腾讯技术文章合集

本文档内容来自「腾讯技术工程」



腾讯技术工程

扫描上方二维码关注回复「腾讯技术」：获取400+页腾讯技术文章精华合集
腾讯技术官方 QQ 交流群：957411539

2020.5

Web网络

腾讯万亿级 Elasticsearch 技术解密

作者： johngqjiang，腾讯 TEG 后台开发工程师

Elasticsearch（ES）作为开源首选的分布式搜索分析引擎，通过一套系统轻松满足用户的日志实时分析、全文检索、结构化数据分析等多种需求，大幅降低大数据时代挖掘数据价值的成本。腾讯在公司内部丰富的场景中大规模使用 ES，同时联合 Elastic 公司在腾讯云上提供内核增强版的 ES 云服务，大规模、丰富多样的的使用场景推动着腾讯对原生 ES 进行持续的高可用、高性能、低成本优化。今天给大家分享近期在 Elastic 中国开发者大会上的演讲内容：腾讯万亿级 Elasticsearch 技术解密。

一、ES 在腾讯的应用场景



本次分享的主要内容包含：首先介绍 ES 在腾讯的丰富应用场景及各种场景的典型特点；然后给出我们在大规模、高压力、丰富多样的使用场景下遇到的挑战；针对这些挑战，我们重点介绍腾讯在 ES 内核方面进行的高可用性、低成本、高性能等优化实践；最后简单分享我们在 ES 未来规划以及开源贡献方面的思考。

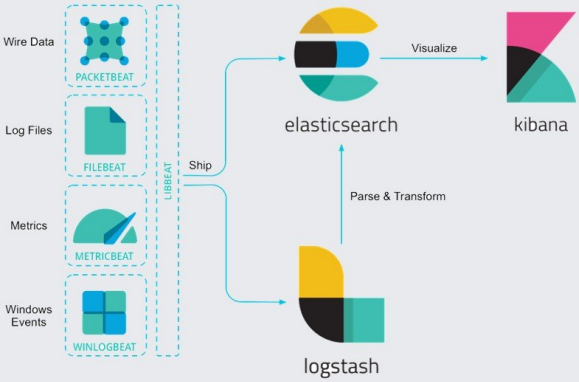
日志实时分析

典型场景

- 业务日志：用户行为
- 运营日志：慢查询、异常日志
- 审计日志
- ……

主要特点

- 完整的解决方案，易运维
- 实时性：从日志产生到可访问，十秒级
- 交互式分析：万亿级日志，搜索秒级响应
- 全文搜索：基于倒排索引，搜索分析灵活



4

我们先来看下 ES 在腾讯的应用场景。最初我们使用 ES 于日志实时分析场景，典型日志如下：运营日志，比如慢日志、异常日志，用来定位业务问题；业务日志，比如用户的点击、访问日志，可以用来分析用户行为；审计日志，可以用于安全分析。ES 很完美的解决了日志实时分析的需求，它具有如下特点：

- Elastic 生态提供了完整的日志解决方案，任何一个开发、运维同学使用成熟组件，通过简单部署，即可搭建起一个完整的日志实时分析服务。
- 在 Elastic 生态中，日志从产生到可访问一般在 10s 级。相比于传统大数据解决方案的几十分钟、小时级，时效性非常高。
- 由于支持倒排索引、列存储等数据结构，ES 提供非常灵活的搜索分析能力。
- 支持交互式分析，即使在万亿级日志的情况下，ES 搜索响应时间也是秒级。

日志是互联网行业最基础、最广泛的数据形式，ES 非常完美的解决了日志实时分析场景，这也是近几年 ES 快速发展的一个重要原因。

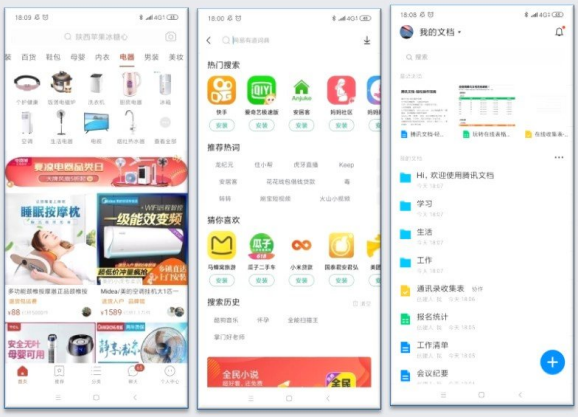
搜索服务

典型场景

- 商品搜索
- APP搜索
- 站内搜索
- ……

主要特点

- 高性能：10w级QPS，20ms级平响
- 强相关：搜索结果高度匹配用户意图
- 高可用：可用性达9999，跨机房容灾



5

第二类使用场景是搜索服务，典型场景包含：商品搜索，类似京东、淘宝、拼多多中的商品搜索；APP 搜索，支持应用商店里的应用搜索；站内搜索，支持论坛、在线文档等搜索功能。我们支持了大量搜索服务，它们主要有以下特点：

- 高性能：单个服务最大达到 10w+ QPS，平响 20ms~，P95 延时小于 100ms。

- 强相关：搜索体验主要取决于搜索结果是否高度匹配用户意图，需要通过正确率、召回率等指标进行评估。
- 高可用：搜索场景通常要求 4 个 9 的可用性，支持单机房故障容灾。任何一个电商服务，如淘宝、京东、拼多多，只要故障一个小时就可以上头条。



第三类使用场景是时序数据分析，典型的时序数据包含：Metrics，即传统的服务器监控；APM，应用性能监控；物联网数据，智能硬件、工业物联网等产生的传感器数据。这类场景腾讯很早就开始探索，在这方面积累了非常丰富的经验。这类场景具有以下特点：

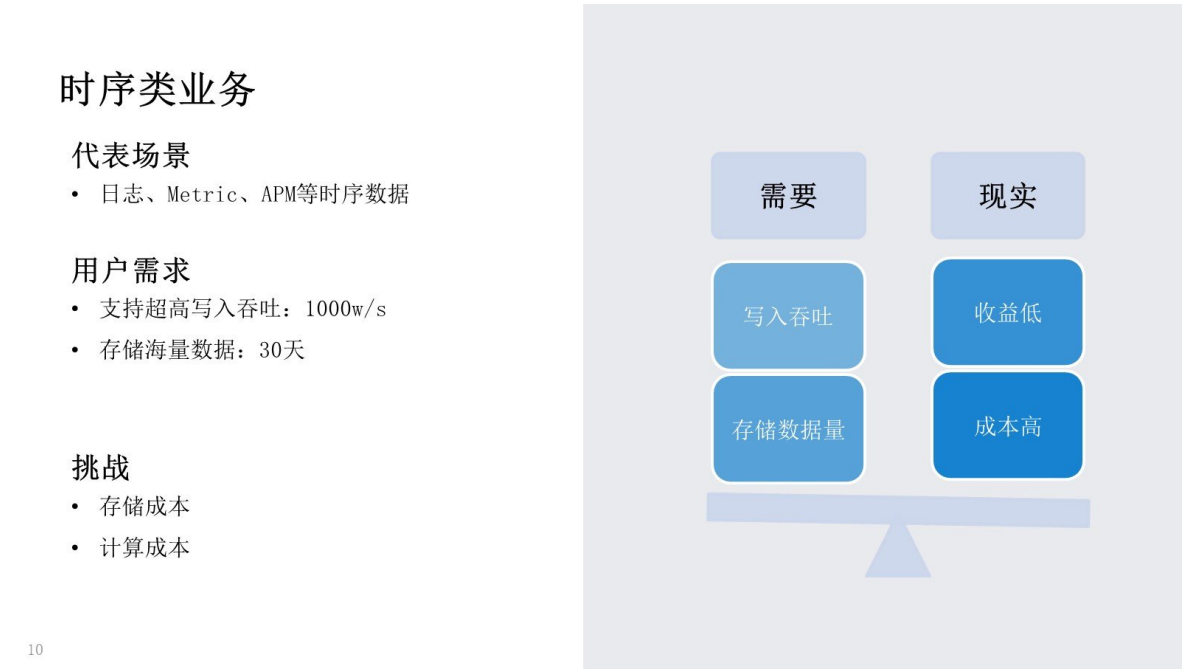
- 高并发写入：线上单集群最大规模达到 600+节点、1000w/s 的写入吞吐。
- 高查询性能：要求单条曲线 或者单个时间线的查询延时在 10ms~。
- 多维分析：要求灵活、多维度的统计分析能力，比如我们在查看监控的时候，可以按照地域、业务模块等灵活的进行统计分析。

二、遇到的挑战

前面我们介绍了 ES 在腾讯内部的广泛应用，在如此大规模、高压力、丰富使用场景的背景下，我们遇到了很多挑战，总体可以划分为两类：搜索类和时序类。



首先，我们一起看看搜索类业务的挑战。以电商搜索、APP 搜索、站内搜索为代表，这类业务非常重视可用性，服务 SLA 达到 4 个 9 以上，需要容忍单机故障、单机房网络故障等；同时要求高性能、低毛刺，例如 20w QPS、平响 20ms、P95 延时 100ms。总之，在搜索类业务场景下，核心挑战点在于高可用、高性能。



另一类我们称之为时序类业务挑战，包含日志、Metrics、APM 等场景。相比于搜索类业务重点关注高可用、高性能，时序类业务会更注重成本、性能。比如时序场景用户通常要求高写入吞吐，部分场景可达 1000w/s
WPS；在这样写入吞吐下，保留 30 天的数据，通常可达到 PB 级的存储量。而现实是日志、监控等场景的收益相对较低，很可能用户用于线上实际业务的机器数量才是 100 台，而监控、日志等需要 50 台，这对多数用户来说，基本是不可接受的。所以在时序类业务中，主要的挑战在于存储成本、计算成本等方面。

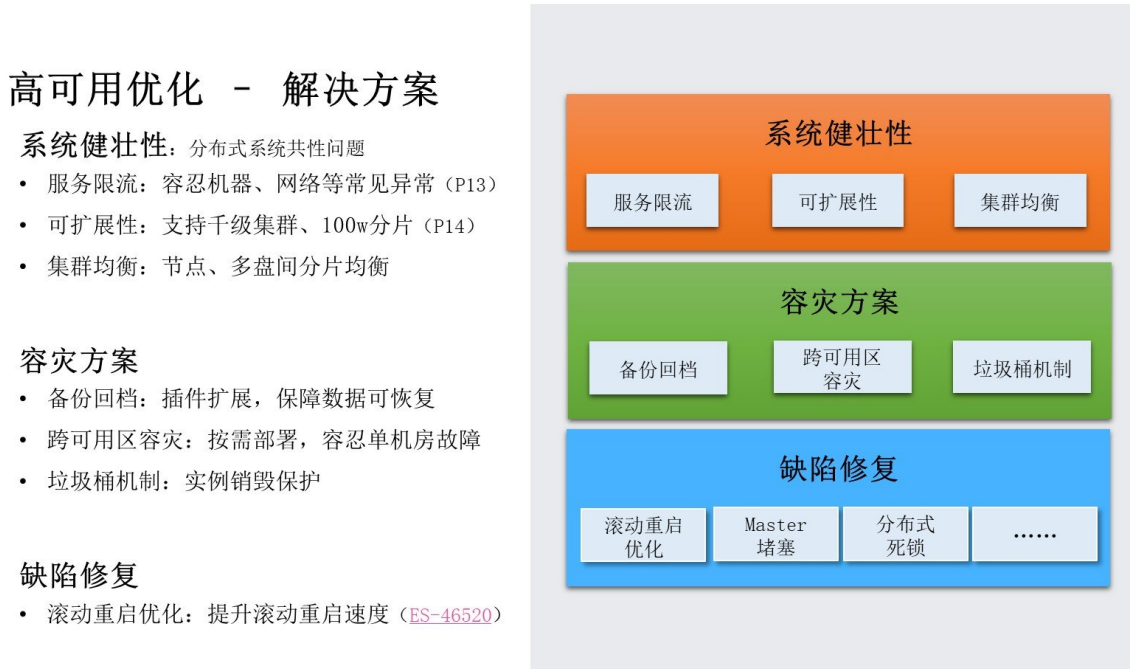
前面我们介绍了在搜索类、时序类业务场景下遇到的高可用、低成本、高性能等挑战，下面针对这些挑战，我们重点分享腾讯在 ES 内核方面的深入实践。

三、ES 优化实践



首先，我们来看看高可用优化，我们把高可用划分为三个维度：

- 系统健壮性：是指 ES 内核自身的健壮性，也是分布式系统面临的共性难题。例如，在异常查询、压力过载下集群的容错能力；在高压力场景下，集群的可扩展性；在集群扩容、节点异常场景下，节点、多硬盘之间的数据均衡能力。
- 容灾方案：如果通过管控系统建设，保障机房网络故障时快速恢复服务，自然灾害下防止数据丢失，误操作后快速恢复等。
- 系统缺陷：这在任何系统发展过程中都会持续产生，比如说 Master 节点堵塞、分布式死锁、滚动重启缓慢等。

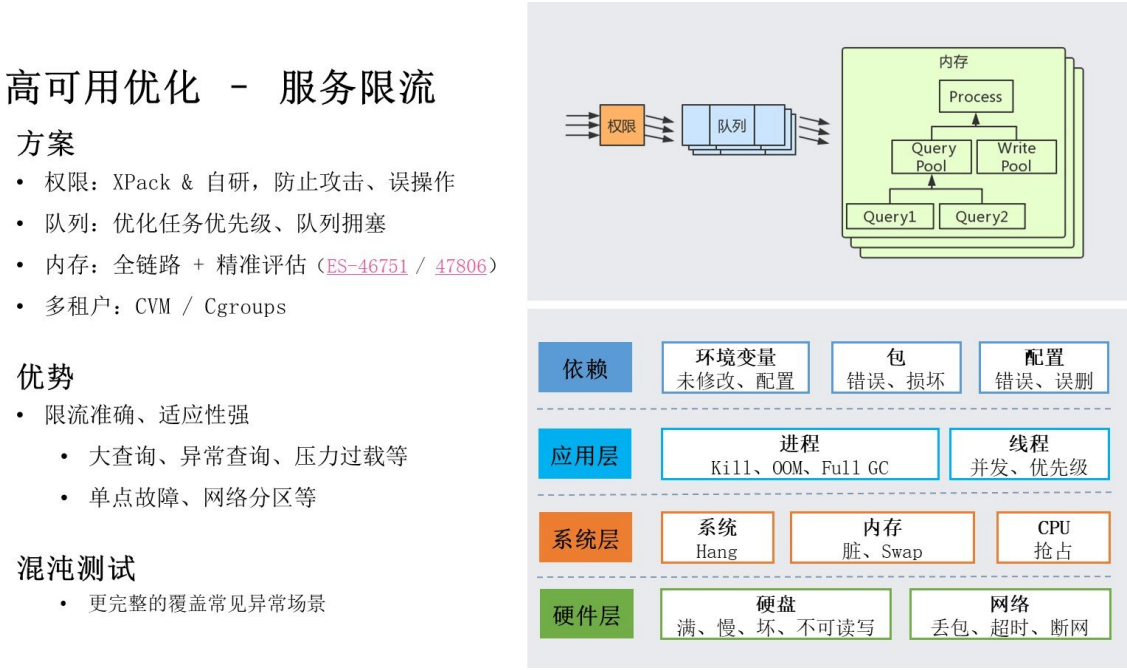


针对上述问题，下面来介绍我们在高可用方面的解决方案：

系统健壮性方面，我们通过服务限流，容忍机器网络故障、异常查询等导致的服务不稳定，后面展开介绍。通过优化集群元数据管控逻辑，提升集群扩展能力一个数量级，支持千级节点集群、百万分片，解决集群可扩展性问题；集群均衡方面，通过优化节点、多硬盘间的分片均衡，保证大规模集群的压力均衡。

容灾方案方面，我们通过扩展 ES 的插件机制支持备份回档，把 ES 的数据备份回档到廉价存储，保证数据的可恢复；支持跨可用区容灾，用户可以按需部署多个可用区，以容忍单机房故障。垃圾桶机制，保证用户在欠费、误操作等场景下，集群可快速恢复。

系统缺陷方面，我们修复了滚动重启、Master 阻塞、分布式死锁等一系列 Bug。其中滚动重启优化，可加速节点重启速度 5+倍，具体可参考 PR ES-46520；Master 堵塞问题，我们在 ES 6.x 版本和官方一起做了优化。



这里我们展开介绍下服务限流部分。我们做了 4 个层级的限流工作：权限层级，我们支持 XPack 和自研权限来防止攻击、误操作；队列层级，通过优化任务执行速度、重复、优先级等问题，解决用户常遇到的 Master 任务队列堆积、任务饿死等问题；内存层级，我们从 ES 6.x 开始，支持在 HTTP 入口、协调节点、数据节点等全链路上进行内存限流，同时使用 JVM 内存、梯度统计等方式精准控制；多租户层级，我们使用 CVM/Cgroups 方案保证多租户间的资源隔离。

这里详细介绍下聚合场景限流问题，用户在使用 ES 进行聚合分析时，经常遇到因聚合分桶过多打爆内存的问题。官方在 ES 6.8 中提供 max_buckets 参数控制聚合的最大分桶数，但这个方式局限性非常强。在某些场景下，用户设置 20 万个分桶可以正常工作，但在另一些场景下，可能 10 万个分桶内存就已经打爆，这主要取决于单分桶的大小，用户并不能准确把握该参数设置为多少比较合适。我们在聚合分析的过程中，采用梯度算法进行优化，每分配 1000 个分桶检查一次 JVM 内存，当内存不足时及时中断请求，保证 ES 集群的高可用。具体可参考 PR ES-46751 /47806。

我们当前的限流方案，能够大幅提升在异常查询、压力过载、单节点故障、网络分区等场景下，ES 服务的稳定性问题。但还有少量场景没有覆盖完全，所以我们目前也在引入混沌测试，依赖混沌测试来覆盖更多异常场景。

成本优化

成本分析

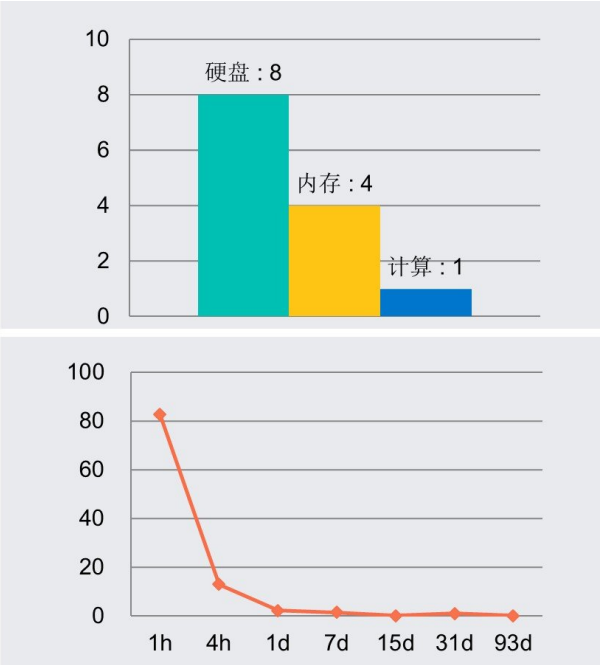
- 代表场景：日志等时序数据
- 成本比例关系

主要瓶颈

- 硬盘
- 内存

特性分析

- 冷热特性明显：访问近多远少
- 历史数据查询聚合结果
- 长时间范围查询性能差



前面我们介绍了高可用解决方案，下面我们来介绍成本方面的优化实践。成本方面的挑战，主要体现在以日志、监控为代表的时序场景对机器资源的消耗，我们对线上典型的日志、时序业务进行分析，总体来看，硬盘、内存、计算资源的成本比例接近 8:4:1，硬盘、内存是主要矛盾，其次是计算成本。

而对时序类场景进行分析，可以发现时序数据有很明显的访问特性。一是冷热特性，时序数据访问具有近多远少的特点，最近 7 天数据的访问量占比可达到 95%以上；历史数据访问较少，且通常都是访问统计类信息。

成本优化 - 解决方案

硬盘成本

- 冷热分离：使用混合存储平衡成本、性能
- Rollup：预计算 换取 存储、性能（P17）
- 备份归档：更廉价的存储系统
- 更多优化方式：存储裁剪、生命周期管理等

内存成本（P18）

- LRU Cache：提升内存利用效率
- Off Heap：降低堆内存，提高节点规格



基于这些瓶颈分析和数据访问特性，我们来介绍成本优化的解决方案。

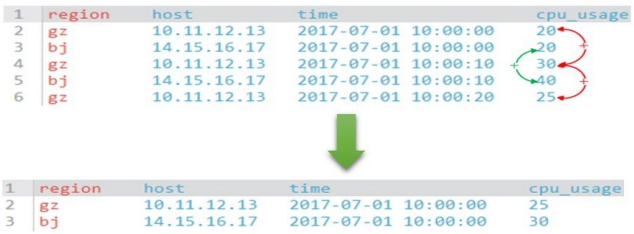
硬盘成本方面，由于数据具有明显的冷热特性，首先我们采用冷热分离架构，使用混合存储的方案来平衡成本、性能；其次，既然对历史数据通常都是访问统计信息，那么以通过预计算来换取存储和性能，后面会展开介绍；如果历史数据完全不使用，也可以备份到更廉价的存储系统；其他一些优化方式包含存储裁剪、生命周期管理等。

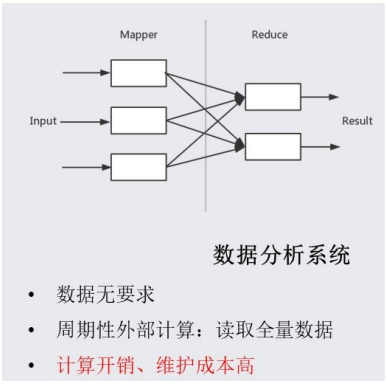
内存成本方面，很多用户在使用大存储机型时会发现，存储资源才用了百分之二十，内存已经不足。其实基于时序数据的访问特性，我们可以利用 Cache 进行优化，后面会展开介绍。

成本优化 - Rollup

方案

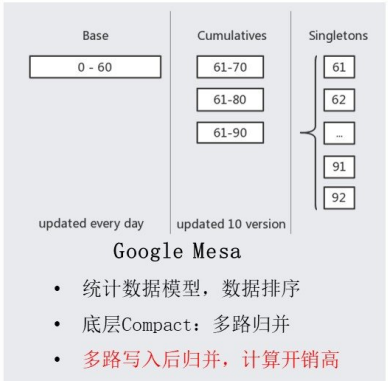
- 类似 Cube / 物化视图
- 预聚合：换取 成本和性能（数量级）





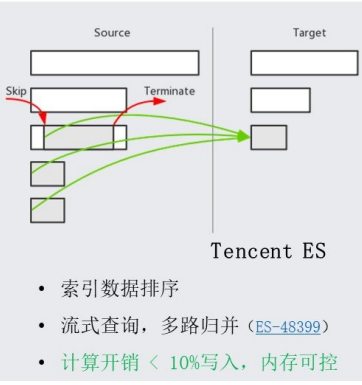
数据分析系统

- 数据无要求
- 周期性外部计算：读取全量数据
- 计算开销、维护成本高



Google Mesa

- 统计数据模型，数据排序
- 底层Compact：多路归并
- 多路写入后归并，计算开销高



Tencent ES

- 索引数据排序
- 流式查询，多路归并（ES-48399）
- 计算开销 < 10%写入，内存可控

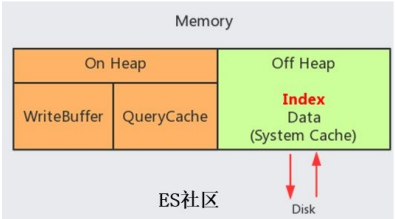
我们展开介绍下 Rollup 部分。官方从 ES 6.x 开始推出 Rollup，实际上腾讯在 5.x 已经开始这部分的实践。Rollup 类似于大数据场景下的 Cube、物化视图，它的核心思想是通过预计算提前生成统计信息，释放掉原始粒度数据，从而降低存储成本、提高查询性能，通常会有数据级的收益。这里举个简单的例子，比如在机器监控场景下，原始粒度的监控数据是 10 秒级的，而一个月之前的监控数据，一般只需要查看小时粒度，这即是一个 Rollup 应用场景。

在大数据领域，传统的方案是依赖外部离线计算系统，周期性的读取全量数据进行计算，这种方式计算开销、维护成本高。谷歌的广告指标系统 Mesa 采用持续生成方案，数据写入时系统给每个 Rollup 产生一份输入数据，并对数据进行排序，底层在 Compact/Merge 过程中通过多路归并完成 Rollup，这种方式的计算、维护成本相对较低。ES 从 6.x 开始支持数据排序，我们通过流式查询进行多路归并生成 Rollup，最终计算开销小于全量数据写入时 CPU 开销的 10%，内存使用小于 10MB。我们已反馈内核优化至开源社区，解决开源 Rollup 的计算、内存瓶颈，具体可参考 PR ES-48399。

成本优化 - 内存优化

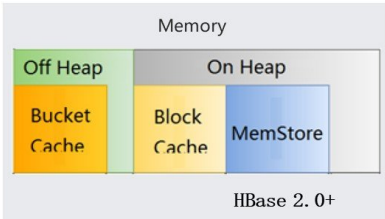
方案

- LFU Cache：提升内存利用效率，大小动态可调
- Off Heap：降低堆内存使用
- Cache复用： Weak Reference降低Cache查找1000~倍



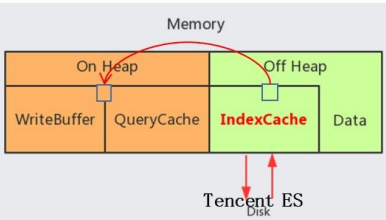
ES社区

- 按需加载，依赖系统缓存
- 查询性能倍数级衰退



HBase 2.0+

- 重点推介Off Heap
- 堆外内存性能接近堆内

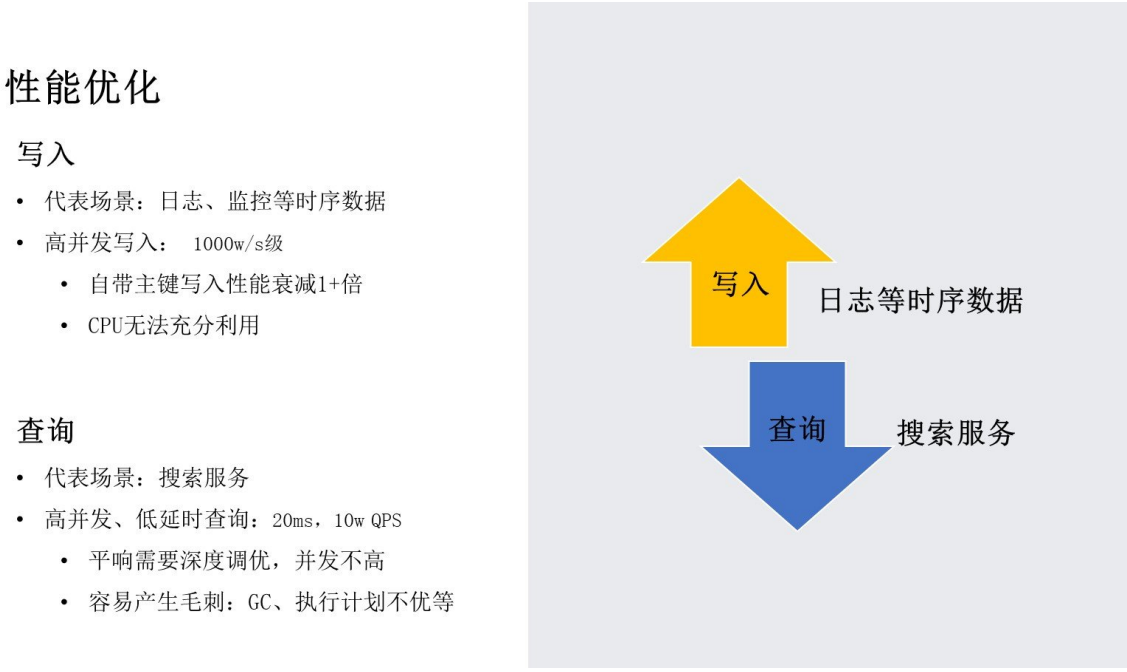


Tencent ES

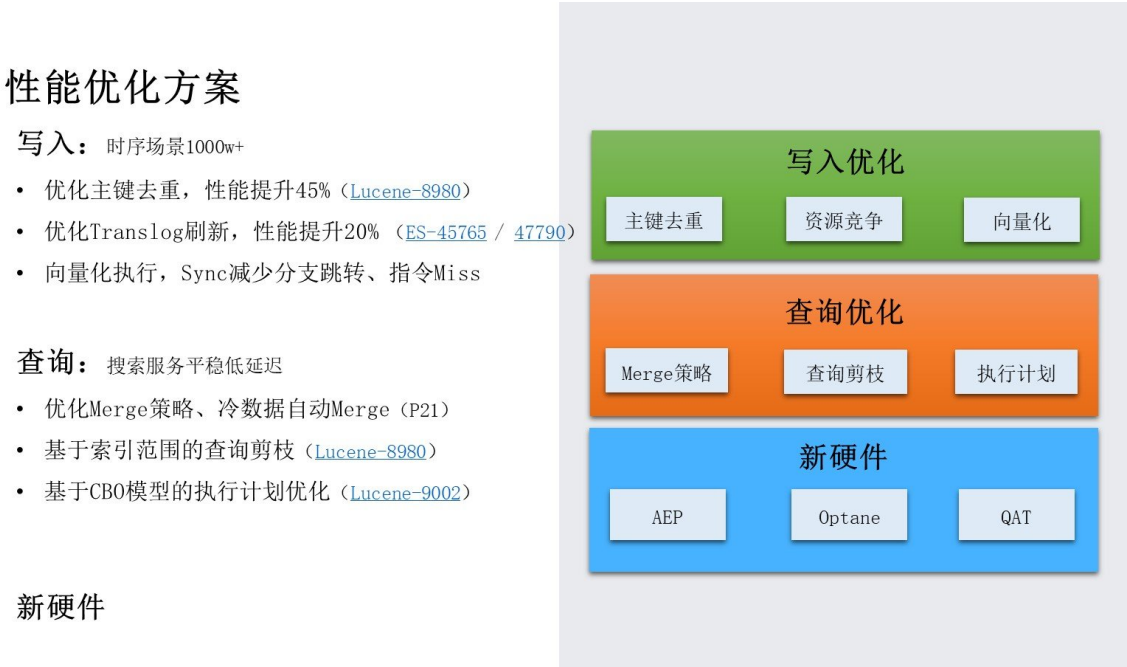
- 内存利用效率提升80%，充分利用硬盘
- 查询性能损耗不超过2%
- 降低GC开销30%

接下来，我们展开介绍内存优化部分。前面提到很多用户在使用大存储机型时，内存优先成为瓶颈、硬盘不能充分利用的问题，主要瓶颈在于索引占用大量内存。但是我们知道时序类场景对历史数据访问很少，部分场景下某些字段基本不使用，所我们可以通过引入 Cache 来提高内存利用效率。

在内存优化方面，业界的方案是什么样的呢？ES 社区从 7.x 后支持索引放于堆外，和 DocValue 一样按需加载。但这种方式不好的地方在于索引和数据的重要性完全不同，一个大查询很容易导致索引被淘汰，后续查询性能倍数级的衰减。Hbase 通过缓存 Cache 缓存索引、数据块，提升热数据访问性能，并且从 HBase 2.0 开始，重点介绍其 Off Heap 技术，重点在于堆外内存的访问性能可接近堆内。我们基于社区经验进行迭代，在 ES 中引入 LFU Cache 以提高内存的利用效率，把 Cache 放置在堆外以降低堆内存压力，同时通过 Weak Reference、减少堆内外拷贝等技术降低损耗。最终效果是内存利用率提升 80%，可以充分利用大存储机型，查询性能损耗不超过 2%，GC 开销降低 30%。



前面我们介绍了可用性、成本优化的解决方案，最后我们来介绍性能方面的优化实践。以日志、监控为代表的时序场景，对写入性能要求非常高，写入并发可达 1000w/s。然而我们发现在带主键写入时，ES 性能衰减 1+倍，部分压测场景下，CPU 无法充分利用。以搜索服务为代表的场景，对查询性的要求非常高，要求 20w QPS, 平响 20ms，而且尽量避免 GC、执行计划不优等造成的查询毛刺。

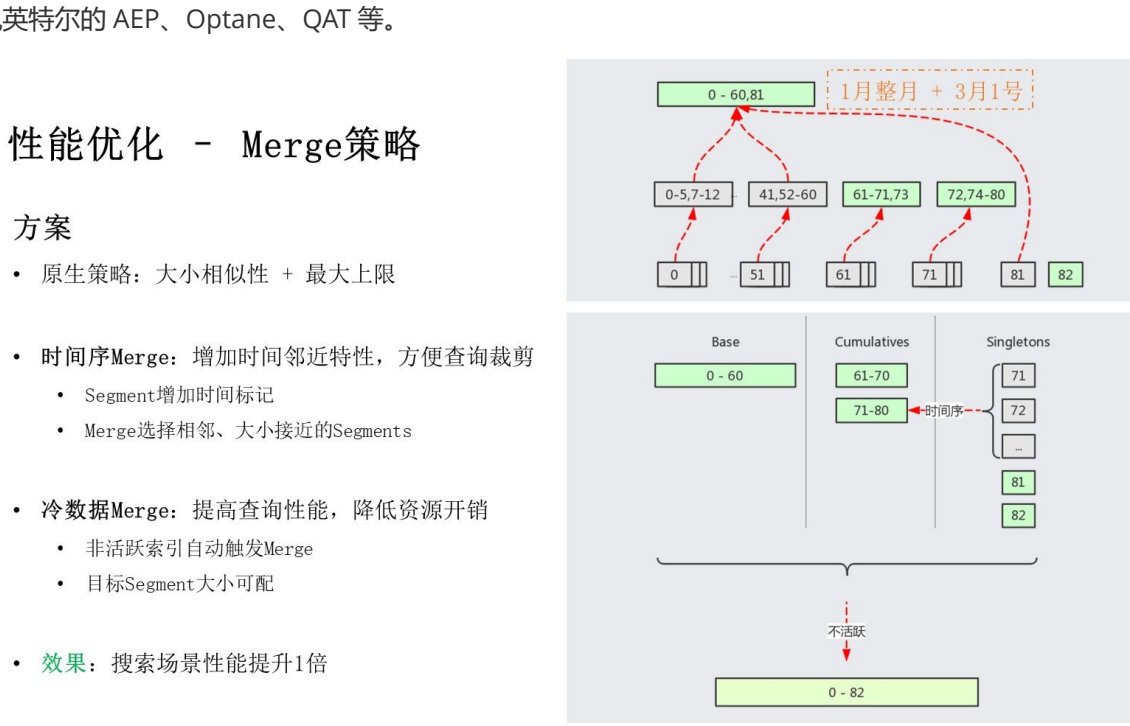


针对上述问题，我们介绍下腾讯在性能方面的优化实践：

写入方面，针对主键去重场景，通过利用索引进行裁剪，加速主键去重的过程，写入性能提升 45%，具体可参考 PR Lucene-8980。对于部分压测场景下 CPU 不能充分利用的问题，通过优化 ES 刷新 Translog 时的资源抢占，提升性能提升 20%，具体可参考 PR ES-45765 /47790。我们正在尝试通过向量化执行优化写入

性能，通过减少分支跳转、指令 Miss，预期写入性能可提升 1 倍。

查询方面，我们通过优化 Merge 策略，提升查询性能，这部分稍后展开介绍。基于每个 Segment 记录的 min/max 索引，进行查询剪枝，提升查询性能 30%。通过 CBO 策略，避免查询 Cache 操作导致查询耗时 10+倍的毛刺，具体可参考Lucene-9002。此外，我们也在尝试通过一些新硬件来优化性能，比如说英特尔的 AEP、Optane、QAT 等。



接下来我们展开介绍下 Merge 策略优化部分。ES 原生的 Merge 策略主要关注大小相似性和最大上限，大小相似性是指 Merge 时尽量选择大小相似的 Segments 进行 Merge，最大上限则考虑尽量把 Segment 拼凑到 5GB。那么有可能出现某个 Segment 中包含了 1 月整月、3 月 1 号的数据，当用户查询 3 月 1 号某小时的数据时，就必须扫描大量无用数据，性能损耗严重。

我们在 ES 中引入了时序 Merge，在选择 Segments 进行 Merge 时，重点考虑时间因素，这样时间相近的 Segments 被 Merge 到一起。当我们查询 3 月 1 号的数据时，只需要扫描个别较小的 Segments 就好，其他的 Segments 可以快速裁剪掉。

另外，ES 官方推荐搜索类用户在写入完成之后，进行一次 Force Merge，用意是把所有 Segments 合并为一个，以提高搜索性能。但这增加了用户的使用成本，且在时序场景下，不利于裁剪，需要扫描全部数据。我们在 ES 中引入了冷数据自动 Merge，对于非活跃的索引，底层 Segments 会自动 Merge 到接近 5GB，降低文件数量的同时，方便时序场景裁剪。对于搜索场景，用户可以调大目标 Segment 的大小，使得所有 Segments 最终 Merge 为一个。我们对 Merge 策略的优化，可以使得搜索场景性能提升 1 倍。

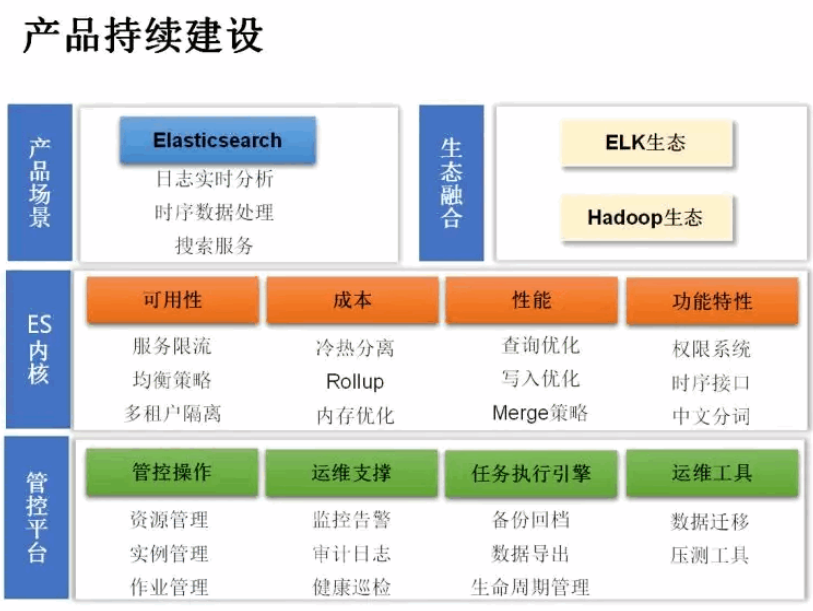
前面介绍完毕我们再 ES 内核方面的优化实践，最后我们来简单分享下我们在开源贡献及未来规划方面的思考。

四、未来规划及开源贡献



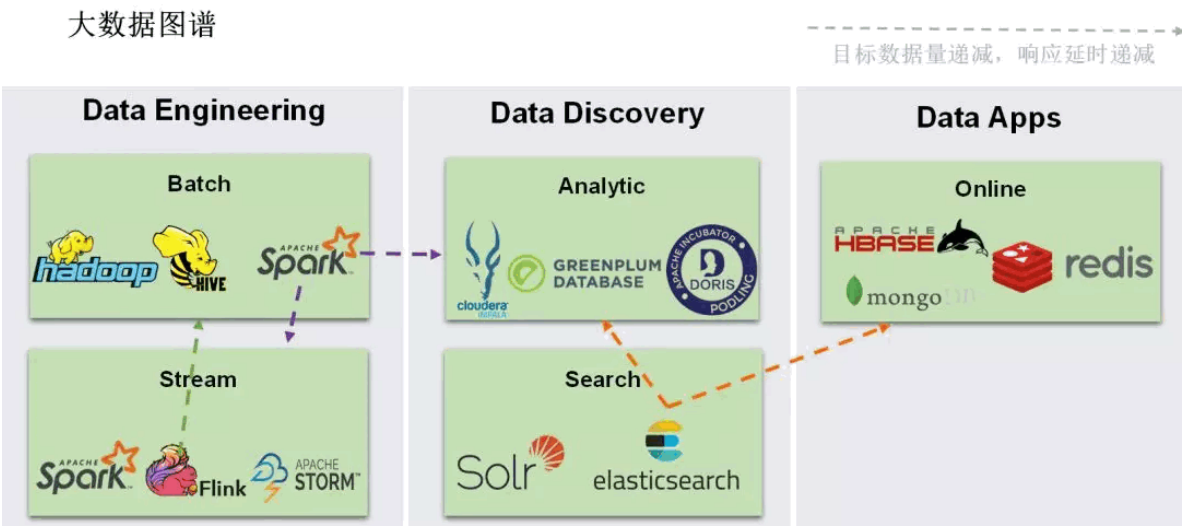
近半年我们向开源社区提交了 10+PR，涉及到写入、查询、集群管理等各个模块，部分优化是和官方开发同学一起来完成的，前面介绍过程中，已经给出相应的 PR 链接，方便大家参考。我们在公司内部也组建了开源协同的小组，来共建 Elastic 生态。

总体来说，开源的收益利大于弊，我们把相应收益反馈出来，希望更多同学参与到 Elastic 生态的开源贡献中：首先，开源可以降低分支维护成本，随着自研的功能越来越多，维护独立分支的成本越来越高，主要体现在与开源版本同步、快速引入开源新特性方面；其次，开源可以帮助研发同学更深入的把控内核，了解最新技术动态，因为在开源反馈的过程中，会涉及与官方开发人员持续的交互。此外，开源有利于建立大家在社区的技术影响力，获得开源社区的认可。最后 Elastic 生态的快速发展，有利于业务服务、个人技术的发展，希望大家一起参与进来，助力 Elastic 生态持续、快速的发展。



未来规划方面，这次分享我们重点介绍了腾讯在 ES 内核方面的优化实践，包含高可用、低成本、高性能等方面。此外，我们也提供了一套管控平台，支持线上集群自动化管控、运维，为腾讯云客户提供 ES 服务。但是从线上大量的运营经验分析，我们发现仍然有非常丰富、高价值的方向需要继续跟进，我们会持续继续加强对产品、内核的建设。

长期探索



长期探索方面，我们结合大数据图谱来介绍。整个大数据领域，按照数据量、延时要求等特点，可以划分为三部分：第一部分是 Data Engineering，包含我们熟悉的批量计算、流式计算；第二部分是 Data Discovery，包含交互式分析、搜索等；第三个部分是 Data Apps，主要用于支撑在线服务。

虽然我们把 ES 放到搜索领域内，但是也有很多用户使用 ES 支持在线搜索、文档服务等；另外，我们了解到有不少成熟的 OLAP 系统，也是基于倒排索引、行列混存等技术栈，所以我们认为 ES 未来往这两个领域发展的可行性非常强，我们未来会在 OLAP 分析和在线服务等方向进行重点探索。

腾讯Elasticsearch海量规模背后的内核优化剖析

作者： 黄华，腾讯 TEG 云架构平台部研发工程师

背景

Elasticsearch 在腾讯内部广泛应用于日志实时分析、结构化数据分析、全文检索等场景，目前单集群规模达到千级节点、万亿级吞吐，同时腾讯联合 Elastic 公司在腾讯云上提供了内核增强版 ES 云服务。海量规模、丰富的应用场景推动着腾讯对原生 ES 进行持续的高可用、高性能、低成本等全方位优化。本次分享主要剖析腾讯对 Elasticsearch 海量规模下的内核优化与实践，希望能和广大 ES 爱好者共同探讨推动 ES 技术的发展。

目录

本次分享内容主要分为以下几个部分：首先介绍 ES 在腾讯海量规模应用的背景，然后是我们遇到的痛点与挑战，接下来针对这些痛点与挑战，剖析我们对 ES 做的内核优化，最后介绍我们的开源贡献以及未来的规划。

目录



ES在腾讯的海量规模背景
遇到的挑战与痛点
腾讯ES内核优化剖析
开源贡献及未来规划

一、ES在腾讯的海量规模背景

应用场景



先来看看 ES 在腾讯的主要应用场景。ES 是一个实时的分布式搜索分析引擎，目前很多用户对 ES 的印象还是准实时，实际上在6.8版本之后官方文档已经将 near real-time 改为了 real-time：
"Elasticsearch provides real-time search and analytics for all types of data."
ES 在写入完毕刷新之前，是可以通过 getById 的方式实时获取文档的，只是在刷新之前 FST 还没有构建，还不能提供搜索的能力。
目前 ES 在腾讯主要应用在三个方面：

- **搜索服务：** 例如像腾讯文档基于 ES 做全文检索，我们的电商客户拼多多、蘑菇街等大量的商品搜索都是基于 ES。
- **日志分析：** 这个是 ES 应用最广泛的领域，支持全栈的日志分析，包括各种应用日志、数据库日志、用户行为日志、网络数据、安全数据等等。ES 拥有一套完整的日志解决方案，可以秒级实现从采集到展示。
- **时序分析：** 典型的场景是监控数据分析，比如云监控，整个腾讯云的监控都是基于 ES 的。此外还包括物联网场景，也有大量的时序数据。时序数据的特点是写入吞吐量特别高，ES 支持的同时也提供了丰富的多维统计分析算子。

当然除了上面的场景之外，ES 本身在站内搜索、安全、APM等领域也有广泛的应用。

目前 ES 在腾讯公有云、专有云以及内部云上面均有提供服务，可以广泛的满足公司内外客户的业务需求。公有云上的使用场景非常丰富，专有云主要实现标准化交付和自动化运维，腾讯内部云上的 ES 都是 PB 级的超大规模集群。



