调用（即作为 TCP client 端）都成功了，但是作为 TCP server 的 bind(0) + listen() 操作却有很多没成功，报错信息是端口不足。

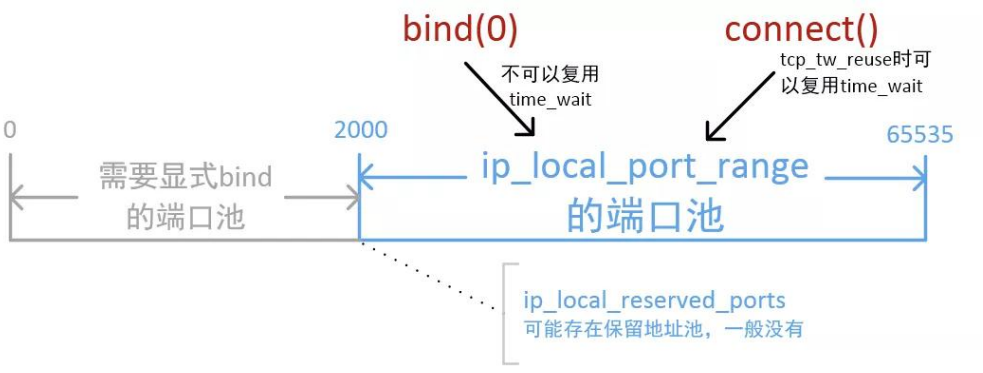
由于我们在源码中，使用了 bind(0) + listen() 的方式（而不是 bind 某一个固定端口），即由操作系统随机选择监听端口号，问题的根因，正是这里。connect() 调用依然能从 net.ipv4.ip_local_port_range 池子里捞出端口来，但是 bind(0) 却不行了。为什么，因为两个看似行为相似的系统调用，底层的实现行为却是不一样的。

源码之前，了无秘密：bind() 系统调用在进行随机端口选择时，判断是否可用是走的 inet_csk_bind_conflict ，其中排除了存在 time_wait 状态连接的端口：



而 connect() 系统调用在进行随机端口的选择时，是走 __inet_check_established 判断可用性的，其中不但允许复用存在 TIME_WAIT 连接的端口，还针对存在 TIME_WAIT 的连接的端口进行了如下判断比较，以确定是否可以复用：

一张图总结一下：



于是答案就明了了，bind(0) 和 connect()冲突了，ip_local_port_range 的池子里被 50W 个 connect() 遗留的 time_wait 占满了，导致 bind(0) 失败。知道了原因，修复方案就比较简单了，将 bind(0) 改为 bind 指定 port，然后在应用层自己维护一个池子，每次从池子中随机地分配即可。

（四）总结

Q：Linux 中究竟有多少个端口是可以被有效使用的?

A：Linux 一共有 65535 个端口可用，其中 ip_local_port_range 范围内的可以被系统随机分配，其他需要指定绑定使用，同一个端口只要 TCP 连接四元组不完全相同可以无限复用。

Q：什么在 tcp_tw_reuse=1 情况下，端口依然不够用？

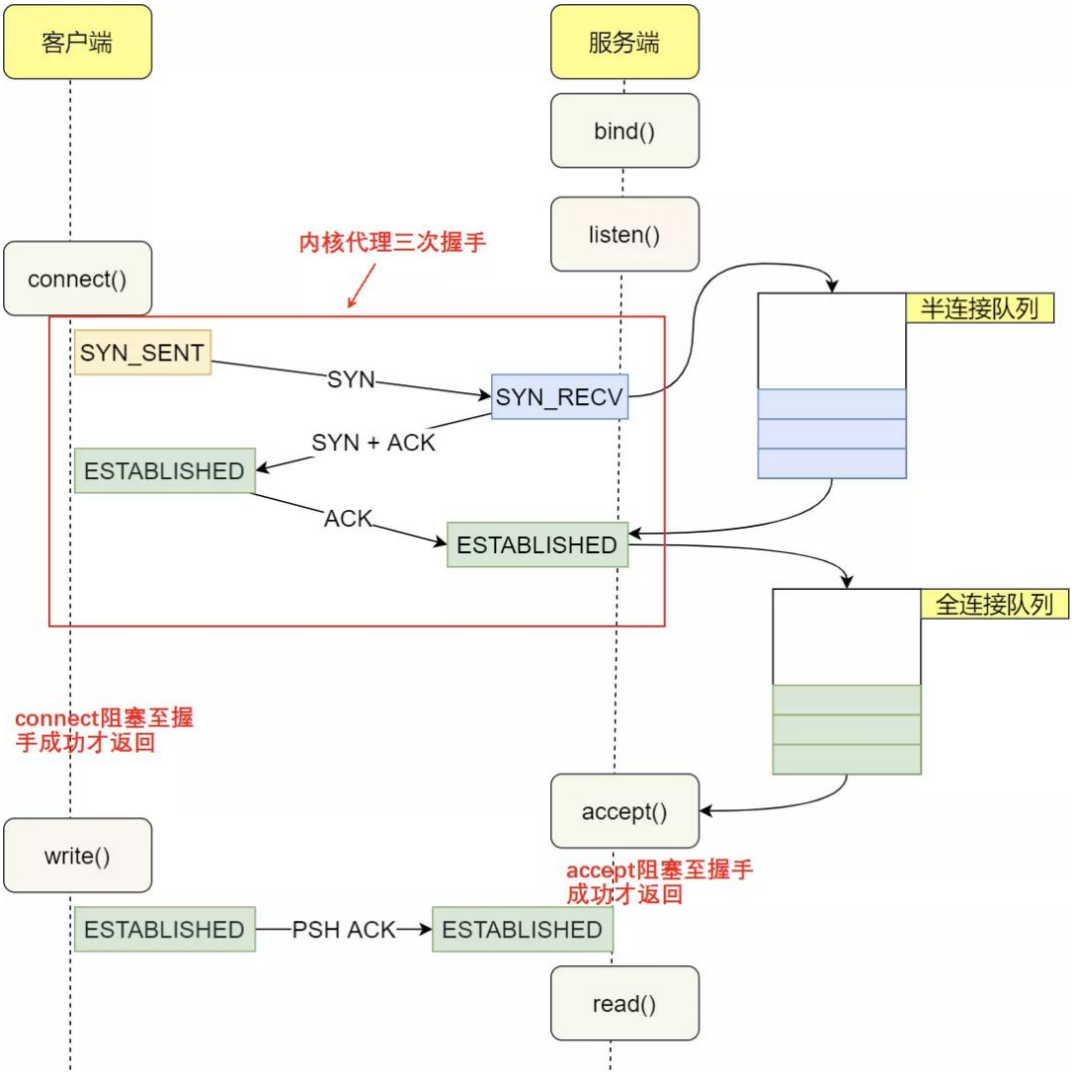
A：connect() 系统调用和 bind(0) 系统调用在随机绑定端口的时候选择限制不同，bind(0) 会忽略存在 time_wait 连接的端口。