二、痛点与挑战

在这些丰富的应用场景，以及海量的规模背景下，我们也遇到了很多的痛点与挑战。主要覆盖在可用性、性能、成本以及扩展性方面。



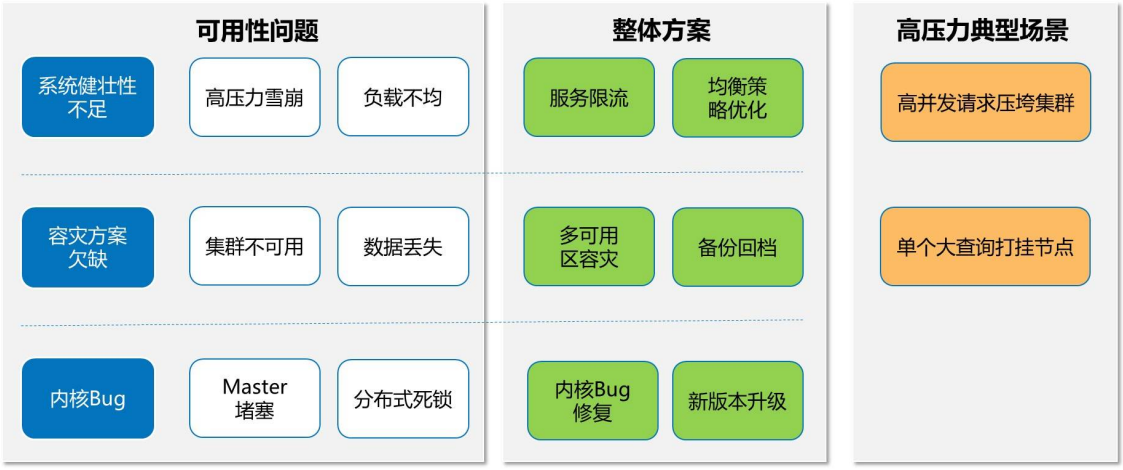
- **可用性：** 最常见的问题是节点因高负载 OOM，或者整个集群因高负载而雪崩。这些痛点使我们很难保障 SLA，尤其是在搜索场景， 可用性要求4个9以上。
- **性能：** 搜索场景一般要求平响延时低于 20 毫秒，查询毛刺低于 100 毫秒。在分析场景，海量数据下，虽然实时性要求没那么高，但请求响应时间决定了用户体验，资源消耗决定了性能边界。
- **成本：** 很多用户都比较关注 ES 的存储成本，因为 ES 确实数据类型较多，压缩比比较低，存储成本比较高，但是优化的空间还是很大的。另外还包括内存成本，ES 有大量的索引数据需要加载到内存提供高性能的搜索能力。那么对于日志、监控等海量场景，成本的挑战就更大。

- **扩展性：** 日志、时序等场景，往往索引会按周期滚动，长周期会产生大量的索引和分片，超大规模集群甚至有几十上百万的分片、千级节点的需求。而目前原生版本 ES 只能支持到万级分片、百级节点。随着大数据领域的飞速发展，ES 最终是要突破 TB 的量级，跨越到 PB 的量级，扩展性就成为了主要的瓶颈与挑战。

三、腾讯 ES 内核优化剖析

ES 使用姿势、参数调优等在社区有很多的案例和经验可以借鉴，但很多的痛点和挑战是无法通过简单的调优来解决的，这个时候就需要从内核层面做深度的优化，来不断完善这个优秀的开源产品。接下来就是本次分享的核心部分，我们来看看腾讯是如何在内核层面对 ES 做优化的。

可用性优化

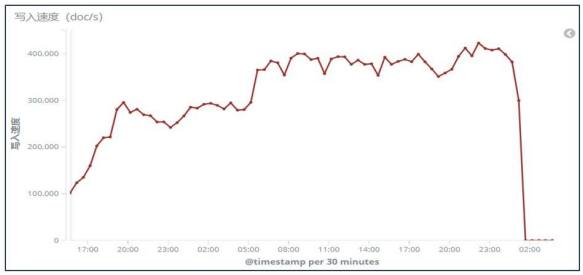


首先介绍可用性优化部分。总体来说，原生版本在可用性层面有三个层面的问题：

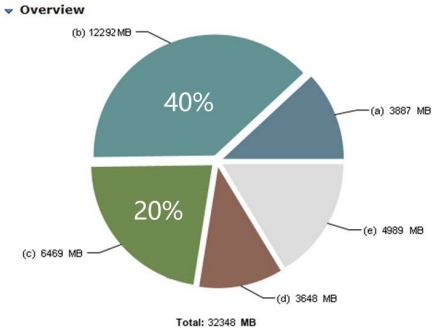
- **系统健壮性不足：** 高压力下集群雪崩，主要原因是内存资源不足。负载不均会导致部分节点压力过载，节点 OOM。我们在这个层面的方案主要是优化服务限流和节点均衡策略。
- **容灾方案欠缺：** ES 本身提供副本机制提升数据安全性，对于多可用区容灾还是需要云平台额外实现。即使有副本机制，甚至有跨集群复制（CCR），但还是不能阻挡用户误操作导致的数据删除，所以还需要额外提供低成本的备份回挡能力。
- **内核 Bug：** 我们修复了 Master 任务堵塞、分布式死锁、滚动重启速度慢等一系列内核可用性相关的问题，并及时提供新版本给用户升级。

接下来针对用户在可用性层面常遇到的两类问题展开分析。一类是高并发请求压垮集群，另一类是单个大查询大挂节点。

可用性优化场景一：高并发请求压垮集群



内部某日志集群，写入量一天突增5倍



集群现象：

- 多个节点 **Old GC** 卡住脱离集群
- 集群状态 **RED**（数据主副本分片均不可用）
- 集群停写

JVM内存分布：

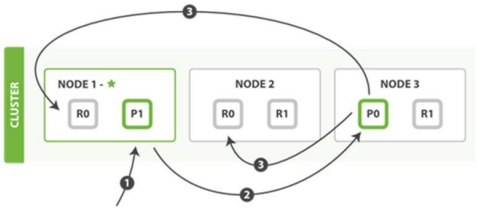
- 反序列化**前**的批量写入请求 **20%** （Netty 缓存）
- 反序列化**后**拆分的分片级批量写入请求 **40%**

先来看第一类场景，高并发请求压垮集群。例如早期我们内部一个日志集群，写入量一天突增 5 倍，集群多个节点 Old GC 卡住脱离集群，集群 RED，写入停止，这个痛点确实有点痛。我们对挂掉的节点做了内存分析，发现大部分内存是被反序列化前后的写入请求占用。我们来看看这些写入请求是堆积在什么位置。

可用性优化场景一：高并发请求压垮集群

场景分析

High Level 写入流程

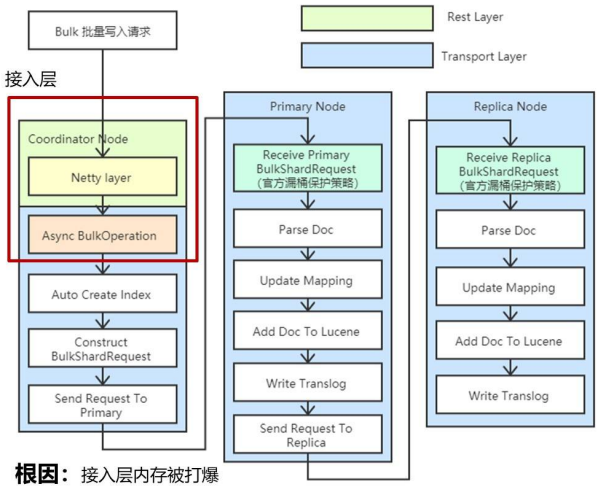


❶ 用户写入请求发送至任意节点

❷ 请求转发至**主分片**所在节点处理

❸ 主分片处理完并发转发给**从分片**处理

Internal 写入流程分析



根因：接入层内存被打爆

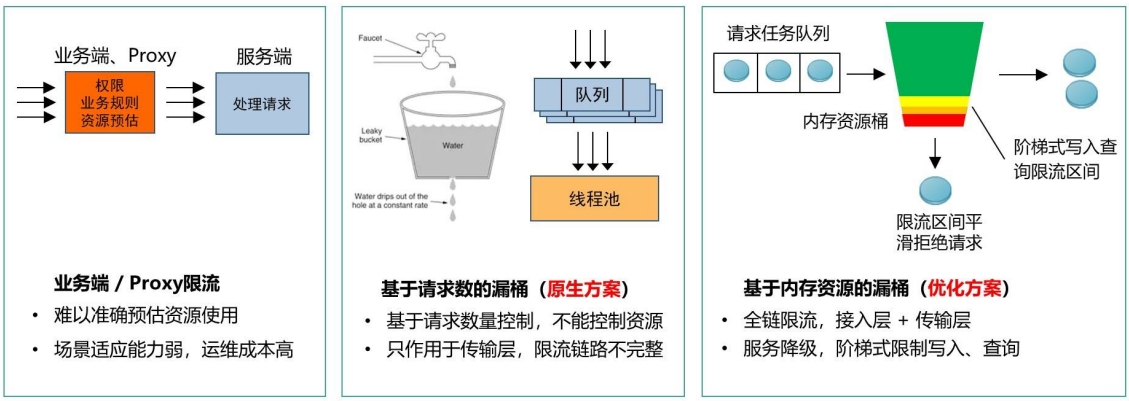
ES high level 的写入流程，用户的写入请求先到达其中一个数据节点，我们称之为数据节点。然后由该协调节点将请求转发给主分片所在节点进行写入，主分片写入完毕再由主分片转发给从分片写入，最后返回给客户端写入结果。右边是更细节的写入流程，而我们从堆栈中看到的写入请求堆积的位置就是在红色框中的接入层，节点挂掉的根因是协调节点的接入层内存被打爆。

找到了问题的原因，接下来介绍我们的优化方案。

可用性优化场景一：高并发请求压垮集群

优化方案 - 服务限流

- 除了并发请求数量，还要能精准控制内存资源
- 通用性要强，能作用于各个层级实现全链限流



业务端 / Proxy限流

- 难以准确预估资源使用
- 场景适应能力弱，运维成本高

基于请求数的漏桶（原生方案）

- 基于请求数量控制，不能控制资源
- 只作用于传输层，限流链路不完整

基于内存资源的漏桶（优化方案）

- 全链限流，接入层 + 传输层
- 服务降级，阶梯式限制写入、查询

针对这种高并发场景，我们的优化方案是服务限流。除了要能控制并发请求数量，还要能精准的控制内存资源，因为内存资源不足是主要的矛盾。另外通用性要强，能作用于各个层级实现全链限流。

限流方案，很多数据库使用场景会采用从业务端或者独立的 proxy 层配置相关的业务规则，做资源预估等方式进行限流。这种方式适应能力弱，运维成本高，而且业务端很难准确的预估资源消耗。

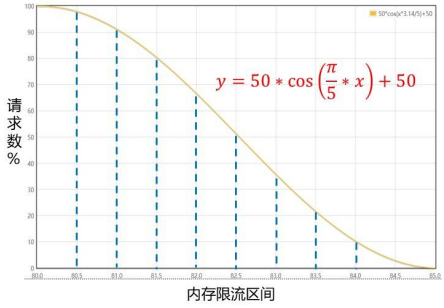
原生版本本身有限流策略，是基于请求数的漏桶策略，通过队列加线程池的方式实现。线程池大小决定的了处理并发度，处理不完放到队列，队列放不下则拒绝请求。但是单纯的基于请求数的限流不能控制资源使用量，而且只作用于分片级子请求的传输层，对于我们前面分析的接入层无法起到有效的保护作用。原生版本也有内存熔断策略，但是在协调节点接入层并没有做限制。

我们的优化方案是基于内存资源的漏桶策略。我们将节点 JVM 内存作为漏桶的资源，当内存资源足够的时候，请求可以正常处理，当内存使用量到达一定阈值的时候分区间阶梯式平滑限流。例如图中浅黄色的区间限制写入，深黄色的区间限制查询，底部红色部分作为预留 buffer，预留给处理中的请求、merge 等操作，以保证节点内存的安全性。

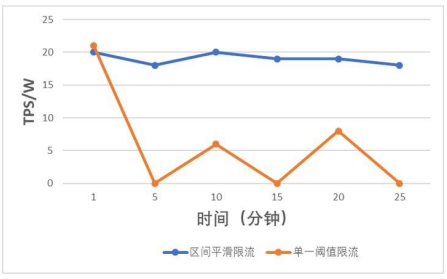
可用性优化场景一：高并发请求压垮集群

挑战：单一阈值限流容易导致请求抖动，如何才能实现平滑限流？

- 设置高低阈值限流区间
- 基于余弦变换实现区间平滑限流



平移缩放余弦函数变换得出目标函数，再分段采样



单一阈值限流易出现抖动，区间平滑限流性能平稳

限流方案里面有一个挑战是，我们如何才能实现平滑限流？因为采用单一的阈值限流很容易出现请求抖动，例如请求一上来把内存打上去马上触发限流，而放开一点点请求又会涌进来把内存打上去。我们的方案是设置了高低限流阈值区间，在这个区间中，基于余弦变换实现请求数和内存资源之间的平滑限流。当内存资源足够的时候，请求通过率 100%，当内存到达限流区间逐步上升的时候，请求通过率随

之逐步下降。而当内存使用量下降的时候，请求通过率也会逐步上升，不会一把放开。通过实际测试，平滑的区间限流能在高压力下保持稳定的写入性能。

我们基于内存资源的区间平滑限流策略是对原生版本基于请求数漏桶策略的有效补充，并且作用范围更广，覆盖协调节点、数据节点的接入层和传输层，并不会替代原生的限流方案。

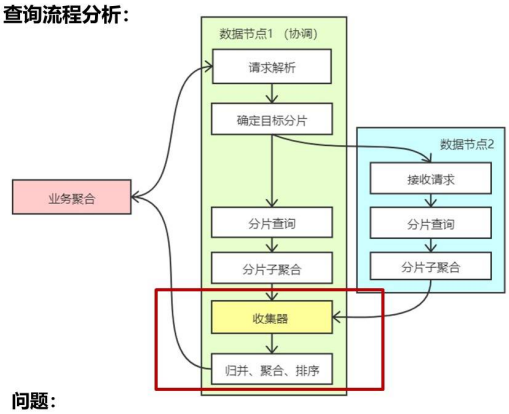
可用性优化场景二：单个大查询打挂节点

场景分析

两层嵌套聚合，结果百万级

```
{
  "aggs": {
    "dimension1": {
      "terms": {
        "field": "user",
        "size": 1000000
      },
      "aggs": {
        "dimension2": {
          "terms": {
            "field": "domain",
            "size": 1000000
          },
          "aggs": {
            "top1": {
              "top_hits": {
                "size": 1
              }
            }
          }
        }
      }
    }
  }
}
```

查询流程分析：



问题：

- 大量汇聚结果反序列化后内存膨胀
- 二次汇聚产生新的结果集打爆内存

接下来介绍单个大查询打挂节点的场景。例如我们在分析场景，做多层嵌套聚合，有时候请求返回的结果集比较大，那么这个时候极有可能这一个请求就会将节点打挂。我们对聚合查询流程进行分析，请求到达协调节点之后，会拆分为分片级子查询请求给目标分片所在数据节点进行子聚合，最后协调节点收集到完整的分片结果后进行归并、聚合、排序等操作。这里的主要问题点是，协调节点大量汇聚结果反序列化后内存膨胀，以及二次聚合产生新的结果集打爆内存。

可用性优化场景二：单个大查询打挂节点

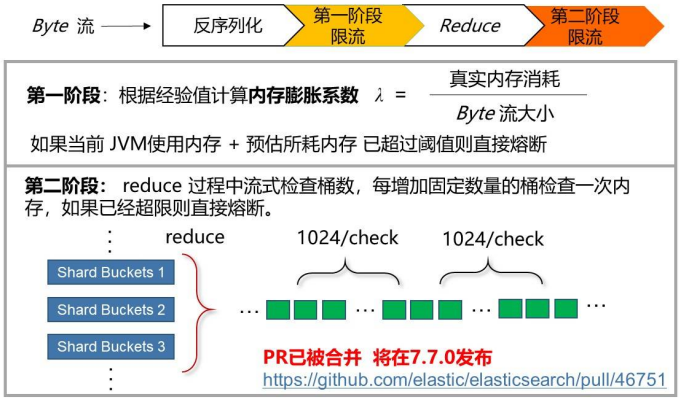
优化方案 - 内存膨胀预估 + 流式检查

原生方案：限制最大返回结果桶数

```
ElasticsearchStatusException[Elasticsearch exception
[type=search_phase_execution_exception, reason=]]; nested:
ElasticsearchException[Elasticsearch exception
[type=too_many_buckets_exception, reason=Trying to create too many buckets. Must be less than or equal to: [10000] but was [10001]. This limit can be set by changing the [search.max_buckets] cluster level setting.]];
```

挑战：

- 分析场景结果桶数十万、百万是常态，默认1万不够
- 调整不灵活，大了内存会崩掉，小了满足不了业务需求



优点：用户无需关心最大结果桶数，只要内存足够就可以最大化满足业务需求

不足：大请求被拒绝，牺牲了用户的查询体验（batch reduce 缓解）

针对上面单个大查询的问题，下面介绍我们的优化方案。优化方案的要点是内存膨胀预估加流式检查。我们先来看下原生方案，原生版本是直接限制最大返回结果桶数，默认一万，超过则请求返回异常。这种方式面临的挑战是，在分析场景结果数十万、百万是常态，默认一万往往不够，调整不灵活，调大了内存可能还是会崩掉，小了又不能满足业务需求。

我们的优化方案主要分为两个阶段：

- 第一阶段：在协调节点接收数据节点返回的响应结果反序列化之前做内存膨胀预估，基于接收到的网络 byte 流大小做膨胀预估，如果当前 JVM 内存使用量加上响应结果预估的使用量超过阈值则直接熔断请求。
- 第二阶段：在协调节点 reduce 过程中，流式检查桶数，每增加固定数量的桶（默认 1024 个）检查一次内存，如果超限则直接熔断。流式检查的逻辑在数据节点子聚合的过程同样生效。

这样用户不再需要关心最大桶数，只要内存足够就能最大化地满足业务需求。不足之处是大请求还是被拒掉了，牺牲了用户的查询体验，但是我们可以通过官方已有的 batch reduce 的方式缓解，就是当有 100 个分片子结果的时候，每收到部分就先做一次聚合，这样能降低单次聚合的内存开销。上面流式聚合的整体方案已经提交给官方并合并了，将在最近的 7.7.0 版本中发布。

前面介绍了两种比较典型的用户常遇到的可用性问题。接下来对整个可用性优化做一个总结。



首先我们结合自研的优化方案和原生的方案实现了系统性的全链路限流。左图中黄色部分为自研优化，其它为原生方案。覆盖执行引擎层、传输层和接入层。另外我们对内存也做了相关的优化，内存利用率优化主要是针对写入场景，例如单条文档字段数过多上千个，每个字段值在写入过程中都会申请固定大小的 buffer，字段数过多的时候内存浪费严重，优化方案主要是实现弹性的内存 buffer。内存回收策略，这里不是指 GC 策略，主要是对于有些例如读写异常的请求及时进行内存回收。JVM GC 债务管理主要是评估 JVM Old GC 时常和正常工作时常的比例来衡量 JVM 的健康情况，特殊情况会重启 JVM 以防止长时间 hang 死。

可用性优化效果，我们将公有云的 ES 集群整体可用性提升至 4 个 9，内存利用率提升 30%，高压力场景稳定性有大幅提升，基本能保证节点不会 OOM，集群不会雪崩。

下面部分是我们可用性优化相关的 PR。除了前面介绍的协调节点流式检查和内存膨胀预估以外，还包括单个查询内存限制，这个也很有用，因为有些场景如果单个查询太大会影响其它所有的请求。以及滚动重启速度优化，大集群单个节点的重启时间从 10 分钟降至 1 分钟以内，这个优化在 7.5 版本已经被合并了。如果大家遇到大集群滚动重启效率问题可以关注。

接下来介绍性能优化。

性能优化

场景分析

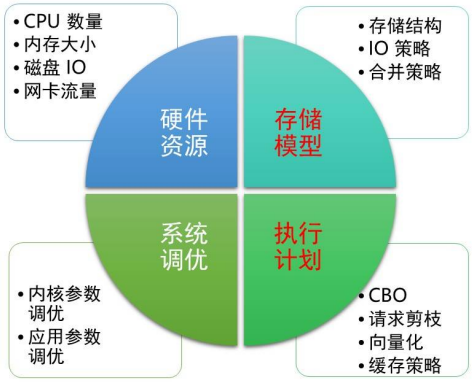
写入

- 代表场景：日志、监控等海量时序数据
- 高并发：每秒千万级吞吐，
  带id写入性能衰减1倍

查询

- 搜索服务：高并发、低延时，十万级 QPS，
  平响20ms，P95<100ms
- 聚合分析：大查询，内存、CPU开销高

性能影响面



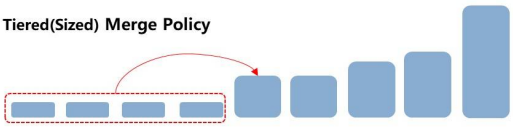
性能优化的场景主要分为写入和查询。写入的代表场景包括日志、监控等海量时序数据场景，一般能达到千万级吞吐。带 id 的写入性能衰减一倍，因为先要查询记录是否存在。查询包含搜索场景和分析场景，搜索服务主要是高并发，低延时。聚合分析主要以大查询为主，内存、CPU开销高。

我们看下性能的影响面，左半部分硬件资源和系统调优一般是用户可以直接掌控的，比如资源不够扩容，参数深度调优等。右半部分存储模型和执行计划涉及到内核优化，用户一般不容易直接调整。接下来我们重点介绍一下这两部分的优化。

性能优化 – 存储模型

ES底层Lucene基于LSM Tree的数据文件

Tiered(Sized) Merge Policy



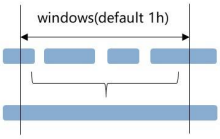
原生合并策略：

- 按文件大小相似性合并
- 默认一次固定合并10个文件

优点： 合并高效，快速降低文件数
不足： 数据不连续，文件裁剪能力弱

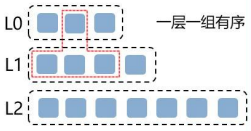
业内典型的合并策略：

Time-Window Merge Policy



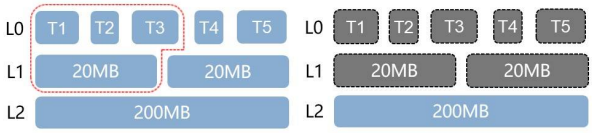
典型产品： Cassandra、HBase
优点： 数据按时间序合并，支持表内TTL
不足： 只适合时序场景；文件大小可能不一致，影响合并效率

Leveled Merge Policy



典型产品： LevelDB，RocksDB
优点： 查询高效
不足： 写放大较严重，影响TPS

优化方案：提升数据连续性、收敛文件数量，提升文件裁剪能力



数据连续性：按时间序分层合并

- **文件之间按创建时间排序**
- 除L0层外**每层按目标文件大小合并**，不固定单次合并文件数量
- **不足：** 因大小不足有少量未合并文件

文件收敛：冷分片持续合并

- **持续合并**超过5min不写入的分片
- 控制合并并发
- 控制合并范围，过冷数据不再合并

优化效果：搜索场景查询性能提升 40%

首先是存储模型优化。我们知道 ES 底层 Lucene 是基于 LSM Tree 的数据文件。原生默认的合并策略是按文件大小相似性合并，默认一次固定合并 10 个文件，近似分层合并。这种合并方式的最大优点是合并高效，可以快速降低文件数；主要问题是数据不连续，这样会导致我们在查询的时候文件裁剪的能力很弱，比如查询最近一小时的数据，很有可能一小时的文件被分别合并到了几天前的文件中去了，导致需要遍历的文件增加了。

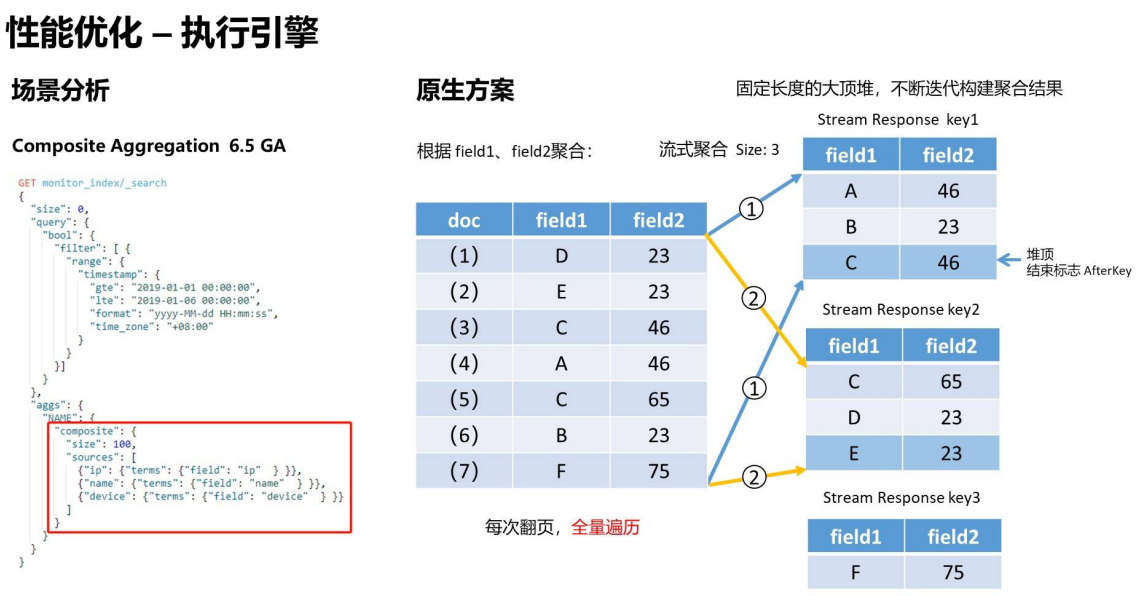
业内典型的解决数据连续性的合并策略，比如以 Cassandra、HBase 为代表的基于时间窗口的合并策略，优点是数据按时间序合并，查询高效，且可以支持表内 TTL；不足是限制只能是时序场景，而且文件大小可能不一致，从而影响合并效率。还有一类是以 LevelDB、RocksDB 为代表的分层合并，一层一组有序，每次抽取部分数据向下层合并，优点是查询高效，但是写放大比较严重，相同的数据可能会

被多次合并，影响写入吞吐。

最后是我们的优化合并策略。我们的目标是为了提升数据连续性、收敛文件数量，提升文件的裁剪能力来提高查询性能。我们实现的策略主要是按时间序分层合并，每层文件之间按创建时间排序，除了第一层外，都按照时间序和目标大小进行合并，不固定每次合并文件数量，这样保证了合并的高效性。对于少量的未合并的文件以及冷分片文件，我们采用持续合并的策略，将超过默认五分钟不再写入的分片进行持续合并，并控制合并并发和范围，以降低合并开销。

通过对合并策略的优化，我们将搜索场景的查询性能提升了 40%。

前面介绍了底层文件的存储模型优化，我们再来向上层看看执行引擎的优化。



我们拿一个典型的场景来进行分析。ES 里面有一种聚合叫 Composite 聚合大家可能都比较了解，这个功能是在 6.5 版本正式 GA 发布的。它的目的是为了支持多字段的嵌套聚合，类似 MySQL 的 group by 多个字段；另外可以支持流式聚合，即以翻页的形式分批聚合结果。用法就像左边贴的查询时聚合操作下面指定 composite 关键字，并指定一次翻页的长度，和 group by 的字段列表。那么每次拿到的聚合结果会伴随着一个 after key 返回，下一次查询拿着这个 after key 就可以查询下一页的结果。

那么它的实现原理是怎样的呢？我们先来看看原生的方案。比如这里有两个字段的文档，field1 和 field2，第一列是文档 id 。我们按照这两个字段进行 composite 聚合，并设定一次翻页的 size 是 3。具体实现是利用一个固定 size 的大顶堆，size 就是翻页的长度，全量遍历一把所有文档迭代构建这个基于大顶堆的聚合结果，如右图中的 1 号序列所示，最后返回这个大顶堆并将堆顶作为 after key。第二次聚合的时候，同样的全量遍历一把文档，但会加上过滤条件排除不符合 after key 的文档，如右图中 2 号序列所示。

很显然这里面存在性能问题，因为每次拉取结果都需要全量遍历一遍所有文档，并未实现真正的翻页。接下来我们提出优化方案。

性能优化 – 执行引擎

优化方案：Index Sorting + AfterKey 跳转 + Early Termination

```
{
  "monitor_index": {
    "settings": {
      "index": {
        "refresh_interval": "30s",
        "translog": {
          "sync_interval": "10s",
          "durability": "async"
        },
        "sort": {
          "field": [
            "ip",
            "name",
            "device"
          ],
          "order": [
            "asc",
            "asc",
            "asc"
          ]
        },
        "number_of_replicas": "1",
        "number_of_shards": "6"
      }
    }
  }
}
```

写入时基于group key进行排序 6.5 GA

doc	field1	field2
(1)	A	46
(2)	B	23
(3)	C	46
(4)	C	65
(5)	D	23
(6)	E	23
(7)	F	75

1.提前结束

2.AfterKey跳转

3. 提前结束

Stream Response key1

field1	field2
A	46
B	23
C	46

结束标志 AfterKey

Stream Response key2

field1	field2
C	65
D	23
E	23

Stream Response key3

field1	field2
F	75

优化效果：聚合查询性能提升 3 – 7 倍（PR已合并）

https://github.com/elastic/elasticsearch/pull/48399 **7.6.0**

https://github.com/elastic/elasticsearch/pull/48130

我们的优化方案主要是利用 index sorting 实现 after key 跳转以及提前结束（early termination）。数据有序才能实现真正的流式聚合，index sorting 也是在 6.5 版本里面引入的，可以支持文档按指定字段排序。但遗憾的是聚合查询并没有利用数据有序性。我们可以进行优化，此时大顶堆我们仍然保留，我们只需要按照文档的顺序提取指定 size 的文档数即可马上返回，因为数据有序。下一次聚合的时候，我们可以直接根据请求携带的 after key 做跳转，直接跳转到指定位置继续向后遍历指定 size 的文档数即可返回。这样避免了每次翻页全量遍历，大幅提升查询性能。这里有一个挑战点，假设数据的顺序和用户查询的顺序不一致优化还能生效吗？实际可以的，逆序场景不能实现 after key 跳转因为 lucene 底层不能支持文档反向遍历，但提前结束的优化仍然生效，仍然可以大幅提升效率。这个优化方案我们是和官方研发协作开发的，因为我们在优化的同时，官方也在优化，但我们考虑的更全面覆盖了数据顺序和请求顺序不一致的优化场景，最终我们和官方一起将方案进行了整合。该优化方案已经在 7.6 合并，大家可以试用体验。

前面从底层的存储模型到上层的执行引擎分别举例剖析了优化，实际上我们在性能层面还做了很多的优化。从底层的存储模型到执行引擎，到优化器，到上层的缓存策略基本都有覆盖。下图中左边是优化项，中间是优化效果，右边是有代表性的优化的 PR 列表。

性能优化总结

系统性性能优化

缓存策略	开销/命中率	GC 策略
优化器	CBO	RBO
执行引擎	查询剪枝	IO 重排
存储引擎	合并策略	文件裁剪

优化效果

- 查询毛刺 **750ms ->50ms**
- FST OffHeap, Tencent Kona JDK

- 点查询加范围查询性能提升**一倍**
- 内置通用的基于规则优化的模板

- 带排序场景聚合性能提升**3-7倍**
- 顺序抓取查询性能**10%**

- 搜索场景查询性能提升**40%**
- 带主键的写入性能提升**一倍**

性能优化相关的PR：

聚合查询优化（ES 7.6.0）

https://github.com/elastic/elasticsearch/pull/48399

Translog刷新优化（ES 7.4.0）

https://github.com/elastic/elasticsearch/pull/45765

https://github.com/elastic/elasticsearch/pull/47790

文件裁剪优化（Lucene 8.3.0）

https://github.com/apache/lucene-solr/pull/884

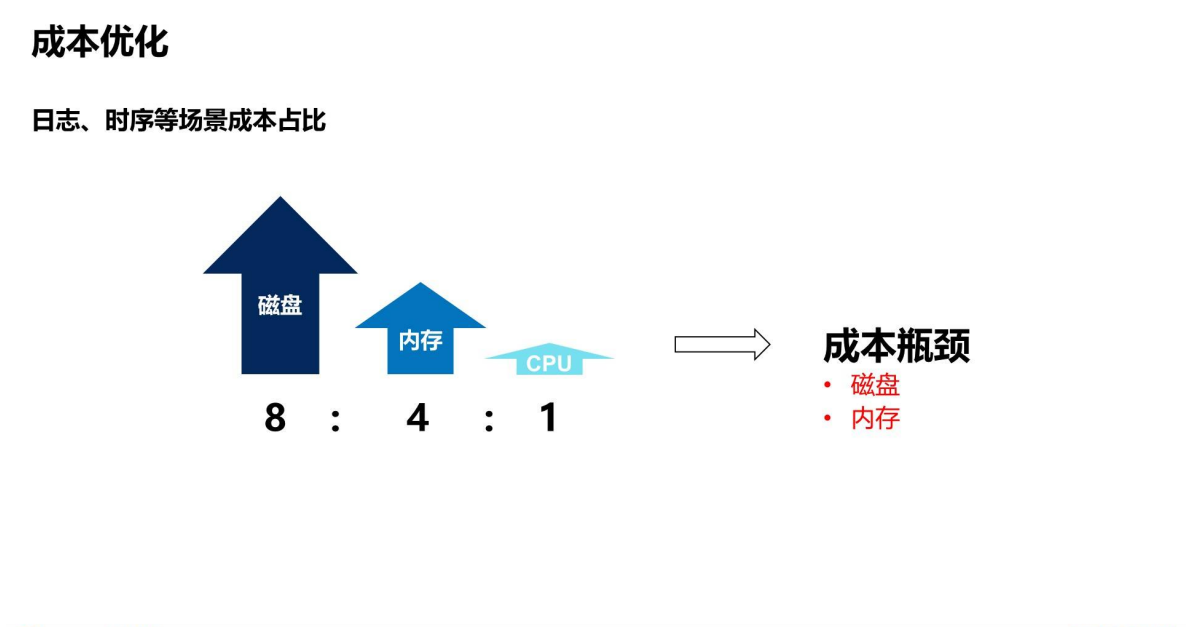
缓存策略优化 (Lucene 8.4.0)

https://github.com/apache/lucene-solr/pull/940

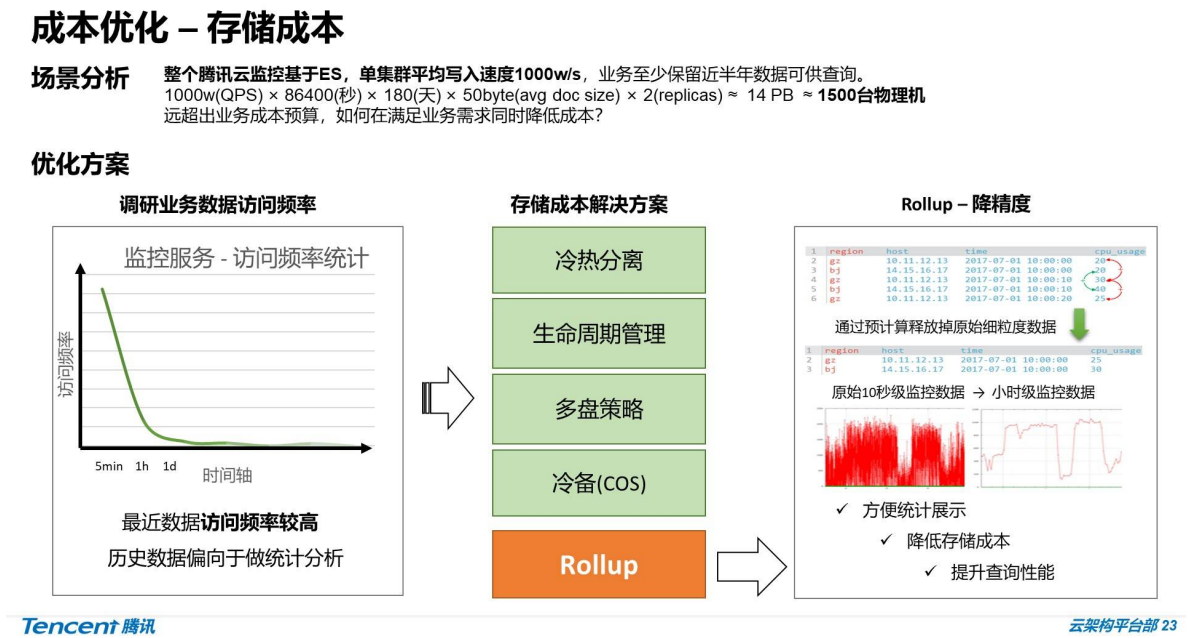
这里简单再介绍一下其它的 PR 优化，中间这个 translog 刷新过程中锁的粗化优化能将整体写入性能提升 20%；这个 lucene 层面的文件裁剪优化，它能将带 id 写入场景性能提升一倍，当然查询也是，因为带 id 的写入需要先根据 id 查询文档是否存在，它的优化主要是在根据 id 准备遍历查询一个 segment 文件的时候，能快速根据这个 segment 所统计的最大最小值进行裁剪，如果不在范围则快速裁剪跳过，避免遍历文档；最下面的一个 PR 是缓存策略的优化，能避免一些开销比较大的缓存，大幅的降低查询毛刺。

上面这些性能优化项在我们腾讯云的 ES 版本中均有合入，大家可以试用体验。

接下来我们再看成本优化。在日志、时序等大规模数据场景下，集群的 CPU、内存、磁盘的成本占比是 1 比 4 比 8。例如一般 16 核 64GB，2-5 TB 磁盘节点的成本占比大概是这个比例。因此成本的主要瓶颈在于磁盘和内存。



成本优化的主要目标是存储成本和内存成本，我们先来看下存储成本。



我们先来看一个场景，整个腾讯云监控是基于 ES 的，单个集群平均写入千万每秒，业务需要保留至少半年的数据供查询。我们按照这个吞吐来计算成本，1000万 QPS 乘以时间乘以单条文档平均大小再乘以主从两个副本总共大约 14 PB 存储，大约需要 1500 台热机型物理机。这显然远远超出了业务成本预算，那我们如何才能既满足业务需求又能实现低成本呢?

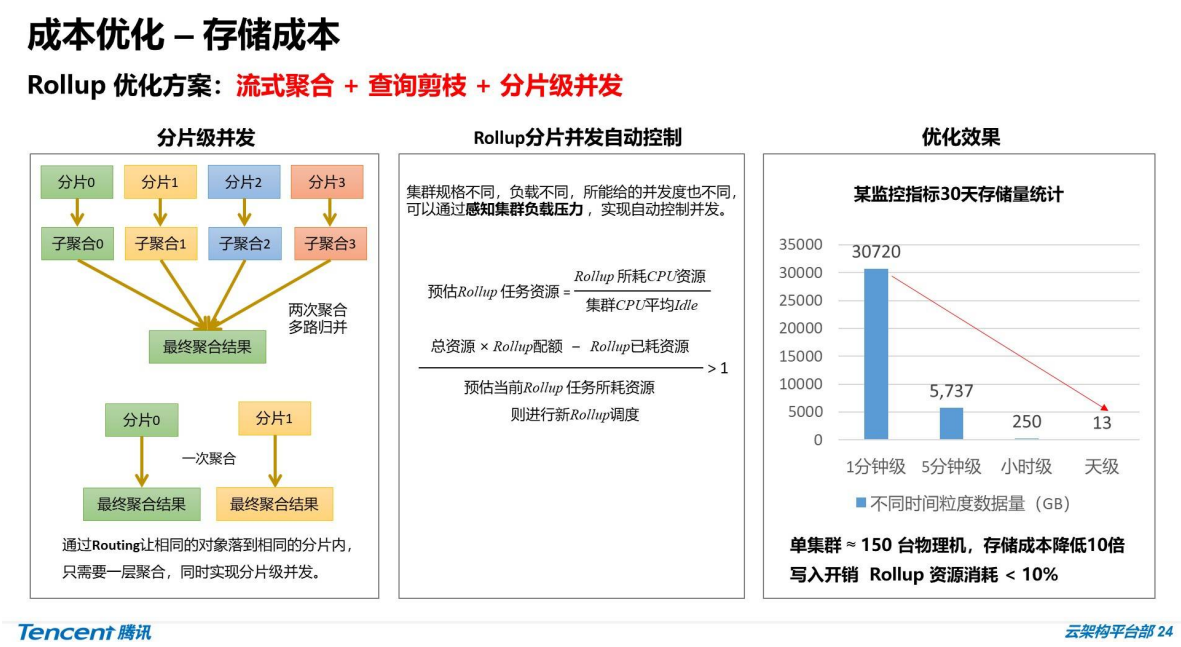
来看下我们的优化方案，首先我们对业务数据访问频率进行调研，发现最近的数据访问频率较高，例如最近 5 分钟的，一小时的，一天的，几天的就比较少了，超过一个月的就更少了，历史数据偏向于统计分析。

首先我们可以通过冷热分离，把冷数据放到 HDD 来降低成本，同时利用官方提供的索引生命周期管理来搬迁数据，冷数据盘一般比较大我们还要利用多盘策略来提高吞吐和数据容灾能力。最后将超冷的数据冷备到腾讯云的对象存储 COS 上，冷备成本非常低，1GB 一个月才一毛多。

上面这些我们都可以从架构层面进行优化。是否还有其它优化点呢？基于前面分析的数据访问特征，历史数据偏向统计分析，我们提出了 Rollup 方案。Rollup 的目的是对历史数据降低精度，来大幅降低存储成本。我们通过预计算来释放原始细粒度的数据，例如秒级的数据聚合成小时级，小时级聚合成天级。这样对于用户查询时间较长的跨度报表方便展示，查询几天的秒级数据太细没法看。另外可以大幅降低存储成本，同时可以提升查询性能。

我们在 17 年的时候就实现了 Rollup 的方案并投入给了腾讯云监控使用，当然目前官方也出了 Rollup 方案，目前功能还在体验中。

下面介绍一下我们最新的 Rollup 方案的要点。



总体来说 Rollup 优化方案主要是基于流式聚合加查询剪枝结合分片级并发来实现其高效性。流式聚合和查询剪枝的优化我们前面在性能优化部分已经介绍了，我们新的 Rollup 也利用了这些优化，这里不再展开。下面介绍一下分片级并发，及并发自动控制策略。

正常的聚合查询，需要将请求发送给每个分片进行子聚合，在到协调节点做汇聚，两次聚合多路归并。我们通过给数据添加 routing 的方式让相同的对象落到相同的分片内，这样就只需要一层聚合，因为分片数据独立，多个数据对象可以实现分片级并发。 另外我们通过对 Rollup 任务资源预估，并感知集群的负载压力来自动控制并发度，这样对集群整体的影响能控制在一定的范围。右边的图是我们的优化效果，某个统计指标 30 天的存储量，天级的只需要 13 GB，小时级的只需要 250 GB，细粒度的会多一些，总体存储量下降了将近 10 倍。单个集群 150 台左右物理机即可搞定，成本缩减 10 倍。整体写入开销 rollup 资源消耗在 10% 以下。

前面是存储成本优化，下面介绍内存成本优化。

成本优化 – 内存成本

内存成本分析

某日志集群	磁盘总量	磁盘使用率	内存总量	堆内存总量	堆内存使用率	FST内存
name	disk.total	disk.used_percent	ram.max	heap.max	heap.percent	fst
node-1	42.9tb	33.05	62.2gb	31.8gb	72	16.8gb
node-2	42.9tb	32.80	62.5gb	31.8gb	67	16.8gb
node-3	35.8tb	38.64	62.2gb	31.8gb	74	16.6gb

线上集群分析：

- 堆内存使用率很高
- 而磁盘使用率却比较低

堆内存使用率为什么会高?

- FST 占用绝大部分堆内存：50% ~ 70%
- FST 与数据量成正比：10TB 磁盘： 10GB ~ 15GB FST

我们通过对线上集群进行分析，发现很多场景堆内内存使用率很高，而磁盘的使用率比较低。堆内存使用率为什么这么高呢？其中的 FST 即倒排索引占据了绝大部分堆内内存，而且这部分是常驻内存的。每 10 TB 的磁盘 FST 的内存消耗大概在 10 GB 到 15 GB 左右。

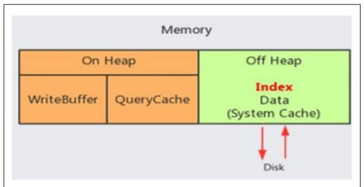
我们能不能对 FST 这种堆内占用比较大的内存做优化？我们的想法是把它移至堆外（off-heap），按需加载，提升堆内内存利用率，降低 GC 开销，提升单个节点管理磁盘的能力。

成本优化 – 内存成本

方案调研

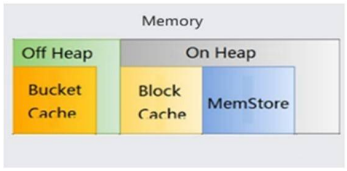
将 FST 从堆内移到堆外（OffHeap）

原生方案



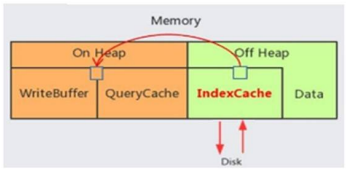
原理： 依赖系统缓存，按需加载文件
优点： 实现简单
不足： 淘汰策略由系统控制，海量数据场景性能N倍损耗

HBase 2.0+



原理： 借助堆外内存，释放堆内存
优点： 数据缓存移至堆外，释放堆内存
不足： 淘汰策略完全依赖LRU，索引依然占据堆内存

优化方案



原理： 借助堆外内存，释放堆内存
优点： Cache保障FST空间，精准淘汰策略提高内存使用率，多级Cache提高性能
不足： 堆外内存管理开发成本高

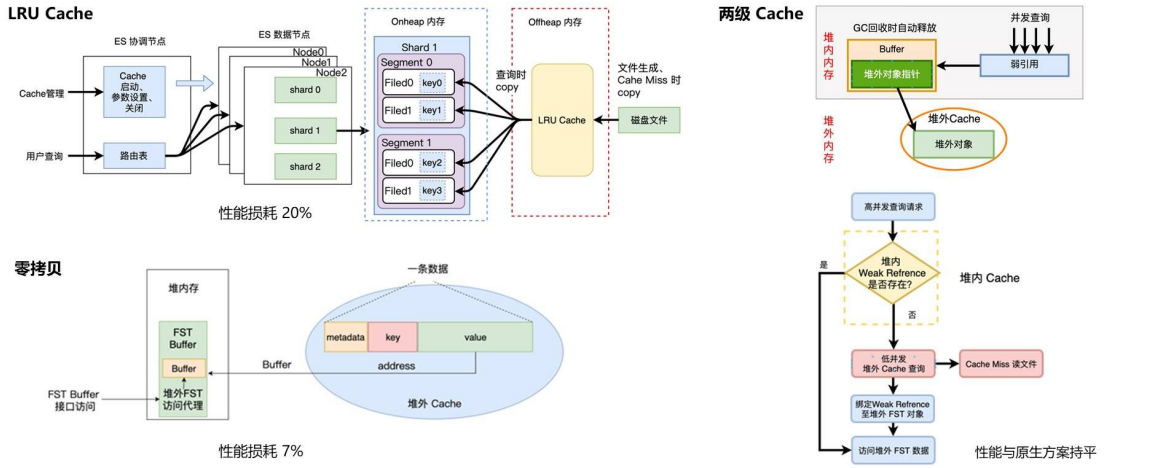
我们来看下 off-heap 相关的方案。首先原生版本目前也实现了 off-heap，方案是将 FST 对象放到 MMAP 中管理，这种方式实现简单，我们早期也采用了这种方式实现，但是由于 MMAP 属于 page cache 可能被系统回收掉，导致读盘操作，从而带来性能的 N 倍损耗，容易产生查询毛刺。

HBase 2.0 版本中也实现了 off-heap，在堆外建立了 cache，脱离系统缓存，但只是把数据放到堆外，索引仍然在堆内，而且淘汰策略完全依赖 LRU 策略，冷数据不能及时的清理。

我们的优化方案也是在堆外建立 cache，保证 FST 的空间不受系统影响，另外我们会实现更精准的淘汰策略，提高内存使用率，再加上多级 cache 的管理模式来提升性能。这种方式实现起来比较复杂但收益还是很明显的，下面我们来看一下详细的实现。

成本优化 – 内存成本

优化方案：LRU Cache + 零拷贝 + 两级 Cache



我们的方案是通过 LRU cache + 零拷贝 + 两级 cache 的方式实现的。首先 LRU cache 是建立在堆外，堆内有访问 FST 需求的时候从磁盘加载到 cache 中。由于 Lucene 默认的访问 FST 的方式是一个堆内的 buffer，前期我们采用了直接从堆外拷贝到堆内的 buffer 方式实现，压测发现查询性能损耗 20%，主要是堆外向堆内 copy 占了大头。

因此我们有了第二阶段优化，将 Lucene 访问 FST 的方式进行了改造，buffer 里面不直接存放 FST，而存放堆外对象的一个指针，这样实现了堆内和堆外之间的零拷贝，这里的零拷贝和我们说的 linux 中的用户态和内核态的零拷贝是两个概念。这样实现后我们压测发现查询性能还是有7%的损耗，相较于堆内的 FST 场景。我们有没办法做到极致呢?

我们通过分析发现，这 7% 的损耗主要是根据 key 去查找堆外对象的过程，包括计算 hash，数据校验等。我们第三阶段的优化就是利用 Java 的弱引用建立第二层轻量级缓存。弱引用指向堆外的地址，只要有请求使用，这个 key 就不会被回收可以重复利用而无需重新获取。一旦不在使用，这个 key 就会被 GC 回收掉，并回收掉堆外对象指针。问题来了，堆外对象指针回收之后我们怎么清理堆外这部分内存呢？让其 LRU 淘汰？这样显然会浪费一部分内存空间。最好的办法是在堆内对象地址回收的时候直接回收堆外对象，但是 Java 没有析构的概念。这里我们利用了弱引用的 Reference Queue，当对象要被 GC 回收的时候会将对象指向的堆外内存清理掉，这样完美解决了堆外内存析构的问题，保证了堆外内存的精准淘汰，提升内存利用率。最后通过压测我们发现性能基本和原生方案 FST 在堆内的场景持平。

下面是内存优化相关的效果和收益：

成本优化 – 内存成本

优化效果：内存开销、数据管理能力、GC优势明显，性能持平略有优势

方案对比	FST 存放位置	FST内存占用量	单个 FST 堆内存占用量	单节点最大磁盘数据量
原生方案	堆内内存	全量存储在内存中，内存占用量大	MB 级别（源生FST数据结构）	10TB（需调优）
优化方案	堆外内存	Cache LRU 淘汰冷数据内存占用量小	100Byte左右（Cache Key 大小）	50TB

性能测试，基于官方 esrally：

写入性能对比	内存使用量（MB）	GC时长（s）	TPS	90%时延（ms）	99%时延（ms）
原生方案	402.59	20.453	198051	463.201	617.701
优化方案	102.217	18.969	201188	455.124	618.379
Diff	优74.6%	优7.26%	优1.58%	优1.74%	劣0.11%

查询性能对比	内存使用量（MB）	GC时长（s）	QPS	90%时延（ms）	99%时延（ms）
原生方案	401.806	20.107	200.057	3.96062	11.1894
优化方案	101.004	19.228	200.087	3.87805	11.2316
Diff	优74.9%	优4.37%	-	优2.00%	劣0.38%

通过我们的内存优化后，内存开销、数据管理能力、GC 优势明显，性能持平略有优势。单个 FST 堆内占用只需要 100 个 byte 左右即 cache key 的大小。单节点磁盘管理能力，32GB heap 能到 50 TB 左右，相较原生版本 5-10 TB（需要深度调优）有 10 倍的提升。利用官方提供的 esrally 进行性能压测，发现堆内存使用量有大幅的下降，GC 时长也有缩减，性能基本持平。

接下来是最后一块内核优化内容，扩展性优化。

扩展性优化

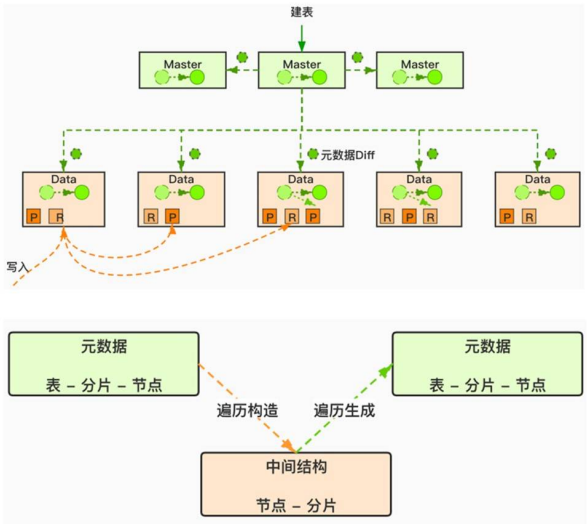
场景分析

ES元数据管理模型：master管理元数据，其它所有节点同步元数据

元数据同步流程：（以建表为例）
- Master分配分片
- Master生成临时元数据及Diff
- **Send**：发送Diff至其他节点
  - 其他节点应用Diff产生临时元数据
- **Commit**：等待大多数Master节点响应
  - 临时元数据生效
  - 其他节点对比新旧元数据执行任务

瓶颈点
- Master节点：分配分片
  - 正反向转换、遍历：均衡策略
  - 新元数据：全量拷贝
- 数据节点：创建分片
  - 全量对比新旧元数据

限制：分片数受限(3-5w)、节点数百级



我们先来看一下场景。ES 的元数据管理模型是，master 管理元数据，其它所有节点同步元数据。我们以建索引流程为例，来看看元数据的同步流程。首先是 master 分配分片，然后产生 diff 的元数据，发送给其它节点，等待大多数 master 节点返回，master 发送元数据应用请求，其它节点开始应用元数据，并根据新旧元数据推导出各自节点的分片创建任务。

这里面的瓶颈点主要有以下几点：

- Mater 节点在分配分片的时候，需要做一遍元数据的正反向转换。我们知道路由信息是由分片到节点的映射，而我们在做分片分配的时候需要节点到分片的映射，需要知道每个节点上的分片分布。分片分配完毕又需要将节点到分片的映射转换回来，因为我们元数据只发布分片到节点的映射。这个转换过程涉及多次的全量遍历，大规模分片性能存在瓶颈。
- 在每次索引创建的过程中，会涉及多次的元数据同步，在大规模的节点数场景，会出现同步瓶颈，上千节点，部分节点假设有一点网络抖动或 Old GC 可能导致同步失败。

基于上面的瓶颈，目前原生版本只能支持大约 3-5 万分片，性能已经到达极限，创建索引基本到达 30 秒+ 甚至分钟级。节点数只能到 500 左右基本是极限了。

为此，我们提出了扩展性优化方案。

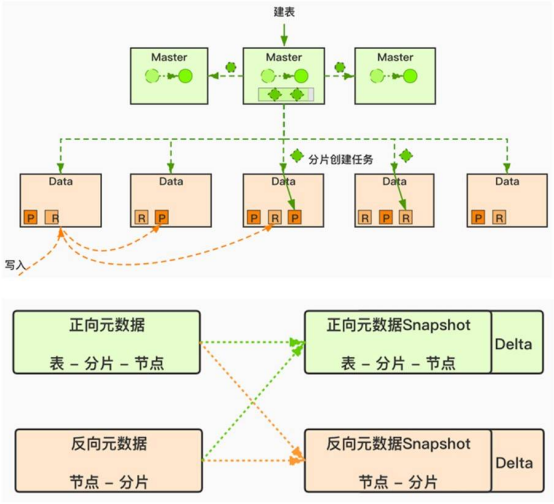
扩展性优化

优化方案

Master节点
- 创建分片任务式下发，避免多次全节点同步
- 增量化维护正反向元数据，避免拷贝、遍历

数据节点
- 依赖任务创建分片
- 周期性与元数据对账

优化结果
- 分片数达到百万级
- 节点数达到千级
- 新建索引 < 5秒
- 统计管理接口秒级



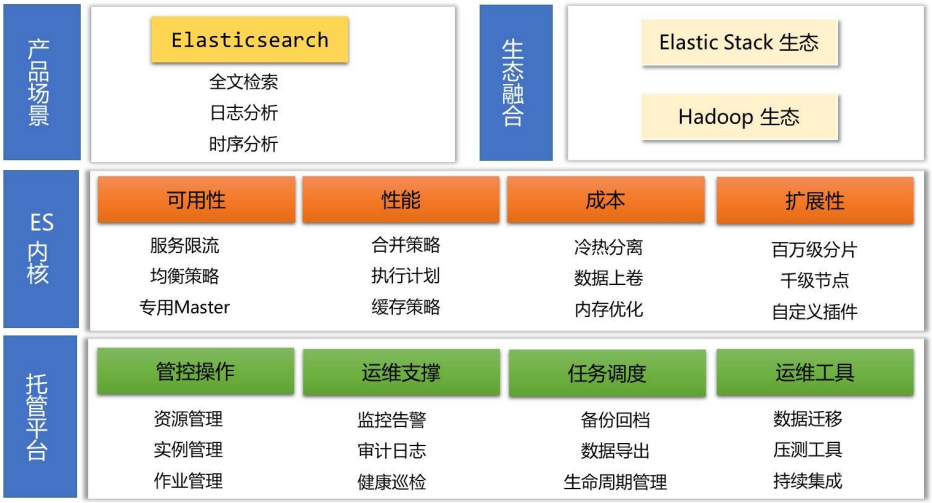
主要的优化内容包括：

- **分片创建任务式定向下发：** 对于创建分片导致的元数据同步瓶颈，我们采用任务下发的方式，定向下发分片创建任务，避免多次全节点元数据同步。
- **元数据增量维护：** 分配分片的过程中多次正反向遍历，我们采用增量化的数据结构维护的方式，避免全量的遍历。
- **统计缓存策略：** 统计接口的性能，我们采用缓存策略避免多次重复的统计计算，大幅降低资源开销。

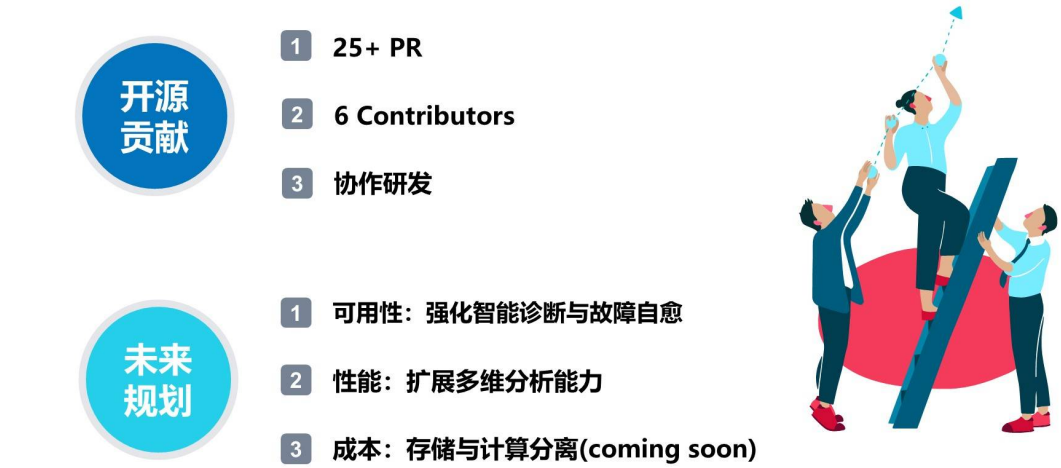
最终我们将集群的分片数扩展到百万级，节点数扩展到千级，新建索引基本稳定在 5 秒以下，统计接口秒级响应。

前面就是所有的内核优化的内容。ES 是一款很优秀的开源大数据产品，我们将持续的建设。我们对公司内外提供了完整的托管平台，对 ES 内核各个层面做了系统性的增强优化，助力 Elastic Stack 在大数据生态中覆盖更多的场景，发展的更好。

产品持续建设



四、开源贡献及未来规划



在腾讯内部，我们实现了 ES 产品开源协同，共同优化完善 ES，避免不同的团队重复踩坑。同时我们也将优秀的方案积极的贡献给社区，和官方及社区的 ES 爱好者们共同推动 ES 的发展。以腾讯 ES 内核研发为代表的团队，截至目前我们共提交了二十多个 PR，其中有 70% 被合并，共有 6 位 ES/Lucene 的 contributor。

未来，我们将持续的优化 ES，包括可用性，性能和成本等方面。可用性方面我们会加强 ES 的故障自愈能力，朝着自动驾驶的目标发展；性能方面，搜索场景 ES 是绝对的王者，多维分析领域还有很多可优化的地方，我们希望能进一步扩展 ES 在分析领域的应用场景。成本方面，存储与计算分离正在研发中，基于腾讯自研的共享文件系统 CFS，到时会进一步缩减成本，提升性能。

本次分享的内容就全部结束了，感谢大家的关注！
线上 PPT 的链接：https://elasticsearch.cn/slides/259
分享过程中相关问题的答疑：https://elasticsearch.cn/article/13768

深入理解浏览器原理

作者：龙付成，2015年加入腾讯，资深高级前端工程师。负责过QQ浏览器游戏平台、天宫活动系统、搜索页面生成引擎、领域组件库等项目。爱好技术研究、总结和分享，曾在腾讯课堂直播《Web前端安全与实践》课程。

> 本文从市面主流的浏览器及相应的内核引擎开始，介绍了Chromium为代表的浏览器架构及Blink内核的功能架构。Chromium为多进程架构，用户从启动运行浏览器后，先后经过页面导航、渲染、资源加载、样式计算、布局、绘制、合成到栅格化，最后完成GPU展示。而页面渲染完成后，浏览器如何响应页面操作事件也进行了深入的介绍。良心推荐！
>
> 本文第二至五部分内容根据 Mariko Kosaka 的英文原版《Inside look at modern web browser》(见参考文献)，进行翻译、理解、总结提炼、条理化、加入应用示例、进行相关知识补充扩展而来。

全文链接：https://mp.weixin.qq.com/s/QqpPGWf3IVEDN1t80CZ06Q

一文读懂 HTTP/1HTTP/2HTTP/3

作者：charryhuang，腾讯 CSIG 前端开发工程师

> 从 1989 年万维网（www）诞生，HTTP（HyperText Transfer Protocol）经历了众多版本迭代，WebSocket 也在期间萌芽。1991 年 HTTP0.9 被发明。1996 年出现了 HTTP1.0。2015 年 HTTP2 正式发布。2020 年 HTTP3 或能正式使用。以下将会简单介绍。

HTTP1.1 与 HTTP2

HTTP1.1 的缺陷

1. 高延迟 — 队头阻塞(Head-Of-Line Blocking)
2. 无状态特性 — 阻碍交互
3. 明文传输 — 不安全性
4. 不支持服务端推送

队头阻塞

队头阻塞是指当顺序发送的请求序列中的一个请求因为某种原因被阻塞时，在后面排队的所有请求也一并被阻塞，会导致客户端迟迟收不到数据。

针对队头阻塞：

1.将同一页面的资源分散到不同域名下，提升连接上限。虽然能公用一个 TCP 管道，但是在一个管道中同一时刻只能处理一个请求，在当前的请求没有结束之前，其他的请求只能处于阻塞状态。

2.减少请求数量

3.内联一些资源：css、base64 图片等

4.合并小文件减少资源数

无状态特性

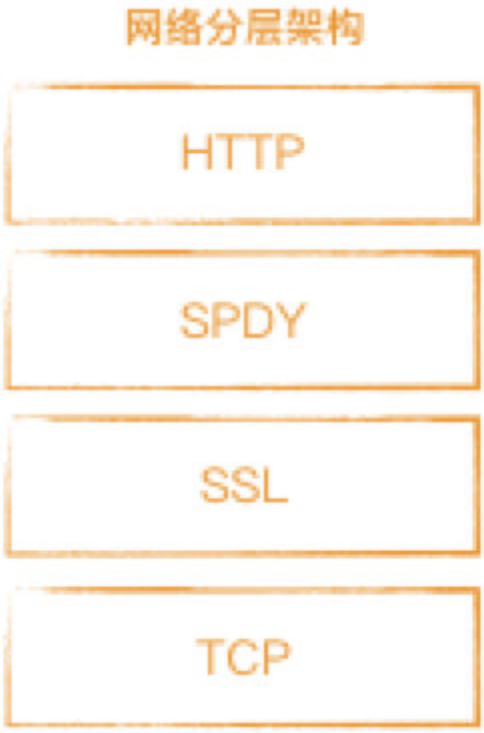
无状态是指协议对于连接状态没有**记忆能力**。纯净的 HTTP 是没有 cookie 等机制的，每一个连接都是一个新的连接。上一次请求验证了用户名密码，而下一次请求服务器并不知道它与上一条请求有何关联，换句话说就是**掉登录态**。

不安全性

传输内容没有加密，中途可能被篡改和劫持。

SPDY 协议

SPDY 是由 google 推行的改进版本的 HTTP1.1 （那时候还没有 HTTP2）。



特性：

1. 多路复用 — 解决队头阻塞
2. 头部压缩 — 解决巨大的 HTTP 头部
3. 请求优先级 — 先获取重要数据
4. 服务端推送 — 填补空缺
5. 提高安全性

多路复用

SPDY 允许在一个连接上无限制并发流。因为请求在一个通道上，TCP 效率更高（参考 TCP 拥塞控制中的**慢启动**）。更少的网络连接，发出更密集的包。

头部压缩

使用专门的 HPACK 算法，每次请求和响应只发送差异头部，一般可以达到 50%~90% 的高压缩率。

请求优先级

虽然无限的并发流解决了队头阻塞的问题，但如果带宽受限，客户端可能会因防止堵塞通道而阻止请求。在网络通道被非关键资源堵塞时，高优先级的请求会被优先处理。

服务端推送

服务端推送（ServerPush），可以让服务端主动把资源文件推送给客户端。当然客户端也有权利选择是否接收。

提高安全性

支持使用 HTTPS 进行加密传输。

HTTP2

HTTP2 基于 SPDY，专注于性能，最大的一个目标是在用户和网站间只用一个连接。

新增特性：

1. 二进制分帧 - HTTP2 性能增强的核心
2. 多路复用 - 解决串行的文件传输和连接数过多

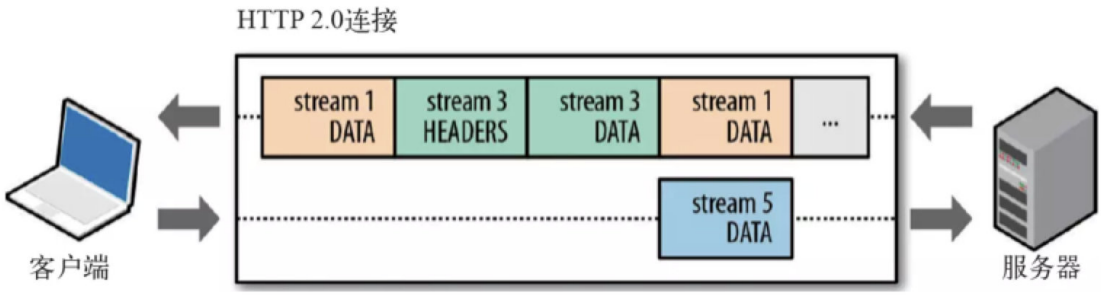
二进制分帧

首先，HTTP2 没有改变 HTTP1 的语义，只是在应用层使用二进制分帧方式传输。因此，也引入了新的通信单位：**帧、消息、流。**

分帧有什么好处？服务器单位时间接收到的请求数变多，可以提高并发数。最重要的是，为多路复用提供了底层支持。

多路复用

一个域名对应一个连接，一个流代表了一个完整的**请求-响应**过程。**帧**是最小的数据单位，每个**帧**会标识出该帧属于哪个**流**，**流**也就是多个帧组成的数据流。多路复用，就是在一个 TCP 连接中可以存在多个流。演示



HTTP2 的缺陷

1. TCP 以及 TCP+TLS 建立连接的延时
2. TCP 的队头阻塞并没有彻底解决
3. 多路复用导致服务器压力上升
4. 多路复用容易 Timeout

建连延时

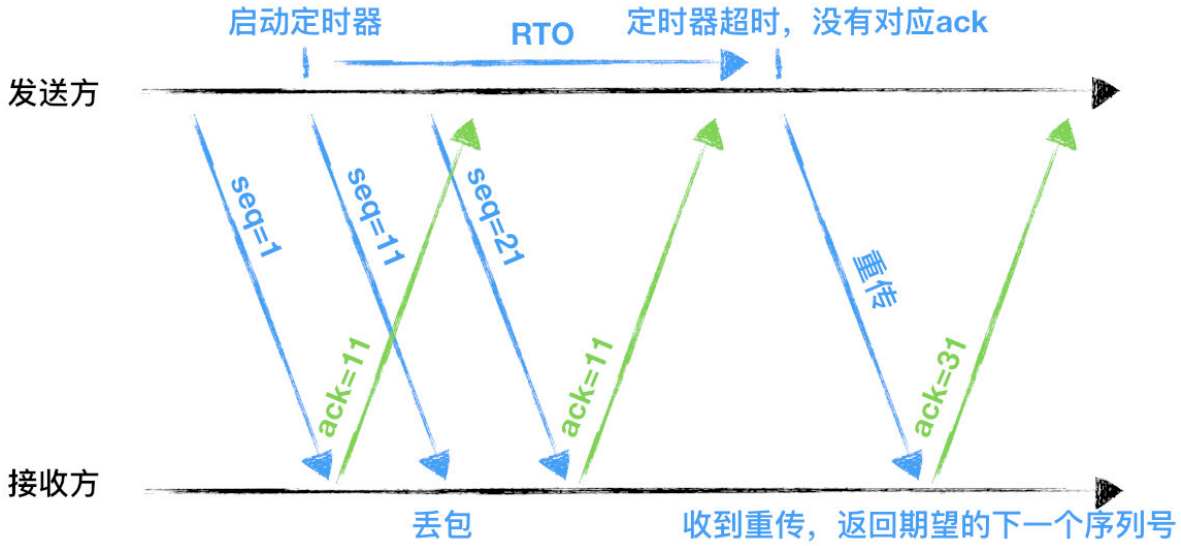
TCP 连接需要和服务器进行**三次握手**，即消耗完 1.5 个 RTT 之后才能进行数据传输。

TLS 连接有两个版本—— TLS1.2 和 TLS1.3，每个版本建立连接所花的时间不同，大致需要 1~2 个 RTT。

RTT（Round-Trip Time）:往返时延。表示从发送端发送数据开始，到发送端收到来自接收端的确认（接收端收到数据后便立即发送确认），总共经历的时延。

队头阻塞没有彻底解决

TCP 为了保证可靠传输，有一个“超时重传”机制，丢失的包必须等待重传确认。HTTP2 出现丢包时，整个 TCP 都要等待重传，那么就会阻塞该 TCP 连接中的所有请求。



RTO：英文全称是 Retransmission TimeOut，即重传超时时间。RTO 是一个动态值，会根据网络的改变而改变。RTO 是根据给定连接的往返时间 RTT 计算出来的。 接收方返回的 ack 是希望收到的下一组包的序列号。

多路复用导致服务器压力上升

多路复用没有限制同时请求数。请求的平均数量与往常相同，但实际会有许多请求的短暂爆发，导致瞬时 QPS 暴增。

多路复用容易 Timeout

大批量的请求同时发送，由于 HTTP2 连接内存在多个并行的流，而网络带宽和服务器资源有限，每个流的资源会被稀释，虽然它们开始时间相差更短，但却都可能超时。

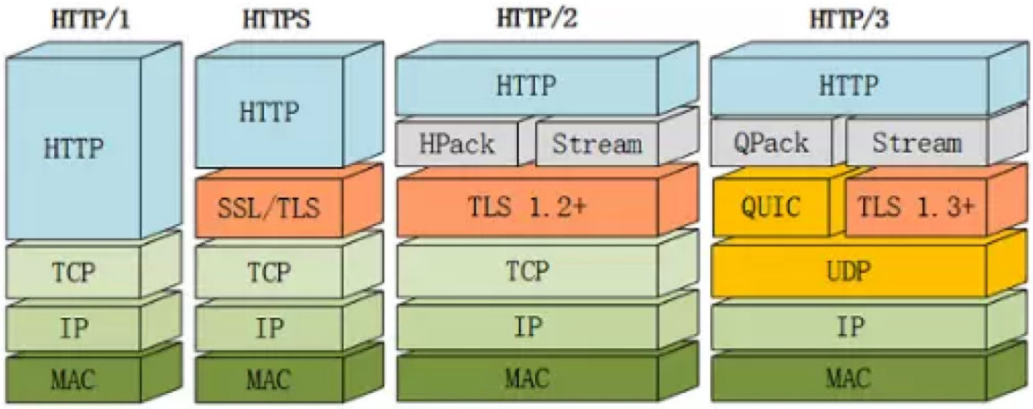
即使是使用 Nginx 这样的负载均衡器，想正确进行节流也可能很棘手。 其次，就算你向应用程序引入或调整排队机制，但一次能处理的连接也是有限的。如果对请求进行排队，还要注意在响应超时后丢弃请求，以避免浪费不必要的资源。引用

QUIC

简介

Google

在推 SPDY 的时候就已经意识到了这些问题，于是就另起炉灶搞了一个基于 UDP 协议的 QUIC 协议。而这个就是 HTTP3。它真正“完美”地解决了“队头阻塞”问题。



主要特点

1. 改进的拥塞控制、可靠传输
2. 快速握手
3. 集成了 TLS 1.3 加密
4. 多路复用
5. 连接迁移

改进的拥塞控制、可靠传输

从拥塞算法和可靠传输本身来看，QUIC 只是按照 TCP 协议重新实现了一遍，那么 QUIC 协议到底改进在哪些方面呢？主要有如下几点：

1. 可插拔 — 应用程序层面就能实现不同的拥塞控制算法。

一个应用程序的不同连接也能支持配置不同的拥塞控制。 应用程序不需要停机和升级就能实现拥塞控制的变更，可以针对不同业务，不同网络制式，甚至不同的 RTT，使用不同的拥塞控制算法。

关于应用层的可插拔拥塞控制模拟，可以对 socket 上的流为对象进行实验。

2. 单调递增的 Packet Number — 使用 Packet Number 代替了 TCP 的 seq。

每个 Packet Number 都严格递增，也就是说就算 Packet N 丢失了，重传的 Packet N 的 Packet Number 已经不是 N，而是一个比 N 大的值。而 TCP 重传策略存在二义性，比如客户端发送了一个请求，一个 RTO 后发起重传，而实际上服务器收到了第一次请求，并且响应已经在路上了，当客户端收到响应后，得出的 RTT 将会比真实 RTT 要小。当 Packet N 唯一之后，就可以计算出正确的 RTT。

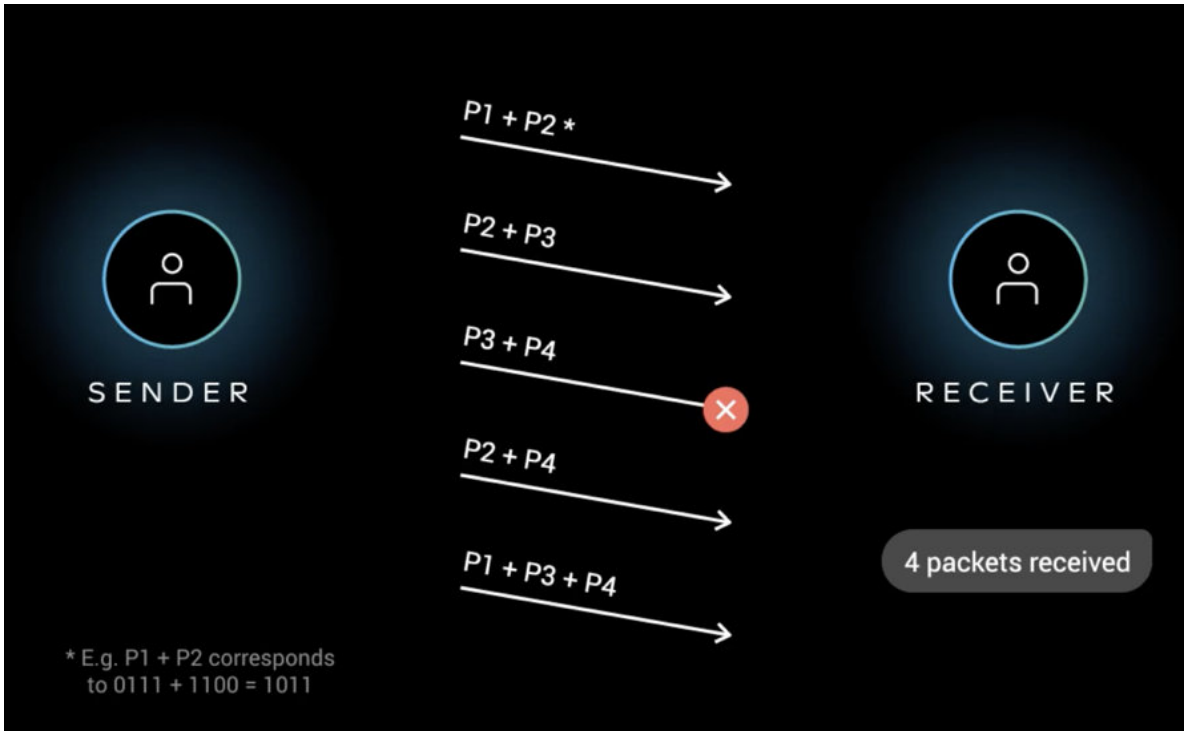
3. 不允许 Reneging — 一个 Packet 只要被 Ack，就认为它一定被正确接收。

Reneging 的意思是，接收方有权把已经报给发送端 SACK（Selective Acknowledgment） 里的数据给丢了（如接收窗口不够而丢弃乱序的包）。

QUIC 中的 ACK 包含了与 TCP 中 SACK 等价的信息，但 QUIC 不允许任何（包括被确认接受的）数据包被丢弃。这样不仅可以简化发送端与接收端的实现难度，还可以减少发送端的内存压力。

4. 前向纠错（FEC）

早期的 QUIC 版本存在一个丢包恢复机制，但后来由于增加带宽消耗和效果一般而**废弃**。FEC 中，QUIC 数据帧的数据混合原始数据和冗余数据，来确保无论到达接收端的 n 次传输内容是什么，接收端都能够恢复所有 n 个原始数据包。FEC 的实质就是异或。示意图：



5. 更多的 Ack 块和增加 Ack Delay 时间。

QUIC 可以同时提供 256 个 Ack Block，因此在重排序时，QUIC 相对于 TCP（使用 SACK）更有弹性，这也使得在**重排序**或**丢失**出现时，QUIC 可以在网络上保留更多的在途字节。在丢包率比较高的网络下，可以提升网络的恢复速度，减少重传量。

TCP 的 Timestamp 选项存在一个问题：发送方在发送报文时设置发送时间戳，接收方在确认该报文段时把时间戳字段值复制到确认报文时间戳，但是没有计算接收端接收到包到发送 Ack 的时间。这个时间可以简称为 Ack Delay，会导致 RTT 计算误差。现在就是把这个东西加进去计算 RTT 了。

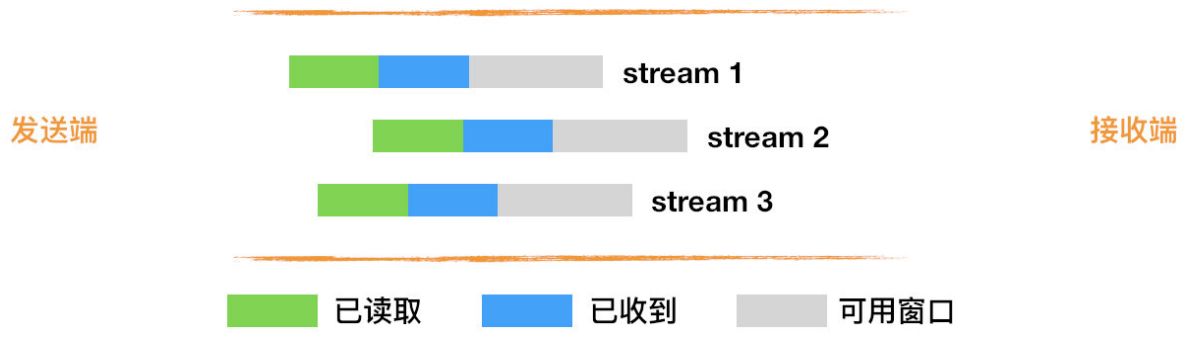
6. 基于 stream 和 connection 级别的流量控制。

为什么需要两类流量控制呢？主要是因为 QUIC 支持多路复用。 Stream 可以认为就是一条 HTTP 请求。 Connection 可以类比一条 TCP 连接。多路复用意味着在一条 Connetion 上会同时存在多条 Stream。

QUIC 接收者会通告每个流中最多想要接收到的数据的绝对字节偏移。随着数据在特定流中的发送，接收和传送，接收者发送 WINDOW_UPDATE 帧，该帧增加该流的通告偏移量限制，允许对端在该流上发送更多的数据。

除了每个流的流控制外，QUIC 还实现连接级的流控制，以限制 QUIC 接收者愿意为连接分配的总缓冲区。连接的流控制工作方式与流的流控制一样，但传送的字节和最大的接收偏移是所有流的总和。

最重要的是，我们可以在内存不足或者上游处理性能出现问题时，通过流量控制来限制传输速率，保障服务可用性。



快速握手

由于 QUIC 是基于 UDP 的，所以 QUIC 可以实现 0-RTT 或者 1-RTT 来建立连接，可以大大提升首次打开页面的速度。

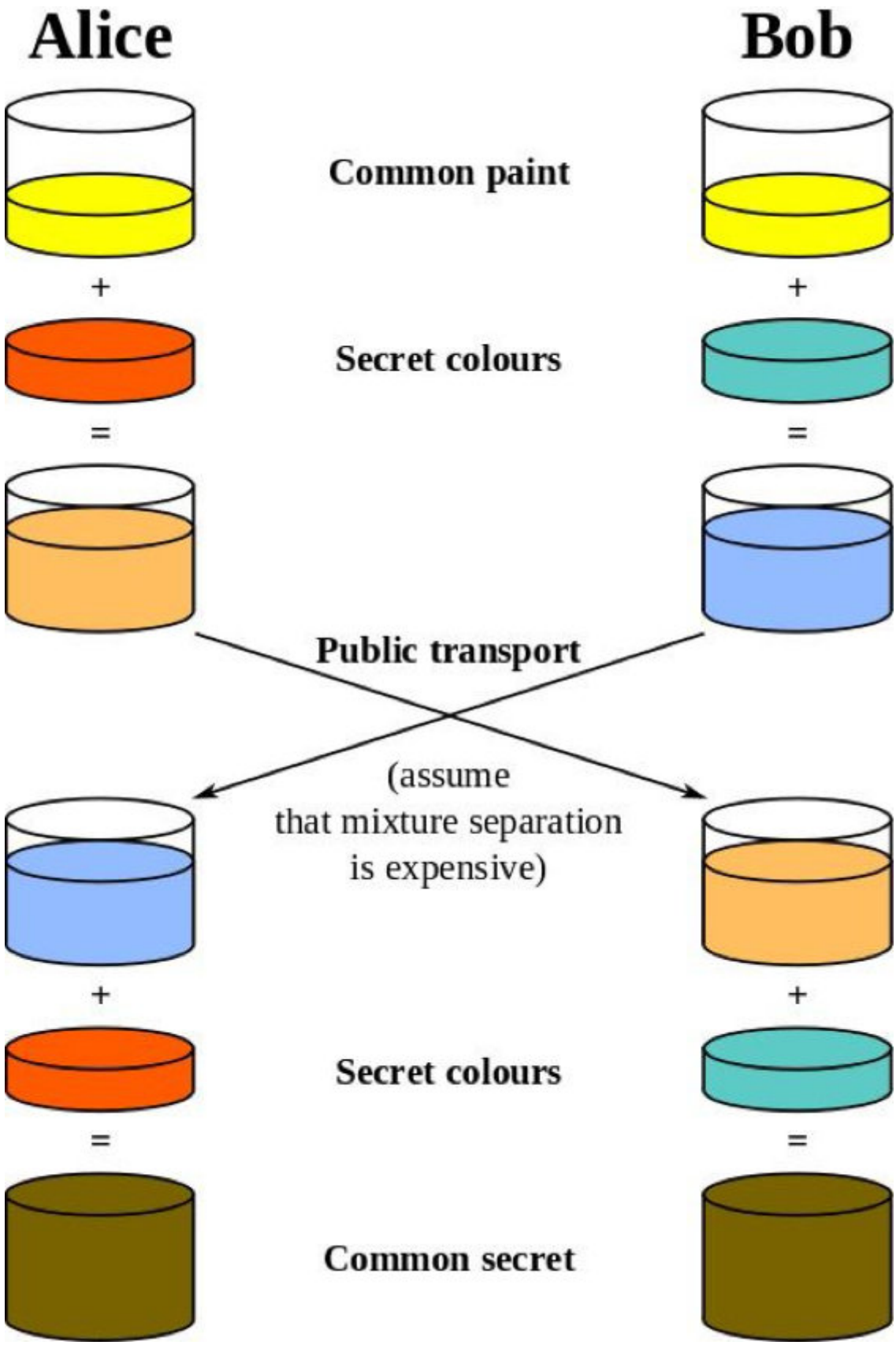
集成了 TLS 1.3 加密

TLS 1.3 支持 3 种基本密钥交换模式：

```
(EC)DHE (基于有限域或椭圆曲线的 Diffie-Hellman)
PSK - only
PSK with (EC)DHE
```

在完全握手情况下，需要 1-RTT 建立连接。 TLS1.3 恢复会话可以直接发送加密后的应用数据，不需要额外的 TLS 握手，也就是 0-RTT。

TLS 1.3 0-RTT 简单原理示意（基于 DHE）：



但是 TLS1.3 也并不完美。TLS 1.3 的 0-RTT 无法保证前向安全性(Forward secrecy)。简单讲就是，如果当攻击者通过某种手段获取到了 Session Ticket Key，那么该攻击者可以解密以前的加密数据。

要缓解该问题可以通过设置使得与 Session Ticket Key 相关的 DH 静态参数在短时间内过期（一般几个小时）。

多路复用

QUIC 是为多路复用从头设计的，携带个别流的的数据的包丢失时，通常只影响该流。QUIC 连接上的多个 stream 之间并没有依赖，也不会有底层协议限制。假如 stream2 丢了一个包，也只会影响 stream2 的处理。

连接迁移

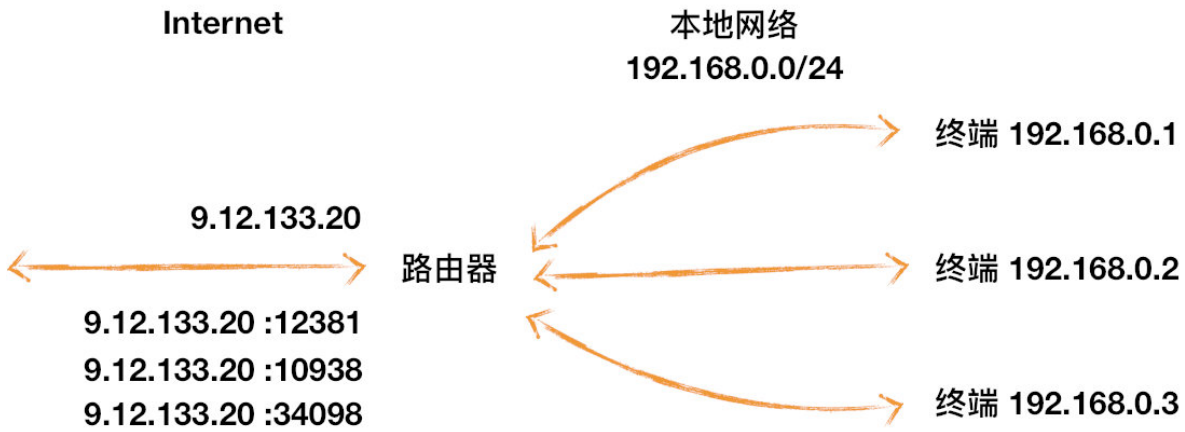
TCP 是按照 4 要素（客户端 IP、端口, 服务器 IP、端口）确定一个连接的。而 QUIC 则是让客户端生成一个 Connection ID （64 位）来区别不同连接。只要 Connection ID 不变，连接就不需要重新建立，即便是客户端的网络发生变化。由于迁移客户端继续使用相同的会话密钥来加密和解密数据包，QUIC 还提供了迁移客户端的自动加密验证。

挑战

NAT 问题

NAT 概念

为了解决 IP 地址不足的问题，NAT 给一个局域网络只分配一个 IP 地址，这个网络内的主机，则分配私有地址，这些私有地址对外是不可见的，他们对外的通信都要借助那个唯一分配的 IP 地址。所有离开本地网络去往 Internet 的数据报的源 IP 地址需替换为相同的 NAT，区别仅在于端口号不同。



原因

TCP 和 UDP 的报文头部不同导致 NAT 问题的出现。

NAT 设备的端口记忆问题

对于基于 TCP 的 HTTP、HTTPS 传输，NAT 设备可以根据 TCP 报文头的 SYN/FIN 状态位，知道通信什么时候开始，什么时候结束，对应记忆 NAT 映射的开始和结束。

但是基于 UDP 传输的 HTTP3 ，不存在 SYN/FIN 状态位。NAT 设备的记忆如果短于用户会话时间，则用户会话会中断。NAT 设备的记忆时间如果长于用户会话时间，则意味着 NAT 设备的端口资源会被白白占用。

最直接的解决方案是，在 QUIC 的头部模仿 TCP 的 SYN/FIN 状态，让沿途的 NAT 设备知道会话什么时候开始、什么时候结束。但这需要升级全球所有的 NAT 设备的软件。

另外一个可行的方案是，让 QUIC 周期性地发送 Keepalive 消息，刷新 NAT 设备的记忆，避免 NAT 设备自动释放。

NAT 设备禁用 UDP

在一些 NAT 网络环境下（如某些校园网），UDP 协议会被路由器等中间网络设备禁止，这时客户端会直接降级，选择 HTTPS 等备选通道，保证正常业务请求。

NGINX 负载均衡问题

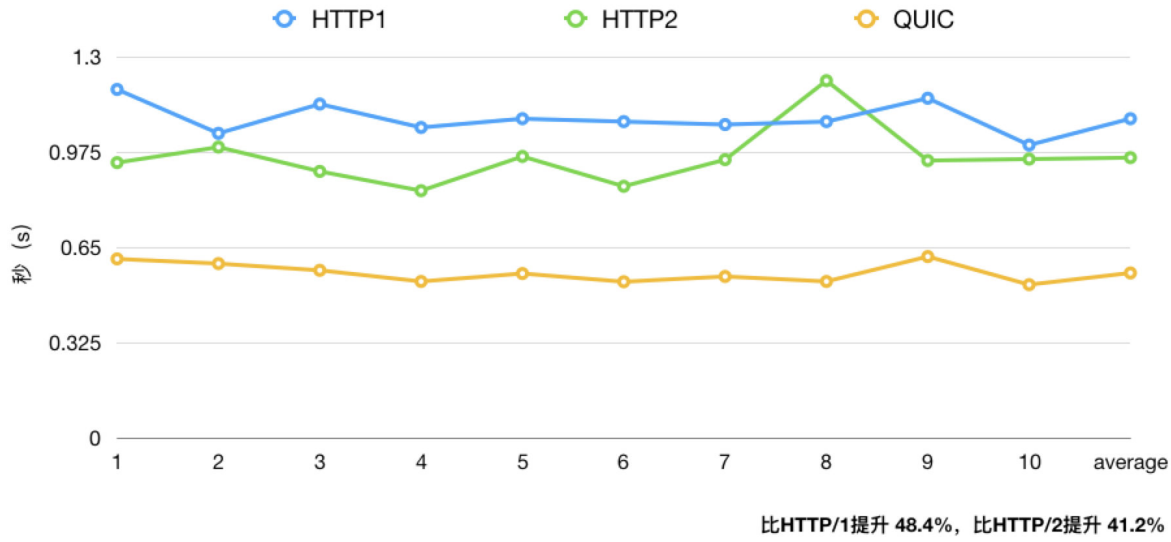
概念

QUIC 客户端存在网络制式切换，就算是同一个移动机房，可能第一次业务请求时会落到 A 这台服务器，后续再次连接，就会落到 B 实例上，重复走 1-RTT 的完整握手流程。

全局握手缓存

为所有 QUIC 服务器实例建立一个全局握手缓存。当用户网络发生切换时，下一次的业务请求无论是落到哪一个机房或哪一台实例上，握手建连都会是 0-RTT。

历代 HTTP 速度测试



比HTTP/1提升 48.4%，比HTTP/2提升 41.2%

结尾

从古至今实时数据传输（音频、视频、游戏等）都面临卡顿、延迟等问题，而 QUIC 基于 UDP 的架构和改进的重传等特性，能够有效的提升用户体验。目前 B 站 也已经接入 QUIC。

如果想要自己体验 QUIC，可以使用 Libquic、Caddy 等。另外 github 上面也有 C++版本的 QUIC 实现，利用 Nodejs 的 C++ 模块，前端工程师也可以快速实现一个 node-quic。

参考资料

1. http2.0 原理详细分析
2. HPACK: HTTP/2 里的沉默杀手
3. QPACK：HTTP /3 的头压缩
4. DH 算法
5. 前向安全（ForwardSecrecy）
6. TLS 1.3 VS TLS 1.2，让你明白 TLS 1.3 的强大
7. CaddyWeb 服务器 QUIC 部署
8. 关于 QUIC 的各种尝试
9. 使用 QUIC 协议实现实时视频直播 0 卡顿
10. 解密 HTTP/2 与 HTTP/3 的新特性
11. Web通信协议，你还需要知道： SPDY 和 QUIC
12. 如何看待 HTTP/3？

腾讯是如何支撑 6W 员工远程工作的?