这个案例告诉我们，如果对某一个知识点比如 time_wait，比如 Linux 究竟有多少 Port 可用知道一点，但是只是一知半解，就很容易陷入思维陷阱，忽略真正的 Root Case，要掌握就要透彻。

案例三：诡异的幽灵连接

（一）背景知识

TCP 三次握手，SYN、SYN-ACK、ACK 是所有人耳熟能详的常识，但是具体到 Socket 代码层面，是如何和三次握手的过程对应的，恐怕就不是那么了解了，可以看一下如下图，理解一下（图源：小林 coding）：



这个过程的关键点是，在 Linux 中，一般情况下都是内核代理三次握手的，也就是说，当你 client 端调用 connect() 之后内核负责发送 SYN，接收 SYN-ACK，发送 ACK。然后 connect() 系统调用才会返回，客户端侧握手成功。

而服务端的 Linux 内核会在收到 SYN 之后负责回复 SYN-ACK 再等待 ACK 之后才会让 accept() 返回，从而完成服务端侧握手。于是 Linux 内核就需要引入半连接队列（用

于存放收到 SYN，但还没收到 ACK 的连接）和全连接队列（用于存放已经完成 3 次握手，但是应用层代码还没有完成 accept() 的连接）两个概念，用于存放在握手中的连接。

（二）问题现象

我们的分布式数据库在初始化阶段，每两个节点之间两两建立 TCP 连接，为后续数据传输做准备。但是在节点数比较多时，比如 320 节点的情况下，很容易出现初始化阶段卡死，经过代码追踪，卡死的原因是，发起 TCP 握手侧已经成功完成的了 connect() 动作，认为 TCP 已建立成功，但是 TCP 对端却没有握手成功，还在等待对方建立 TCP 连接，从而整个集群一直没有完成初始化。

（三）关键点分析

看过之前的背景介绍，聪明的小伙伴一定会好奇，假如我们上层的 accpet() 调用没有那么及时（应用层压力大，上层代码在干别的），那么全连接队列是有可能会满的，满的情况会是如何效果，我们下面就重点看一下全连接队列满的时候会发生什么。

当全连接队列满时，connect() 和 accept() 侧是什么表现行为？

实践是检验真理的最好途径

我们直接上测试程序。

client.c：

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>
#define MAXLINE 4096

int main(int argc, char** argv)
{
    int sockfd, n;
    char recvline[4096], sendline[4096];
    struct sockaddr_in servaddr;

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(6666);
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);

    for (n = 0; n < 100; n++)
    {
        if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
            printf("create socket error: %s(errno: %d)\n",
                strerror(errno),errno);
            return 0;
        }

        //客户端不停的向服务端发起新连接，成功之后继续发，没成功会阻塞在这里
        if(connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0)
        {
            printf("connect error: %s(errno: %d)\n",strerror(errno),errno);
            return 0;
        }

        printf("connected to server: %d\n", n);

        close(sockfd);
    }

    return 0;
}
```

server.c：

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>

#define MAXLINE 4096

int main(int argc, char* argv[])
{
    int listenfd, connfd;
    struct sockaddr_in servaddr;
    char buff[4096];
    int  n;

    if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 )
    {
        printf("create socket error: %s(errno: %d)\n", strerror(errno), errno);
        return 0;
    }

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(6666);

    if(bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1)
    {
        printf("bind socket error: %s(errno: %d)\n", strerror(errno), errno);
        return 1;
    }

    if(listen(listenfd, 10) == -1)
    {
        printf("listen socket error: %s(errno: %d)\n", strerror(errno), errno);
        return 2;
    }

    if( (connfd = accept(listenfd, (struct sockaddr*)NULL, NULL)) == -1)
    {
        printf("accept socket error: %s(errno: %d)", strerror(errno), errno);
        return 3;
    }

    printf("accepet a socket\n");

    //服务端仅accept一次，之后就不再accept，此时全连接队列会被堆满
    sleep(1000);

    close(connfd);
    close(listenfd);
    return 0;
}
```

通过执行上述代码，我们观察 Linux 3.10 版本内核在全连接队列满的情况下的现象。神奇的事情发生了，服务端全连接队列已满，该连接被丢掉，但是客户端 connect() 系统调用却已经返回成功，客户端以为这个 TCP 连接握手成功了，但是服务端却不知道，这个连接犹如幽灵一般存在了一瞬又消失了：