疫情期间，绝大部分企业和绝大多数的员工都进入了远程工作模式。这突如其来的变化，让很多企业的 IT 和安全部门以及员工都措手不及。除了绞尽脑汁的想办法紧急扩容或限流设计容量本就很小的企业 VPN，还面临企业的安全防护边界被打破，黑客入侵和敏感信息泄露的挑战，而员工对企业 VPN 的吐槽声音也越来越大……很多同行找 TEG企业IT 咨询腾讯是如何支撑 6W 员工全尺寸工作的，于是企业IT 人决定将腾讯 iOA NGN 的零信任无边界网络实践技术介绍给大家，希望对需要了解的人有所帮助。

全文链接：https://mp.weixin.qq.com/s/jyyNsuv9xnMxLhsRBCgaLQ

新一代海量数据搜索引擎 TurboSearch

本文作者：sololzluo，腾讯 AI Lab 开发工程师

一. TurboSearch 简介

AI Lab多年一直在搜索领域进行深耕和积累，继搜搜网页搜索之后，陆续服务于微信搜一搜（公众号文章、朋友圈、视频）、应用宝搜索、地图搜索、音乐搜索、视频搜索、手 Q、QQ 群等精品垂直搜索业务，以及云搜中小数据搜索业务。

从网页搜索继承下来的搜索系统，经过多年的需求迭代，越来越难以支撑结构级新特性更新。因此我们投入精力对整体系统重构和优化，重新构建了大规模、轻量级、松耦合、可裁剪、低运营成本 **具有完整解决方案** 的新一代搜索系统 TurboSearch 。主要有以下特性：

- 完整的 *分布式*、*海量* 搜索系统及运营解决方案
- 支持便捷 *私有化部署*
- 高性能索引及 *并行检索*
- 支持 *多粒度索引* 及检索
- 支持普通检索、分类检索、*WAND* 及精细化的检索层过滤逻辑
- 核心组件 *解耦*，支持横向扩展，能力可裁剪
- 无缝对接 AI Lab各项 *NLP 能力*，涵盖 Query 分析及排序等多个领域
- 支持场景丰富，**除传统的网页和各类非结构化垂类场景外**，同时 *可扩展* 到多模态向量搜索场景

与业界部分开源引擎框架 ElasticSeach，Solr 等不同的是，TurboSearch 更倾向于面向在线 *高并发、大规模、低时延* 的检索需求，同时能够平行扩展到多模态场景，并提供完整的搜索运营能力。TurboSearch 在将会分层次和分阶段逐步在公司内部开源。

对比项	ElasticSearch	TurboSearch
应用级组件化接口	●	○
丰富的社区和接入支持	●	○
实时索引、磁盘索引	●	●
插件支持	●	●
多集群结果融合	○	●
多粒度索引	○	●
丰富的在线干预	○	●
全流程诊断	○	●
海量数据磁盘索引高并发	○	●
基础 NLP 能力	○	●
多层打分排序召回优化	○	●
向量搜索拓展	○	●
关系链搜索支持	○	●

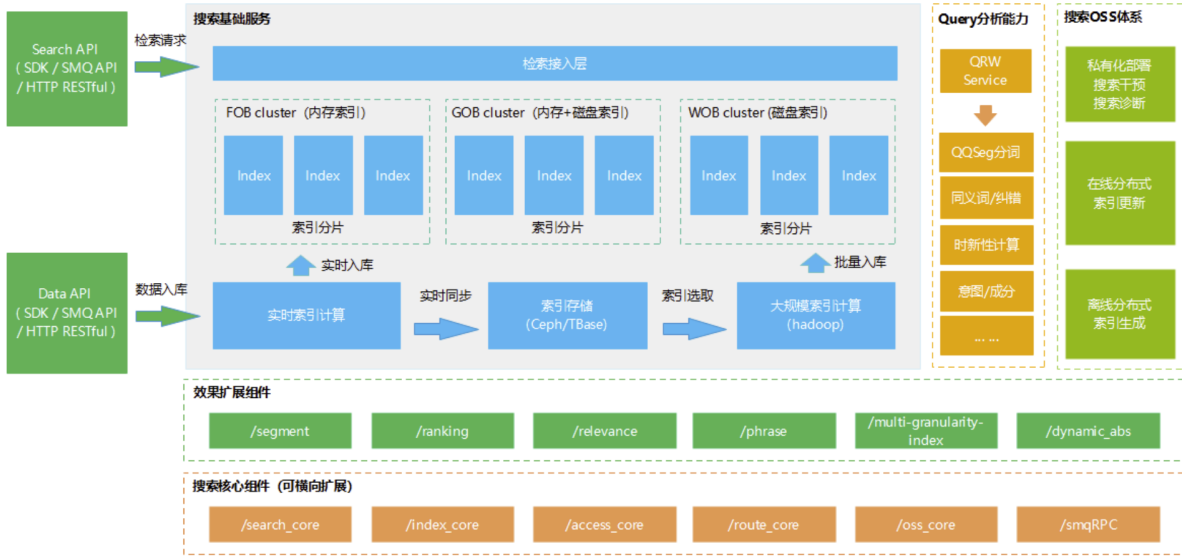
在 多模态/向量检索 领域，AI Lab已经推出 *GNES 检索系统*，聚焦于内容对象的 Encoding，以及多种算法模型的平台化整合。同样在向量检索领域，**TurboSearch 会逐步从索引层面，探索针对大规模向量数据集的高性能检索。**并从向量索引、及系统化运营层面为 GNES 提供支持。

二. 引擎框架介绍

TurboSearch 引擎主要有六大核心能力：

- ***搜索核心组件***：基础核心能力抽象和组件化，便于扩展，如索引计算、检索核心等。同时为了降低多进程资源开销，构建了多线程 C++ 检索通信框架 smqRPC。
- ***搜索基础服务***：基于搜索核心组件分层包装的检索服务，主要包括离线索引、在线检索及检索接入三大层次。支持包括内存、磁盘索引在内的分布式索引及检索环境。
- ***搜索 OSS 体系***：包括离线索引生成、在线索引滚动更新、检索干预、ABTest 等多项能力。后续将进一步完善包括好例、离线效果评估等其它精细化运营系统。
- ***效果扩展组件***：搜索效果随业务场景而变化，我们将打分排序解耦剥离，内置部分基础相关性排序功能，也可自定义排序。
- ***API 组件***：提供包括 SDK、smq 协议访问及 HTTP RESTful 接口等多种访问方式。
- ***Query 分析能力***：除了基础的分词之外，也具备同义词、纠错、时新性计算、意图识别、成分分析、非比留、新词发现等全面能力。

TurboSearch 基础框架：



三. 引擎特性

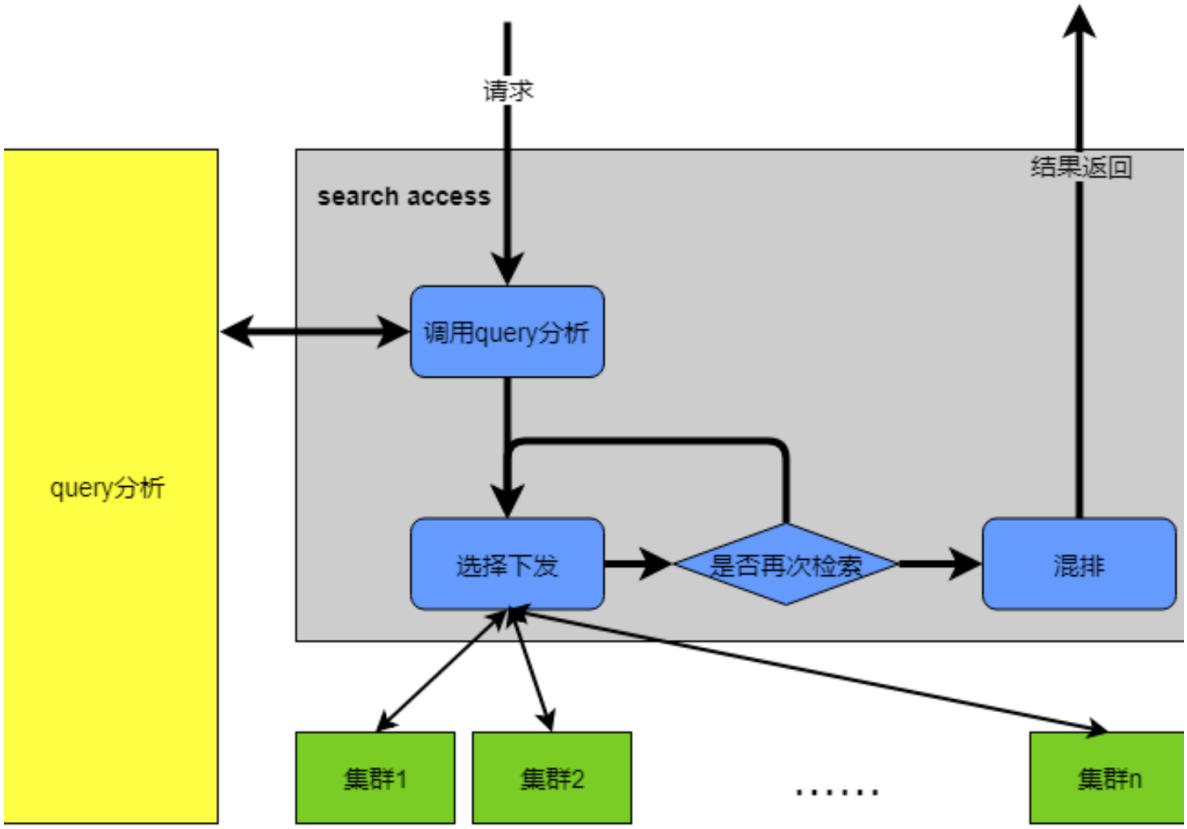
1. Query 检索召回

检索多次下发

一个 Query 的搜索过程可以分为以下几个主要部分：

- **Query 分析（适用于网页和垂搜）**，包括切词、纠错、拓展、改写等。
- **检索召回**，包括倒排求交、截断合并等。
- **打分排序**，在 TurboSearch 中包含多层 rank 提高召回率。

然而，单次 Query 召回往往并不能达到搜索预期。比如，搜索 “*吃鸡*”，只召回吃饭相关的文章可能难以命中用户意图。将其拓展为 “*和平精英*”，或其他热点事件 Query，并将多次拓展结果融合，更容易命中用户意图。因此 TurboSearch 应对这样的 NLP 拓展能力，原生支持多次下发结果融合。



2. QRW & 分层打分排序

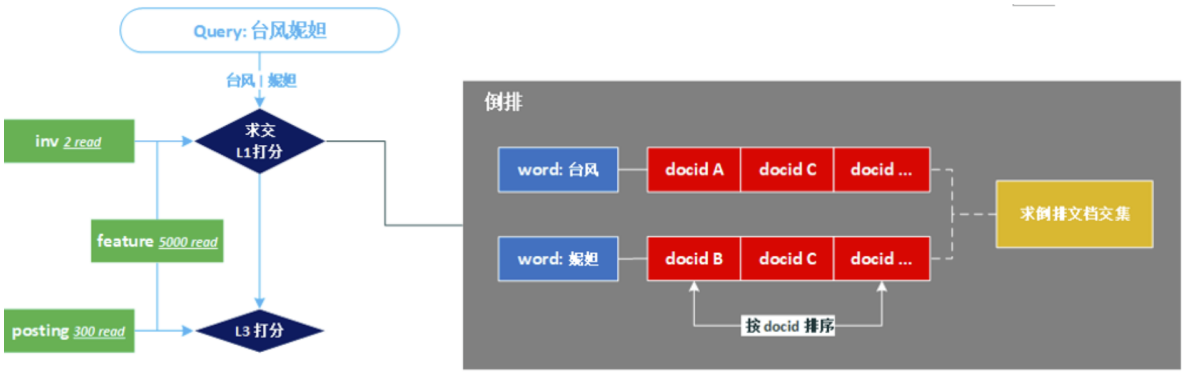
QRW 是 AI Lab多年积累的 Query 分析 NLP 服务，除了覆盖垂搜所需的 纠错 同义词/非比留/基础相关性等基础能力之外，也涵盖了全网搜索所需具备的全部能力，如 Query 改写/时新性/意图识别/成分分析/文本分档 等等。

在排序和召回层面，TurboSearch 设计了 5 层 Rank 来最大化提高召回率，**从 L0 - L4 覆盖离线、倒排求交、精计算、全局精排等多个层面，为每个可能漏招环节做保障。**

3. 高性能检索

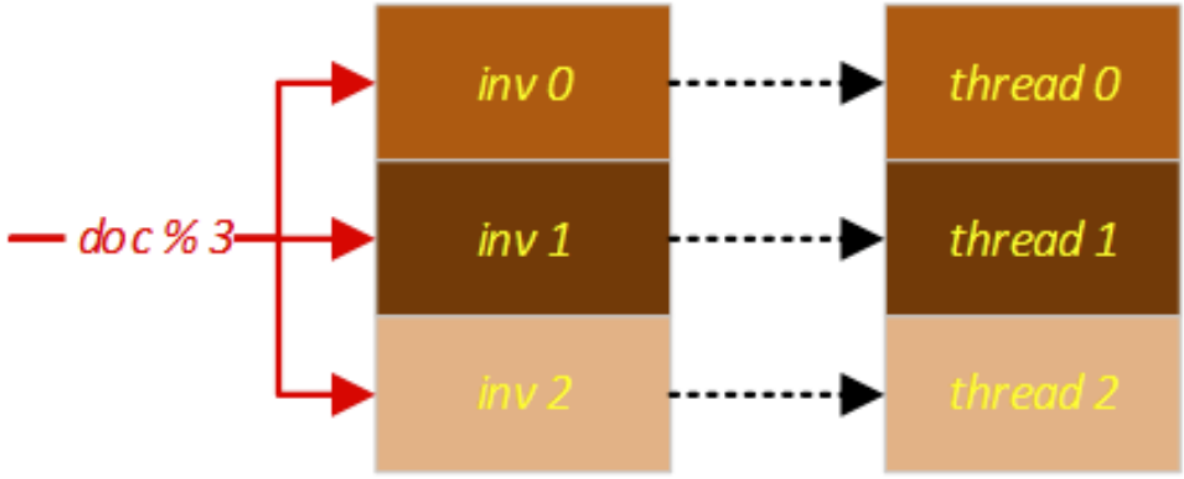
并行检索

文本检索召回的基础即倒排求交：



最耗时部分集中在 *求交+L1 打分* 和 *L3 打分*。在传统的程序设计中，这两部分均在单线程中串行执行。使得在高频词检索求交时，**单次请求耗时难以控制。**

TurboSearch 针对两个高耗时流程，采用 **多线程并行处理。**将倒排索引切分，来并行化检索求交+L1：

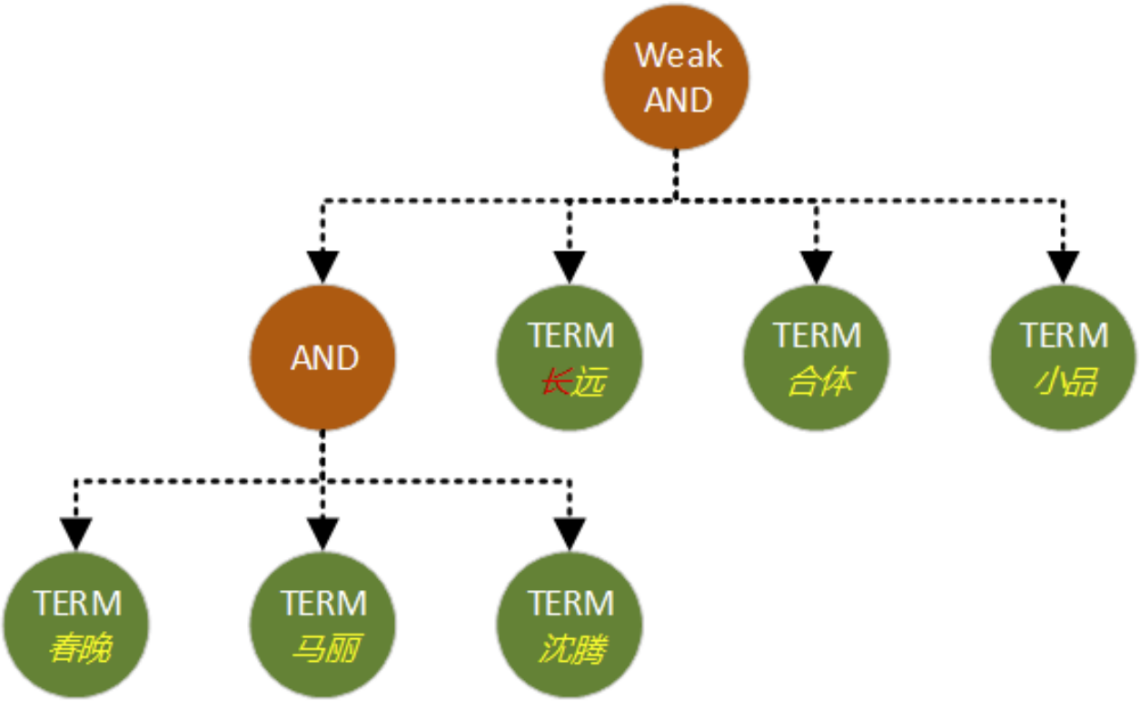


我们做了一些 **特殊的无锁多线程结果合并设计**，避免合并结果等待导致闲置 CPU 的问题。负载未达 100% 时，平均检索耗时可大幅降低（*数据集为长文本新闻数据 250w*）：

对比项	老架构	TurboSearch 3 线程	TurboSearch 5 线程
10 核心 *TOP query* 检索	1800 QPS	2700QPS	2700QPS
80% 负载平均耗时	5.5ms	***2.6ms***	***1.9ms***

Weak-AND

Weak-AND 在广告或推荐等小数据集召回场景下，已经有较多的应用。**在海量数据检索中**，TurboSearch 正探索其在 *长 Query 召回场景* 下的应用。**通过结合 Weak-AND 与 AND**，平衡召回率和检索性能。



对比项	AND	全词 Weak-AND	Weak-AND + AND
Query 修改词召回率	52%	***85%***	***84%***
Query 添加词召回率	68%	***95%***	***92%***
Query 删除词召回率	34%	***67%***	***70%***
长 Query 集合检索平均耗时	60ms	***150ms***	***100ms***

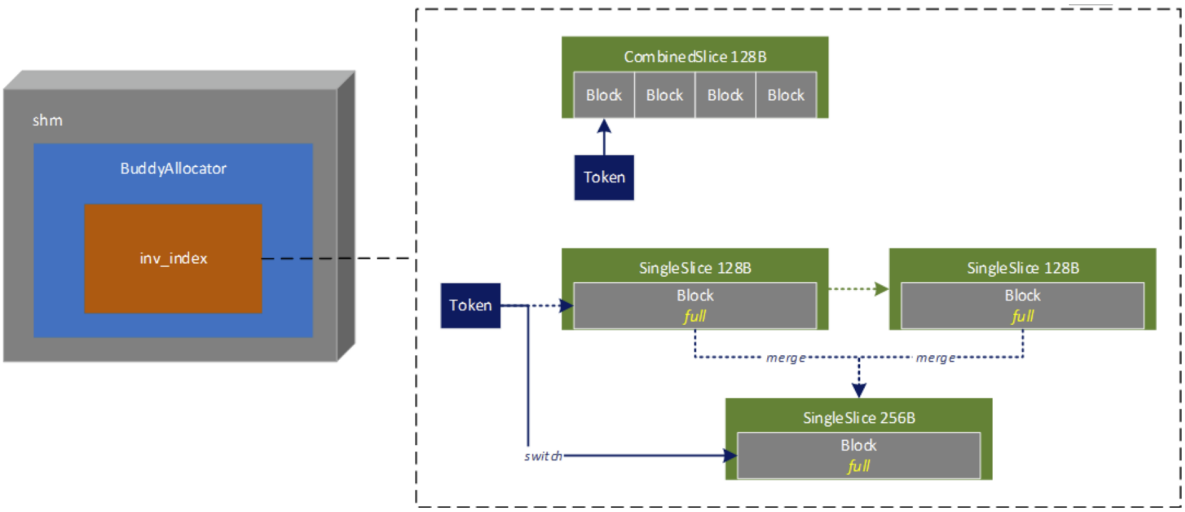
Weak-AND 的性能优化和场景探索将持续进行。

倒排性能优化

求交召回过程中，倒排的索引结构设计，对求交耗时影响较大。内存实时索引倒排在设计上具有以下特性：

- 倒排索引需要支持 *高性能同时读写*，写入新文档和读取倒排求交的能力。
- 需要写入 *共享内存* 避免进程停止导致索引需要重新加载。
- 倒排链中，存储块越多，性能越差，*尽量避免倒排块数量过多*。

TurboSearch 对内存倒排索引做以下设计：



其中：

- BuddyAllocator 单线程运行，***分配释放处理能力达 1000w/s。***
- 分超小块 CombinedChunk 和普通 SingleSlice，***解决超短倒排存储率问题。***

- 小块 SingleSlice 合并为大 SingleSlice，***解决超长倒排中倒排块过多问题。***

对比老架构固定块倒排索引：

对比项	老架构	TurboSearch 3 线程	TurboSearch 5 线程
存储率	100%（基准）	***200%***	***200%***
数据加载速度	100%（基准）	100%	100%
10 核心 *随机语法树* 检索	5w QPS	***10w QPS***	***10w QPS***

4. 多粒度索引

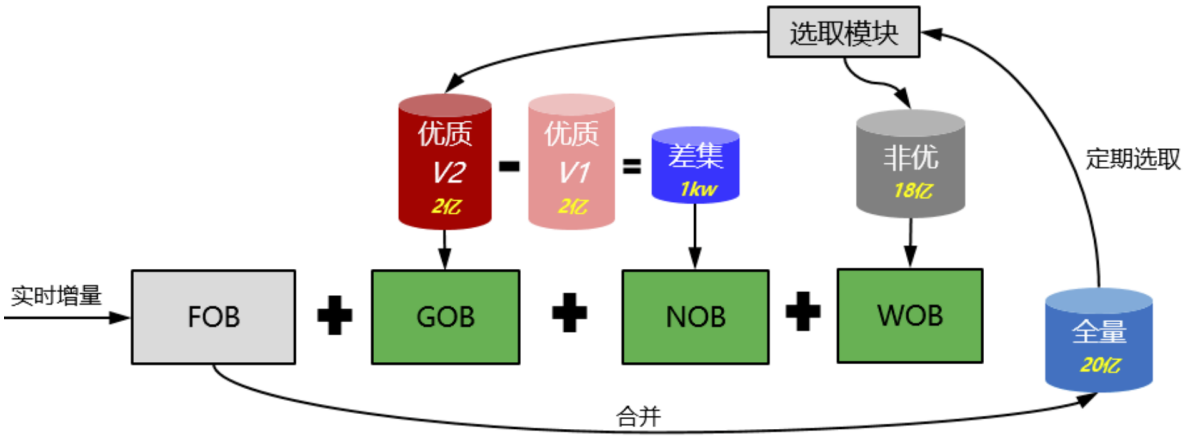
不同于 N-gram 这种暴力索引方式，**多粒度索引专注于文档与 Query 中的隐性词组发现**，对正常分词补充。检索时先进行粗粒度词召回，如果粗粒度无结果或结果偏少，将再次进行细粒度词召回。通过这个方式来解决松散召回导致的紧邻结果截断问题。

如 "*海底捞万象城店*" 对应的粗粒度索引为 "*P:海底捞 万象城 店*"，保证结果能紧邻命中召回，如果在粗粒度检索无结果时，将再次使用 "*海底捞*"、"*万象城*"、"*店*" 进行检索召回。既保证了准确性，也能兼顾召回率。

5. 海量数据索引支持

对于海量数据搜索业务场景，脱胎于网页搜索的 **TurboSearch 继承三种类型的索引集群结构:**

- **FOB**，全内存索引，支持实时增删。
- ***GOB/NOB***，内存倒排+正排，磁盘摘要，不支持实时增删。
- ***WOB***，磁盘索引，不支持实时增删。



根据不用搜索业务数据场景需求，可将各类索引集群组合达到设计目标。

6. 核心逻辑功能插件拓展

TurboSearch 引擎考虑到自定义功能开发拓展，目前对以下核心功能做了插件支持：

- *过滤库 filter*
- *打分库 score*
- *求交 intersect*
- *语法树 syntax*
- *分词库 segment*

7. 私有化部署

TurboSearch 整体设计上支持私有化部署。在公司内网环境运营时，可使用已有的服务组件，如 CL5（名字服务）、Sumeru（资源管理） 等。**然而在私有化部署场景下，这些公共服务难以一同打包部署。**因此 TurboSearch 对这些功能 均有 **内建相应能力，可选择使用，**并基于以下设计支持私有化实现：

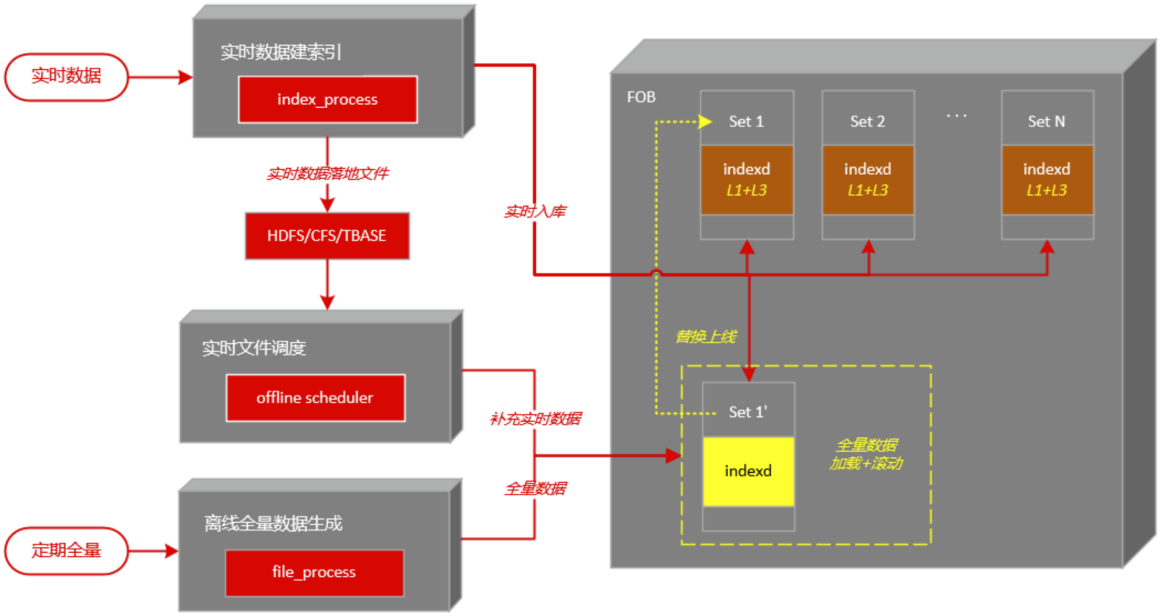
- 开发和部署上，较少的内部环境和外部项目依赖。
- 从运营系统、DB 环境、服务模块均可支持 Docker 部署。
- 完善独立设计的路由管理和资源管理。

四. 系统运营

1. 离线、在线运营架构

以较小数据量的 FOB（实时内存索引系统）集群为例，**离线、在线运营系统通过以下设计保证稳定持续服务**：

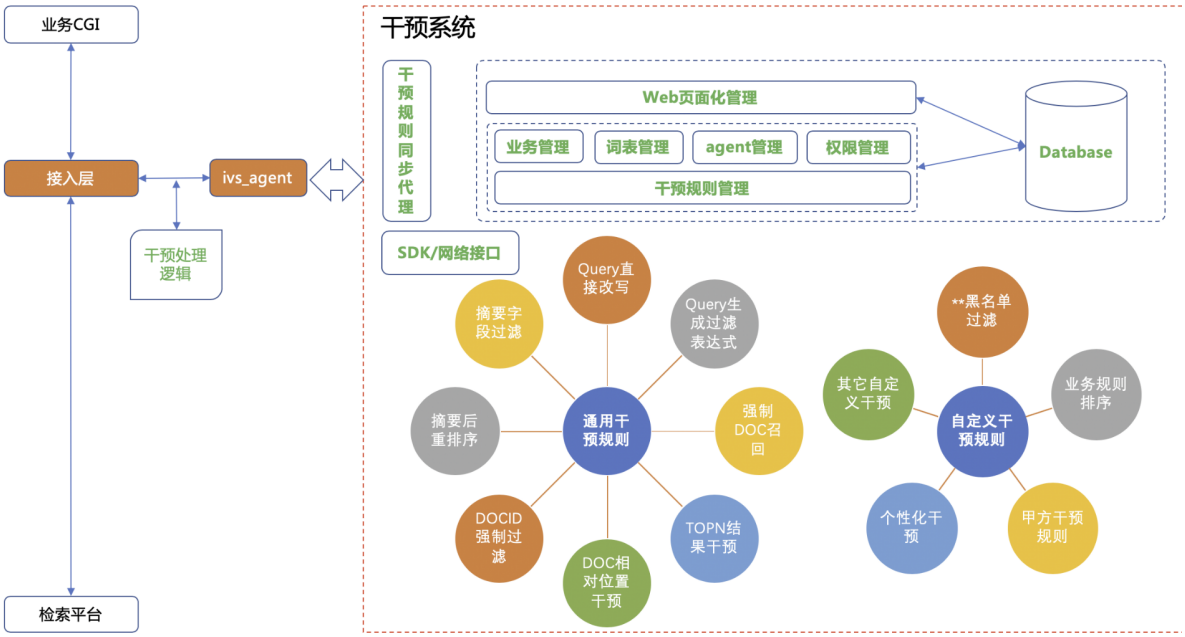
- 全量数据平滑无缝版本更新，确保线上服务 *不受数据滚动影响*。
- 实时数据与全量滚动无缝衔接，确保滚动 *不会导致实时数据缺失*。



2. 干预系统

在现网运营中，检索召回排序无法保证所有 Query 达到最佳。对于一些突发高曝光 Badcase，需要有**临时干预能力**。TurboSearch 在接入层设计了干预系统，并沉淀积累了大量干预策略，可覆盖现网运营大部分干预需求。**主要支持两大类干预类型：

- ***通用干预规则***，系统预先定义和实现了具体的干预处理逻辑。
- **自定义干预规则**，提供干预规则的读写接口，满足不同业务的特定干预需求。



3. 全流程检索、数据诊断

在持续优化的海量数据搜索业务运营过程中，会有持续或突发的 Badcase 需要定位。**而一个海量数据搜索业务中，一般都是 *多集群、多机服务、多层逻辑* 的复杂系统。在整体系统中定位和诊断 Badcase 是一个复杂而困难的工作。**比如一篇文档未被召回有以下多种可能：** **

- **文档数据入库问题**，某些原因数据被删除或未入库。
- **求交篇数阶段问题**，由于在线检索考虑耗时和性能问题，无法做到全求交召回，召回太多被截断。
- **求交超时截断问题**，出于耗时限制，超高频词之间的求交过程常常会出现超时截断。
- **L1 打分低未进入 L3**，L1 取 Top300 进入 L3，因此 L1 打分过低可能导致无法召回。
- **L3 打分低被合并截断**，每一层检索转发 access 服务均会对召回结果按照打分取 TopN 截断返回。
- **L4 打分低或被过滤**，多集群召回融合打分会丢弃掉一些文档。
- **语法树本身无法召回目标文档**，下发的语法树全求交也不可能召回目标文档。
- ...

一篇文章有如此繁多的漏召回可能性。**TurboSearch 从在线服务模块和离线数据流程两个方面入手，设计全流程的诊断，来协助快速定位召回和数据问题。**

五. 应用场景和展望

目前 TurboSearch 已可应用在 传统文本搜索、关系链搜索、LBS 位置相关搜索、中心聚类向量搜索等场景。在持续改进引擎现有功能之外，我们还会做更多的探索：

- 持续优化 WAND 检索性能，以及分析拓展其使用的 Query 场景。
- 探索基于倒排索引，引入 **知识实体的链接类搜索**，比如搜索 “腾讯总办”，可从索引层面召回相应结果。
- 在 多模态/向量检索 领域，**探索新结构索引**，比如图式、树式索引来优化向量检索性能和效果。
- 逐步开源开放整个引擎，并构建协同开发生态。

全面理解搜索 Query：当你在搜索引擎中敲下回车后，发生了什么?

作者：joelchen，腾讯 PCG 应用研究员

1. 前言

Query 理解（QU，Query Understanding），简单来说就是从词法、句法、语义三个层面对 query 进行结构化解析。这里 query 从广义上来说涉及的任务比较多，最常见的就是我们在搜索系统中输入的查询词，也可以是 FAQ 问答或阅读理解中的问句，又或者可以是人机对话中用户的聊天输入。本文主要介绍在搜索中的 query 理解，会相对系统性地介绍 query 理解中各个重要模块以及它们之间如何 work 起来共同为搜索召回及排序模块服务，同时简单总结个人目前了解到业界在各个模块中的一些实现方法。

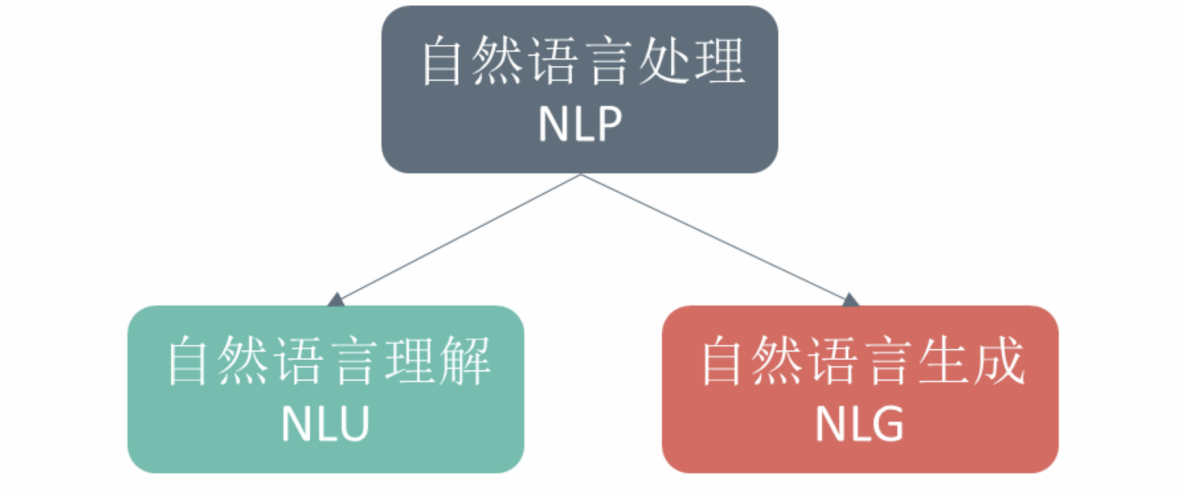
2. 相关概念

2.1 NLP

自然语言处理（NLP，Natural Language Processing）是集语言学、统计学、计算机科学，人工智能等学科于一体的交叉领域，目标是让计算机能在处理理解人类自然语言的基础上进一步执行结构化输出或语言生成等其他任务，其涉及的基础技术主要有：词法分析、句法分析、语义分析、语用分析、生成模型等。诸如语音识别、机器翻译、QA 问答、对话机器人、阅读理解、文本分类聚类等任务都属于 NLP 的范畴。

这些任务从变换方向上来看，主要可以分为自然语言理解（NLU，Natural Language Understanding）和自然语言生成（NLG，Natural Language Generation）两个方面，其中 NLU 是指对自然语言进行理解并输出结构化语义信息，而 NLG 则是多模态内容（图像、语音、视频、结构/半结构/非结构化文本）之间的相互生成转换。

一些任务同时涵盖 NLU 和 NLG，比如对话机器人任务需要在理解用户的对话内容（NLU 范畴）基础上进行对话内容生成（NLG 范畴），同时为进行多轮对话理解及与用户交互提示这些还需要有对话管理模块（DM，Dialogue Management）等进行协调作出对话控制。本文要介绍的搜索 query 理解大部分模块属于 NLU 范畴，而像 query 改写模块等也会涉及到一些 NLG 方法。



2.2 搜索引擎

在介绍搜索 query 理解之前，先简单介绍下搜索引擎相关知识。除了传统的通用搜索 Google、Baidu、Bing 这些，大部分互联网垂直产品如电商、音乐、应用市场、短视频等也都需要搜索功能来满足用户的需求查询，相较于推荐系统的被动式需求满足，用户在使用搜索时会通过组织 query 来主动表达诉求，为此用户的搜索意图相对比较明确。但即便意图相对明确，要做好搜索引擎也是很有挑战的，需要考虑的点及涉及的技术无论在深度还是广度上都有一定难度。

一个基本的搜索系统大体可以分为离线挖掘和在线检索两部分，其中包含的重要模块主要有：Item 内容理解、Query 理解、检索召回、排序模块等。整个检索系统的目标可以抽象为给定 query，检索出最能满足用户需求的 item，也即检索出概率：$\mathrm{P}\left(\mathrm{I}{\mathrm{i}} \middle| \mathrm{Q} \right)$ *最大的 item* $\mathrm{I}{\mathrm{i}}$

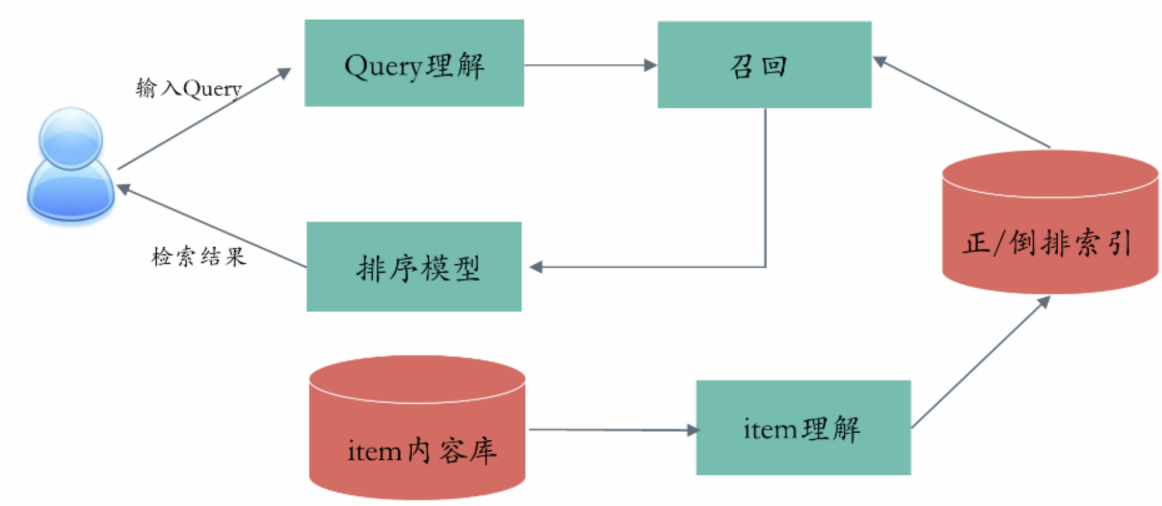
根据贝叶斯公式展开：

$$\mathrm{argmax}P\left(I_i|Q\right) = argmax\frac{P(Q|I_i)P(I_i)}{P(Q)}argmxP(Q|I_i)P(I_i)$$

其中 $P(I{i})$ *表示 item 的重要程度，对应 item 理解侧的权威度、质量度等挖掘，*$P(Q|I{i})$ 表示 item I_{i} 能满足用户搜索需求 Q 的程度，对 query 分词后可以表示为：

$$P\left(Q|I_i\right) = P(t_1, \ldots, t_n|I_i)$$

这部分概率对应到基于 query 理解和 item 理解的结果上进行 query 和 item 间相关性计算，同时涉及到点击调权等排序模块。



2.2.1 离线挖掘

在离线侧，我们需要做一些基础的离线挖掘工作，包括 item 内容的获取、清洗解析、item 内容理解（语义 tag、权威度计算、时间因子、质量度等）、用户画像构建、query 离线策略挖掘、以及从搜索推荐日志中挖掘 item 之间的语义关联关系、构建排序模型样本及特征工程等。进行 item 内容理解之后，对相应的结构化内容执行建库操作，分别构建正排和倒排索引库。其中，正排索引简单理解起来就是根据 itemid 能找到 item 的各个基本属性及 term 相关（term 及其在 item 中出现的频次、位置等信息）的详细结构化数据。

相反地，倒排索引就是能根据分词 term 来找到包含该 term 的 item 列表及 term 在对应 item 中词频、位置等信息。通常会对某个 item 的 title、keyword、anchor、content 等不同属性域分别构建倒排索引，同时一般会根据 item 资源的权威度、质量度等纵向构建分级索引库，首先从高质量库中进行检索优先保证优质资源能被检索出来，如果高质量检索结果不够再从低质量库中进行检索。为了兼顾索引更新时效性和检索效率，一般会对索引库进行横向分布式部署，且索引库的构建一般分为全量构建和增量更新。常见的能用于快速构建索引的工具框架有 Lucene 全文检索引擎以及基于 Lucene 的 Solr、ES（Elastic Search）等。

除了基本的文本匹配召回，还需要通过构建 query 意图 tag 召回或进行语义匹配召回等多路召回来提升搜索语义相关性以及保证召回的多样性。

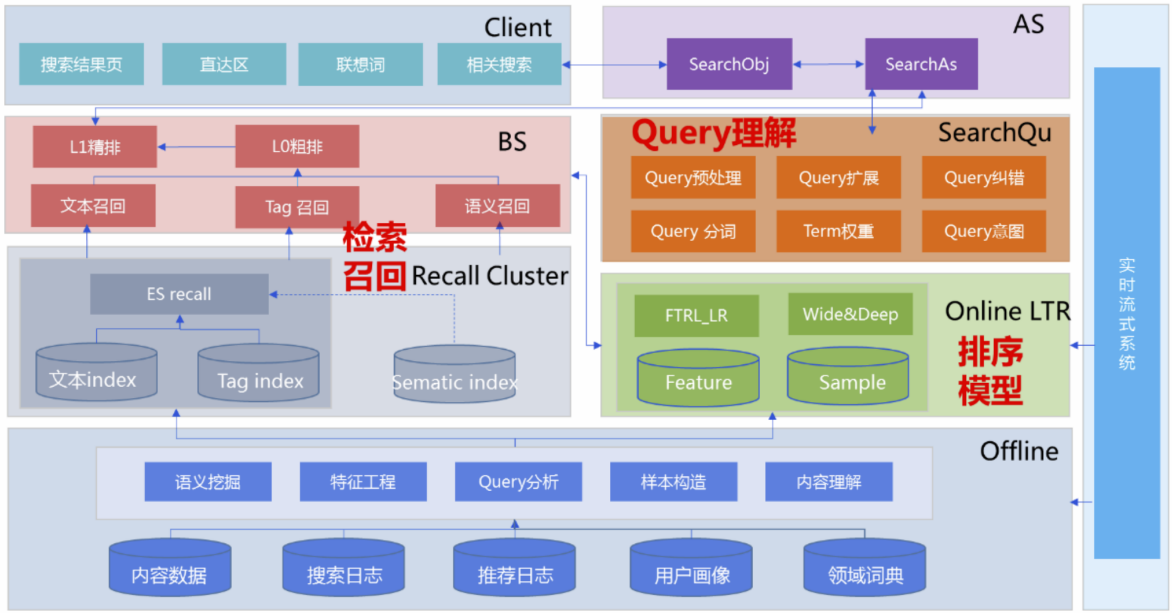
2.2.2 在线检索

线上执行检索时大体可以分为基础检索（BS）和高级检索（AS）两个过程，其中 BS 更注重 term 级别的文本相关性匹配及粗排，AS 则更注重从整体上把控语义相关性及进行精排等处理。以下面这个框图为例，介绍下一个相对简单的在线检索过程。对于从 client 发起的线上搜索请求，会由接入层传入 SearchObj（主要负责一些业务逻辑的处理），再由 SearchObj 传给 SearchAS（负责协调调用 qu、召回和排序等模块），SearchAS 首先会去请求 SearchQU 服务（负责搜索 query 理解）获取对 query 理解后的结构化数据，然后将这些结构化数据传给基础召回模块 BS，BS 根据切词粒度由粗到细对底层索引库进行一次或多次检索，执行多个索引队列的求交求并拉链等操作返回结果。

同时 BS 还需要对文本、意图 tag、语义召回等不同路召回队列根据各路召回特点采用多个相关性度量（如：BM25 文本相似度、tag 相似度、语义相关度、pagerank 权威度、点击调权等）进行 L0 粗排截断以保证相关性，然后再将截断后的多路召回进行更精细的 L1 相关性融合排序，更复杂一些的搜索可能会有 L0 到 LN 多层排序，每层排序的侧重点有所不同，越高层次 item 数变得越少，相应的排序方法也会更复杂。

BS 将经过相关性排序的结果返回给 SearchAS，SearchAS 接着调用 SearchRank 服务进行 ctr/cvr 预估以对 BS 返回的结果列表进行 L2 重排序，并从正排索引及摘要库等获取相应 item 详情信息进一步返回给 SearchObj 服务，与此同时 SearchObj 服务可以异步去请求广告服务拉取这个 query 对应的广告召回队列及竞价相关信息，然后就可以对广告或非广告 item 内容进行以效果（pctr、pcvr、pcpm）为导向的排序从而确定各个 item 内容的最终展示位置。

最后在业务层一般还会支持干预逻辑以及一些规则策略来实现各种业务需求。在实际设计搜索实验系统时一般会对这些负责不同功能的模块进行分层管理，以上面介绍的简单检索流程为例主要包括索引召回层、L0 排序截断层、L1 相关性融合排序层、L2 效果排序层、SearchAS 层、业务层等，各层流量相互正交互不影响，可以方便在不同层进行独立的 abtest 实验。



当然，具体实现搜索系统远比上面介绍的流程更为复杂，涉及的模块及模块间的交互会更多，比如当用户搜索 query 没有符合需求的结果时需要做相关搜索 query 推荐以进行用户引导、检索无结果时可以根据用户的画像或搜索历史做无结果个性化推荐等，同时和推荐系统一样，搜索周边涉及的系统服务也比较庞杂，如涉及日志上报回流、实时计算能力、离/在线存储系统、abtest 实验平台、报表分析平台、任务调度平台、机器学习平台、模型管理平台、业务管理后台等，只有这些平台能 work 起来，整个搜索系统才能正常运转起来。

2.2.3 多模语义搜索

随着资源越来越丰富，如何让用户从海量的信息及服务资源中能既准确又快速地找到诉求变得越来越重要，如 2009 年百度提出的框计算概念目标就是为了能更快更准地满足用户需求。同时，随着移动互联网的到来，用户的输入方式越来越多样，除了基本的文本输入检索，实现通过语音、图片、视频等内容载体进行检索的多模态搜索也变得越来越有必要。由于用户需求的多样或知识背景不同导致需求表达的

千差万别，要做到极致的搜索体验需要漫长的优化过程，毕竟目前真正的语义搜索还远未到来。

3.Query 理解

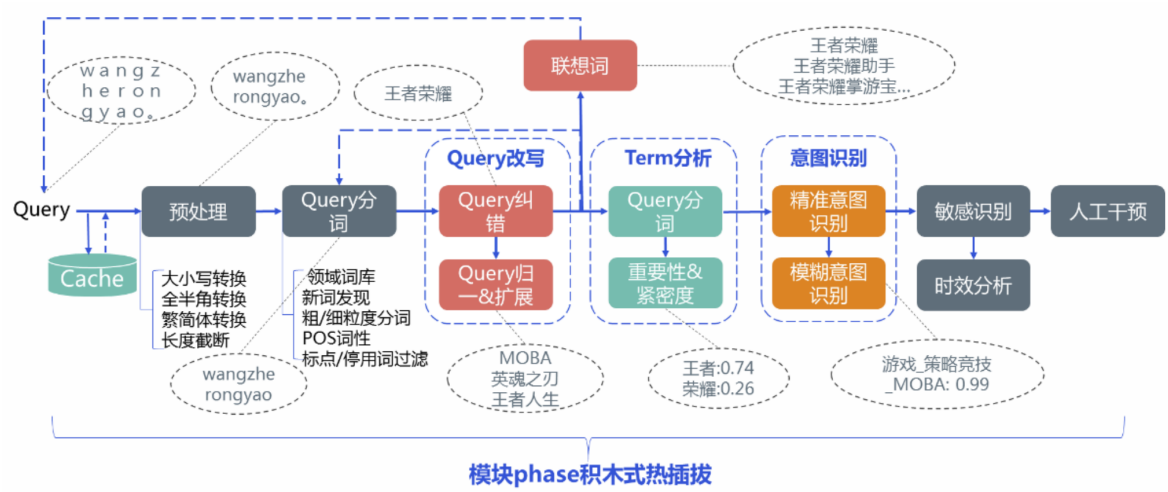
从前面的介绍可以知道，搜索三大模块的大致调用顺序是从 query 理解到检索召回再到排序，作为搜索系统的第一道环节，query 理解的结果除了可以用于召回也可以给排序模块提供必要的基础特征，为此 qu 很大程度影响召回和排序的质量，对 query 是进行简单字面上的理解还是可以理解其潜在的真实需求很大程度决定着一个搜索系统的智能程度。目前业界的搜索 QU 处理流程还是以 pipeline 的方式为主，也即拆分成多个模块分别负责相应的功能实现，pipeline 的处理流程可控性比较强，但存在缺点就是其中一个模块不 work 或准确率不够会对全局理解有较大影响。为此，直接进行 query-item 端到端地理解如深度语义匹配模型等也是一个值得尝试的方向。

3.1 Pipeline 流程

搜索 Query 理解包含的模块主要有：query 预处理、query 纠错、query 扩展、query 归一、联想词、query 分词、意图识别、term 重要性分析、敏感 query 识别、时效性识别等。以下图为例，这里简单介绍下 query 理解的 pipeline 处理流程：线上来一个请求 query，为缓解后端压力首先会判断该 query 是否命中 cache，若命中则直接返回对该 query 对应的结构化数据。若不命中 cache，则首先会对 query 进行一些简单的预处理，接着由于 query 纠错可能会用到分词 term 信息进行错误检测等先进行 query 分词并移除一些噪音符号，然后进行 query 纠错处理，一些情况下局部纠错可能会影响到上下文搭配的全局合理性，为此还可能会需要进行二次纠错处理。

对 query 纠错完后可以做 query 归一、扩展及联想词用于进行扩召回及帮助用户做搜索引导。接着再对 query 做分词及对分词后的 term 做重要性分析及紧密度分析，对无关紧要的词可以做丢词等处理，有了分词 term 及对应的权重、紧密度信息后可以用于进行精准和模糊意图的识别。除了这些基本模块，有些搜索场景还需要有对 query 进行敏感识别及时效性分析等其他处理模块。最后还需要能在 cms 端进行配置的人工干预模块，对前面各个模块处理的结果能进行干预以快速响应 badcase 处理。

当然，这个 pipeline 不是通用的，不同的搜索业务可能需要结合自身情况做 pipeline 的调整定制，像百度这些搜索引擎会有更为复杂的 query 理解 pipeline，各个模块间的依赖交互也更为复杂，所以整个 qu 服务最好能灵活支持各个模块的动态热插拔，像组装乐高积木一样进行所需模块的灵活配置，下面对各个模块做进一步详细的介绍。



3.2 Query 预处理

Query 预处理这个模块相对来说比较简单，主要对 query 进行以下预处理从而方便其他模块进行分析：

1. 全半角转换：将在输入法全角模式下输入的 query 转换为半角模式的，主要对英文、数字、标点符号有影响，如将“ｗｅｃｈａｔ１２３”全角输入转成半角模式下的“wechat 123”。
2. 大小写转换：统一将大写形式的字母转成小写形式的。

3. 繁简体转换：将繁体输入转成简体的形式，当然考虑到用户群体的差异以及可能存在繁体形式的资源，有些情况还需要保留转换前的繁体输入用于召回。
4. 无意义符号移除：移除诸如火星文符号、emoji 表情符号等特殊符号内容。
5. Query 截断：对于超过一定长度的 query 进行截断处理。

3.3 Query 分词

3.3.1 分词技术

Query 分词就是将 query 切分成多个 term，如："手机淘宝"切分成"手机 淘宝"两个 term。分词作为最基础的词法分析组件，其准确性很大程度影响 qu 后面的各个处理，如分词及对应词性信息可以用于后续的 term 重要性分析和意图识别等多个模块。同时，qu 的分词及其粒度需要与 item 侧索引构建的分词及粒度保持一致，从而才能进行有效地召回。目前分词技术相对来说已经比较成熟，主要做法有基于词典进行前后向最大匹配、对所有成词情况构造 DAG、hmm/crf 序列标注模型以及深度学习模型+序列标注等。

目前无论学术界还是工业界开放的分词工具或服务还是比较多的，如主要有腾讯内部的 QQSeg、百度 LAC、Jieba 分词、清华 THULAC、北大 pkuseg、中科院 ICTCLAS、哈工大 PyLTP、复旦 FNLP、Ansj、SnowNLP、FoolNLTK、HanLP、斯坦福 CoreNLP、Jiagu、IKAnalyzer 等。这些分词工具在功能和性能上会存在一定的差异，具体使用时要结合业务需求定制选择，需要考虑的点主要包括：切词准确性、粒度控制、切词速度、是否带有 NER、NER 识别速度、是否支持自定义词典等，由于没对所有这些分词工具做各项性能指标的具体评测，这里暂时不做过多的比较。

在搜索中的 query 切词一般会做粒度控制，分为细粒度和 phrase 粗粒度两个级别，比如 query"下载深海大作战"按 phrase 粗粒度切分可以为"下载 深海大作战"，按细粒度切分为"下载 深海 大 作战"。在进行召回时可以优先用 phrase 粗粒度的切词结果进行召回能得到更精准相关的结果同时减少多个 term 拉链合并的计算量。当 phrase 粗粒度分词进行召回结果不够时，可以采用拆分后的细粒度分词进行二次重查扩召回。

3.3.2 新词发现

一般来说，使用已有的开源切词工具已经有比较好的切分精度了，但是对于一些新出现的网络词汇可能不能及时识别覆盖，尤其是对于一些垂直搜索有比较多业务专名词的情况，这时候需要对这些未登录词做新词发现。一些切词工具自带有新词发现功能，比如 Jieba 采用 HMM 进行新词发现。此外简单地，可以采用基于统计的方法来进行新词发现，通过统计语料中的词语 tfidf 词频、凝聚度和自由度等指标来进行无监督新词挖掘，当词语的凝聚度和自由度达到一定阈值且已有分词不能切分到一起时可以人工评估后加进词库。其中凝聚度即点互信息：

$$Solid = log\frac{P(w_i w_j)}{P(w_i)P(w_j)}$$

用于衡量两个 term 共现的概率，两个 term 经常出现在一起，则可以将它们组合成一个词语整体的可能性也更大。自由度取 term 左右邻熵的较小值：

$$Freedeg = min(entropy\left(w_{\text{left}}\right), entropy(w_{\text{right}}))$$

衡量当前 term 左右两边字集合的随机程度，如果左右字集合越随机则这个 term 独立成词的可能性也更大。还有的做法是对 query 进行切词后构建词之间的关系矩阵，然后进行矩阵分解，得到各个词在主特征空间的投影向量，通过投影向量计算词之间的相似度并设定相应阈值构造 0-1 切分矩阵，基于该矩阵进行是否成词的判断。再有就是可以将新词发现转化为序列标注问题，训练 BiLSTM-CRF、BERT-CRF 等模型预测成词的起始、中间及结束位置，或者转化为 ngram 词在给定句子语境中是否成词或不成词二分类问题。

3.4 紧密度分析

Term 紧密度，主要用于衡量 query 中任意两个 term 之间的紧密程度，如果两个 term 间紧密度比较高，则这两个 term 在召回 item 中出现的距离越近相对来说越相关。以相邻的两个 term 为例，由于分词工具本身存在准确率问题，某两个 term 可能不会被分词工具切分到一起，但它们之间的关系比较紧密，也即紧密度比较高，如果在召回时能将在 item 中同时出现有这两个 term 且出现位置挨在一起或比较靠近的 item 进行召回，相关性会更好。

以前面的搜索 query“下载深海大作战”为例，经分词工具可能切分成“下载 深海 大 作战”，但其实“大”和“作战”的紧密度很高，从文本相关性角度来看，召回“喵星大作战”app 要一定程度比“大人物作战”会更相关。当然，在 query 中并不是两个 term 的距离越近紧密度越高，可以通过统计搜索 log 里 term 之间的共现概率来衡量他们的紧密程度。

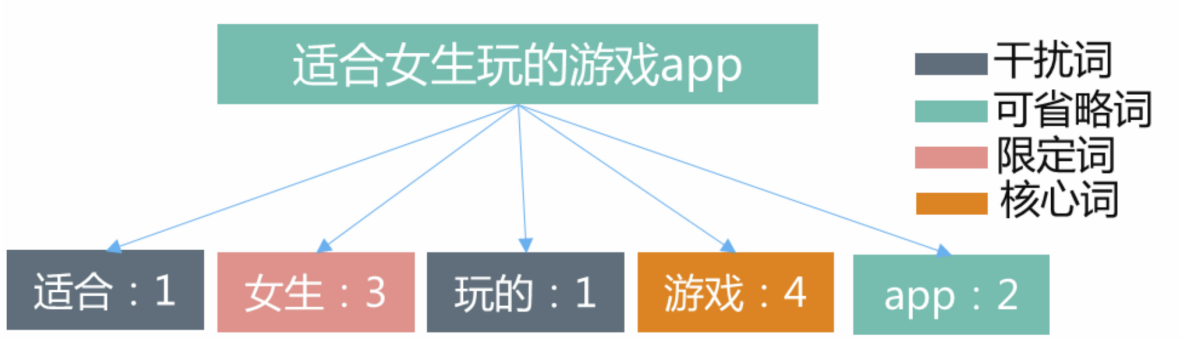
在进行召回时，传统的相关性打分公式如 OkaTP、BM25TP、newTP、BM25TOP 等在 BM25 基础上进一步考虑了 proximity 计算，但主要采用两个 term 在 item 中的距离度量，如：$\text{tpi}\left(t{i},t{j} \right) = \frac{1.0}{\text{dist}{\lbrack t{i},t{j}\rbrack}^{2}}$ 有了 query 中的 term 紧密度，在召回构造查询索引的逻辑表达式中可以要求紧密度高的两个 term 需共同出现以及在 proximity 计算公式中融合考虑进去，从而保证 query 中紧密度高的两个 term 在 item 中出现距离更近更相关。

3.5 Term 重要性分析

考虑到不同 term 在同一文本中会有不一样的重要性，在做 query 理解及 item 内容理解时均需要挖掘切词后各个 term 的重要性，在进行召回计算相关性时需要同时考虑 query 及 item 侧的 term 重要性。如对于 query“手机淘宝”，很明显 term“淘宝”要比“手机”更重要，为此赋予“淘宝”的权重应该比“手机”高。Term 重要性可以通过分等级或 0.0~1.0 的量化分值来衡量，如下图的 case 所示我们可以将 term 重要性分为 4 个级别，重要性由高到低分别是：核心词、限定词、可省略词、干扰词。对于重要级别最低的 term 可以考虑直接丢词，或者在索引库进行召回构造查询逻辑表达式时将对应的 term 用“or”逻辑放宽出现限制，至于计算出的在 query 和 item 内容中的 term 重要性分值则可以用于召回时计算基础相关性，如简单地将 BM25 公式

$$\sum_{t\in Q}\log\frac{N-n\left(t\right)+0.5}{n\left(t\right)+0.5}\times\frac{tf(t)(k_1+1)}{\mathrm{tf}\left(t\right)+k_1(1-b+b\frac{\mathrm{dl}}{\mathrm{avgdl}})}\frac{qf(t)(k_2+1)}{\mathrm{qf}\left(t\right)+k_2}$$

中 term 在 item 及 query 中的词频 tf(t)、qf(t)乘上各自的 term 权重。



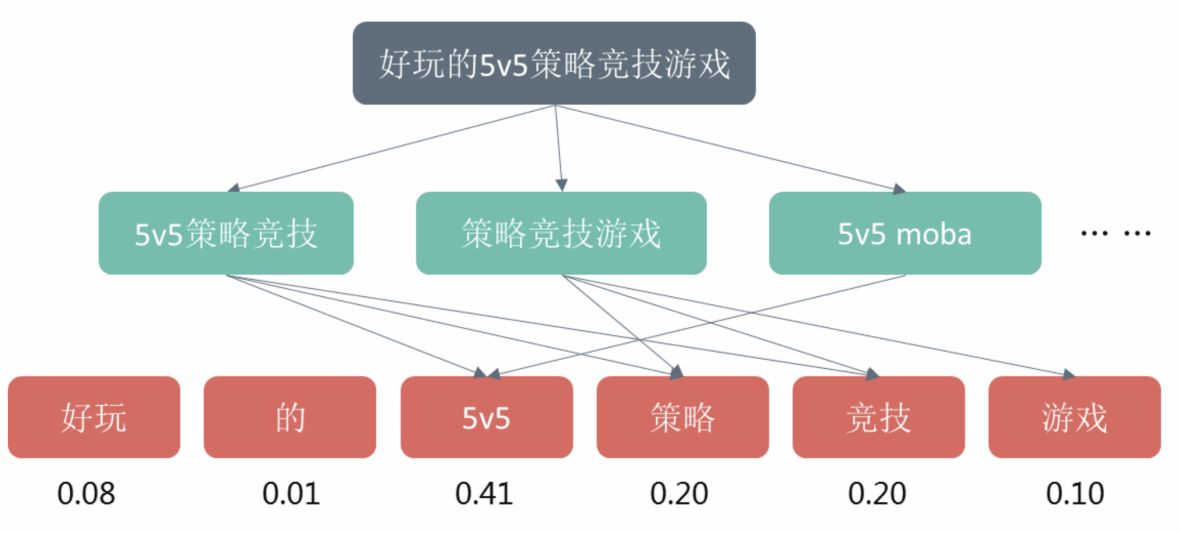
其中 item 内容侧的 term 重要性可以采用 LDA 主题模型、TextRank 等方法来挖掘，至于 query 侧的 term 重要性，比较容易想到的方法就是把它视为分类或回归问题来考虑，通过训练 svm、gbdt 等传统机器学习模型即可进行预测。模型样本的构造除了进行人工标注还可以通过用户的搜索点击日志来自动构造。大概做法是将某个 query 对应搜索结果的用户点击频次作为同时出现在 query 及搜索结果 title 中相应 term 的重要性体现，首先通过共同点击信息或二部图方法对 query 进行聚类，再将一个 query 对应有不同点击 title 以及同一 term 在簇内不同 query 中的点击 title 频次结果加权考虑起来，同时排除掉一些搜索 qv 比较不置信的 query 及 title 对应的结果，最后将累计频次进行归一化及分档得到样本对应 label。

至于特征方面，则可以从词法、句法、语义、统计信息等多个方向进行构造，比如：term 词性、长度信息、term 数目、位置信息、句法依存 tag、是否数字、是否英文、是否停用词、是否专名实体、是否重要行业词、embedding 模长、删词差异度、前后词互信息、左右邻熵、独立检索占比（term 单独作为 query 的 qv/所有包含 term 的 query 的 qv 和）、iqf、文档 idf、统计概率 $P\left(

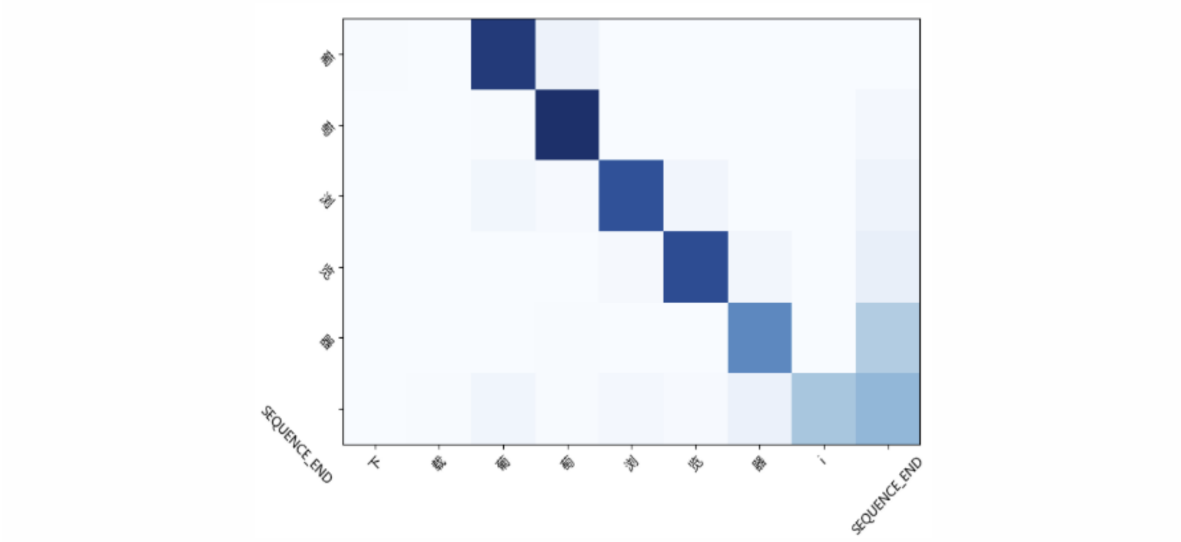
t \middle| d \right) = \frac{P\left(d \middle| t \right)*P(t)}{P(d)}$ 以及短语生成树得到 term 权重等。

其中删词差异度的做法是先训练得到 query 的 embedding 表示，然后计算移除各个 term 之后的 query 与原 query 的 embedding 相似度差值用于衡量 term 的重要性，如果移除某个 term 之后相似度差异很大，代表这个 term 比较重要。而短语生成树的做法是通过从搜索 session 序列及搜索点击行为中挖掘出 query 之间的相关关系按 query 长度降序自顶向下进行构造得到，其中树的边和结点均有一定的权重。

这里假设在一定共现关系情况下越短的 query 越是整体意图的核心表达，所以和越下层结点连接越多的 term 重要性越大，仅和较上层或根结点有连接的 term 重要性相对较小，通过用 iqf 等初始化叶子结点，然后自底向上进行结点分值计算并进行多轮迭代使得 term 权重相对稳定。如下图所示 query“好玩的 5v5 策略竞技游戏”构造的短语生成树示例。



分类回归的方法很好地利用了用户的点击行为反馈，通过精细的特征工程往往能得到比较好的结果。还有的方法就是利用深度学习模型来学习 term 重要性，比如通过训练基于 BiLSTM+Attention 的 query 意图分类模型或基于 Seq2Seq/Transformer 训练的 query 翻译改写模型得到的 attention 权重副产物再结合其他策略或作为上述分类回归模型的特征也可以用于衡量 term 的重要性。



3.6 搜索引导

受限于用户的先验知识的参差不齐或输入设备引入的噪音，用户输入的 query 可能不足以表达用户真正的需求进而影响用户搜到想要的结果。为此，除了保证搜索结果的相关性，一个完善的搜索引擎还需要给用户提供一系列搜索引导功能，以减少用户的搜索输入成本，缩短用户找到诉求的路径。搜索引导按功能的不同主要可以分为：搜索热词、搜索历史、query 改写、搜索联想词，一些电商等垂搜可能还带

有类目属性等搜索导航功能。由于搜索热词及搜索历史功能相对比较好理解，这里不做过多阐述。

3.6.1 Query 改写

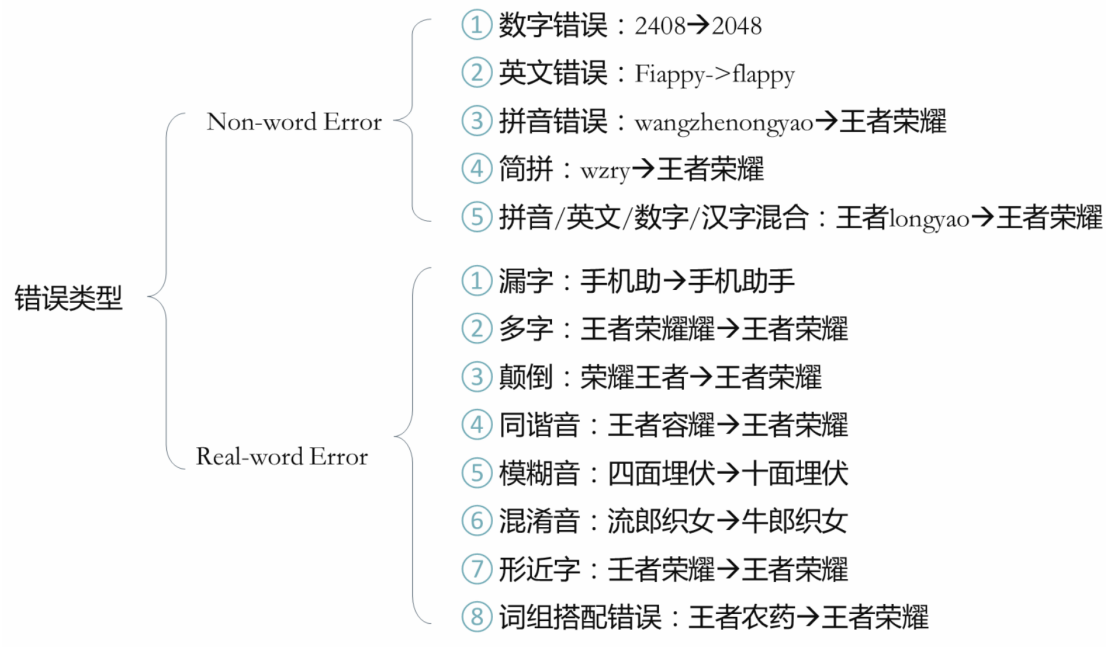
Query 改写这个概念比较泛，简单理解就是将源 query 改写变换到另一个 query。按照改写功能的不同，query 改写可以分为 query 纠错、query 归一、query 扩展三个方向。其中 query 纠错负责对存在错误的 query 进行识别纠错，query 归一负责将偏门的 query 归一变换到更标准且同义的 query 表达，而 query 扩展则负责扩展出和源 query 内容或行为语义相关的 query 列表推荐给用户进行潜在需求挖掘发现。

3.6.1.1 Query 纠错

Query 纠错，顾名思义，也即对用户输入 query 出现的错误进行检测和纠正的过程。用户在使用搜索过程中，可能由于先验知识掌握不够或输入过程引入噪音（如：语音识别有误、快速输入手误等）输入的搜索 query 会存在一定的错误。如果不对带有错误的 query 进行纠错，除了会影响 qu 其他模块的准确率，还会影响召回的相关性及排序的合理性，最终影响到用户的搜索体验。

除了搜索场景，query 纠错还可以应用于输入法、人机对话、语音识别、内容审核等应用场景。不同的业务场景需要解决的错误类型会不太一样，比如 ASR 语音识别主要解决同谐音、模糊音等错误，而输入法及搜索等场景则需要面临解决几乎所有错误类型，如同谐音、模糊音（平舌翘舌、前后鼻音等）、混淆音、形近字（主要针对五笔和笔画手写输入法）、多漏字等错误。根据 query 中是否有不在词典中本身就有错误的词语（Non-word），可以将 query 错误类型主要分为 Non-word 和 Real-word 两类错误。

其中，Non-word 错误一般出现在带英文单词或数字的 query 中，由于通过输入法进行输入，不会存在错误中文字的情况，所以中文 query 如果以字作为最小语义单元的话一般只会存在 Real-word 错误，而带英文数字的 query 则可能存在两类错误。下图对这两大类的常见错误类型进行归类及给出了相应的例子。

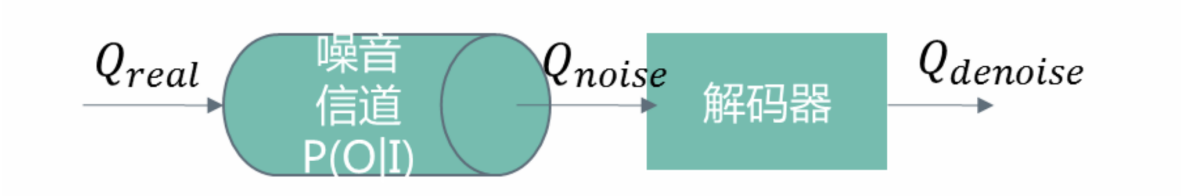


从原理上，Query 纠错可以用噪音信道模型来理解，假设用户本意是搜索 $$Q{\text{real}}$$，*但是 query 经过噪音信道后引进了一定的噪音，此时纠错过程相当于构建解码器将带有噪音干扰的 query* $Q{\text{noise}}$ 进行最大去噪还原成 $Q{*text{denoise}}$，*使得* $Q{\text{denoise}} \approx Q_{\text{real}}$。

对应的公式为：

$$Q_{\text{real}} = \text{argmax}_{Q_{\text{denoise}}} P(Q_{\text{denoise}}|Q_{\text{noise}}) = \text{argmax}_{Q_{\text{denoise}}} \frac{P(Q_{\text{noise}}|Q_{\text{denoise}}) \times P(Q_{\text{denoise}})}{P(Q_{\text{noise}})} \propto \text{argmax}_{Q_{\text{denoise}}} P(Q_{\text{noise}}|Q_{\text{denoise}}) \times P(Q_{\text{denoise}})$$

其中 $P(Q{\text{denoise}})$ 表示语言模型概率，$P(Q{\text{noise}}|Q_{\text{denoise}})$ 表示写错概率，进行纠错一般都是围绕着求解这两个概率来进行的。



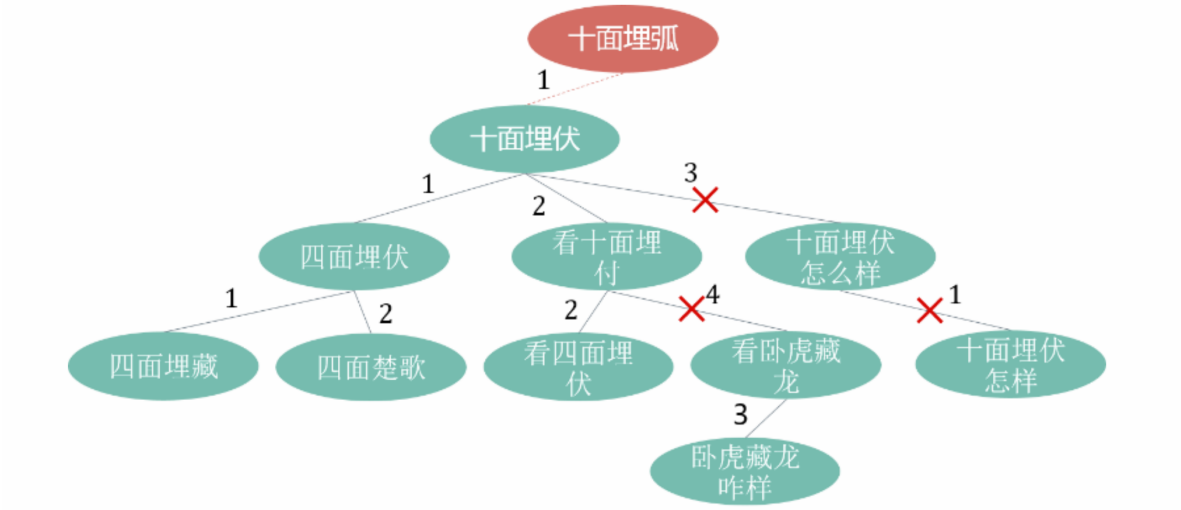
纠错任务主要包含错误检测和错误纠正两个子任务，其中错误检测用于识别错误词语的位置，简单地可以通过对输入 query 进行切分后检查各个词语是否在维护的自定义词表或挖掘积累的常见纠错 pair 中，若不在则根据字型、字音或输入码相近字进行替换构造候选并结合 ngram 语言模型概率来判断其是否存在错误，这个方法未充分考虑到上下文信息，可以适用于常见中文词组搭配、英文单词错误等的检测。进一步的做法是通过训练序列标注模型的方法来识别错误的开始和结束位置。

至于错误纠正，即在检测出 query 存在错误的基础上对错误部分进行纠正的过程，其主要包括纠错候选召回、候选排序选择两个步骤。在进行候选召回时，没有一种策略方法能覆盖所有的错误类型，所以一般通过采用多种策略方法进行多路候选召回，然后在多路召回的基础上通过排序模型来进行最终的候选排序。

对于英文单词错误、多漏字、前后颠倒等错误可以通过编辑距离度量进行召回，编辑距离表示从一个字符串变换到另一个字符串需要进行插入、删除、替换操作的次数，如“apple”可能错误拼写成“appel”，它们的编辑距离是 1。由于用户的搜索 query 数一般是千万甚至亿级别的，如果进行两两计算编辑距离的话计算量会非常大，为此需要采用一定的方法减小计算量才行。比较容易想到的做法是采用一些启发式的策略，如要求首字（符）一样情况下将长度小于等于一定值的 query 划分到一个桶内再计算两两 query 间的编辑距离，此时可以利用 MapReduce 进一步加速计算。

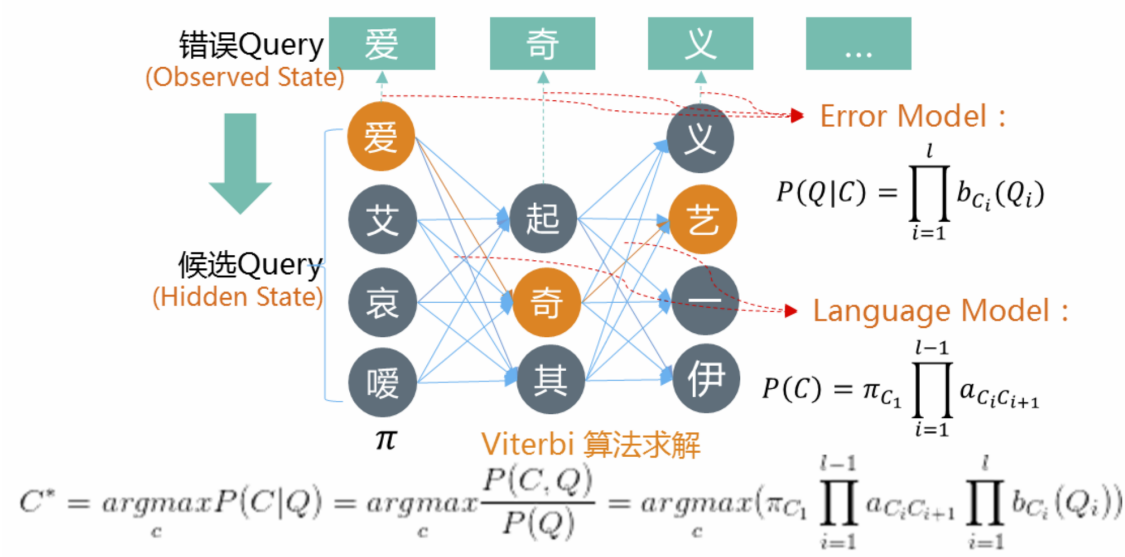
当然这种启发式的策略可能会遗漏掉首字（符）不一样的 case，如在前面两个位置的多漏字、颠倒等错误。还有的办法就是利用空间换时间，如对 query 进行 ngram 等长粒度切分后构建倒排索引，然后进行索引拉链的时候保留相似度 topn 的 query 作为候选。又或者利用编辑距离度量满足三角不等式 d（x，y）+d（y，z）>=d（x，z）的特性对多叉树进行剪枝来减少计算量。

首先随机选取一个 query 作为根结点，然后自顶向下对所有 query 构建多叉树，树的边为两个结点 query 的编辑距离。给定一个 query，需要找到与其编辑距离小于等于 n 的所有 query，此时自顶向下与相应的结点 query 计算编辑距离 d，接着只需递归考虑边值在 d-n 到 d+n 范围的子树即可。如下图所示需要查找所有与 query“十面埋弧”编辑距离小于等于 1 的 query，由于“十面埋弧”与“十面埋伏”的编辑距离为 1，此时只需考虑边值在 1-1 到 1+1 范围的子树，为此“十面埋伏怎么样”作为根结点的子树可以不用继续考虑。



对于等长的拼音字型错误类型还可以用 HMM 模型进行召回，HMM 模型主要由初始状态概率、隐藏状态间转移概率及隐藏状态到可观测状态的发射概率三部分组成。如下图所示，将用户输入的错误 query“爱奇义”视为可观测状态，对应的正确 query“爱奇艺”作为隐藏状态，其中正确 query 字词到错误 query 字词的发射关系可以通过人工梳理的同谐音、形近字混淆词表、通过编辑距离度量召回的相近英文单词以及挖掘好的纠错片段对得到。

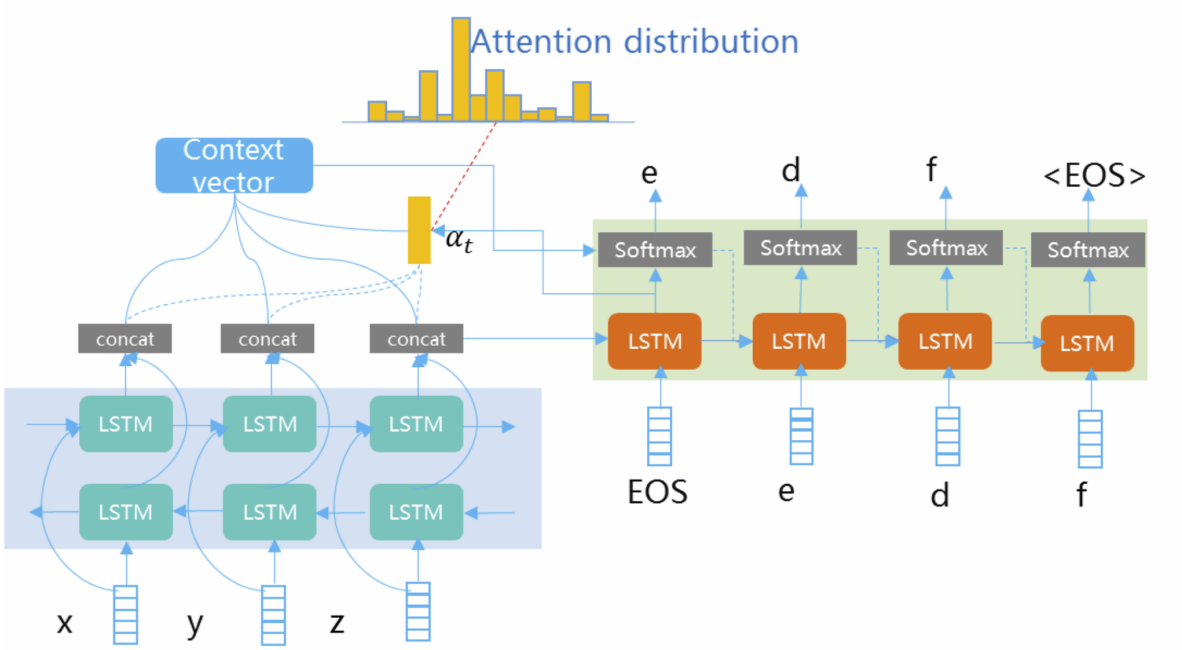
至于模型参数，可以将搜索日志中 query 进行采样后作为样本利用 hmmlearn、pomegranate 等工具采用 EM 算法进行无监督训练，也可以简单地对搜索行为进行统计得到，如通过 nltk、srilm、kenlm 等工具统计搜索行为日志中 ngram 语言模型转移概率，以及通过统计搜索点击日志中 query-item 及搜索 session 中 query-query 对齐后的混淆词表中字之间的错误发射概率。训练得到模型参数后，采用维特比算法对隐藏状态序列矩阵进行最大纠错概率求解得到候选纠错序列。



进一步地，我们还可以尝试深度学习模型来充分挖掘搜索点击行为及搜索 session 进行纠错候选召回，如采用 Seq2Seq、Transformer、Pointer-Generator Networks 等模型进行端到端的生成改写，通过引入 attention、copy 等机制以及结合混淆词表进行受限翻译生成等优化，可以比较好地结合上下文进行变长的候选召回。

另外结合 BERT 等预训练语言模型来进行候选召回也是值得尝试的方向，如在 BERT 等预训练模型基础上采用场景相关的无监督语料继续预训练，然后在错误检测的基础上对错误的字词进行 mask 并预测该位置的正确字词。Google 在 2019 年提出的 LaserTagger 模型则是另辟蹊径将文本生成建模为序列标注任务，采用 BERT 预训练模型作为 Encoder 基础上预测各个序列位置的增删留标签，同样适用于 query 纠错这种纠错前后大部分文本重合的任务。

另外，爱奇艺在同一年提出的适用于繁简体中文拼写检纠错的 FASPell 模型尝试在利用 BERT 等预训练语言模型生成纠错候选矩阵（DAE）的基础上结合生成候选的置信度及字符相似度对候选进行解码过滤（CSD）出候选纠错目标，在中文纠错任务上也取得了一定进展。



在多路召回的基础上，可以通过从词法、句法、语义及统计特征等多个方面构造特征训练 GBDT、GBRank 等排序模型将预测出最优的纠错候选。由于进行多路纠错候选召回计算量相对比较大且直接进行线上纠错会存在较大的风险，可以在离线挖掘好 query 纠错 pair 后按搜索 qv 优先进行对头中部 query 进行人工审核准入，然后放到线上存储供调用查询。线上应用时，当 qu 识别到用户输入 query 存在错误并进行相应纠错后，对于不那么置信的结果可以将纠错结果透传给召回侧进行二次检索使用，对于比较置信的纠错结果可以直接展示用纠错后 query 进行召回的结果并给到用户搜索词确认提示（如下图所示），如果纠错 query 不是用户真实意图表达的话，用户可以继续选择用原 query 进行搜索，此时用户的反馈行为也可以用于进一步优化 query 纠错。



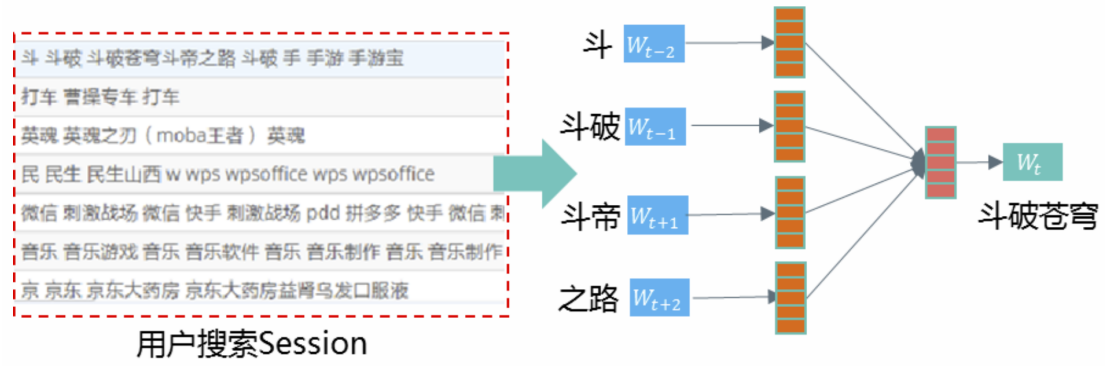
3.6.1.2 Query 扩展

Query 扩展，即通过挖掘 query 间的语义关系扩展出和原 query 相关的 query 列表。Query 列表的结果可以用于扩召回以及进行 query 推荐帮用户挖掘潜在需求，如下图在百度搜索"自然语言处理"扩展出的相关搜索 query：



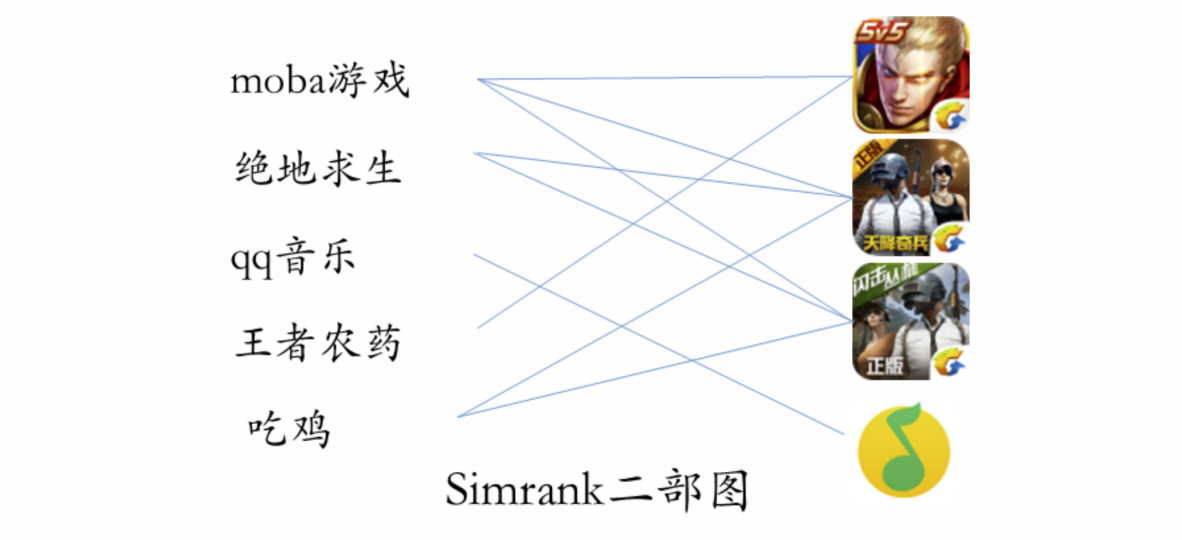
搜索场景有丰富的用户行为数据，我们可以通过挖掘搜索 session 序列和点击下载行为中 query 间的语义关系来做 query 扩展。如用户在进行搜索时，如果对当前搜索结果不满意可能会进行一次或多次 query 变换重新发起搜索，那么同一搜索 session 内变换前后的 query 一般存在一定的相关性，为此可以通过统计互信息、关联规则挖掘等方法来挖掘搜索 session 序列中的频繁共现关系。

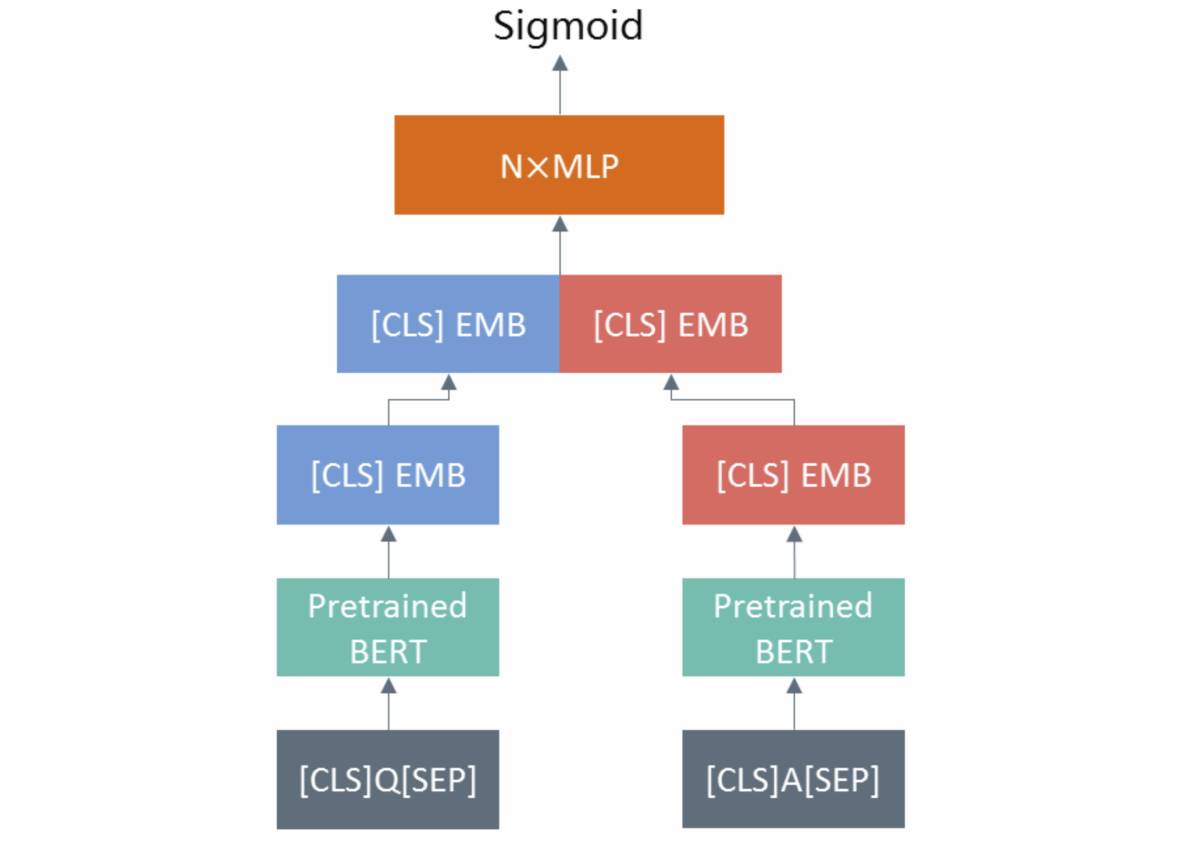
或者把一个用户搜索 session 序列当成文章，其中的每个 query 作为文章的一个词语，作出假设：如果两个 query 有相同的 session 上下文，则它们是相似的，然后通过训练 word2vec、fasttext 等模型将 query 向量化，进而可以计算得到 query 间的 embedding 相似度。对于长尾复杂 query，通过 word2vec 训练得到的 embedding 可能会存在 oov 的问题，而 fasttext 由于还考虑了字级别的 ngram 特征输入进行训练，所以除了可以得到 query 粒度的 embedding，还可以得到字、词粒度的 embedding，此时通过对未登录 query 切词后的字、词的 embedding 进行简单的求和、求平均也可以得到 query 的 embedding 表示。还可以参考 WR embedding 的做法进一步考虑不同 term 的权重做加权求平均，然后通过减去主成分的映射向量以加大不同 query 间的向量距离，方法简单却比较有效。