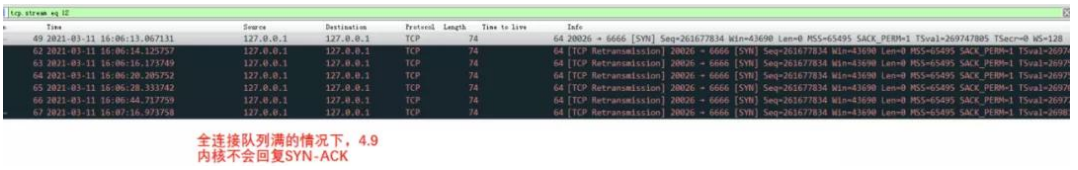
这个问题对应的抓包如下：

正如问题中所述的现象，在一个 320 个节点的集群中，总会有个别节点，明明 connect() 返回成功了，但是对端却没有成功，因为 3.10 内核在全连接队列满的情况下，会先回复 SYN-ACK，然后移进全连接队列时才发现满了于是丢弃连接，这样从客户端看来 TCP 连接成功了，但是服务端却什么都不知道。

Linux 4.9 版本内核在全连接队列满时的行为

在 4.9 内核中，对于全连接队列满的处理，就不一样，connect() 系统调用不会成功，一直阻塞，也就是说能够避免幽灵连接的产生：

抓包报文交互如下，可以看到 Server 端没有回复 SYN-ACK，客户端一直在重传 SYN：



事实上，在刚遇到这个问题的时候，我第一时间就怀疑到了全连接队列满的情况，但是悲剧的是看的源码是 Linux 3.10 的，而随手找的一个本地日常测试的 ECS 却刚好是 Linux 4.9 内核的，导致写了个 demo 测试例子却死活没有复现问题。排除了所有其他原因，再次绕回来的时候已经是一周之后了（这是一个悲伤的故事）。

（四）总结

Q：当全连接队列满时，connect() 和 accept() 侧是什么表现行为？

A：Linux 3.10 内核和新版本内核行为不一致，如果在 Linux 3.10 内核，会出现客户端假连接成功的问题，Linux 4.9 内核就不会出现问题。

这个案例告诉我们，实践是检验真理的最好方式，但是实践的时候也一定要睁大眼睛看清楚环境差异，如 Linux 内核这般稳定的东西，也不是一成不变的。唯一不变的是变化，也许你也是可以来数据库内核玩玩底层技术的。

如何准备阿里技术面试？终面官现身说法！

作者：春招火热进行中的

春暖花开的季节，阿里巴巴的春招面试正如火如荼地进行着。相信同学们也在面试这块做了许多准备，那么，参加阿里的面试需要注意些什么？今天，我们特别邀请到资深终面官永叔给同学们送上最实用的面试秘籍。

嘉宾简介：永叔，资深算法专家，5 届校招终面官，历任职于淘宝、阿里妈妈、搜索、优酷等多个部门，主要研究方向为大规模分布式机器学习算法，多模态交互搜索推荐系统，算法博弈论。

Q：面试官看简历，最关注哪些部分？

A：导师和实验室、研究方向都会关注，还有他的论文，我们都会提前去下载来看，并提前准备问题。其他信息我们会也看看有没有加分项。细节上，邮箱的名字能看出来有些同学很重视细节。TIPS：简历第一页一定要把最关键的信息写上，简历篇幅不要太长。

Q：面试过这么多同学，您对同学们有什么面试忠告？

A：面试的基本要点很多，很多同学容易犯的一些小问题，我总结几个点分享给大家：

- 一定不要迟到，这是起码的尊重。对面试官也是这样的要求。
- 对简历内容要有准备。对自己的突出点，包括技术/个性上的亮点，要练习表达，避免临时组织语言。
- 尊重事实，如实回答。每个同学擅长点不同，面试官风格也不同，问到自己不清楚的地方，请不要试图去强掰，实事求是回答就好。
- 心态放平。碰到压力面试的时候，不要试图去挑起 PK 的氛围。在面试过程中，面试官的最终目标是希望帮助面试同学，找到问题最优解。做好自己，平时多加练习。

Q：面试到底面哪些维度？同学们需要怎么准备？

A：阿里的用人理念是非凡人、平常心、做非凡事。我们期待的人员特质是聪明、乐观、皮实、自省。下面用技术类的同学举个例子：

计算机类的同学，我们更关注基础。公司有一套完整的流程体系去培养一个工程师，面试更多是基础素质的考察，比如概率、矩阵等。不用特意准备，但基础的知识还是需要去复习的。

此外，我们更多看的是你思考的路径，思考的工具和方法，你应对问题的反应如何，过程中融入一些软技能的考察。当然，我们还会考察学生的潜力。自己的项目经历，论文等，这一块的考察我们首先会确认真实性，更多的会关注细节。千万不要把别人的项目写到自己的项目里面，一旦发现后果很严重。

另外，Coding 能力是必须的，建议一定要练习，并且我们有速度要求。面试官面试过程中会要求在线写代码，实时同步过程。主要考察编码风格、准确性、熟练程度。毕竟没有哪个面试官会用一个在工作技能上无法和团队合作开展工作的人。

Q：学历是不是招聘门槛？

A：我们只以能力论英雄，不会看学历。这几年的面试过程中，我们发现一个现象：很多优秀的同学，在日常学习中会主动去了解、重视企业需要的能力项，并不断通过自学习去拉近自己与目标值的距离。这个是非常好的趋势。