至于搜索点击下载行为，可以通过构建 query-item 的点击下载矩阵，然后采用协同过滤或 SVD 矩阵分解等方法计算 query 之间的相似度，又或者构建 query 和 item 之间的二部图（如下图示例），若某个 query 节点和 item 节点之间存在点击或下载行为则用边进行连接，以 ctr、cvr 或归一化的点击下载次数等作为连接边的权重，然后可以训练 swing、simrank/wsimrank++等图算法迭代计算 query 间的相似度，也可以采用 Graph Embedding 的方法来训练得到 query 结点间的 embedding 相似度。

更进一步地，我们还可以利用搜索点击下载行为构造弱监督样本训练基于 CNN/LSTM/BERT 等子网络的 query-item 语义匹配模型得到 query 和 item 的 embedding 表示，如此也可以计算 query pair 间的 embedding 相似度。对于将 query 进行 embedding 向量化的方法，可以先离线计算好已有存量 query 的 embedding 表示，然后用 faiss 等工具构建向量索引，当线上有新的 query 时通过模型 inference 得到对应的 embedding 表示即可进行高效的近邻向量检索以召回语义相似的 query。





在给用户做搜索 query 推荐时，除了上面提到的跟用户当前输入 query 单点相关 query 推荐之外，还可以结合用户历史搜索行为及画像信息等来预测用户当前时刻可能感兴趣的搜索 query，并在搜索起始页等场景进行推荐展示。此时，可以通过 LSTM 等网络将用户在一段 session 内的搜索行为序列建模为用户 embedding 表示，然后可以通过构建 Encoder-Decoder 模型来生成 query，或采用语义匹配的方法将用户 embedding 及 query embedding 映射到同一向量空间中，然后通过计算 embedding 相似度的方法来召回用户可能感兴趣的 query。

3.6.1.3 Query 归一

Query 归一和 query 纠错在概念上容易混淆，相较于 query 纠错是对存在错误的 query 进行纠正，query 归一则主要起到对同近义表达的 query 进行语义归一的作用。一些用户的 query 组织相对来说比较冷门，和 item 侧资源的语义相同但文字表达相差较大，直接用于召回的话相关性可能会打折扣，这时如果能将这些 query 归一到相对热门同义或存在对应资源的 query 会更容易召回相关结果。如将“腾讯台球”归一到“腾讯桌球”，“华仔啥时候出生的？”、“刘德华出生年月”、“刘德华什么是出生的”这些 query 都可以归一到“刘德华出生日期”相对标准的 query。其中涉及到的技术主要有同义词挖掘及语义对齐替换，如“华仔”对应的同义词是“刘德华”，“啥时候出生的”对应的同义词是“出生日期”。

同义词的挖掘是一个积累的过程，最直接的获取方式是利用业界已经有一些比较有名的知识库，如英文版本的 WordNet、中文版本的知网 HowNet、哈工大的同义词词林等，或者可以利用一些开放的中文知识图谱（如：OpenKG、OwnThink 等）或从抓取百度/维基百科站点数据然后提取出其中的别名、简称等结构化信息直接获得，对于百科中无结构化数据可以简单通过一些模板规则（如："XX 俗称 XX"、"XX 又名 XX"等）来提取同义词。同时，还可以在知识库中已有同义词种子的基础上通过一些方法进一步扩充同义词，如韩家炜老师团队提出的通过构建分类器来判断实体词是否属于某个同义词簇的方法来进一步扩充同义词集。

除了利用结构化数据或规则模板，还可以在构造平行语料基础上通过语义对齐的方式来挖掘同义词。对于搜索场景来说，可以通过挖掘丰富的行为数据来构造平行语料，如利用搜索 session 行为相关语料训练无监督的 word2vec、wordrank 等词向量模型来衡量词语间的相似度，不过这些模型更多是学习词语间在相同上下文的共现相似，得到的相似度高的词语对不一定是同义词，有可能是同位词、上下位词甚至是反义词，此时需要通过引入监督信号或外部知识来优化词向量，如有方法提出通过构建 multi-task 任务在预测目标词的同时预测目标词在句子中表示的实体类型以加入实体的语义信息来提升词向量之间的语义相似性。

进一步地，还可以利用前面介绍的二部图迭代、深度语义匹配、Seq2Seq 翻译生成等 query 扩展方法从搜索点击弱监督行为中先挖掘出语义表达相近的 query-query、item-item 或 query-item 短语对，然后再将语义相近的 query/item 短语对进行语义对齐，对齐的话可以采用一些规则的方法，也可以采用传统的统计翻译模型如 IBM-M2 进行对齐，语义对齐后从中抽取出处于相同或相近上下文中的两个词语作为同义词对候选，然后结合一些统计特征、词语 embedding 相似度以及人工筛选等方式进行过滤筛选。

考虑到同一个词语在不同的上下文中可能表达不同的语义，同义词语间的关系也是上下文相关的，此时如果通过对齐挖掘粗粒度的同义片段对能进一步消除歧义。线上对 query 进行归一时，则和离线同义词挖掘的过程相反，对 query 进行分词后读取线上存储的同义词表做同义词候选替换，对替换网格进行对齐生成候选 query，最后通过结合语言模型概率及在当前上下文的替换概率或者构造特征训练 GBDT 等模型的方式对候选 query 进行排序得到最终的归一 query。



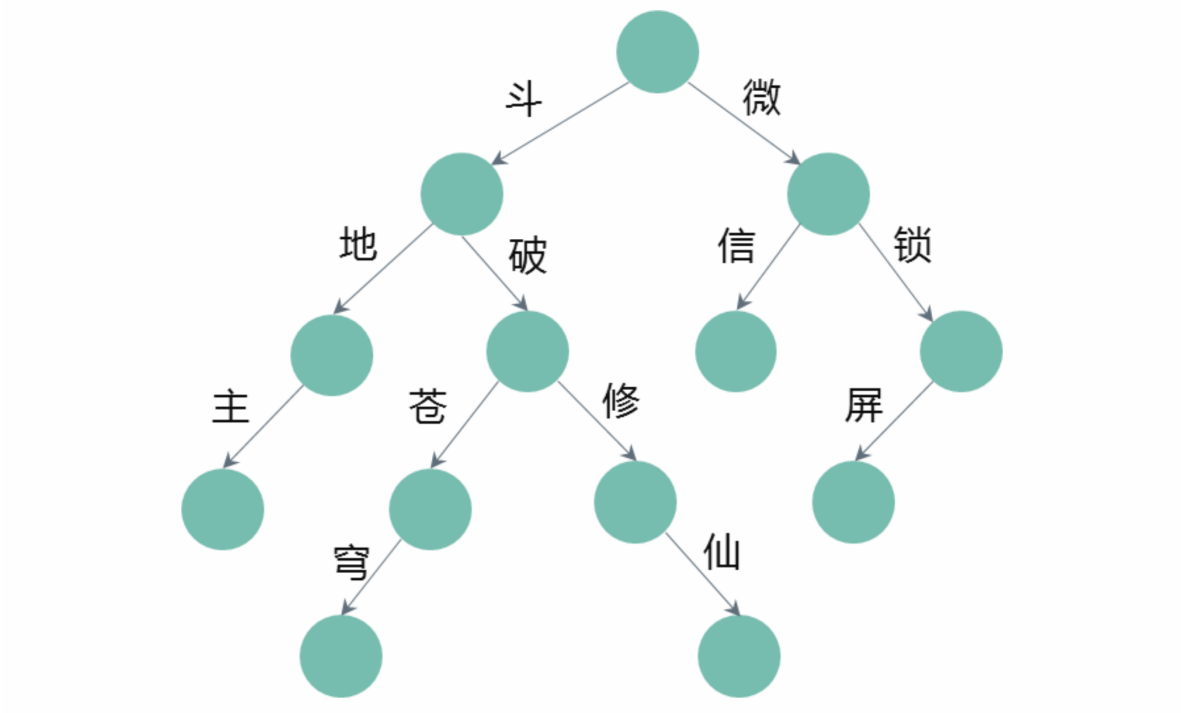
3.6.2 搜索联想词

联想词，顾名思义，就是对用户输入 query 进行联想扩展，以减少用户搜索输入成本及输错可能，能比较好地提升用户搜索体验。联想结果主要以文本匹配为主，文本匹配结果不足可以辅以语义召回结果提升充盈率。考虑到用户在输入搜索 query 时意图相对明确，一般会从左到右进行 query 组织，为此基于这个启发式规则，目前联想词中文本匹配召回又以前缀匹配优先。



虽然联想词涉及的是技术主要是简单的文本匹配，在匹配过程中还需要考虑效率和召回质量，同时中文输入可能会有拼音输入的情况（如上图所示）也需要考虑。由于用户在搜索框中输入每一个字时都会发起一起请求，联想词场景的请求 pv 是非常大的。为加速匹配效率，可以通过对历史搜索 query 按 qv 量这些筛选并预处理后分别构建前后缀 trie 树用于对用户线上输入的 query 进行前缀及中后缀匹配召

回，然后对召回的结果进行排序，如果是仅简单按 qv 降序排序，可以在 trie 树结点中存放 qv 信息并用最小堆等结构进行 topk 召回。



当然仅按 qv 排序还不够，比如可能还需要考虑用户输入上文 query 后对推荐的下文 query 的点击转化、下文 query 在结果页的点击转化以及 query 的商业化价值等因素。同时一些短 query 召回的结果会非常多，线上直接进行召回排序性能压力较大，为此可以先通过离线召回并进行粗排筛选，再将召回结果写到一些 kv 数据库如 redis、共享内存等存储供线上直接查询使用。离线召回的话，可以采用 AC 自动机同时进行高效的前中后缀匹配召回。AC 自动机（Aho-Corasic）是基于 trie 数+KMP 实现的多模匹配算法，可以在目标文本中查找多个模式串的出现次数以及位置。此时，离线召回大致的流程是：

（1）从历史搜索 query 中构造前缀 sub-query，如 query“酷我音乐”对应的 sub-query 有中文形式的“酷”、“酷我”、“酷我音”、“酷我音乐”及拼音字符串“ku”、“kuwo”等，同时可以加上一些专名实体或行业词，如应用垂搜中的“音乐”、“视频”等功能需求词；

（2）利用所有的 sub-query 构建 AC 自动机；

（3）利用构建的 AC 自动机对历史搜索 query 及其拼音形式分别进行多模匹配，从中匹配出所有的前中后缀 sub-query，进而得到<sub-query，query>召回候选。

（4）按照一定策略（一般在前缀基础上）进行候选粗排并写到线上存储。（5）线上来一个请求 sub-query，直接查询存储获取召回结果，然后再基于训练的 pctr 预估模型或 pcpm 商业化导向进行重排，此时可以通过引入用户侧、context 侧等特征实现个性化排序。

3.7 意图识别

搜索意图识别是 qu 最重要却也最具挑战的模块，存在的难点主要有：

（1）用户输入 query 不规范：由于用户先验知识的差异，必然导致用户在通过自然语言组织表达同一需求时千差万别，甚至可能会出现 query 表达错误、遗漏等情况；

（2）歧义性&多样性：用户的搜索 query 表达不够明确带来的意图歧义性或用户本身搜索意图存在多样性，比如：搜索 query“择天记”可能是想下载仙侠玄幻类游戏，可能是玄幻小说类 app，也可能是想看择天记电视剧而下视频类 app。此时衍生出来的另一个问题是当某个 query 存在多个意图可能时，需要合理地量化各个意图的需求强度；

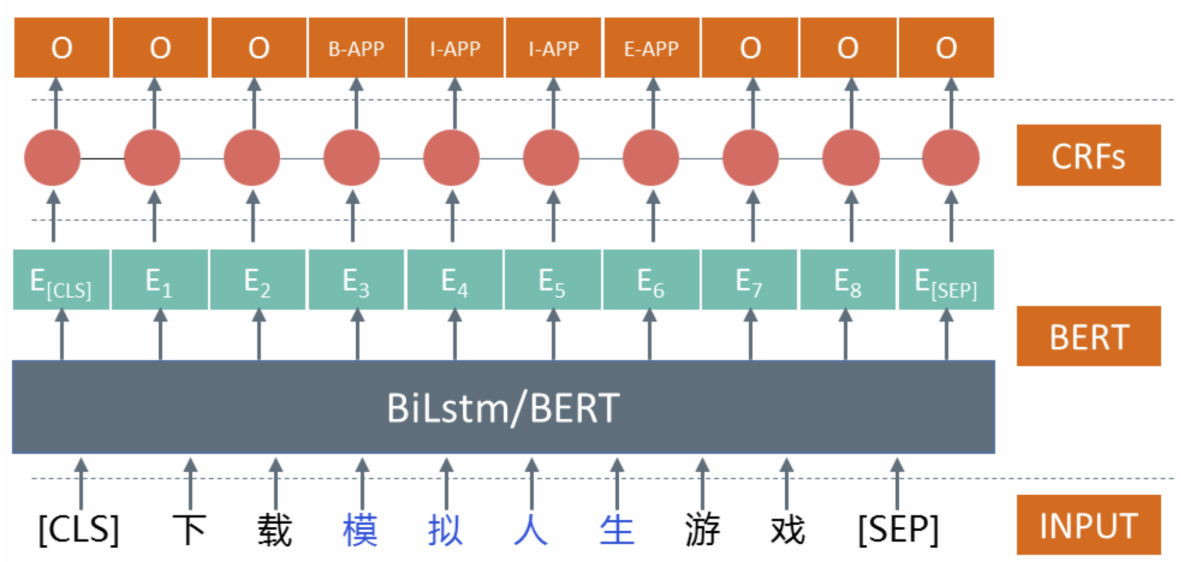
（3）如何根据用户及其所处 context 的不同实现个性化意图，比如用户的性别、年龄不同，搜索同一 query 的意图可能不一样，用户当前时刻搜索 query 的意图可能和上一时刻搜索 query 相关等。

根据用户意图明确程度的差别，搜索意图识别又可以细分为精准意图和模糊意图识别。

3.7.1 精准意图

所谓精准意图，是指用户通过 query 所表达的意图已经非常明确了，其需求可以比较置信地锁定为一个资源目标。精准意图需求在垂直搜索中尤为常见，以应用市场 app 搜索为例，用户搜索 query“下载王者荣耀”很明确就是想下载“王者荣耀”app，这时候将“王者荣耀”展现在结果列表首位才合理。当然一般排序模型拟合足够好的情况下也能将对应的精准资源排在首位，但是以下一些情况可能会引起排序不稳定进而导致对应精准资源没能置顶在首位的问题：1）长尾 query 行为特征稀疏，模型学习不够充分；2）引入用户个性化特征，排序结果因人而异；3）以商业化为导向的排序影响相关性体验。为此，需要一定策略识别出精准首位意图并将它们高优置顶，同时可以通过直达区产品形态给用户快速直达需求的搜索体验。

对于垂直搜索来说，精准意图一般是给定一个 query，找到与其意图精准对应的 item，可以通过文本匹配和 top 后验转化筛选出候选 item，然后通过从文本匹配、行为反馈、语义相似等方向构造样本特征训练 GBDT 等模型对<query,item>样本 pair 进行是否精准二分类。也可以尝试类似 DSSM 的语义匹配网络对 query 和 item 进行语义匹配。对于长尾 query 且完全文本包含 item 的情况，由于行为量不够丰富利用分类模型可能无法召回且直接进行文本匹配提取可能存在歧义性，此时可以视为 NER 任务通过训练 BiLSTM-CRF、BERT-CRF 等序列标注模型进行 item 实体的识别，再结合一些启发性策略及后验行为进行验证。



在 Google、百度等通用搜索中，用户可能会输入一些知识问答型的 query，此时用户的意图也比较明确，就是问题对应的答案。如搜索 query“刘德华的妻子是谁”，通过召回带“刘德华”、“妻子”字样的网页，用户估计也能找到答案，但如果能直接给出这个 query 的答案的话体验会更好，如下面百度搜索给出的结果。这时候可以归为 QA 问答任务来处理，一般需要结合知识图谱来做，也即 KBQA（Knowledge Based Question Answer）。传统的的 KBQA 做法是先对 query 进行词法句法以及语义解析，识别出其中的主要实体概念，再基于这些主题概念构造相应的查询逻辑表达式去知识库中进行查询及推理得到想要的答案。之后陆续有提出将问题和知识库中候选答案映射成分布式向量进行匹配，以及利用 CNN、LSTM、记忆网络等模型对应问题及候选答案向量建模优化等方法来进行 KBQA。



3.7.2 模糊意图

模糊意图就是指用户的搜索意图不会具体到某个目标资源，而是代表一类或多类意图，比如用户搜索“视频 app”，此时没有明确想下某款 app，可将其泛化到“视频类”tag 意图。模糊意图识别一般可以采用基于模板规则、行为统计反馈、分类模型等方法，这里主要会从意图分类及槽位解析两个方向进行阐述。

3.7.2.1 意图分类

在构建 query 意图分类模型之前，需要先制定一套意图标签体系用于全面覆盖用户意图需求。这里需要对 query 侧和 item 侧的标签体系进行统一以便于在预测出某个 query 的意图 tag 分布后直接用 tag 去倒排索引中召回属于这些 tag 的 item。

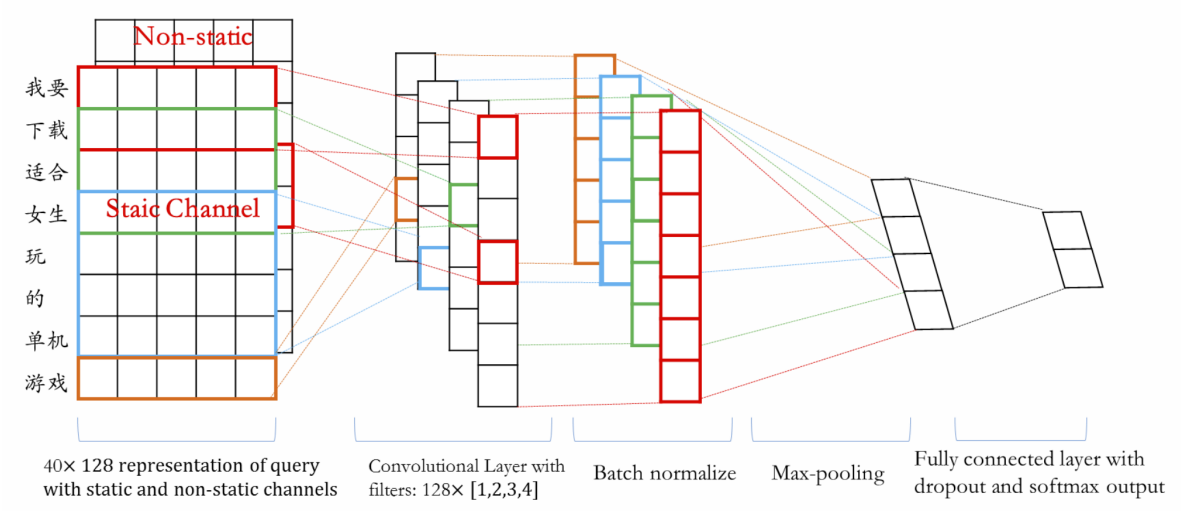
由于搜索 query 一般相对来说长度较短且意图存在多样性，意图分类可以归结为短文本多标签分类任务。在意图分类样本构造方面，可以通过关联用户搜索点击行为分布及进行 item 理解获得的 tag 或站点所属行业分类信息等自动构造样本，对于可能存在的样本类别不平衡的问题，需要对样本进行重降采样等处理，或采用主动学习等方法进行高效的样本标注。

至于模型方面，传统的文本分类主要采用向量空间模型 VSM 或进行其他特征工程来表征文本，然后用贝叶斯、SVM、最大熵等机器学习模型进行训练预测。随着 Word2vec、GloVe、Fasttext 等分布式词向量技术的兴起，打破了传统 nlp 任务需要做大量特征工程的局面，通过分布式向量对文本进行表示后再接入其它网络结构进行端到端分类训练的做法成为了主流。

如采用比较简单又实用的浅层网络模型 Fasttext，Fasttext 是从 Word2vec 衍生出来的，架构和 Word2vec 的类似，核心思想是将整篇文档的词及 n-gram 向量叠加平均得到文档向量，然后使用文档向量做 softmax 多分类。相对 Word2vec 的优点是输入层考虑了字符级 ngram 特征可以一定程度解决 oov 问题以及输出层支持进行有监督的任务学习。使用 Fasttext 训练简单，且线上 inference 性能也很高，但也正因为采用相对简单的浅层网络结构其准确率也相对较低。为此，进一步地可以尝试一些深度神经网络模型，如：TextRNN、TextCNN、Char-CNN、BiLSTM+Self-Attention、RCNN、C-LSTM、HAN、EntNet、DMN 等。

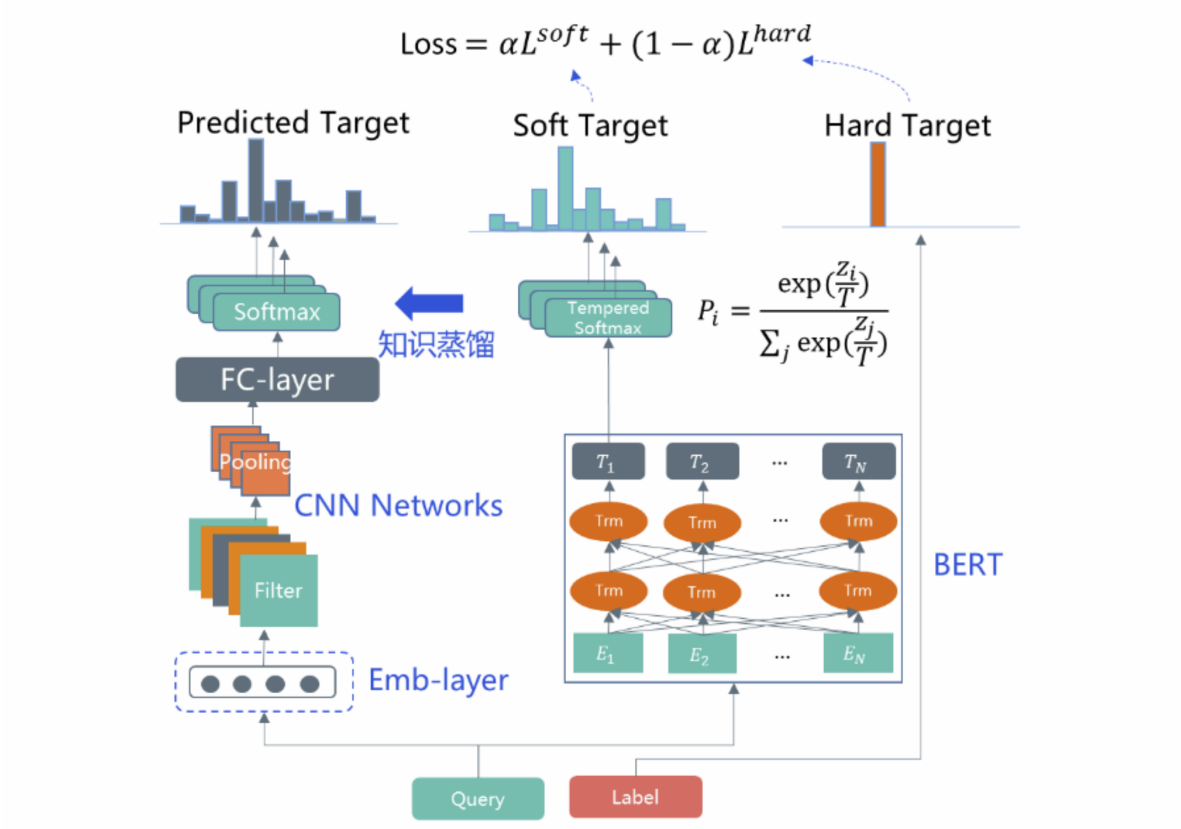
这些模型通过 CNN/RNN 网络结构提炼更高阶的上下文语义特征以及引入注意力机制、记忆存储机制等可以有效地优化模型的分类准确率。其实，对于 query 短文本分类来说采用相对简单的 TextRNN/TextCNN 网络结构就已经能达到较高的准确率了，其中 TextRNN 通过使用 GRU/LSTM 编码单元能更好地捕获词序和较长长度的上下文依赖信息，但由于采用 RNN 网络训练耗时相对较长。TextCNN 则主要通过不同 size 的卷积核捕获不同局部窗口内的 ngram 组合特征，然后一般通过 max-pooling 或 kmax-pooling 保留变长文本中一个或多个位置的最强特征转换为固定长度的向量再做 sigmoid/softmax 分类。

同时为进一步提升网络性能及加速模型收敛，还可以在网络中进一步考虑 dropout 及 batch normalize 等防过拟合机制，以及考虑在输入层融入 Word2vec、GloVe、Fastext 等模型预训练得到的 embedding，如下图在 cnn 输入层中加入预训练 embedding 组成双通道输入。虽然 TextCNN 对于捕获长程依赖信息方面会不如 TextRNN，考虑到 query 一般长度相对较短所以影响相对还好，而且其训练速度及在线 inference 性能也都比较符合要求。



除了从零开始训练或引入无监督预训练的隐式 embedding 表示，还可以通过引入显式的知识概念进一步丰富文本的语义表达，在有比较丰富的领域知识库的情况下进行 NER 实体识别，然后在模型的输入中可以融入这些实体知识特征，通过引入外部知识来优化分类的模型有 KPCNN、STCKA 等。由于 Word2vec、GloVe 等模型训练得到的词语 embedding 对不同的上下文来说都是固定的，无法解决一词多义等问题，基于此陆续提出的 ELMO、GPT、BERT 等深度预训练语言模型渐渐成为了 nlp 任务的标配，这些模型及其各种演进版本在多个 GLUE 基准中均取得了进一步的突破。

通过在大规模语料上预训练得到的 BERT 等模型能较好地动态捕获词语在不同上下文中的前后向语义表达，基于这些预训练模型在意图分类任务上进行 finetune 学习能进一步提升模型分类准确率。同样地，像 ERNIE 等模型通过引入外部知识也能进一步提升模型效果，尤其在一些垂直搜索有较多特定领域实体的情况下，可以尝试将这些领域实体知识融入模型预训练及 finetune 过程中。由于 BERT 等模型复杂度较高，进行在线 inference 时耗时也相对较高可能达不到性能要求，为此需要在模型精度和复杂度上做个权衡，从模型剪枝、半精度量化、知识蒸馏等方向进行性能优化。这里可以尝试通过权重分解及参数共享等方法精简优化的 ALBERT 模型，也可以尝试诸如 DistilBERT、TinyBERT 等知识蒸馏学习模型。



3.7.2.2 槽位解析

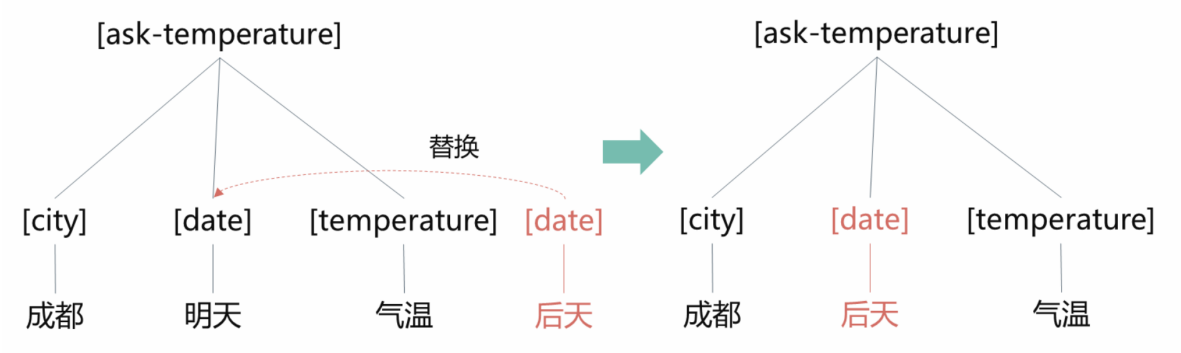
前面介绍的各种基于深度学习模型的意图分类能起到比较好相关性导航的作用，如将 query 意图划分到“天气”、“酒店”、“汽车”等意图体系中。但是针对更加复杂的口语化 query，我们需要进行识别提取出 query 中重要的意图成分以进行更全面的意图理解，此时仅进行意图分类是不够的。比如对于搜索 query“北京飞成都的机票”，意图分类模型可以识别出是“订机票”的意图，但是无法区分出 query 中的出发地和目的地信息，需要通过一定方法识别出“出发地”概念及其对应值是“北京”、“目的地”概念及其对应值是“成都”，基于此可以作出一些决策提供更直观的结果以提升用户搜索体验。



由于自然语言表达充满着歧义性，计算机肯定是无法直接理解的，需要将 query 表示成计算机能够理解的表示。类似于计算机语言无法理解高级编程语言一样，需要通过将高级编程语言代码编译成低级的汇编或二进制代码后计算机才能执行。所以我们也需要一个类似的“编译器”能将 query 按一定文法规则进行确定性的形式化表示，可以将这个过程称之为语义解析，前面提到的传统 KBQA 的做法也需要该技术将 query 转换成相应的形式化表示才能进一步执行推理等操作。

对于 query“北京飞成都的机票”，通过意图分类模型可以识别出 query 的整体意图是订机票，在此基础上进一步语义解析出对应的出发地 Depart=“北京”，到达地 Arrive=“成都”，所以生成的形式化表达可以是：Ticket=Order（Depart，Arrive），Depart={北京}，Arrive={成都}。如果 query 换成是“成都飞北京的机票”，同样的需要解析出 query 的意图是订机票，但是出发地和到达地互换了，所以语义解析过程需要除了需要对概念进行标注识别，还可以通过对概念进行归一来提高泛化性，比如这里“北京”、“成都”都可以归一为[city]概念，形式化表达变为：Ticket=Order（Depart，Arrive），Depart={$city}，Arrive={$city}，其中$city={北京、上海、成都...}。

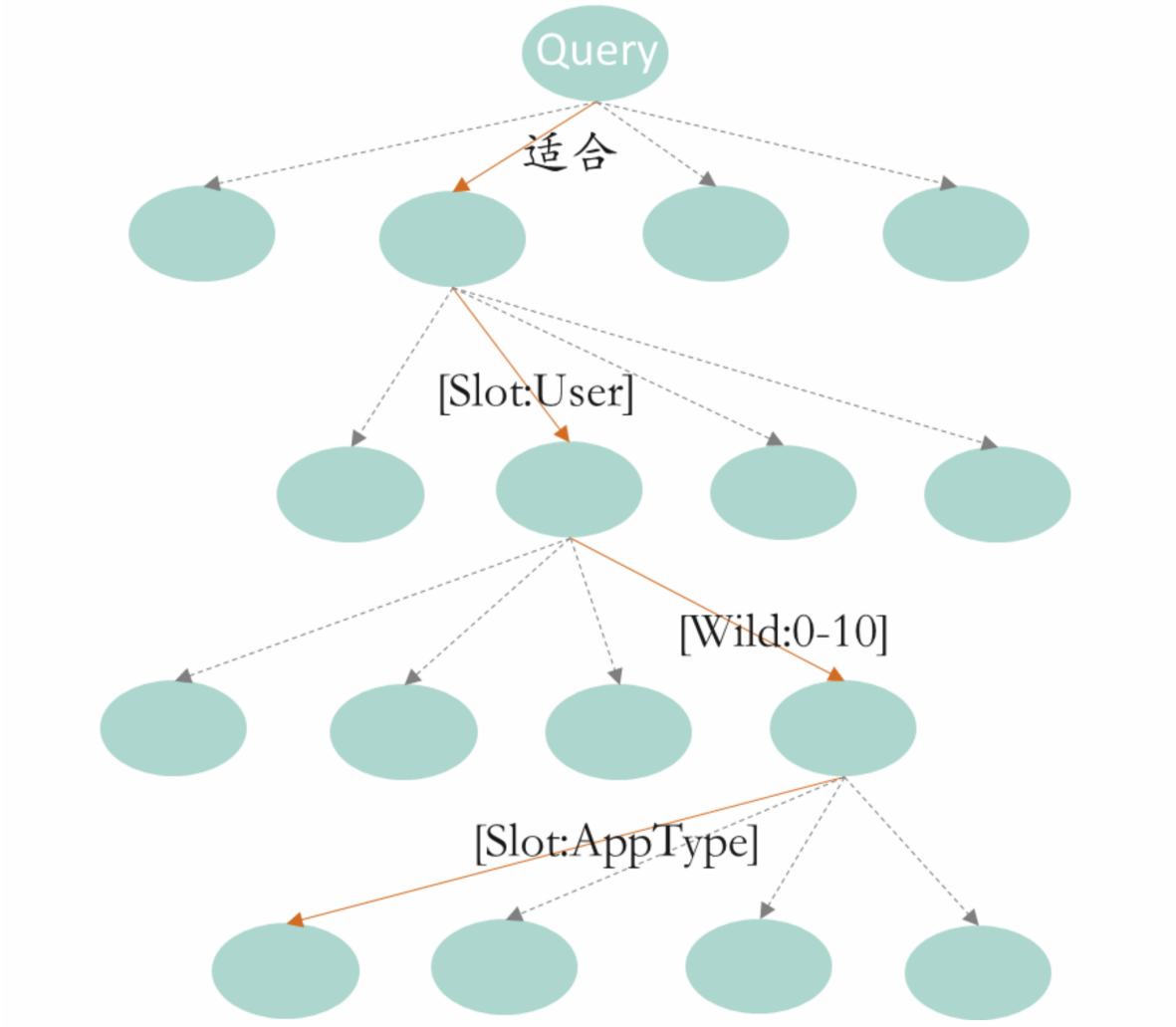
形式化表达可以递归地定义为一些子表达式的组合的形式，为进行语义解析，最主要的是确定一种形式化语言及对应的解析器，通常我们采用确定的上下文无关文法以确保形式化的每一部分都对应有一个解析树。得到 query 对应的形式化表示后，还可以进行一些解析树归并等形式化运算推演。为此，除了可以用来理解搜索 query 的结构化语义，语义解析技术也广泛应用于 QA 问答及聊天机器人中，如多轮对话中比较有挑战的上下文省略和指代消歧问题也可以一定程度通过将下文 query 对应的解析树合并变换到上文 query 对应的解析树中来解决。如下图例子所示，当用户在第一轮对话询问“成都明天气温？”时构造出相应的解析树，接着问“后天如何？”时就可以将日期进行替换后合并到之前的解析树中。

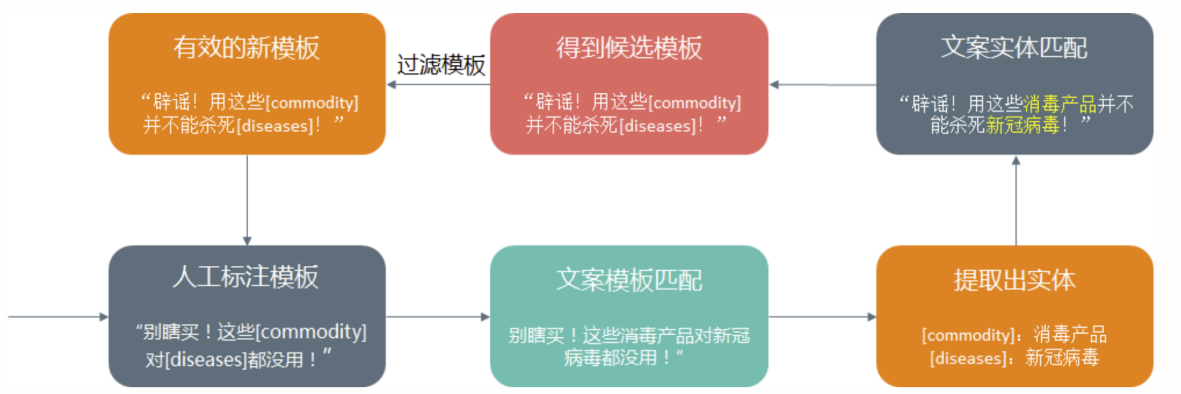


目前学术界和工业界在形式化语言和语义解析器方面均有一定的研究成果，通过一些基于统计、半监督、监督的方法来训练得到语义解析器，其中比较有名的开源语义解析器有 Google 的 SLING、SyntaxNet。简单的做法是通过人工制定的正则表达式和槽位解析的方法来进行语义解析，正则表达式相对好理解，槽位解析是指通过将具有相同模式的 query 归纳成模板，基于模板规则来解析用户 query 意图及意图槽位值。模板构成主要包括：槽位词、固定词、函数、通配符，其中槽位词是指能抽象到某一概念的词语集合，如：“北京”、“上海”、“成都”这些词都可以抽象到城市概念，固定词也即明确的某个词语，可以通过函数来匹配一些诸如数字组合等词，通配符可以匹配任意字符。

我们需要结合领域的业务知识来构造模板，构造过程需要保证模板符合语法句式规范，同时尽量保证其泛化性以覆盖更多的 query 问法。举个简单例子，构造模板：“适合[Slot:User][wild:0-10][Slot:AppType]”及相应的槽位词[Slot:User]={女生，男生，小孩，老人,...}、[Slot:AppType]={单机游戏，益智游戏，moba 游戏，...}，然后通过构建如下图所示的相应 trie 树，可以自上而下遍历及回溯解析出 query“适合女生玩的单机游戏”匹配上了这个模板，从而识别出 query 整体意图是询问女生类单机游戏，其中用户槽位值为“女生”，app 类型槽位值为“单机游戏”，“适合”是固定词，“玩的”匹配通配符[Wild:0-10]，基于这些槽位解析的结果，接下来可以进行一系列的决策。

槽位解析方法的优点是准确率高，可控性强，解释性好，但缺点是需要耗费较多地人力进行对模板及槽位词等进行标注，同时维护起来也比较麻烦。可以考虑结合一些策略方法来一定程度减少人工标注量，如基于前面提到的同义词挖掘技术及词向量化挖掘同位/上下位词等方法来辅助槽位词的标注，以及在人工进行模板标注的基础上采用 bootstrapping 迭代挖掘构造出更多的模板。槽位解析的具体实现可以参考的开源项目 Snips-nlu，进行意图识别的同时从自然语言句子中解析提取结构化槽位信息。





3.8 敏感识别

敏感识别模块主要对 query 进行是否带有色情、反动、赌博、暴力等敏感话题的识别，如果识别出 query 中存在敏感话题可以进行定向改写到相对合适的 query 或者给用户做搜索引导提示等处理。敏感识别可以归为分类问题，最简单的做法就是词表匹配，按不同的敏感话题人工输入一批词库，复杂点就训练一个分类模型进行多分类，传统的 SVM、最大熵分类或者 TextCNN、TextRNN 等模型都能比较好地 work。

3.9 时效性分析

用户的搜索需求可能会显式或隐式地带有一定的时效性要求，如：“最近上映的好看电影”带有显式的时间词“最近”，而“疫情进展”则隐式地表达了解最新情况的需求。时效性大概可以分为三种：持续时效性，周期时效性，准/实时时效性。其中持续时效性是指 query 一直具有时效性，如：“美食推荐”，周期时效性是指具有周期性、季节性的事件或需求，如：“世界杯”、用户在冬季搜索“上衣”等，而准/实时效性是指近期发生或突然发生的事件。

通过分析出 query 的时效性需求等级的不同，在召回 item 时就可以针对性地做一些过滤或者在排序时进行时效性调权。其中显式的时效性需求因为带有时间关键词，可以通过规则匹配或训练分类模型进行判断识别，而隐式表达的时效性则可以通过按时间维度分析历史搜索 qv 行为、实时监测最新搜索 qv 变化情况以及综合考虑搜索意图及当前时间上下文等方法来判断识别。

4.结语

人工智能的发展是循序渐进的，需要经历从计算智能到记忆智能、感知智能、认知智能，最终到达创造智能等多个发展阶段才能称得上是真正的人工智能。目前业界在计算、记忆和感知方面已经做得相对比较成熟，但是在认知智能方面则还需要有进一步突破，而 nlp 恰好是实现认知智能最重要的一环，为此 nlp 也被称为人工智能皇冠上的一颗明珠。同样地，真正的语义搜索也远未到来，对 query 的语义理解能力很大程度决定着整个搜索的智能程度。本文仅为个人在进行 query 理解相关优化项目的一些简单总结，主要从搜索的角度对 query 理解涉及的各个重要模块的概念及其对应的一些方法进行阐述。文中暂未涉及太深的技术探讨，希望能帮助到大家对搜索 qu 相关概念有一个初步的认识，起到抛砖引玉的效果，如有错误及不足之处，还请不吝交流指出。

5.参考文献

[1] Kim, Y. (2014). Convolutional Neural Networks for Sentence Classification. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014), 1746–1751.

[2] Kalchbrenner, N., Grefenstette, E., & Blunsom, P. (2014). A Convolutional Neural Network for Modelling Sentences. Acl, 655–665.

[3] Armand Joulin, Edouard Grave , Piotr Bojanowski, omas Mikolov. Bag of Tricks for Efficient Text Classification. arXiv, 2016.

[4] Dzmitry Bahdanau, KyungHyun Cho, Yoshua Bengio. Neural Machine Translation By Jointly Learning To Align And Translate. ICLR, 2015.

[5] Sanjeev Arora, Yingyu Liang, Tengyu Ma. A Simple but Tough-to-Beat Baseline for Sentence Embeddings,2016.

[6] C Zhou, C Sun, Z Liu, F Lau. A C-LSTM neural network for text classification. arXiv, 2015.

[7] X Zhang, J Zhao, Y LeCun. Character-level convolutional networks for text classification. NIPS, 2015.

[8] S Lai, L Xu, K Liu, J Zhao. Recurrent convolutional neural networks for text classification. AAAI, 2015.

[9] Z Lin, M Feng, CN Santos, M Yu, B Xiang. A structured self-attentive sentence embedding. arXiv, 2017.

[10] J Howard, S Ruder. Universal language model fine-tuning for text classification. arXiv, 2018.

[11] J Devlin, MW Chang, K Lee, K Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv, 2018.

[12] J Wang, Z Wang, D Zhang, J Yan. Combining Knowledge with Deep Convolutional Neural Networks for Short Text Classification. IJCAI, 2017.

[13] J Chen, Y Hu, J Liu, Y Xiao, H Jiang. Deep short text classification with knowledge powered attention. AAAI,2019.

[14] H Ren, L Yang, E Xun. A sequence to sequence learning for Chinese grammatical error correction. NLPCC,2018.

[15] Y Hong, X Yu, N He, N Liu, J Liu. FASPell: A Fast, Adaptable, Simple, Powerful Chinese Spell Checker Based On DAE-Decoder Paradigm. EMNLP, 2019.

[16] I Antonellis, H Garcia-Molina. Simrank++ query rewriting through link analysis of the clickgraph. WWW'08.

[17] G Grigonytė, J Cordeiro, G Dias, R Moraliyski. Paraphrase alignment for synonym evidence discovery. COLING,2010.

[18] X Wei, F Peng, H Tseng, Y Lu, B Dumoulin. Context sensitive synonym discovery for web search queries. CIKM,2009.

[19] S Zhao, H Wang, T Liu. Paraphrasing with search engine query logs. COLING, 2010.

[20] X Ma, X Luo, S Huang, Y Guo. Multi-Distribution Characteristics Based Chinese Entity Synonym Extraction from The Web. IJISA, 2019.

[21] H Fei, S Tan, P Li. Hierarchical multi-task word embedding learning for synonym prediction. KDD,2019.

[22] M Qu, X Ren, J Han. Automatic synonym discovery with knowledge bases. KDD,2017.

[23] J Shen, R Lyu, X Ren, M Vanni, B Sadler. Mining Entity Synonyms with Efficient Neural Set Generation. AAAI,2019.

[24] A Vaswani, N Shazeer, N Parmar. Attention is all you need. NIPS, 2017.

[25] Berant J, Chou A, Frostig R, et al. Semantic Parsing on Freebase from Question-Answer Pairs. EMNLP,2013.

[26] Cai Q, Yates A. Large-scale Semantic Parsing via Schema Matching and Lexicon Extension. ACL,2013.

[27] Bordes A, Chopra S, Weston J. Question answering with subgraph embeddings. arXiv,2014.

[28] Dong L, Wei F, Zhou M, et al. Question Answering over Freebase with Multi-Column Convolutional Neural Networks. ACL,2015.

[29] E Malmi, S Krause, S Rothe, D Mirylenka. Encode, Tag, Realize: High-Precision Text Editing. arXiv, 2019.

机器学习

微信「扫一扫识物」 的背后技术揭秘

作者： breezecheng，腾讯 WXG 应用研究员

> 微信扫码已经深入人心， 微信扫物 12.23 日 ios 版本正式上线，从识别特定编码形态的图片(二维码/小程序码/条形码/扫翻译)，到精准识别自然场景中商品图片(鞋子/箱包/美妆/服装/家电/玩具/图书/食品/珠宝/家具/其他商品)，有哪些难点需要去克服? 扫物以图片(视频)作为媒介，聚合微信内部有价值的生态内容如电商，百科，资讯进行展示， 会催生哪些新的落地场景?本文将细细道来.

一. 扫一扫识物概述

1.1 扫一扫识物是做什么的?

扫一扫识物是指以图片或者视频(商品图：鞋子/箱包/美妆/服装/家电/玩具/图书/食品/珠宝/家具/其他商品)作为输入媒介来挖掘微信内容生态中有价值的信息（电商+百科+资讯，如图 1 所示），并展示给用户。这里我们基本覆盖了微信全量优质小程序电商涵盖上亿商品 SKU，可以支持用户货比 N 家并直接下单购买，百科和资讯则是聚合了微信内的搜一搜、搜狗、百度等头部媒体，向用户展示和分享与该拍摄商品相关的资讯内容。



百闻不如一试，欢迎大家更新 ios 新版本微信 → 扫一扫 → 识物自行体验，也欢迎大家通过识物界面中的反馈按键向我们提交体验反馈。图 2 即为扫物实拍展示。

视频 1

图 2. 扫一扫识物实拍展示

1.2 扫一扫识物落地哪些场景?