Q：哪些因素会导致同学面试不通过？

A： 我们不会因为学生某个能力不足就 PASS 掉。面试是一个逐步肯定的过程，不是一个否定的过程。

一般面试官最后都会问，你有什么问题要问我的吗？其实面试官希望知道的是同学对职位/部门/公司是否有了解，基于你了解的信息你有什么样的问题，如何在这个环节提出高质量的问题，也反应出来你对这个面试是否重视。但很遗憾，这个环节能回答好的同学并不多。最后这个提问机会是给到他展示自己综合能力的机会，但很多人却把这当做面试结果“comfirm”的环节。还有同学拿问题去考面试官，或者纠结于某个问题应该怎么去回答，完全忘记了面试是展示自己，不是学习交流，也不是 PK。

Q：如果碰到特殊情况，同学该怎么做？

A：整个面试的节奏是面试官和同学共同营造的，所以需要有明显的节奏感。有些问题同学确实任何想法都没有的，或者觉得题意不清晰，一定要确认清楚，请求面试官的确认。现场状态不 OK 的情况下，一定要说 NO，不用去迎合面试官，比如电话质量很差。我们要的是展示自己，面试官做的是协助同学展示最好的自己。

Q：说说您面试过哪个同学，让您印象最深刻？

A：2018 年，我面试过一个同学，目前他已经入职我的团队，面试的时候他提到自己的生活习惯，对管理自己的时间还是很有想法的，自律程度让人吃惊。入职之后，这个同学他也是这么做的，每天早上 6-9 点是他的学习时间。每天早上 6 点钟-8 点在家看书，8 点到公司后，继续再看 1 个小时的书，风雨无阻。他学习结束，其他人才刚到公司。工作之余，他还投入在工作论文的发表上，刚投出去 2 篇。业务上，他现在是我们团队的骨干力量。从对他的观察里面，不难发现，优秀的同学基本都来自于高度的自律。

Q：您经历过的最糟糕的面试是什么？

A：我曾经面试过一类候选人，2 分钟的自我介绍候选人滔滔不绝，没有重点，所有问题都会发散，整个面试的过程一直在 show off 自己，但没有 get 到问题的关键点。整个面试体验是非常糟糕的。

Q：很多学生困惑于该怎么选择岗位，您有什么建议呢？

A：尽可能了解这个岗位未来的职业发展是什么样的，是否具备岗位的敲门砖。阿里不同部门内推信息网上有很多，可以找相关的师兄去问。同时，可以去做一些测评，看看自己是否对自我有足够的认知。

一手面经，稳拿 Offer

我是一名应届生,我觉得拿到心仪的 offer 不难

作者：开发者小助手

简介：作者简介——阿里巴巴 nacos 项目管理委员会成员 阿里巴巴 spring-cloud-alibaba 项目提交者 阿里巴巴 nacos-spring-project 项目维护者 阿里巴巴 nacos-springboot-project 项目维护者 spring-cloud/spring-cloud-sleuth 项目贡献者 阿里巴巴云原生日讲师 2019 年第一季阿里巴巴编程之夏学员。

一、起源

不知不觉大学四年时光就过去了，而我，不仅仅是一名应届毕业生，同时，也是一开源项目的 PMC。

与开源结缘是在大三上的时候吧，那个时候是由于与同学承接了一个商业外包，因此使用一个 WxJava 的开源项目，在项目交接之后，我打算用 golang 重新翻译此项目，因此去学习了一下，在学习期间发现了几个小问题以及贡献了一个优化，算正式与开源结缘了吧。

真正完全参与开源，是从大三下开始，那个时候经常和学长去参与各种技术讲座，比如 Flink、Apache APISIX 网关的宣讲、Service Mesh、分布式 DB、服务治理等等。

然后在四月的一天，学长给我了一个社区群，是有关服务治理的开源项目，而我正好想从理论到玩具的学习方式，转为理论到实践的，将所学真真切切应用在实际当中，从这一天开始，真真切切的开始投入到开源当中。

二、前进

参与开源，其过程就好比 RPG 游戏一般，一路升级打怪，从最开始的在 SDK 侧新增简单的增删改查功能，到参与维护两个 spring 生态组建的维护。这期间，重新学习了 spring 内部的原理。对于 Spring 的整个设计理解，又更近了一步，能够更加灵活的运用 Spring 提供的各种钩子去实现用户对于组件的需求。

期间比较自豪的事情，是发现了 spring-cloud-seluth 的 bug，并提交 PR 进行了 fix，其实发现这个问题的路途，比较曲折，最开始是有用户反馈 zipkin 无法与服务治理中心进行整合，于是我根据带我的 PMC 提供的资料，去 zipkin 社区以及他的源码研究了一下，发现 zipkin 从某个版本开始，他们自己写了一个 webserver，因此无法使用 spring 相关的能力将 zipkin-server 注册到服务治理中心，因此我进行了一个简单的测试，将注册时机进行了简单的调整。

但是由于过于定制化，因此没有进行回馈（其实问题的根本原因倒不是这个），只是将方案告诉给有此问题的相关用户。后面再持续跟进此问题时，发现仍然有 zipkin 与服务治理中心存在整合问题，但是这次是客户端，因此进行长时间的问题跟踪调试，最终确定问题的原因，然后进行反馈，提交 PR 进行修复。

这一次的经历，使得我对于问题的解决，不再是只会埋头谷歌或者百度，而是从问题本身出发，去跟踪、观察问题，并成功解决。

三、突破

有了上面一次的经历，使得我更加有信心参与开源，接着，我从客户端转战服务端，真正切入服务治理中心的核心。

而此时，我已经成为 commiter 了。因此为了能够更好的参与项目，同时符合 commiter 的身份，我重新开始学习项目的源码、设计，纠正了许多之前第一次看源码时出现的理解误区，对于某些功能模块代码的设计有了更深的理解。同时，高可用的思想也在源码中穿插着，使得我后面在实习中，参与项目的改造时，思考了更多的东西。