扫一扫识物的目的是开辟一个用户直达微信内部生态内容的一个新窗口，该窗口以用户扫图片的形式作为输入，以微信生态内容中的百科、资讯、电商作为展示页提供给用户。除了用户非常熟悉的扫操作，后续我们会进一步拓展长按识图操作，将扫一扫识物打造成用户更加触手可及的运用。扫一扫识物的落地场景如下图所示，主要涵盖 3 大部分：

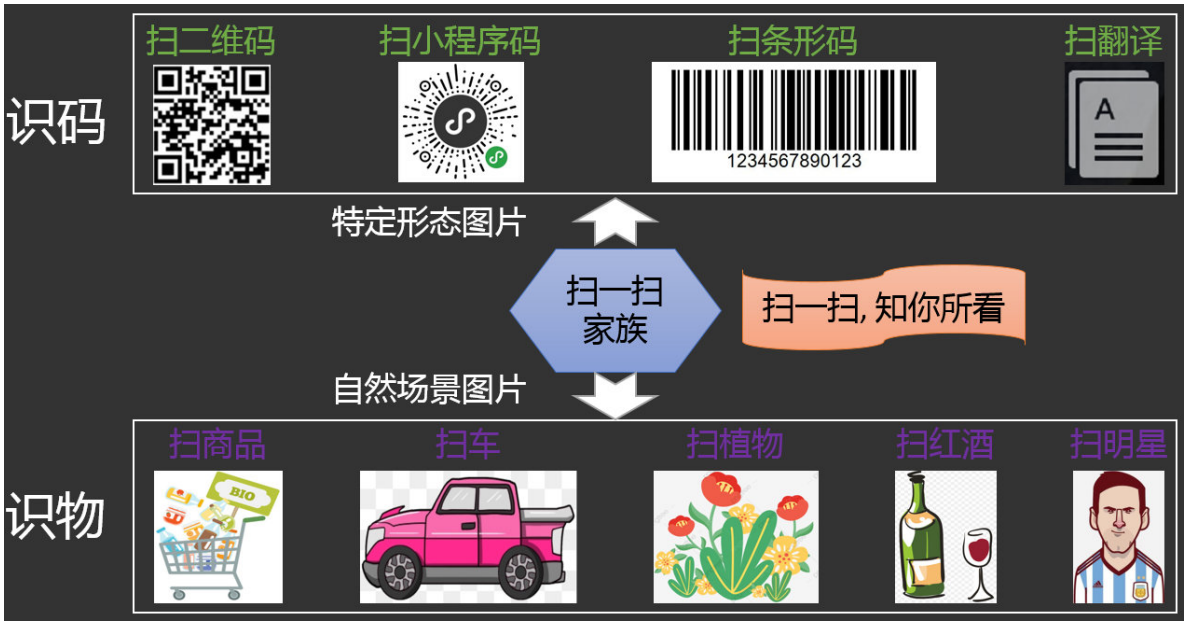
a.科普知识。用户通过扫物体既可以获得微信生态中关于该物体相关的百科、资讯等小常识或者趣闻，帮助用户更好的了解该物体；

b.购物场景。同款搜索功能支持用户对于见到的喜爱商品立即检索到微信小程序电商中的同款商品，支持用户扫即购；

c.广告场景。扫一扫识物可以辅助公众号文章、视频更好的理解里面嵌入的图片信息，从而更好的投放匹配的广告，提升点击率。

1.3 扫一扫识物给扫一扫家族带来哪些新科技?

对于扫一扫，大家耳熟能详的应该是扫二维码、扫小程序码，扫条形码，扫翻译。无论是各种形态的码还是文本字符，都可以将其认为是一种特定编码形态的图片，而识物则是识别自然场景图片，对于扫一扫家族来说是一个质的飞跃，我们希望从识物开始，进一步拓展扫一扫对自然场景图片的理解能力，比如扫酒，扫车，扫植物，扫人脸等等服务，如下图 3 所示。



二. 扫一扫识物技术解析

2.1 扫一扫识物整体框架

下面我们为大家重点介绍扫一扫识物的完整技术实现方案，图 4 展示的是扫一扫的整体框架示意图。该框架主要包含 4 大部分：

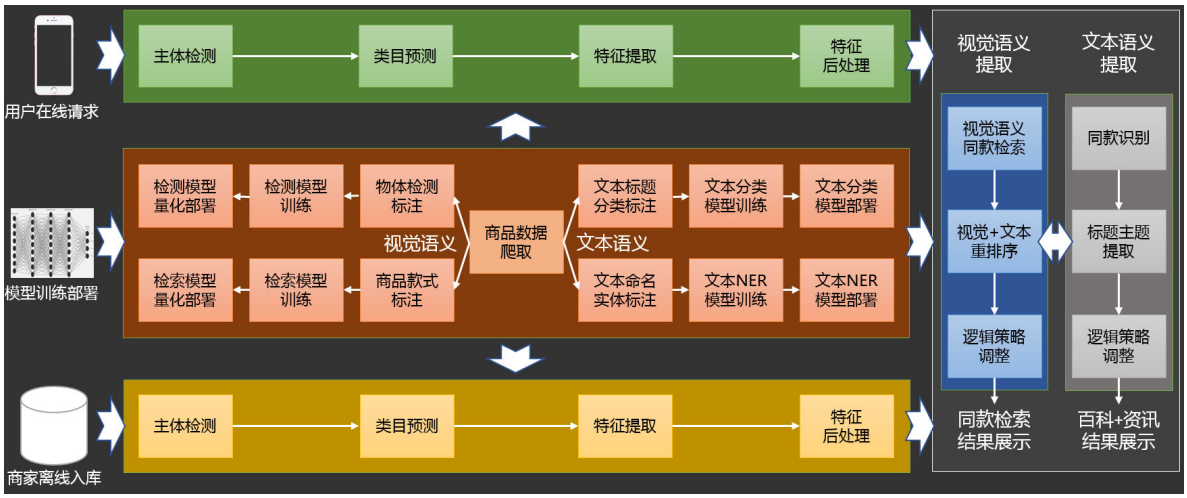
1）用户请求环节；

2）商检离线入库环节；

3）同款检索+资讯百科获取环节；

4）模型训练部署环节。

四大环节抽取核心技术模块可以总结为三个，即为数据构建、算法研发、平台建设，我们将一一道来。



数据构建

AI 时代数据为王，数据构建的目的就是为了更好的服务于 AI 算法，比如对于用户请求图、商家入库图都需要进行主体检测、类目预测、特征提取，都需要有高质量的训练数据来支撑检测模型、分类模型以及检索模型。一言以蔽之，数据决定了整个扫一扫识物性能上限。

算法研发

算法研发是为了充分的利用已有的数据，为识物的每一个环节如检测、类目预测，检索特征提取都在精度、速度上到达最优的折中，从而实现用户任意商品请求都能获得精准的同款召回，以及更加相关的资讯展示。算法研发的好坏决定了扫一扫识物的性能下限。

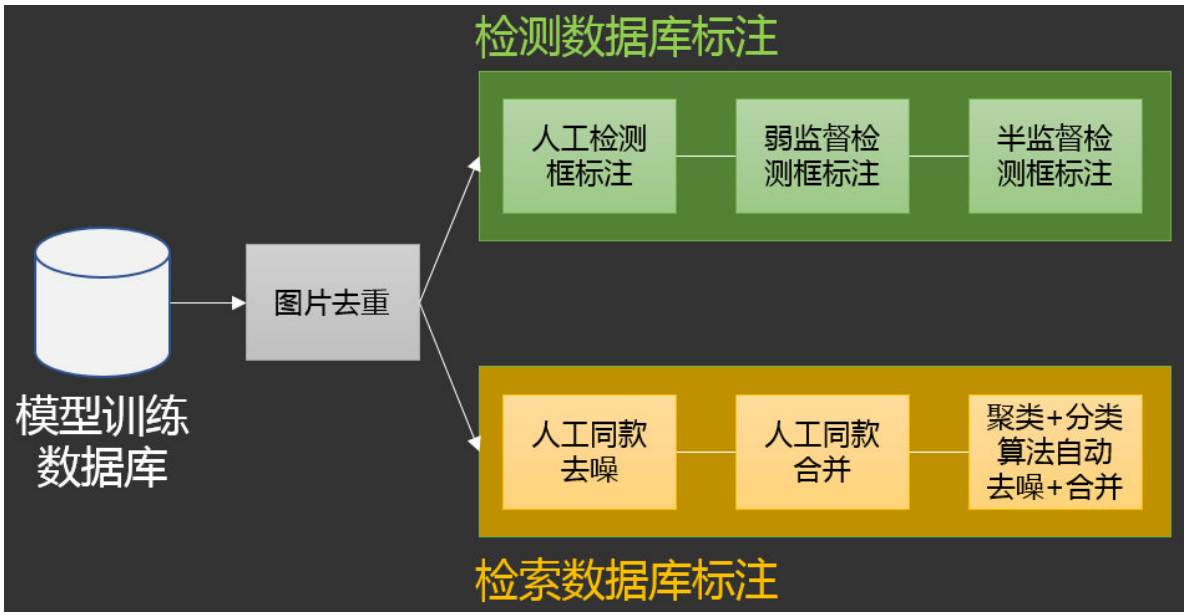
平台建设

无论是数据建设，算法研发，模型上线都离不开一个好的平台支持，我们为扫一扫识物从数据清洗，模型训练，部署，上线打造了一个完整的平台。可以说，平台建设关乎研发效率，决定了扫一扫识物能否实现上线。

2.2 扫一扫识物数据建设

扫一扫识物数据构建分为两大块，一大块是用于模型训练的训练数据建设，另一大块则是支撑用户任意商品检索请求的线上检索库构建。

模型训练数据库构建 训练数据库主要是支援模型训练，如同款检索中需要的物体检测模型，类目预测模型以及同款检索模型等，在百科资讯搜索中则需要训练商品标题文本分类模型，命名实体识别模型等。本篇文章我们重点关注视觉语义这一块的算法策略，对于百科资讯用到 NLP 算法我们在下一篇文章中详细阐述。2.3 章节中我们将阐述数据构建中用到的图片去重，检测数据库标注用到的半监督学习算法，以及检索数据构建提出的半自动同款去噪+合并算法。



在线检索数据库构建 在线检索数据库的覆盖范围至关重要，决定了能否支持用户的任意商品搜索请求。我们采用定向导入、长尾加爬、访问重放、主动发现四种策略不断扩展我们的商家图规模，目前已覆盖 95%+常见商品。这一数字还在不断上涨中，我们希望后续对用户的任意商品请求都能够精确召回同款商品。

2.2.1 图片去重

无论是检测数据库还是检索数据库，第一步都是清理掉重复图片，这样既可以降低存储压力，也能够降低模型训练压力。常用的图片去重算法有如下 2 种：

1）MD5 去重，去除完全相同的图片；

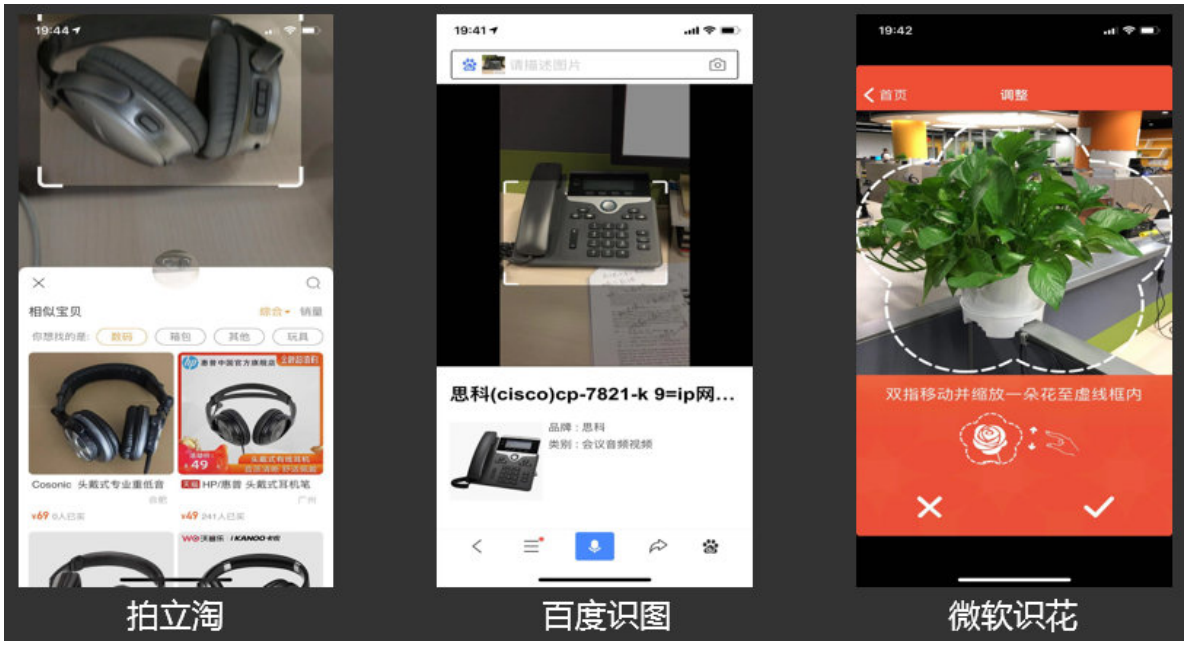
2）哈希去重，除完全重复图外，还能去除在原图上进行了些简单加工的图片，如改变原图的亮度、尺度、对比度、边沿锐化、模糊、色度以及旋转角度，这些图片保留意义也不大，因为在模型训练中采用数据增强策略可以覆盖到。常用的哈希去重主要有 aHash，dHash，pHash 等，详细理解可以参阅相关文章[1，2]，我们重点对比各个去重算法的速度和鲁棒性，如下图 6 所示。

三种图像哈希的比较		时间/s			相似度		
		aHash	pHash	dHash	aHash	pHash	dHash
原图(a)	原图(a)	0.0005	0.0118	0.0001	100.00%	100.00%	100.00%
原图(a)	亮度(b)	0.0008	0.0145	0.0002	92.19%	94.53%	93.75%
原图(a)	缩放(c)	0.0007	0.0288	0.0002	98.44%	97.27%	98.44%
原图(a)	对比度增强(d)	0.0007	0.0149	0.0002	92.19%	97.27%	95.31%
原图(a)	锐化(e)	0.0016	0.0162	0.0003	93.75%	90.23%	90.62%
原图(a)	模糊(f)	0.0011	0.0211	0.0004	93.75%	90.62%	92.19%
原图(a)	色度增强(g)	0.0008	0.0159	0.0002	92.19%	97.27%	95.31%
原图(a)	旋转(h)	0.0008	0.0154	0.0002	50.00%	56.25%	53.12%

对比图 6 中的数字可知，dHash 的去重速度最快，因为只需要计算临近像素的亮度差异，非常简单易处理，而且对于图片简单的 ps 操作具有较好的容忍程度，除图片旋转外，诸如亮度、尺度、对比度、边沿锐化、模糊、色度等较小改变都具有很好的抗干扰能力，有助于我们清理更多的无效图片。最终我们也是采用 dHash 对我们训练库进行去重操作，11 类全量爬虫商品库中，大概清理掉了 30%的重复图片，累计有 300w 左右。

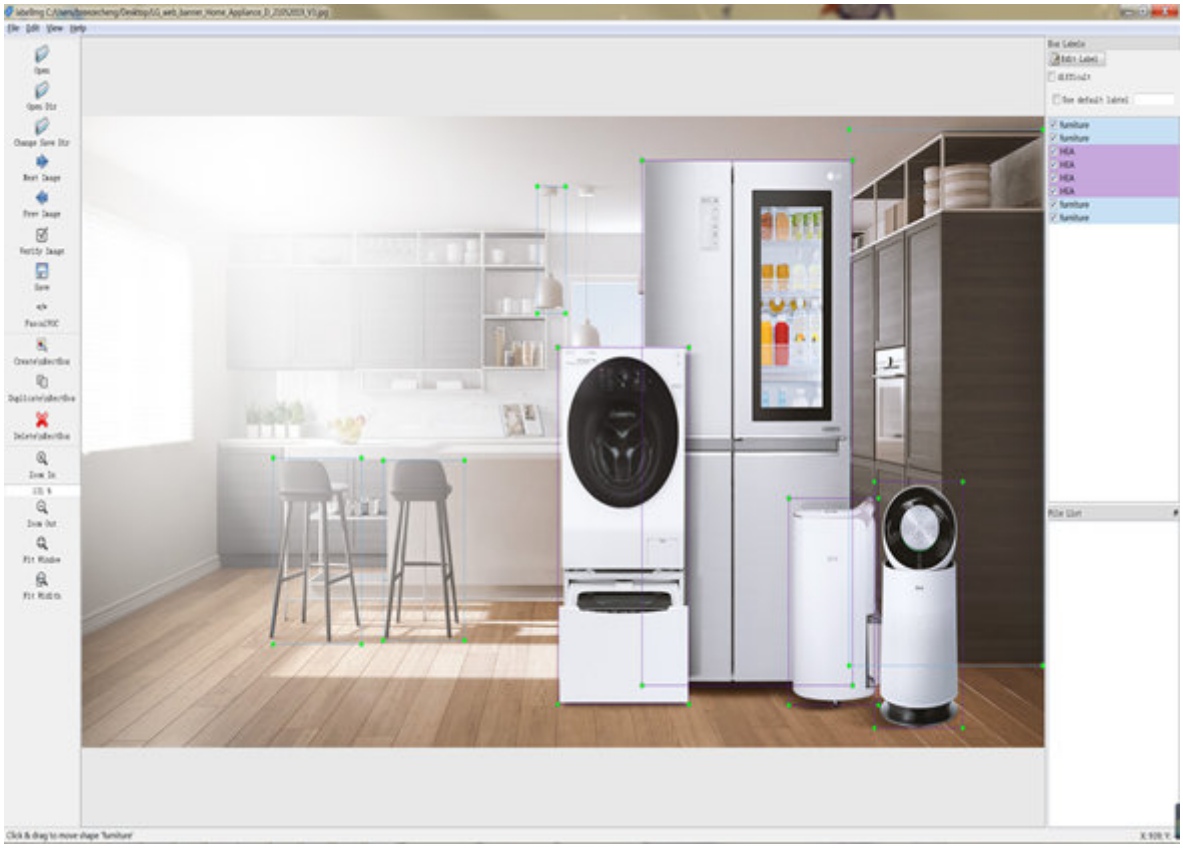
2.2.2 检测数据库构建

从图 4 展示的整体框架可知，扫一扫识物的首要步骤就是主体检测，即先定位用户感兴趣的区域，去除掉背景对后续环节的干扰。主体检测基本是大部分以图搜图产品的公认首要操作，如下图 7 所示的阿里的拍立淘，百度的拍照识图，以及微软的识花小程序。当然主体检测的算法各有差异，如拍立淘采用的是物体检测算法，百度识图采用的是显著性区域预测，微软识花需要用户配合定位。为了解放用户，我们希望算法能够自动定位商品区域，考虑到显著性区域预测很难处理多个商品出现在视野的情况，类似拍立淘，我们采用更加精确的物体检测算法来定位商品位置，并选择置信度最高的商品进行后续的商品检索以及资讯百科展示。

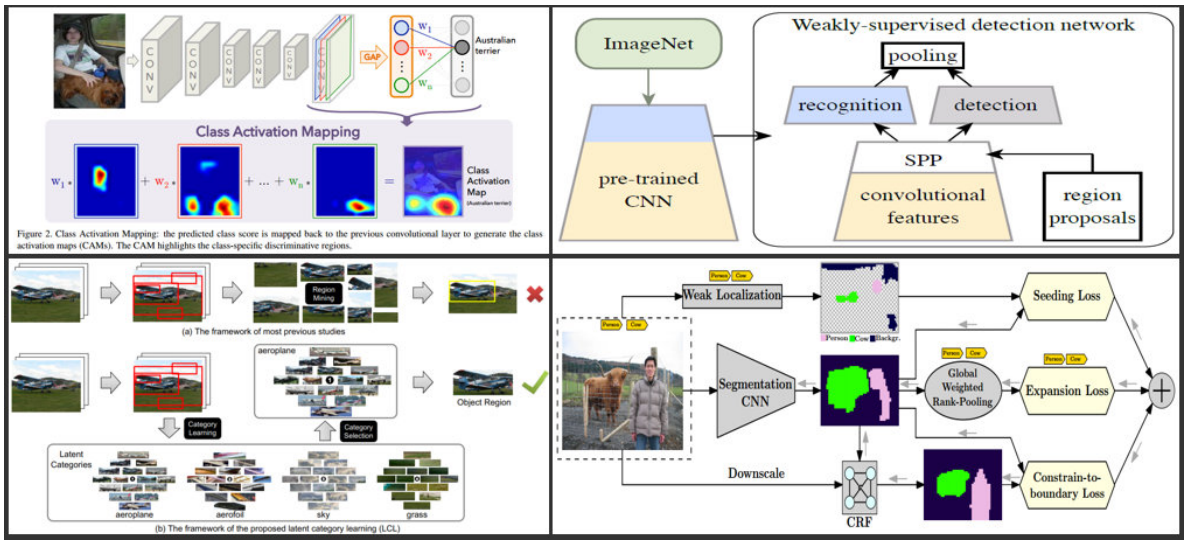


当然物体检测模型离不开检测数据库的支撑，这里我们对比三种标注物体 boundbox 位置和类别的方法，即人工检测标注，弱监督检测标注以及半监督学习检测标注。

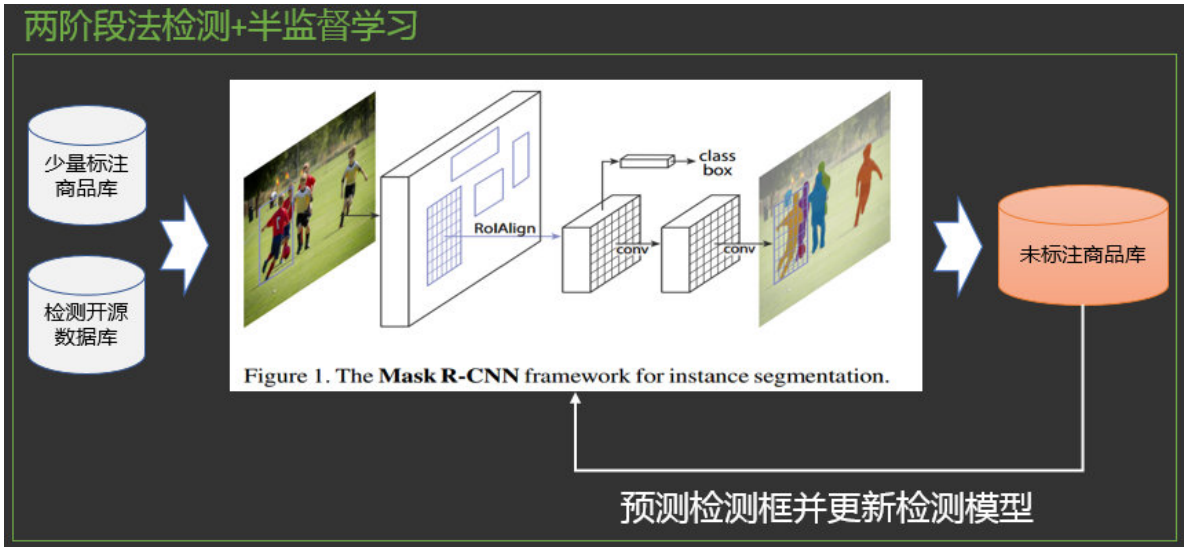
人工检测标注常用的人工检测标注工具 labelimg 如下图所示，我们需要标注爬虫的商品库中 11 类商品出现的位置以及对应的类别标签。考虑到人工标注的时间和金钱成本都非常巨大，我们只采用该策略标注少量的样本，更多的采用后续的算法进行自动检测框标注。



弱监督检测标注该算法的核心思想是标注图片中所含物体的类别相比标注框+类别的时间成本要低很多，如何只利用全图类别信息来自动推断物体的位置信息，从而完成自动检测标注呢？学术界和工业界有大量的研究者对这一方向进行了研究和探索，主要思路都是挖掘图片中的局部显著性区域，用其来表征全图的类别语义信息，如图 9 列举了几篇代表性文章。图 9 左下角算法重点阐述下，它是业内第一篇用深度学习来做弱监督检测的文章，实验室师兄研发，我们还基于该算法参加过 ImageNet14 的竞赛，拿了一个小分支的冠军。尽管愿景很美好，弱监督检测算法有个严重的缺陷，就是很容易拟合到局部区域，比如从大量猫的图片中拟合到的位置是猫脸区域，而很难定位到完整的猫的位置，这对于我们同款商品检索来说是难于接受的，我们需要精确召回同一款商品（细粒度检索），所有信息量都非常重要，因而该算法基本被我们 pass 掉了。

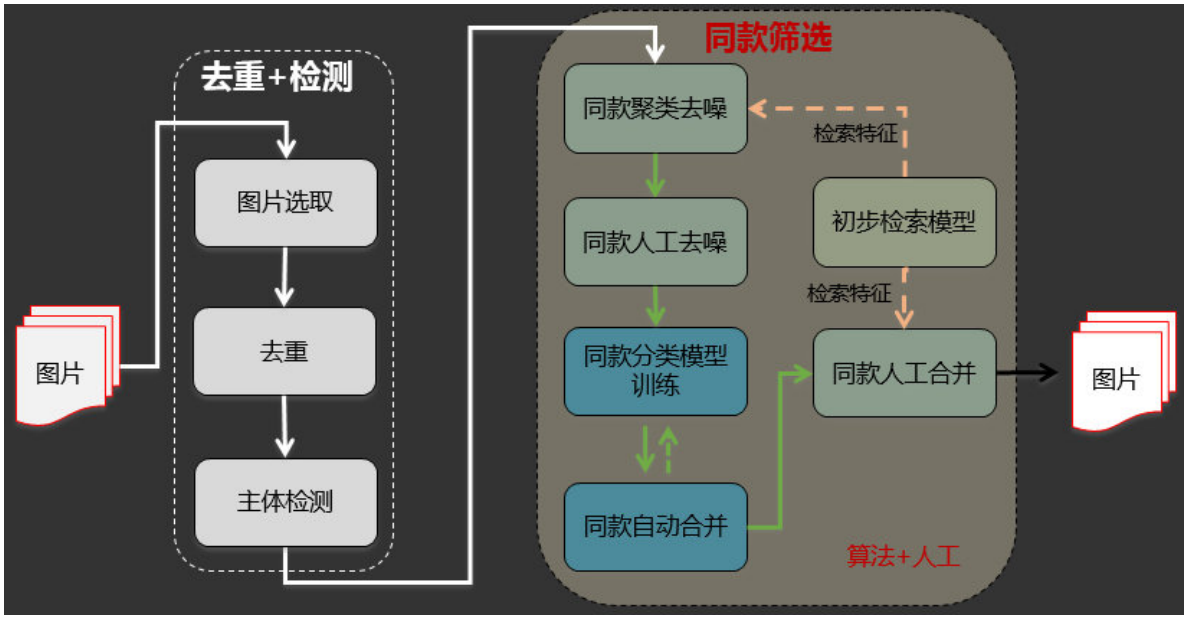


半监督检测标注相比弱监督检测算法，半监督检测算法更加贴近任务本身：基于少量的人工标注检测框+检测开源数据库初始化检测模型，然后用该模型去自动标注剩下的商品数据库，并用增加的标注来重新更新检测模型，之后迭代进行模型更新和自动标注。当然为了控制上述环节持续正向激励，我们为对置信度较低的标注样本进行人工校正，算法示意图如下所示。基于该算法，我们完成了整个商品检测数据库的标注，11 类全量商品库标注了上百万个检测框，其中人工启动标注只有十几万，其他的基本都是基于算法自动标注，大大节省了标注的时间和金钱成本。



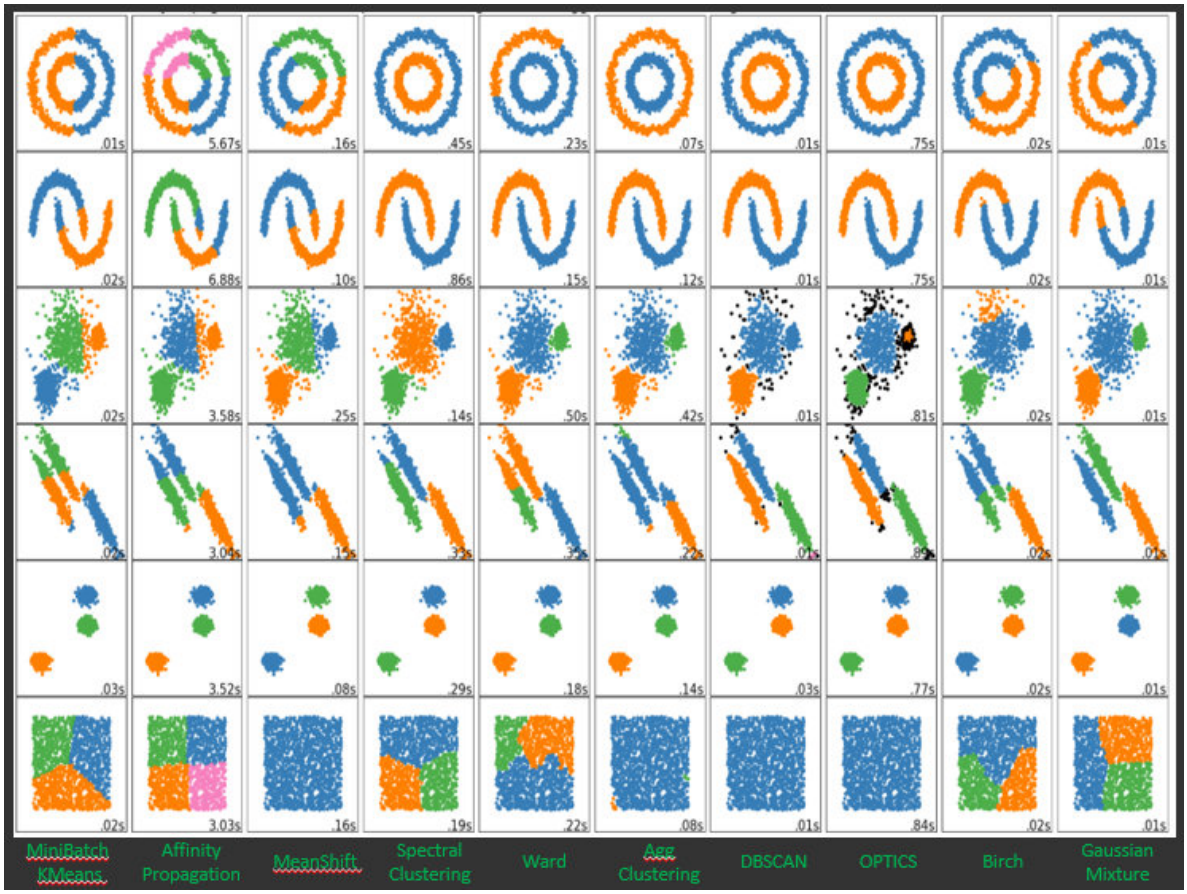
2.2.3 检索数据库构建

完成了图片去重和主体检测之后，大家一个自然的想法就是，能否直接用这批抠图后的商品图结合 SKU 货号进行检索模型的训练，答案是否定的，抠图之后的商品图还存在两大问题：1.同款噪声问题，即同一个 SKU 下面存在非同款的图片，这可能是由于用户上传错误图片、商家展示的是局部细节图、检测抠图错误等因素导致；2.同款合并问题，不同的 SKU 可能对应的是同一个款式商品，不加以合并，会导致分类类别数目急剧膨胀，而且模型难于收敛。因此，在训练检索模型之前，我们需要完成同款去噪和同款合并两个环节，我们提出了基于自动聚类的同款去噪算法和基于混淆分类矩阵的同款合并算法，如下图 11 所示。后续我们将分别对这个算法进行解析。



2.2.3.1 同款去噪

我们采用聚类算法来自动去除每个商品 SKU 中存在的噪声图片。我们研究了常用的聚类算法，如下图 12 所示，



关于这些算法的理论及实现流程，大家感兴趣可以参阅[7]。我们在图 13 中对上述聚类算法在常见指标上进行了对比分析，考虑到商品 SKU 聚类的特性，我们更加关注聚类算法的抗噪声能力，对不同 SKU 特征分布的适应性以及处理速度，综合分析实践后，我们选择了 DBSCAN 作为我们的聚类算法，并对其进行改造来更加适配商品的聚类去噪，我们称其为层次法 DBSCAN。

聚类算法	输入参数	试用数据	可扩展性	抗噪声	时间复杂
MiniBatch KMeans	类目数	类目不多,平坦分部	较高	较低	较低
Afinity Propagation	参考度	类目多,非平坦分布	较低	较低	较高
MeanShift	滑窗半径	类目多,非平坦分布	较低	较低	较高
Spectral Clustering	类目数	类目少,非平坦分布	一般	较低	较高
Ward	类目数	类目多	较高	较低	较高
Agg Clustering	类目数	类目多	较高	较低	较高
DBSCAN	邻域半径,最小样本数	非平坦分布	较高	较高	较低
OPTICS	邻域半径,最小样本数	非平坦分布	较高	较高	较高
Birch	半径T,CF数目B和L	数目多,类目多	高	较低	较低
Gaussian Mixture	类目数	平坦分布	较低	较低	较低

层次法 DBSCAN 主要分为两个环节，分别为 step1.寻找距离最紧致的最大类簇，以及 step2.重访噪声样本， 捞回同款困难样本，增加多样性。下面我简要介绍这两个步骤。

层次法 DBSCAN 步骤 1 该步骤的目的是挑选 SKU 中距离最紧致的最大类簇，因为 SKU 中的同款样本相对噪声样本分布更有规律，数目也一般会更多。算法示意图如下图 14 所示，我们邻域内最小样本数目设置为 2，将所有的核心点联通后，假设有多个类簇，我们选择数目最多的类簇作为我们挑选的商品图片，剩下的样本作为噪声样本。

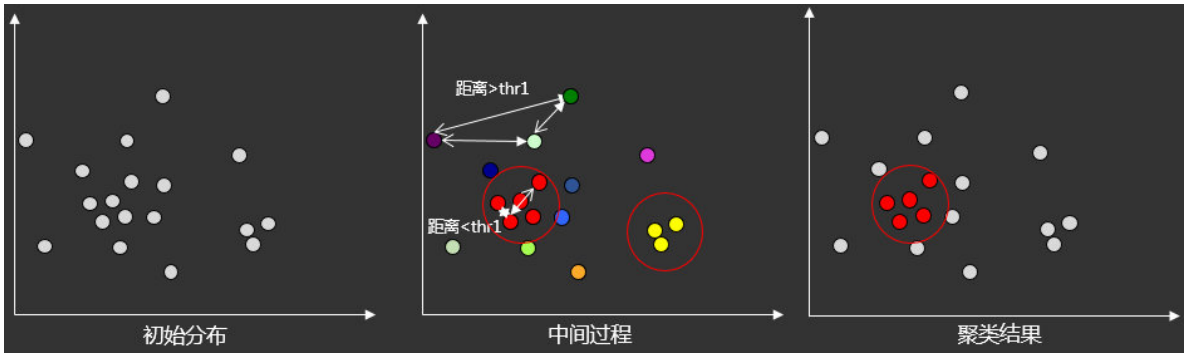
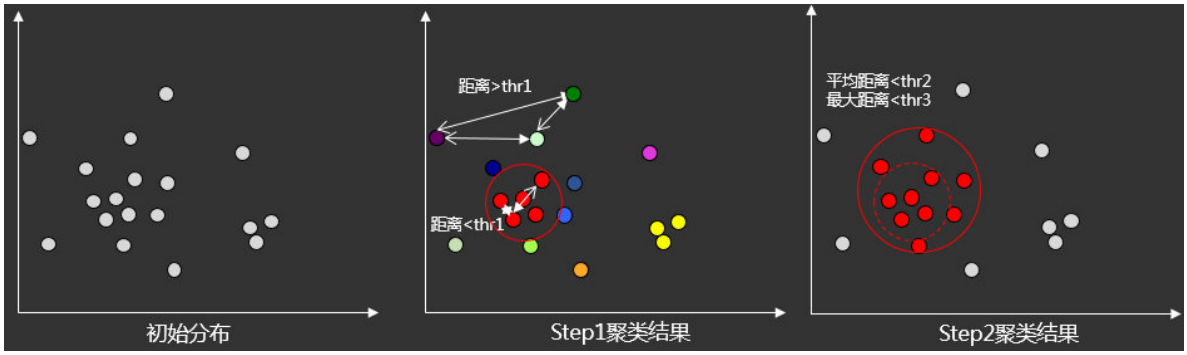


图 15 展示了某个实际 SKU 的步骤一聚类效果，从中我们可以发现最大聚类由于控制的阈值很严格，样本分布很单一化，而在噪声中实际有一些误为噪声的困难正样本（用红圈示意）。这些困难的正样本对于提升检索模型的鲁棒性和泛化能力非常重要，因而我们需要把噪声中的困难正样本捞回到左边的最大类簇中来，即为步骤 2 完成的事情。

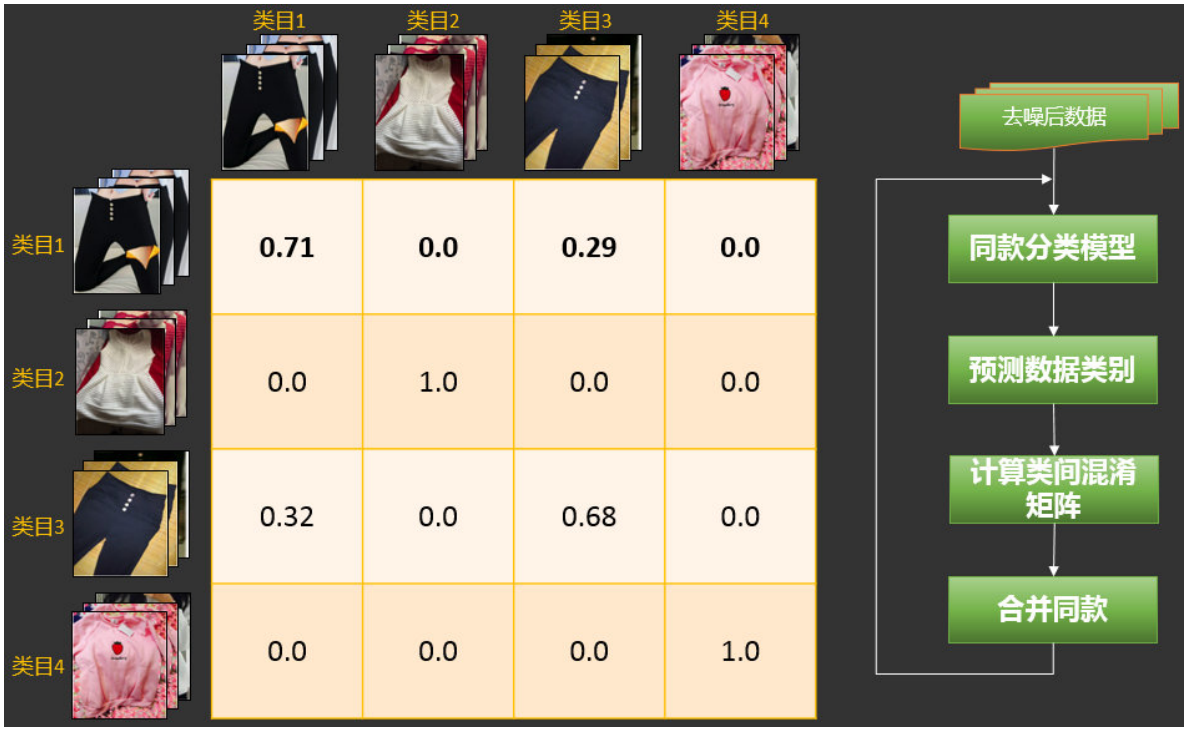


层次法 DBSCAN 步骤 2 为了将噪声样本中的困难正样本捞回，提升训练样本的丰富性，我们需要重访噪声样本，计算噪声样本和最大类簇的距离，并将满足阈值条件的近距离噪声样本重新分配给最大类簇，如下图 16 所示。



2.2.3.2 同款合并

同款合并的目的是为了将不同 SKU 但是同一款式的商品进行合并，从而得到实际有效的款式类目。我们采用分类混淆矩阵来完整自动同款合并。基于合并前的数据我们可以训练初始分类模型，并计算任意两个 SKU 之间的混淆概率。混淆概率定义为 p_ij=c_ij/n_i，其中 p_ij 为第 i 类预测为第 j 类的混淆概率，c_ij 为模型将第 i 类预测为第 j 类的样本个数，n_i 为第 i 类样本的实际个数。某个 SKU 的混淆概率矩阵如下所示，可知，当两个 SKU 实际为同一个款式时，如图 17 左中类目 1 和 3，那么他们之间很难区分，混淆的概率就会偏大。通过设定合并同款的分数阈值，我们能够将同款 SKU 进行合并。实际操作过程中，我们采用下图右边的迭代步骤进行，即先采用高的阈值合并置信度最高的同款，然后更新优化分类模型，并降低阈值合并更多的同款 SKU，上述步骤迭代进行，直到没有同款 SKU 需要合并。



合并完同款之后，我们得到的训练检索数据库的规模如下图所示，总共为 7w+多类目，1kw+训练样本，相比目前主流的开源数据库如 ImageNet（1000 类，130w+）和 OpenImage(6000 类，900w+)，在类别和数目上都要更加丰富。

2.2.3.2 成本收益

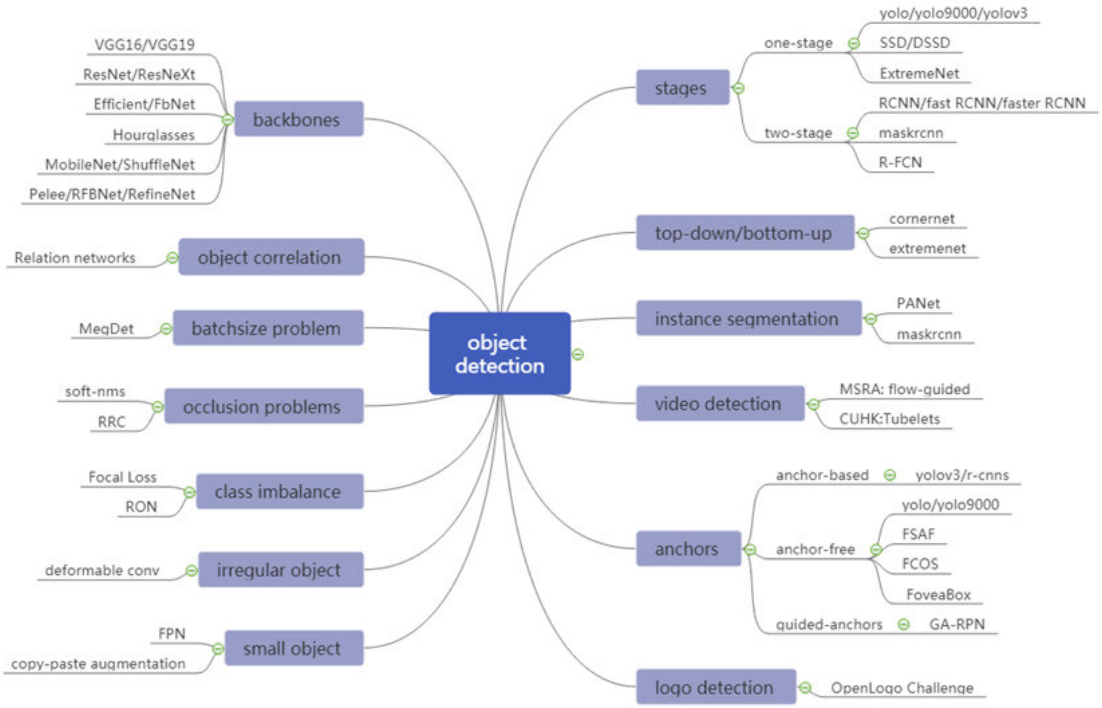
这里我们对采用算法进行同款去噪+同款合并，和采用纯人工清理进行对比在时间和金钱上都有巨大的收益提升，加快了整个项目的迭代速度。

2.3 扫一扫识物算法研发

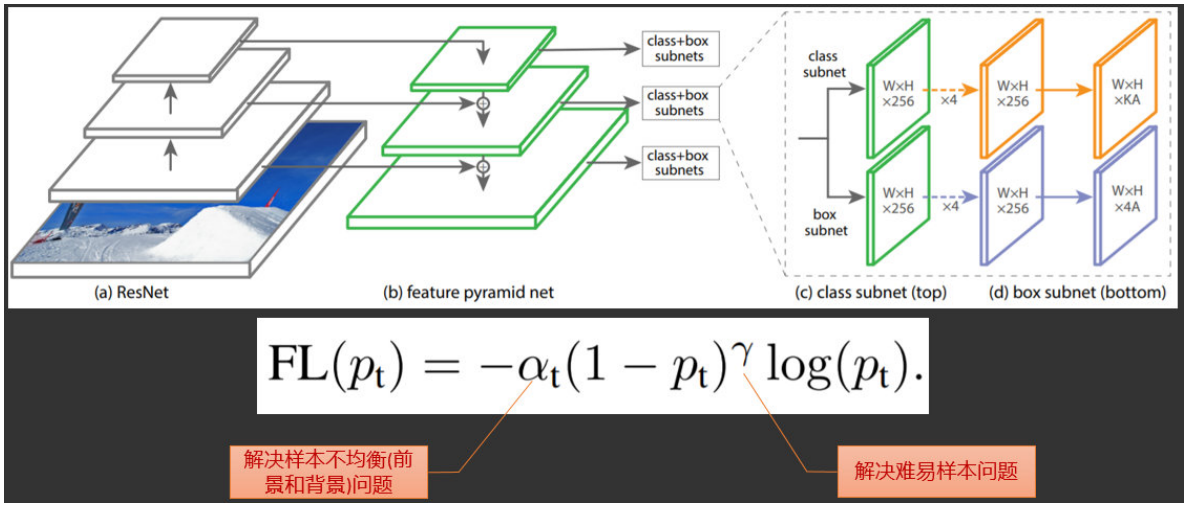
兵马未动，粮草先行，上一章节我们已经讲述了如何备好粮草（清洗高质量的训练数据），那么这一章节自然而然就是亮兵器了（高效利用现有训练数据的算法模型）。按照扫一扫识物整体框架，我们重点介绍视觉同款搜索涉及到的三大模块，即为物体检测、类目预测和同款检索。

2.3.1 物体检测

物体检测是扫一扫识物的第一个环节，我们需要有效的定位用户拍摄图片中的商品位置，剔除掉背景对后续同款检索的干扰。学术界和工业界对物体检测展开了大量的研究，如下图 18 所示，研究者从不同角度对检测算法进行优化，如从速度考虑分为 one-stage 和 two-stage 检测；从 anchor 出发分为 anchor-based、anchor-free、guided anchors，近期 anchor 又开始崛起，在性能上匹配 two-stage，速度上匹配 one-stage；还有从类别不平衡出发，不同尺度物体检测等等出发。



考虑商品检测中，主要重视三个问题：1.速度问题；2.检测标注类别不平衡；3.物体尺度变化差异大，综合这三个问题，我们选择图 19 中的 retinanet [8]作为我们的检测器。众所周知，retinanet 属于 one-stage 检测器，从原图出发直接回归物体的位置和类别，因而速度快，其次它采用金字塔架构，有效的适配多尺度物体检测，最后，retinanet 提出了 focal loss 可以有效的适应类别不平衡问题，以及样本难易问题。后续我们会采用 anchor-free 的策略进一步优化 retinanet 的 head 部分，进一步加速模型的检测速度，这里不展开介绍。



我们将 retinanet-resnet50fpn 和经典的单阶段检测器 yolov3，和两阶段检测器 faster-rcnn-resnet50-fpn 进行对比，如下表格 20 所示。评测数据采用的是外包采集的 7k 张图片，涵盖了 11 大类，对比表格结果可知，retinanet 在速度和精度上达到了很好的折中效果。后续我们会通过 tensorRT 进一步优化 retinanet 的前向速度，剧透下最快可以达到 80FPS。