成为 commiter 之后，不知道是不是初生牛犊不怕虎，我接受了内核模块的重构以及去 MySQL 依赖这两个艰巨的任务。其内核重构设计了一致性协议层的抽象设计、寻址模式的统一、事件机制的统一，其中，最难的莫过于一致性协议层的抽象以及设计了。

其实，但是对于一致性协议了解的不是很多，只是知道 CAP、BASE 理论而已，因此，接过任务之后，开始各种开源项目源码的探究，比如 JRaft、Etcd、Memberlist、hashicorp/raft 等等，同时下载了各类的电子 PDF 进行学习，为我后面的工作打下了一定的理论基础。

四、探索

待秋招以及实习结束之后，我正式开始了相关任务的工作，设计文档编写、基础理论支持、相关项目设计学习、代码编写，其实就是一个需求，从成立到最终产出的全过程，其综合性挺强的，这个时候的代码设计不再是随心所欲了，将一个单机的关系型存储变为一个分布式强一致性的关系型存储，其必须保证数据的一致性、事务的 ACID 性质，需要结合大量的资料以及前人项目的设计进行参考，当时提出的思路方案，就有四五种，其中，为了从数据库内部解决这个问题，还去学习一下 apache derby 的源码——插入一条数据的流程是怎么样的以及他的 master-slave 机制的实现，可以说，通过这些的前期准备以及与其他大佬们的交流，使得我后面的代码编写更加游刃有余。

五、感想

其实对于应届生的我，参与开源项目并且成为 commiter，也算是我的一项优势吧，也正因如此，我在秋招的时候基本是面试一家收获一家公司的 offer，其中也不乏 SP。

参与开源项目，是一个将理论付诸于生产实践的有效途径，它让你需要考虑各种因素，比如接口设计、新老版本的数据兼容、可扩展性、边界因素的思考等等，同时还会使得自己知识面的横向以及纵向的延伸；不仅如此，参与开源，你需要和世界不同的开发者进行思想的碰撞交流，有时候通过交流，能够使得自己对于自己的设计有更深的认识，发现设计上的不足，同时也锻炼了自己口述、文字的能力。

虽然自己没几天就要去某大厂工作了，但是还是希望自己能够保证工作质量同时深入学习工作方向内容的空闲时间，保持对开源参与的热情，从开源中学习，并将自己学习的知识回馈当中。

十年前，他如何自学技术进阿里？

作者：玄惭

简介：阿里云高级 DBA 专家玄惭，讲述十年前通过校招加入阿里的经历和心得，希望对大家有所帮助

一、准备工作

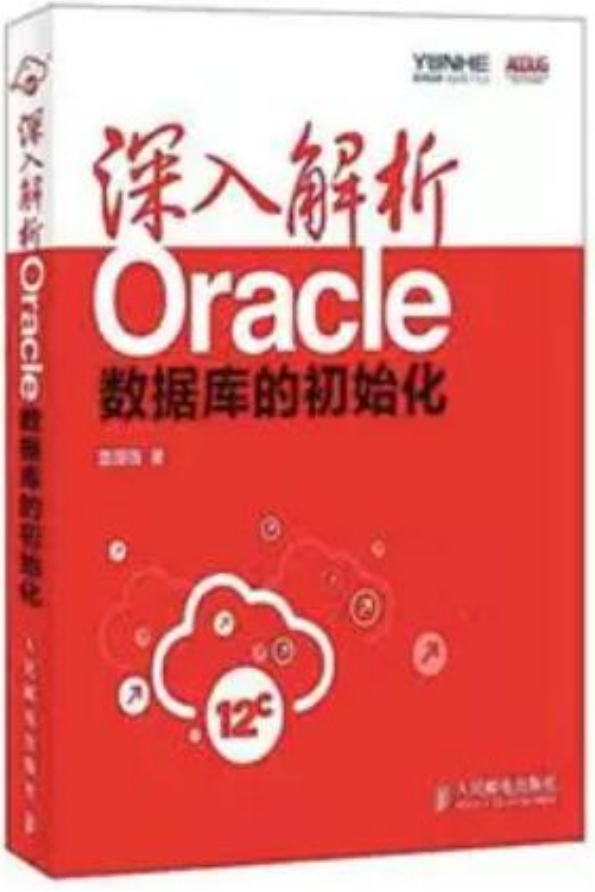
一年一度的校园实习招聘开始了，最近接触了几个找工作的应届生同学。这让我回想当年找工作的时候，遇到了很多好心人，所以一直想写写如何加入阿里的文章，算是对自己有一个交代，也希望能够帮助到找工作的同学。

（一）序：一颗种子的种下

我的母校是四川师范大学，专业是教育技术，在大一下期的一堂专业课上网站设计，我的专业课老师在讲网站开发过程中使用数据库的时候，介绍了这个数据库的管理者 DBA，在当今是属于比较稀缺的技术人员，他们随着经验的不断增加，所获得的报酬也将会越来越大。在当时很多学计算机的人都觉得做程序员是一门年青饭，所以我一下子被打动了，在心里暗暗就下定决心我毕业后就要做一位出色的 DBA，专业老师的不经意一句话，就在我内心中种下了一颗种子，等待着时间发芽成长。

（二）暑期自学数据库

有了这样一个想法之后，暑假里我在图书馆里借几本数据库原理这本书，打算在暑假的时候开始自学数据库，但其实回想起来这些书都应该没有看懂。到了大二，开始到图书馆中去借各种各样的数据库技术书籍，2007 年的时候 Oracle 还是非常流行的数据库，所以自然想成为一名 Oracle DBA，依然还记得最早 Oracle 入门书籍看的是 eygle 盖国强写的书，他坚持不懈的撰写 Oracle 相关的技术文章，让当时一大批 DBA 爱好者受益匪浅。



书看完后，心中会有很多的疑问，一遍看不懂，再看一篇，再不懂，再看一遍，这是我的学习方法。同时我也会自己搭建环境自己进行测试验证，再不懂就到论坛中去提问，最后将问题总结下来写成 Blog。

当时中国最大的 Oracle 技术论坛 ITPUB 云集了国内众多的高手，从论坛中看高手的回答，往往一针见血，认识了很多人，了解了他们的成长轨迹，让人觉得非常佩服。

在大学里学到了一门独特的技能就是通过互联网搜索我想了解的知识。我从互联网上搜寻各种 DBA 相关的资源，搜索到了让我坚定走入 DBA 之门的关键钥匙——《阿里 DBA 成长之路》，后来这篇文章的作者成为了我的一面技术官——我进入阿里后的第一任主管丹臣，每当重新读起他那篇文章，总是能够让人重新燃起成为一名优秀 DBA 的希望。

（三）中国最好的 DBA 技术团队在阿里

当时中国最好的 DBA 技术人才都在阿里，业界出名的 DBA 有冯春培（孔丘），陈吉平（托雷），宁海远（江枫），HelloDBA(张瑞)都在阿里，内心中升起了加入淘宝 DBA 团队的决心，我每天都会关注这些技术牛人和技术团队的 blog，通过 google reader 订阅他们的文章，了解业界最新的技术动态。

我保存了一篇当年淘宝 DBA 团队的 blog-淘宝 DBA 语录，节选如下：

- 作为我们的团队成员，我并不一定需要他有高超的技术（有当然更好），但是，我们的团队成员必须有强烈的责任心，有很强的团队合作能力。
- 我们拥有中国一流的环境，我们拥有中国一流的技术，我们的目标是打造中国一流的团队。
- DBA 未必是一个高薪的职业，但绝对是一个高压力的职业。
- 在遇到问题没有搞清楚具体原因之前，千万不要轻易重启数据库。
- 操作有风险，下手须慎重。
- 在淘宝这样高速发展的公司里，每一天都是一个挑战。我们都不得不去面对一些新的问题，我们唯有不断去提高自己，提供一些新的解决方案，to handle these problems。

每当看到这篇文章的时候，没有一次不会升起对这支团队的敬仰和羡慕，我下定决心立志加入这支团队。

（四）从理论到实践 初涉企业级数据库架构

通过近两年的技术理论自学，对 Oracle 数据库理论有了一定的理解，但是理论归理论，实践归实践，我想验证我所学习和理解的 Oracle 技术，所以我决定参加了社会上的 Oracle 培训，培训实践选择再大三下期，老师是具有多年经验的 Oracle DBA，当时的学费需要 1W 多，对于家里来说也是非常大的一笔开支，当时大学一年的学费也只有 5000 左右，此时仍然非常感恩我的父母对我理想的支持。为期半年的培训，回想起那段培训的经历仍然历历在目，经常是早出晚归，天刚刚渐亮，寝室里的同学还在睡觉，我就要起床赶 2 个小时的公交去城里培训，培训的同学大都是已经参加工作想转行 DBA 的，或者是大四马上要毕业的学长。

通过培训，接触了更多志同道合的人，对 Oracle 的体系结构更加的深入理解，同时也了解企业级的数据库架构是怎么样的，对于我后来的校园招聘的笔试具有非常大的帮助作用，我仍然记得淘宝校园招聘的笔试最后一道题目就是如何构建高可用的企业系统架构。

（五）折戟校园招聘

秋季校园招聘一般在国庆节后陆续开始，我提前 1 个月开始准备校园招聘，了解校园招聘的流程，关注各大互联网公司的校园宣讲时间，复习数据结构，编程算法，数据库原理，网络原理等基础计算机课程，因为这些都是校园招聘第一关需要考察的基础内

容，这也是成为 DBA 必须要要具有的基础理论，往往很多想成为 DBA 的应届生所忽略的，这一点非常重要。

国庆后，我陆续参加了多家互联网公司的校园招聘，随着一次次的校园招聘经历，我也慢慢熟悉了这些互联网公司的招聘流程，网上投递简历—>通知笔试—>一面技术面—>二面综合面试，也可能是群面—>HR 面试，让我没有想到是 10 月份的奔波并没有带来结果，百度笔试挂掉，网易通过了笔试和一面技术面，在二面群面中挂掉，而淘宝做了笔试之后就没有消息了。

这让我有点心灰意冷，难道三年的努力就这样化为灰烬了吗，我开始把精力转向社会招聘，记得为了参加一次面试，从成都东南边坐车到成都西北边，差不多斜线横穿了成都，早上 8 点出发，中午差不多 12 点才到面试的公司，应聘公司的 HR 也被我的诚意打动，没有吃饭等着我做完笔试。

（六）喜从天降 顺利拿到淘宝 DBA 的 offer

在十一月份参加了几次社会招聘后，也没有成功被录用，在一天中午突然接到了来自杭州的一个电话，让我准备视频面试，打来电话的人是我的启蒙导师，也是我后面的第一任主管-丹臣。这突如其来的消息让我兴奋不已，原来淘宝在四川的宣讲会，我的主管并没有来，所以我的试卷是在杭州改的。

约好了时间面试后，我内心非常激动，面试过程也比较顺利，我介绍我学习 Oracle 的经历，在大学里面做过的一些网站项目，意想不到的是，对我启蒙的那篇文章作者《阿里巴巴 DBA 成长之路》就坐在我的面前，而且丹臣也是四川眉山人（我姐姐嫁到了眉山），

这让我很快进入了面试状态，接下来的问题也回答的比较满意，顺利通过了第一技术面试。

紧接着是第二技术面试，面试官是江枫，我一眼就认出来了，他，托雷，eygle 刚刚从美国参加完 OOW 回来，江枫问了我一些 Oracle 基础相关的问题，还记得其中一个问题是讲一讲 Oracle 数据段中 pectfree 和 pctused 这两个参数的含义，在最后一道综合题中没有回答出来，我很诚恳表示没有学习接触过相关技术知识，就这样二面技术面也通过了。

第三面是 HR 面，说来也是天意，我居然也第一眼认出了这位 HR（花名：玄渡），他参加了四川的校园宣讲会，我是在参加笔试的时候不经意记住了这位 HR，所以可以想象整个面试的氛围还是很融洽的，HR 问了一些在团队相处上的问题，最后一个问题是对阿里文化的讲解，我之前看过一本关于阿里巴巴文化的书，对阿里的武侠文化非常热爱，本人也非常喜欢金庸的武侠小说，特别敬仰乔峰这个角色，所以我对 HR 说我加入淘宝后能不能用乔峰这个花名，HR 笑着解释说“这个花名早就已经有人了”。就这样三面也顺利通过，一气呵成。

就在这一天（2009 年 11 月 15 日）我拿到了淘宝 DBA 团队的 offer，当时对我来说那天下午就像做了一场梦，我第一时间给在外省的母亲打了电话，发了短信给我大学里喜欢的女生，那一刻所有的一切付出都有了回报。

（七）命运的安排 阿里云，为了无法计算的价值

2010 年加入淘宝成为 Oracle DBA 之后，恰好经历了阿里云的崛起，2012 年我从

Oracle DBA 转型为 MySQLDBA 开始支持阿里云 RDS，成为 RDS 最早一波创业者，此刻仍能够想起那段这一生都无法忘怀的创业时光。

2012 年 9 月，阿里云数据库开始对外提供服务，紧接着就需要在 11 月份支撑天猫双 11，承担天猫 20%的订单量。如果用一个词来形容 2012 年的双 11，那就是肩挑背扛。很多商家对云需求强烈，但在迁移过程中，还是遇到了一些问题。当时阿里云数据库支持 MySQL 和 SQLServer 两类引擎，这两类数据库的上云迁移都不支持在线，以致用户的业务停机时间会非常长。记得有一个用户由于数据量特别大，为了加快迁移速度，其甚至把硬盘邮寄给了我们。短短一个月，我们就帮助用户手动迁移了数百台规模的数据库实例到云上。当然，这个问题现在已经不再存在，用阿里云的数据库迁移工具可以很方便完成不停机在线迁移工作。

通过这几年的技术演进，阿里云数据库不仅仅承担了 100%天猫的订单处理，我们的产品也变得更加丰富和稳定可靠，涵盖了市面各种主流数据库类型，包括 MySQL、MongoDB、Redis 等，同时我们还自研了能满足高吞吐在线事务处理的关系型云数据库 POLARDB，支持单库容量扩展至上百 TB 以及计算引擎能力及存储能力的秒级扩展能力，对比 MySQL 有 6 倍性能提升。

云计算是全民的云计算，我们忠心希望用户在使用云计算的时候能够像使用水电煤一样简单。我们也会不断地将最佳实践沉淀到产品中，只有这样才能将其作用最大化、规模化、可复制化，让用户真正享受到技术红利，也期待更多的有志之士加入我们一起来完成这一项伟大的事业。

二、总结

回顾我大学里的学习生活，从大一开始立志成为一位 DBA，经过三年的准备，最后加上了一点点运气，这一切的一切好像命中注定一样，我只能谢天谢地，冥冥之中是老天爷在帮助着你。

对于大学里面的学生，我看到过很多同学在大学里找不到方向，盲盲碌碌到了大四的时候要么考研，要么找一份自己不是很喜欢的工作，所以提早立志是非常重要的。最近重温了一部电影《三傻大闹宝莱坞》，找一个自己喜欢的工作比什么都重要，通过自己的努力积累，最终你将会厚积薄发。

关于学习方法，技术一定要多实践，多总结，学会分享，尝试着去帮助需要帮助的人，这样你会成长的更加迅速，我所认识的成长飞快的人都是特别愿意帮助别人，愿意分享的人。

为求职阿里我准备了 4 年，本科生 offer 经验分享！

作者：薛勤

简介：三月春招季马上来了，很多同学都在关注阿里巴巴的校招信息。今天，阿里妹邀请到一位刚刚拿下阿里 offer 的同学，和大家分享他的求职经验。

薛勤，2020 年本科应届毕业生，上大学前就规划好了大学四年，通过自己的努力以弥补平台不足。在大学生涯中，带领团队获得了省级软件设计大赛一等奖。有过两段实习经验，一段在网易杭州，一段在腾讯深圳。2019 年 11 月通过五轮面试拿到了阿里巴巴的正式 offer。他是如何拿到阿里 offer 的？让我们来一起看看他的经验分享吧。

一、学习类：

（一）想学 Java 可以从哪里入手？有什么推荐的书或者网站嘛？

答：从前端入手，JAVA 工程师又称全栈工程师，前端是绕不过去的坎，但是我也不建议你多深入，能带动你学习的兴趣就很不错了。之后开始学习 JAVA 语法、基础类、线程类，把 JDK 主要内容学的差不多再去学习框架，Spring、SpringBoot，做点项目练手加巩固，最后阅读 JDK 源码，各种常考常问的 HashMap 底层原理等等。推荐的书有《深入理解 JAVA 虚拟机》《JAVA 并发编程的艺术》《JAVA 多线程编程核心技术》《JAVA 程序性能优化》

（二）新手练手的项目从哪里找来练习？

答：从一些实战视频或者书籍中来，慕课网，极客学院，淘宝购买，都行。

（三）学习一门新的编程语言时，有什么技巧吗？

答：多去和已会的编程去比较去联系，比如 C 和 JAVA 其实语法上差不多的，JAVA 会更简单，更多的是面向过程和面向对象这两个编程思想上的区别。

（四）学习后端或者前端有必要去培训机构学吗？

答：这个看个人，想一想，培训机构能给你什么，又是不是你所需要的。如果你自学自控能力很强，你完全不需要去培训机构，培训机构能给你的就是学习氛围，另外有不懂的可以问老师，但是培训机构的课程范围也不一定就是最好的学习范围。

二、工作类：

（一）你做的比较难的项目有哪些？有什么经验可以分享？

答：我做的比较难的项目应该是我做的一个分布式限流系统，当时也可以说是一筹莫展，不过现在有很多公司都已经研发了这方面的技术产品并且开源，我们完全可以借鉴参考这些开源项目的思路，站在巨人的肩膀上，看的更远，做的更好。

（二）对本科毕业生而言，前端还是后端比较占优势？

答：无论面试难度还是校招待遇其实都差不多的，我甚至见过在一家公司里前端的校招待遇远远超过后端的情况。

三、面试类：

（一）面试的一些技巧可以说说嘛？

答：面试技巧太多了，总结起来一句话就是努力展现最好的自己，包括自己的硬实力和软实力，把自己的优势尽力展露出来，别哪里不会提哪里，把话题往自己优势上引，一个人 不可能是全才，尽力展现自己最好的一面吧。

（二）笔试一般包括什么？

答：选择题、算法题。算法题必考，有些公司就只有三四道算法题就完了，有些公司还会通过选择题去考察你的知识面。

最后附上视频链接：https://developer.aliyun.com/live/2311

大佬指路，助你成功

阿里研究员毕玄：又是一年校招季，我是这样考察学生的

作者：毕玄

简介：毕玄，阿里巴巴研究员，平台架构部负责人，淘宝服务框架（HSF）作者，异地多活项目负责人，目前致力于资源统一管理调度系统建设。 又一年的校招进行中，近来面试了一些学生，分享一些感受，希望对学生们有些帮助。

对于学生而言，相对而言实际的工作经验或者说工程经验会有些差距，因此就我而言，我主要考察的会是技术基础和技术兴趣这两点，进阶的话我会考察在专业研究方向（通常像硕士、博士都会有专门的研究方向）上达到的高度。

一、技术基础

技术基础要考察的具体的知识点，我在面试前会通过读学生的简历（所以简历非常重要）来决定，一般来说就是就所做过的项目，读过的书来决定要问的知识点，举几个例子来说明下。

例子一

简历里写了写过一些多线程的程序，或者是看过一些并发领域的书，这种情况下通

常我会问多线程中的一些更具体的知识点：线程池实现的细节、降低高并发程序锁竞争的方法等；

例子二

简历里写了写过一些网络通信程序，这种情况下通常我会问 Blocking-IO，NonBlocking-IO、AIO 的不同；连接池、短连接、长连接、连接复用的不同；C10K 的难度等；

例子三

简历里写了做过一些 Java Web 应用的开发，这种情况下通常我会问在这个 Java Web 应用中用到的框架的知识点，例如 Spring 的话 IoC，AOP 的实现原理是什么，或者实在不知道的话可以考虑如果自己要做的话怎么实现等；

例子四

简历里写了对 JVM 有学习，或者通过 JVM 参数调优 Java 程序，这种情况下通常我会问学习的 JVM 点：JIT、GC、参数的具体作用和对应的背后的原理等；

在上面这些技术基础的知识点的考察中，会设计问的问题逐步越来越深，试探学生在这个知识点上掌握到到了什么程度。

二、专业研究方向

硕士、博士都会有专门的研究方向，我通常会问研究的方向是什么，涉及到的知识点，研究的方向在业界目前的一些东西的状况等，例如研究方向是弹性资源调度，我会问 Mesos、Borg 这些事怎么做的；

前面两部分都过关后，我会开始考察技术兴趣。

三、技术兴趣

技术兴趣这个看起来好像很难考察，但我觉得其实是可以体现出来的，例如前面技术基础中考察的用到的东西的背后的知识点，在这个点上我通常还会问其他几个问题。

对项目中用到的东西背后的知识点在项目后一直继续研究的知识点？

有做过什么非项目/非老师授命的技术的东西，为什么做？

有过不吃饭、通宵做过的什么事情的经历，是什么事？（我始终认为一个对技术极度有兴趣的人，一定都会有不吃饭或通宵写代码的经历）

四、总结

对于我而言，技术兴趣会是我要求的学生的基本点，没有兴趣的话基本上我不会招，

不过这点其实是相辅相成的，没有兴趣还能把技术基础做到非常好的人其实很少；在技术兴趣的基础上，技术基础扎实的话基本上就可以招了，在某个专业点上做到了广阔的视野，较深的研究的话我基本上就会列为努力吸引招进来的人才，如果还刚好是对口的专业点，那就要列为必须努力吸引招进来的人才。

从经历的面试来看，很多学生都会在技术基础这点考察上闯关失败，这里我认为多数是因为对技术的兴趣导致，另外一个原因是学习方法，在如今这么发达的信息传播和分享时代，我认为只要擅长用 Google，基本就可以学的还不错，还有就是多写代码练手，很多学生会认为没有实际的场景，没法练，这个其实还是取决于自己，例如想学习写通信程序的，完全可以自己写一个，然后压测，同时对比业界一些成熟的开源的，进而翻代码去学习为什么自己写的不如开源的，又例如想学习写高并发程序，也可以自己写，不断的增加复杂度，做压力测试来不断优化提升自己写的程序的并发能力。

《Java 超神季》

等你来战

阿里云开发者“藏经阁”

海量电子书免费下载