方法	骨干网络	图片大小	检测精度评测(详见附件1)					速度
			AP	AP_50	AP_75	AP_Small	AP_Large	
YOLOV3 +BN	darknet53	416	0.54	--	--	--	--	20FPS
Faster-RCNN +BN	resnet50FPN	480	0.5699	0.7255	0.6453	-1	0.0992	10FPS
Faster-RCNN +GN	resnet50FPN	480	0.5622	0.7336	0.6374	-1	0.1225	10FPS
Faster-RCNN +Deformable	resnet50FPN	480	0.5799	0.7458	0.6596	-1	0.1216	9FPS
RetinaNet +BN	resnet50FPN	480	0.6119	0.7599	0.6705	-1	0.149	12FPS

2.3.1 类目预测

类目预测是为了确定检测抠图中物体的类别，从而方便后续用指定类目的模型进行特征提取和同款索引。大家可能会有所疑虑，因为前面检测器已经识别出了商品的位置和类目，为何不直接用检测器的类目，而是要重新来过一次类目预测呢？原因如下图 21 所示：训练检测器的数据有限，而用户上传的图片可能千奇百怪，那么训练库未出现的子类很容易造成检测器分类错误，其次是类间混淆性也会带来分类错误。



那么该如何提升类目识别的精度呢？这里我们利用海量的线上检索库来提升类目预测的精度。即为我们将用户 query 在检索库中进行一次检索，查询最近邻的 top-20 所属的类目，结合检测器的预测类目，来重新加权投票得到物体的最终类目。最终，通过检测+检索，我们能极大提升类目预测的精度，将近 6 个点的提升。

2.3.2 同款检索

同款检索是扫一扫识物的灵魂。不同于一般的以图搜图，只需要找出相似的图片即可，同款检索属于细粒度检索，需要检索出 query 的精确同款，比如华为 watch GT2 需要搜出来的就是该款手表，不能是其他牌子的手表，这就导致同款检索的挑战非常大，下图 22 列出了同款检索的难点与挑战：1.类间混淆性问题，即如何区分相似款和同款；2.同款召回问题，即同款本身存在较大差异，如何有效的检索出同款。考虑到这些难点，我们提出了 9 大方向来优化我们的同款检索模型。下面一一解释。



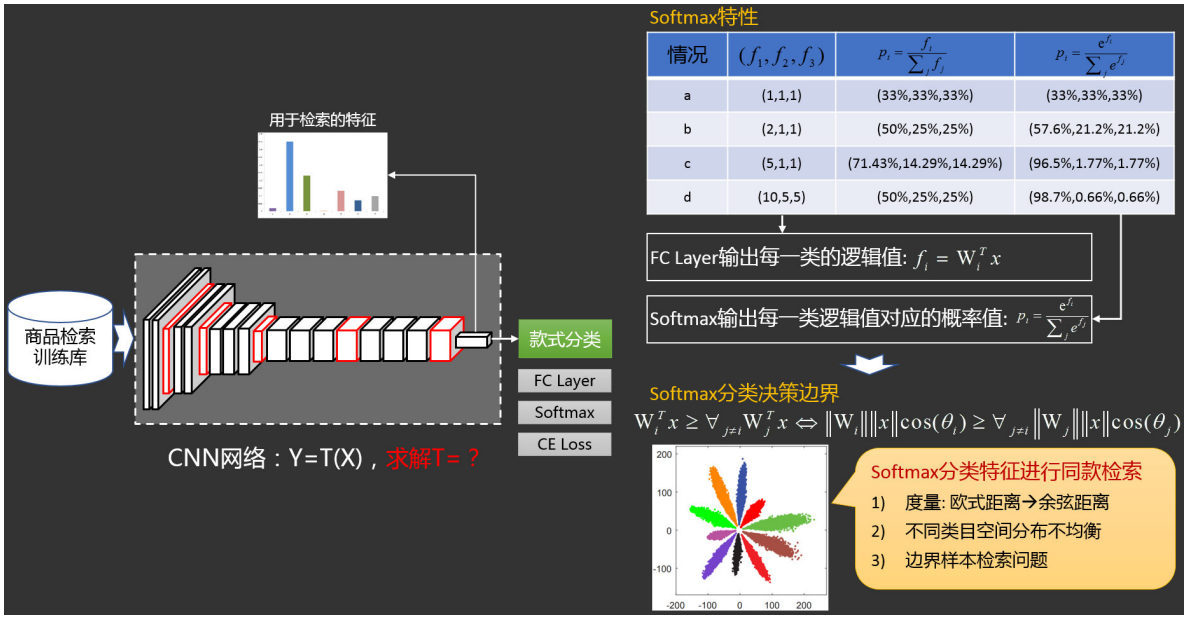
2.3.2.1 同款检索之分类模型

这是我们的 baseline 模型。我们使用的分类模型如下图 23 左侧所示。众所周知，分类模型采用 softmax 将逻辑值映射为类目的概率值，下图右上角表格所示，softmax 能很好的放大逻辑值的差异，将正确类目的概率逼近 1，有利于模型快速收敛。下图中我们还展示了 softmax 分类模型的决策边界，以及在 mnist-10 类目上学习得到的特征分布[9，10]。观察可知，softmax 分类器学习到的特征空间有 3 大特征：

1）特征空间呈现扇形分布，因而用余弦距离检索会优于欧式距离，后面我们会固定采用余弦距离进行同款检索；

2）不同类目分布不均衡。事实上我们希望同等重视各个类目，他们能均匀的分割整个特征空间，有利于模型泛化能力；

3）边界样本检索不准确，从特征图可知，边界样本距离临近类的余弦距离很可能小于该样本到同类之间的距离，造成检索错误。下面，我们重点修整分类模型的所存在的问题。

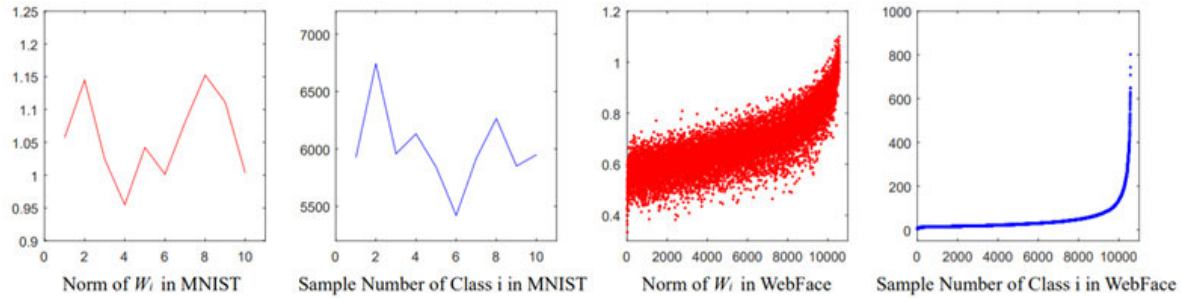


2.3.2.2 同款检索之分类模型改进 1 归一化操作

归一化包括两种，对图 23 中的分类权重 W 进行归一化，以及对特征 x 进行归一化。那么归一化的作用是什么呢？先来说说权重 W 的模长和特征空间分布不均衡的关系。有研究者[10]表明，训练数据库中样本多的类目对应的 W 的模长也会越长，表征模型越重视该类的分类正确程度，而忽视了其他样本数目少的类目，如图 24 所示，MNIST 和 WebFace 两个数据库都验证了上述的映射关系。而实际上我们希望的是每一类都能被平等的重视，特征空间中每一类能够均衡的划分整个空间，因而我们需要对 W 进行归一化，让所有类别的权重一致，即

$$W_k = \frac{W_k}{\|W_k\|_2}$$

：



特征归一化的操作类似，即为：

$$x = \frac{x}{\|x\|_2}$$

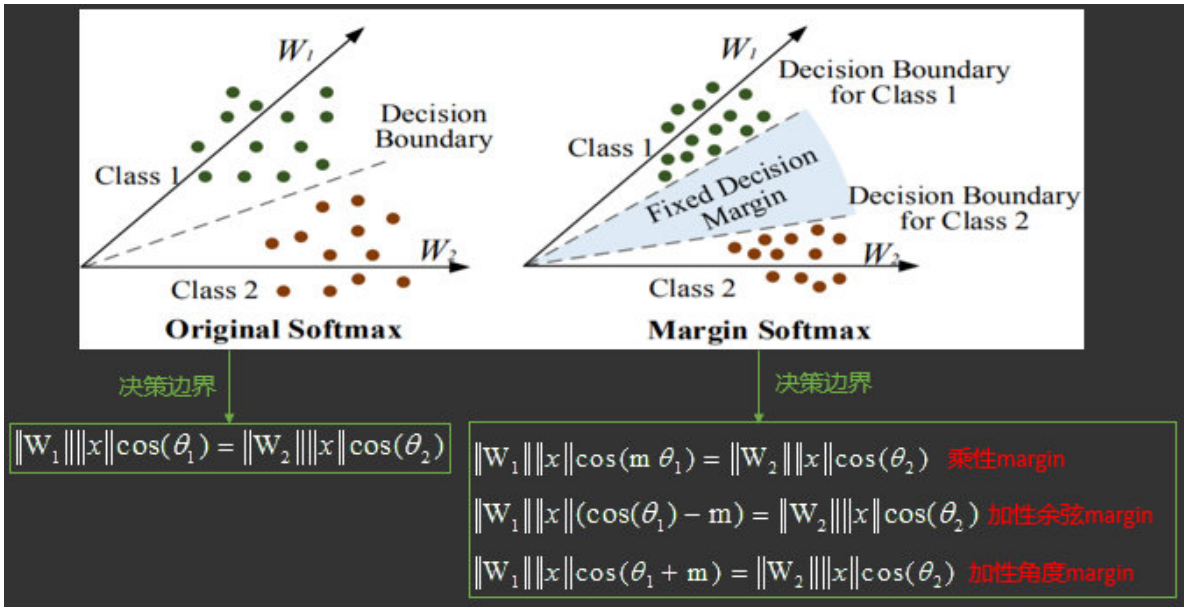
回顾 softmax 分类的决策边界：

$$W_i^T x \geq \forall_{j \neq i} W_j^T x \Leftrightarrow \|W_i\|\|x\|\cos(\theta_i) \geq \forall_{j \neq i} \|W_j\|\|x\|\cos(\theta_j) \Leftrightarrow \cos(\theta_i) \geq \cos(\theta_j)$$

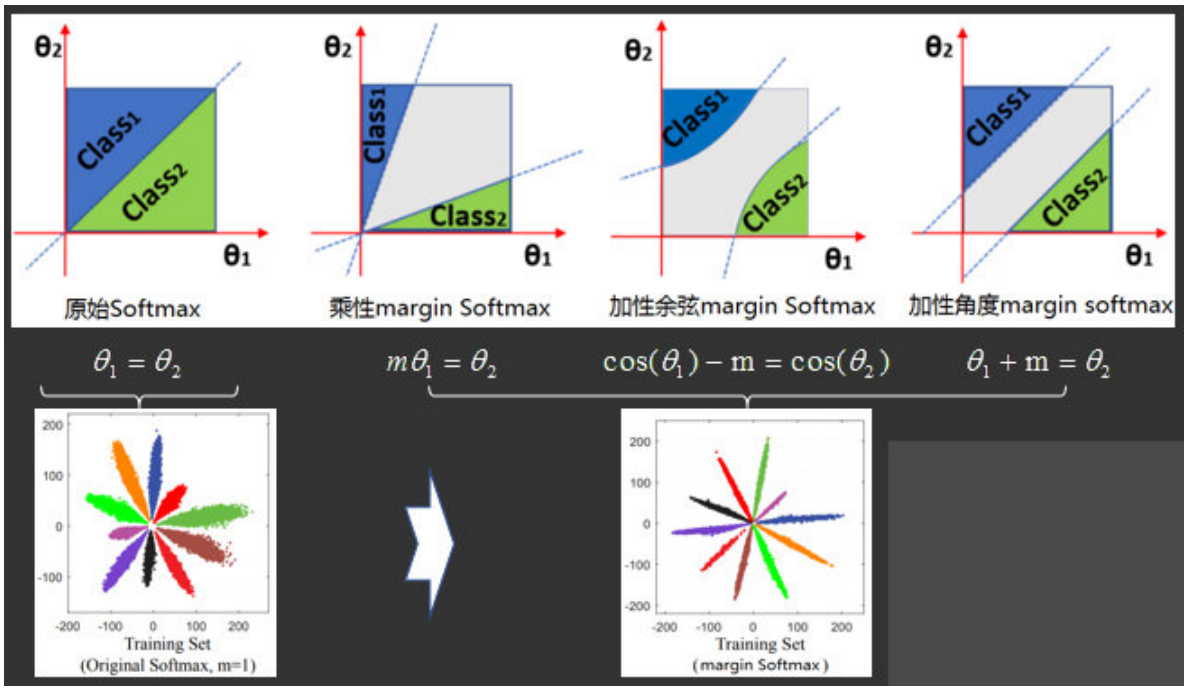
我们将 W 和 x 都进行归一化，因而决策边界只取决于角度，迫使模型训练收敛后特征分布更加扇形化，有利于余弦检索。但是两者同时归一化，会造成模型难于收敛，大家可以思考一番为何？参考图 23 中的 softmax 特性，由于权重和特征都进行了归一化，分类逻辑值最大为 1，最小为-1，同样的三类分类学习中 gt 类目对应的 softmax 概率最大只到 0.78，远小于 1，导致模型仍有较大 loss，不好收敛。解决方法比价简单，对逻辑值乘以一个尺度值 s 即可，扩大差异化，有利于模型收敛。

2.3.2.3 同款检索之分类模型改进 2 角度 Margin

增加角度 Margin 的核心目的是让 softmax 分类的扇形分布更加有利于检索：即为同类更加聚集，不同类更加远离。常见的 3 种增加角度 margin 的策略入下图 25 所示：乘性 margin[10，11]，加性余弦 margin[12]，加性角度 margin[13]。



增加 margin 后，softmax 分类模型得到的类内特征更加紧凑，如下图 26 所示。这里多说几句，相比乘性 margin，加性 margin 更加容易训练，这是因为乘性 margin 将余弦的单调区间从[0，π]缩小为[0，π/m]，梯度更新区间变小了，而加性 margin 单调区间不变，有利于模型收敛。



2.3.2.4 同款检索之分类模型改进 3 排序损失

分类模型的学习目的是类别可区分，和检索任务还是有一定区别，引入排序损失的目的就是显示的去优化检索任务，即为欧式空间中同类样本的距离要小于不同类。一般排序损失和分类损失叠加使用效果要更好，我们设计的模型结构如下图 27 所示：

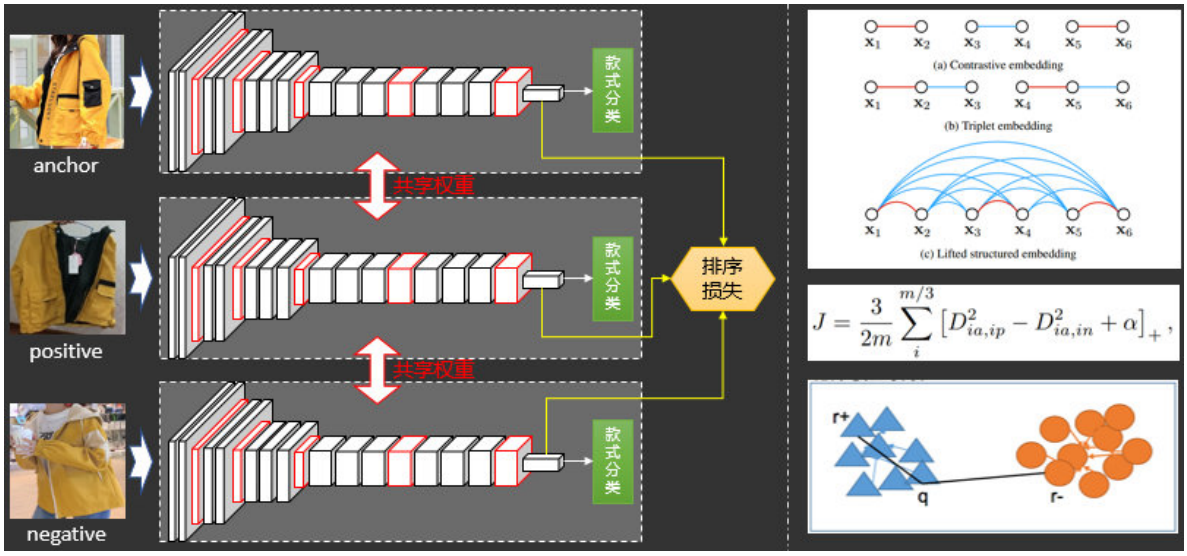


图 27 中右边展示的是常用的排序损失函数，如 contrastive loss， triplet loss， lifted structure loss 等。图中重点展示了三元组损失函数以及优化的可视化目标，即同类距离要比不同类距离小于一个 margin。

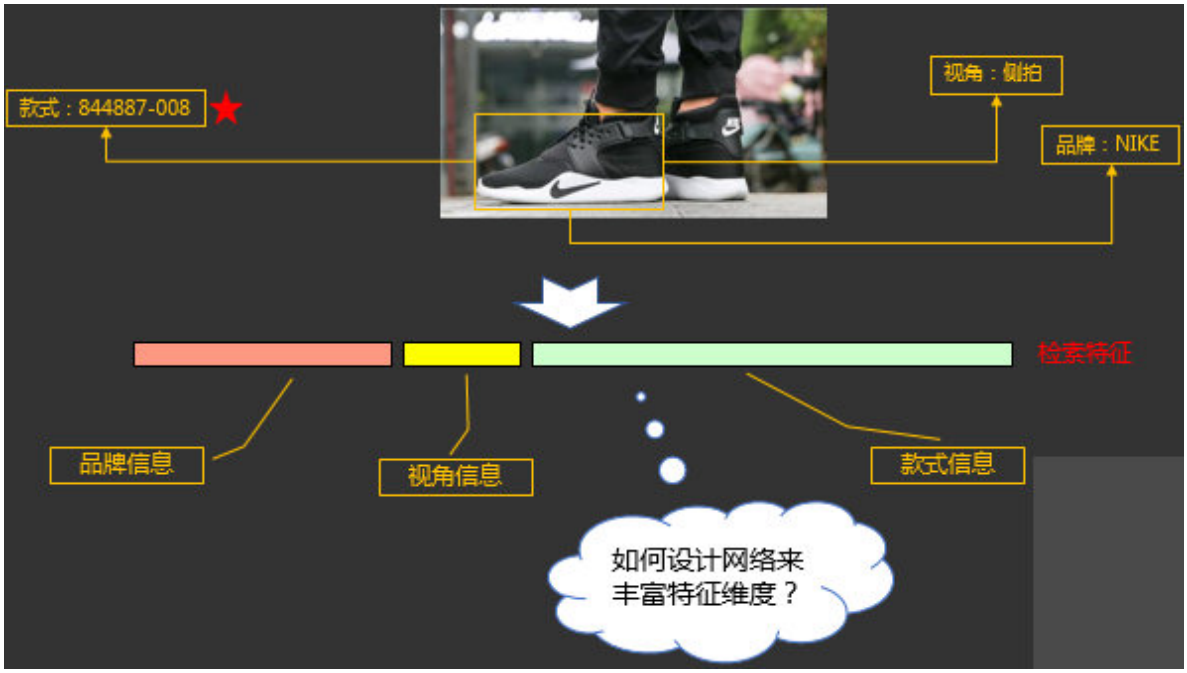
2.3.2.5 同款检索之分类模型及其改进后性能对比

此处我们对分类模型和其改进版本在商品检索任务上进行性能对比。评测集合是我们收集的 11 大类商品库，其中用户评论图作为查询样本，检索样本为同款商家图和非该款式的噪声集合组成，可以较好的模拟线上检索库。图 28 展示了分类模型及其改进版本在珠宝类目上的性能对比，可知：1）增加归一化和角度加性 margin 后，即为 ArcFace[13]，检索性能要优于常用 softmax 分类模型；2）在分类模型基础上增加排序损失，如 Hard Triplet Loss，检索性能优于常用 softmax 分类模型；3）分类+归一化+角度 margin+排序，如最后两行所示，性能进一步提升，但是提升幅度不是特别大。

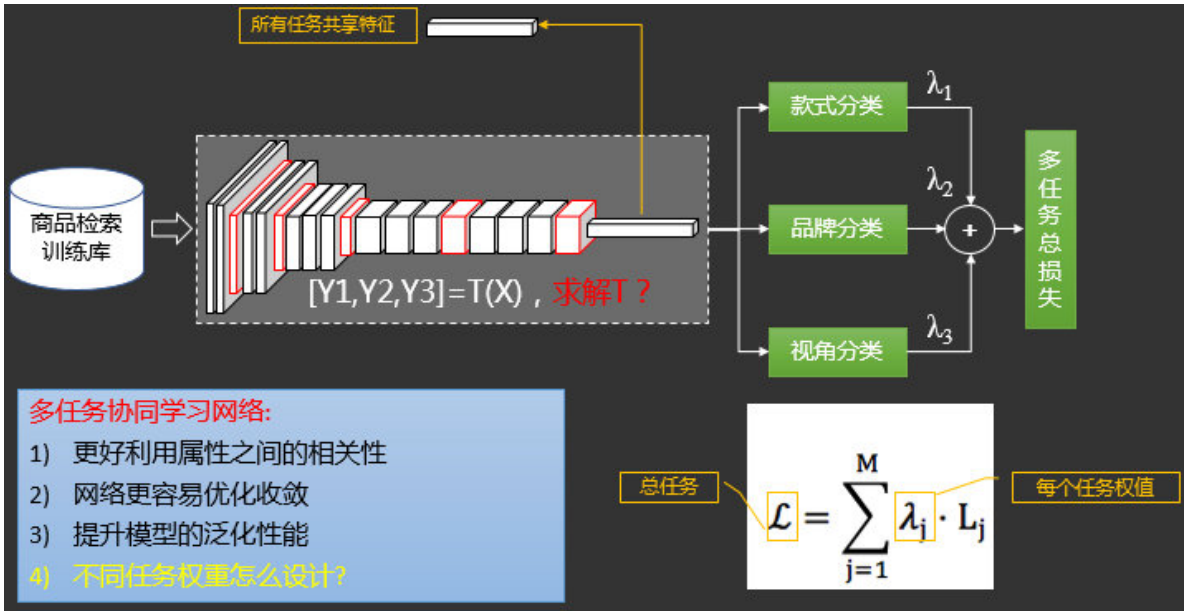
损失函数	评测类目	骨干网络	图片大小	特征维度	检索精度评测(详见附件2)	
					Top-1	ACC
Softmax Loss	珠宝首饰	ResNet152	299x299	2048	56.54%	58.71%
ArcFace	珠宝首饰	ResNet152	299x299	2048	56.75%	58.99%
Softmax Loss + Hard Triplet Loss	珠宝首饰	ResNet152	299x299	2048	57.51%	59.84%
ArcFace + Hard Triple Loss	珠宝首饰	ResNet152	299x299	2048	57.88%	60.40%
ArcFace + Lifted Struture Loss	珠宝首饰	ResNet152	299x299	2048	58.66%	60.41%

2.3.2.6 同款检索之多任务模型

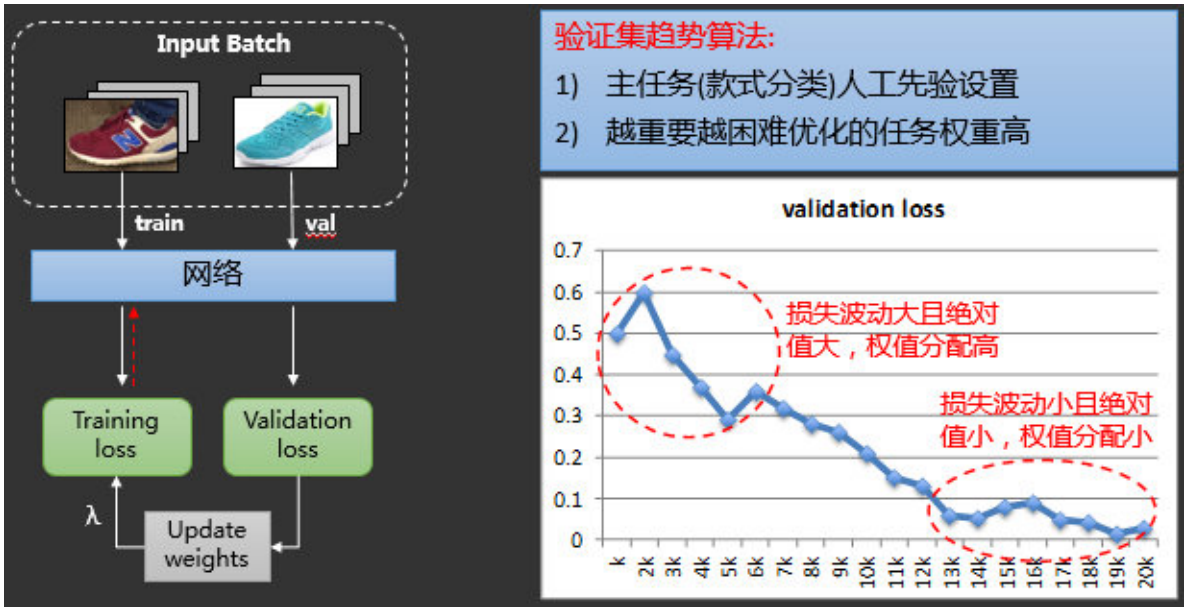
为了进一步提升检索模型的特征表达能力，我们探索让检索特征捕捉更加丰富的商品特性，如在款式的类别上，加上视角、品牌等商品属性，如下图 29 所示。



为了适应多属性标注标注的学习，我们设计如下图 30 的多任务协同学习模型。多任务协同学习模型的好处非常明显，可以更好的利用属性之间的相关性，有利于网络的优化收敛，以及提升模型的泛化能力，这样得到的检索特征更加有利于商品同款检索。这里有个问题，不同任务权重如何设计？当然我们可以手工设置每个任务的权重，但这需要对各个任务理解较为深入加上大



量调参得到，另外的策略是自动得到各个任务的权重，有许多研究者对此进行了研究，这里我们采用验证集趋势算法[14]来自动计算得到各个任务的权重，如下图 31 所示。该算法思想比较直接，即为人工设置高权重用于主任务，如款式分类，其他任务基于其优化难易程度来得到权重，如难优化（损失波动大且绝对值高）的任务权重大，易优化的权重小。使用多任务协同学习后，模

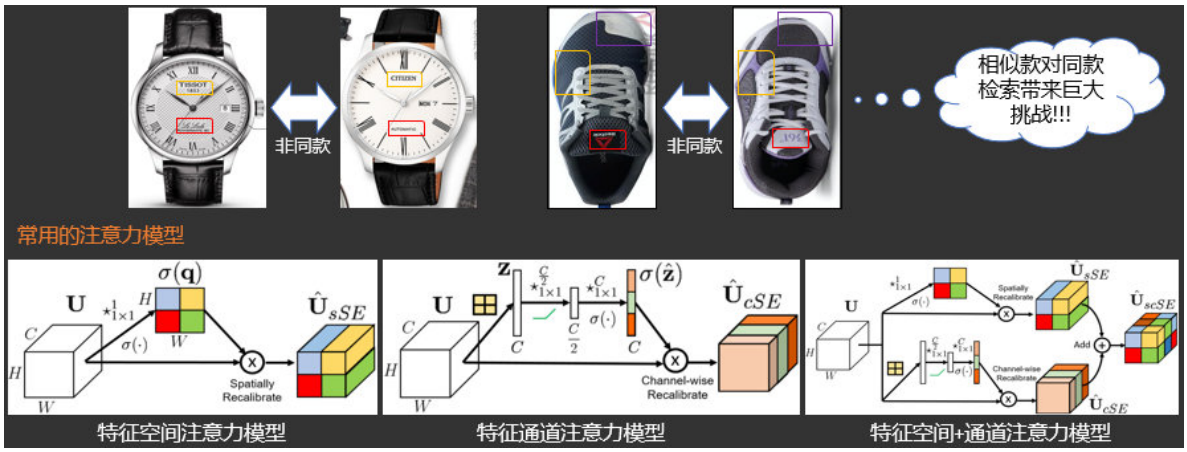


型的检索性能相比单任务模型有较大的提升，如下图 32 所示。

网络架构 (Softmax Loss)	评测类目	骨干网络	图片大小	特征维度	检索精度评测(详见附件2)				
					Top-1	Top-3	Top-5	Top-10	ACC
单任务	鞋子	InceptionV4	299x299	1536	81.80%	87.65%	89.81%	92.31%	80.57%
固定权重多任务	鞋子	InceptionV4	299x299	1536	83.38%	88.67%	90.89%	93.01%	82.15%
自适应权重多任务	鞋子	InceptionV4	299x299	1536	84.41%	89.34%	91.11%	93.54%	83.28%

2.3.2.7 同款检索之注意力模型

在图 22 中我们讲述了同款检索的一个巨大的挑战即为相似款对同款检索的混淆。为了更好的区分同款和相似款，我们希望更加关注一些显著的区域，如下图 33 中手表的 logo，刻写的文字，鞋子的 logo，花纹等。常用的注意力模型分为三种[15]，有特

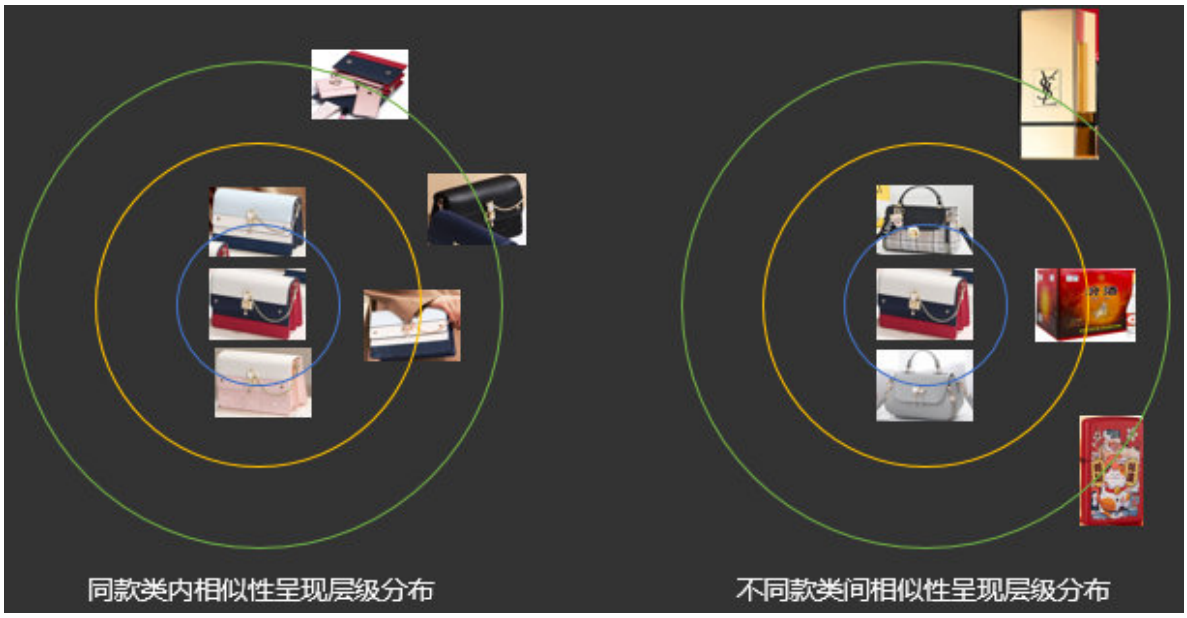


特征空间注意力，特征通道注意力，以及 2 种注意力相结合。通过实验对比可知，增加特征空间注意力后，模型检索性能提升，但是增加特征通道注意力，检索性能反而下降，如下图 34 所示。我们认为空间注意力有利于强化模型重视显著空间区域，而特征通道注意力可能造成了模型过拟合，性能反而有所下降。

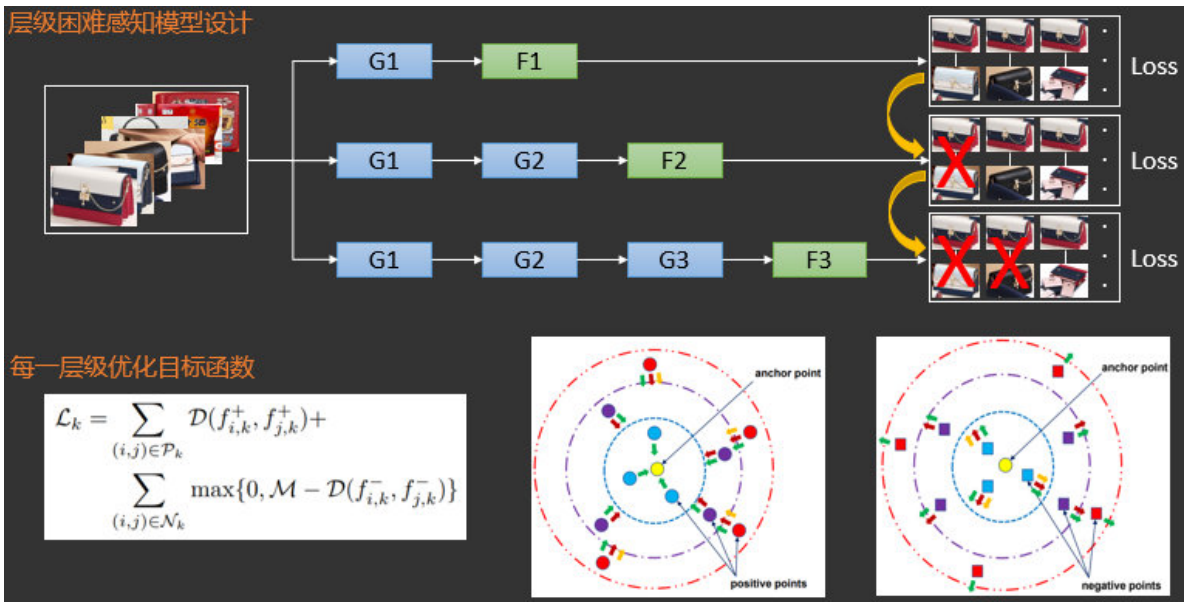
网络架构(Softmax Loss)	评测类目	骨干网络	图片大小	特征维度	检索精度评测(详见附件2)				
					Top-1	Top-3	Top-5	Top-10	ACC
No Attention	鞋子	InceptionV4	299x299	1536	84.41%	89.34%	91.11%	93.54%	83.28%
Spatial Attention	鞋子	InceptionV4	299x299	1536	85.19%	90.32%	92.10%	94.47%	84.30%
Channel Attention	鞋子	InceptionV4	299x299	1536	83.49%	88.21%	89.99%	92.14%	82.20%
Spatial+Channel Attention	鞋子	InceptionV4	299x299	1536	84.20%	89.30%	90.89%	92.90%	82.95%

2.3.2.8 同款检索之层级困难感知模型

同样针对同款检索的两大难点和挑战，即类间混淆性和类内差异性，我们采用层级困难感知模型来充分重视这两大难题，如下图 35 所示的包包。尽管是同款，其相似程度可能有很大差异，如该款式包包由于拍摄视角、遮挡等造成的变化，对于非同款的负样本，也有非常相似的其他款式包包，和差异比较大的其他商品入酒类、口红等。



层级困难感知模型模型[16]结构如下图 36 所示，其排序损失按层级分布，第一层级针对所有的正负样本进行排序学习，第二层负责较为困难的正负样本对，而第三层则负责更加困难的正负样本对，并且对于越困难的样本对，模型网络设计的越深。

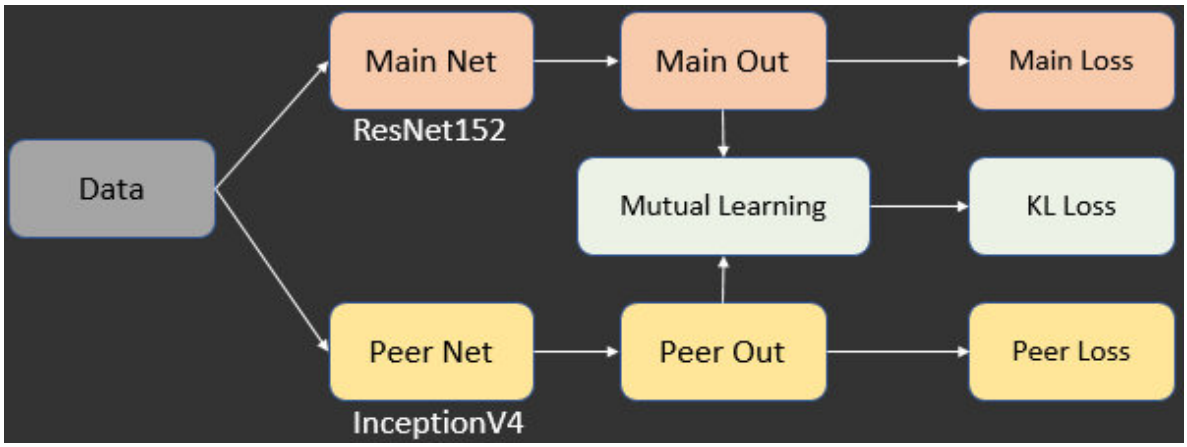


层级困难感知模型通过层级设计，匹配正负样本对的层级分布特性，能够有效的提升模型的同款检索性能，如下图 37 的实验所示。

网络架构	评测类目	骨干网络	图片大小	特征维度	检索精度评测(详见附件2)				
					Top-1	Top-3	Top-5	Top-10	ACC
Softmax Loss	鞋子	ResNet152	256x256	2048	84.92	89.49	91.2	92.98	84.53
Softmax Loss + Lifted Structure	鞋子	ResNet152	256x256	2048	85.69	90.46	92.12	94.49	84.6
Hard-Aware Network	鞋子	ResNet152	256x256	2048: F1	86.41	91	92.55	94.55	85.64
				4096: F1+F2	86.75	91.02	92.4	94.37	85.64

2.3.2.9 同款检索之互学习模型

众所周知，一般竞赛中大家都会融合多个模型的结果来提升精度。但是在实际的项目落地，我们需要同时考虑部署模型的精度和速度，融合多个模型会占用更多计算资源，降低前向速度。那么怎么既然融合多个模型结构的优势，又不增加计算资源呢？这就是互学习模型的核心思想，如下图所示，互学习模型通过 KL 散度 loss 来吸收其他模型的结构优势，部署的时候只需要部署一个模型即可，不增加计算资源。



实验中，我们利用 resnet152 和 inceptionv4 进行互学习，实际部署采用 resnet152 进行检索，其性能如下图 39 所示。互学习除了增加模型训练时间，对模型上线不增加任何负担，但是精度能够较为显著的增加。

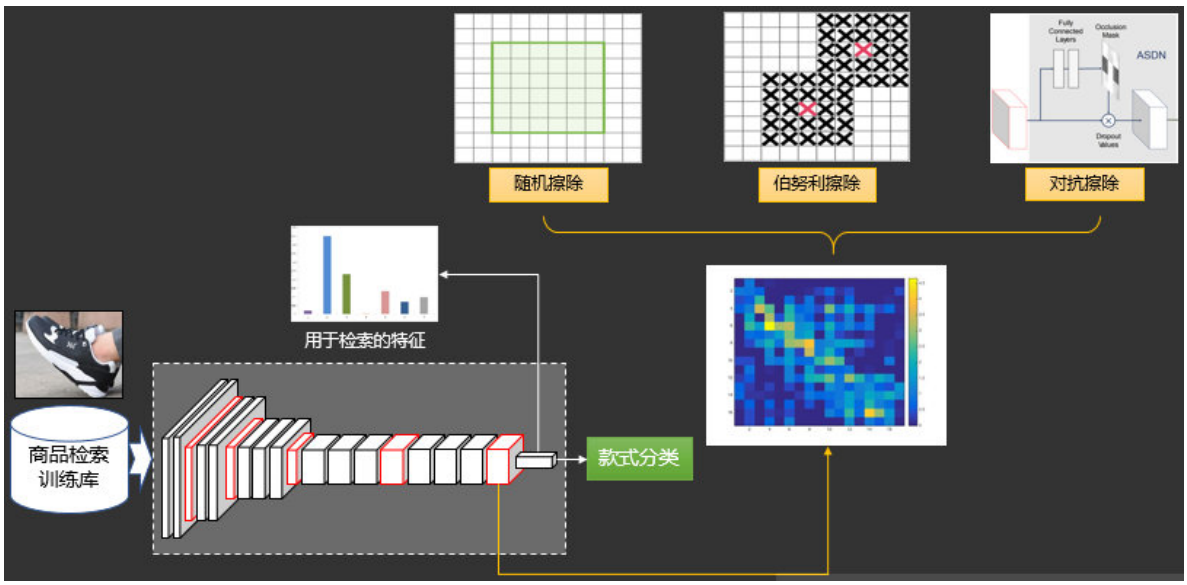
网络架构	评测类目	骨干网络	图片大小	特征维度	检索精度评测(详见附件2)				
					Top-1	Top-3	Top-5	Top-10	ACC
ArcFace+Hard Triplet Loss	珠宝	ResNet152	299x299	2048	57.88%	-	-	-	60.40%
ArcFace+Hard Triplet Loss+KL	珠宝	ResNet152-InceptionV4	299x299	2048	59.45%	-	-	-	61.25%

2.3.2.10 同款检索之局部显著性擦除

通过上述所有策略来提升模型的同款检索性能后，我们发现仍然存在一个问题，就是深度学习模型会过分关注图像的纹理区域，而忽视物体的形状。比如在行旅箱的同款检索中，返回的是印有相同图案的书包，钱包等，而非行旅箱。如何让模型在关注



纹理的同时，也关注下物体的形状信息呢？我们采用局部显著性擦除技术来破坏原图的纹理，迫使模型来关注物体的形状。常见的局部显著性擦除有 3 种，如下图 41 所示，分别为随机擦除，伯努利擦除，对抗擦除。这里我们重点对比了前两者，对抗擦除后



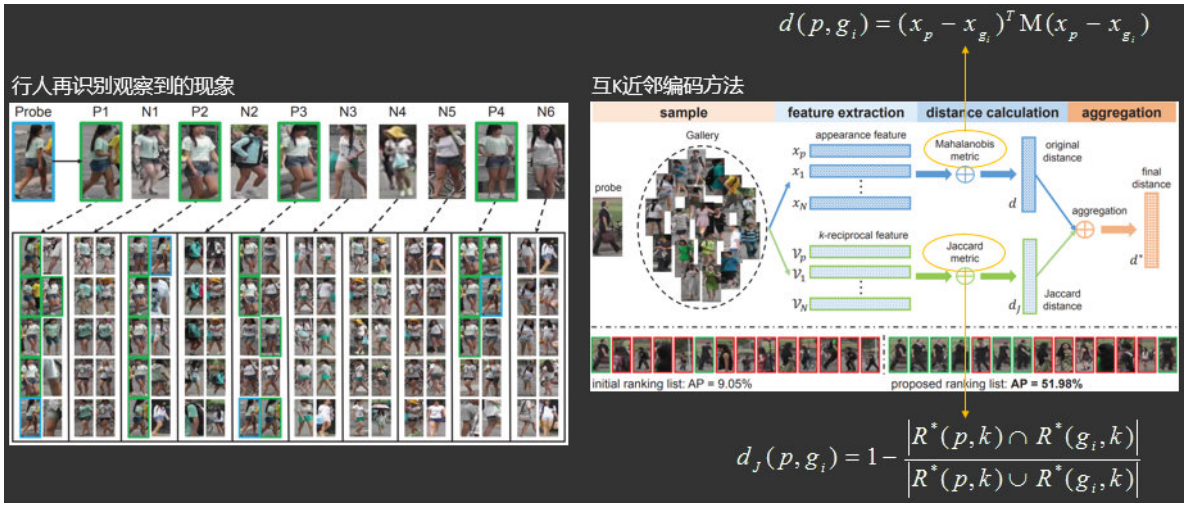
续有时间再补上其性能，实验结果如下图 42 所示，局部显著性擦除能够极大的提升模型的同款检索精度。

网络架构	评测类目	骨干网络	擦除策略	图片大小	特征维度	检索精度评测(详见附件2)				
						Top-1	Top-3	Top-5	Top-10	ACC
ArcFace+Hard Triplet Loss	珠宝	ResNet152	不擦除	299x299	2048	57.88%	-	-	-	60.40%
ArcFace+Hard Triplet Loss+KL	珠宝	ResNet152-InceptionV4	不擦除	299x299	2048	59.45%	-	-	-	61.25%
			随机擦除	299x299	2048	68.52%	78.63%	83.14%	88.19%	69.41%
			伯努利擦除	299x299	2048	67.38%	77.70%	82.8%	88.02%	68.25%
			对抗擦除	299x299	2048	--	--	--	--	--

2.3.2.11 同款检索之互 k 近邻编码重排序

前面我们讲述的都是优化检索模型，这里我们讲述的是如何进一步优化检索模型的检索结果，即重排序。在重排序里面，我个人非常欣赏互 k 近邻算法[17]，非常简单高效。互 k 学习算法最早被提出来用于行人再检索，如下图 43 所示，其核心发现是，对于 query 样本，其检索的 top-10 样本中，正样本（和 query 为同一人）换做查询样本时，其 k 近邻中有原始的 query 样本，而负样本（和 query 非同

一个人），其 k 近邻中没有原始 query 样本，基于该发现，作者在马氏距离的基础上，增加了基于互 k 近邻的距离度量，如图中下方所示，基于该度量可以有效的对原排序进行重排序，将正样本挪前，将负样本挪后。



但是实际上，我们无法直接利用该算法用于商品同款检索，原因在于我们的 query 是用户评论图，而检索图是商家图，他们存在很大的差异，造成互 k 近邻会失效，后续我们重点是如何优化特征度量空间，让模型的域差异减小，然后再利用互 k 近邻来进行重排序。

2.3.2.12 竞品对比

最后，我们基于用户上传的 7k 张图片进行 11 类检测准确度评测， 运行环境 NVIDIA GPU P4，来对比不同竞品的精度差异。对比试验可知，我们的算法要优于京东，接近拍立淘。

2.4 扫一扫识物平台建设

正所谓磨刀不误砍柴工，平台建设对于我们的数据构建，模型学习，模型部署都是至关重要。下面我们一一介绍。

2.4.1 数据清理平台

为了加快人工校验以及人工标注的速度，我们开发了一系列工具来辅助标注和检验模型精度，这里不做过多解释。

2.4.2 模型训练平台

这几年，机器学习平台发展迅猛，很多公司拥有自己的深度学习平台，我们人少钱少，主要是基于开源的深度学习平台来开发符合商品同款检索的特有平台。我们主要是开发了 caffe 和 pytorch 两套同款检索平台，后续重点介绍。

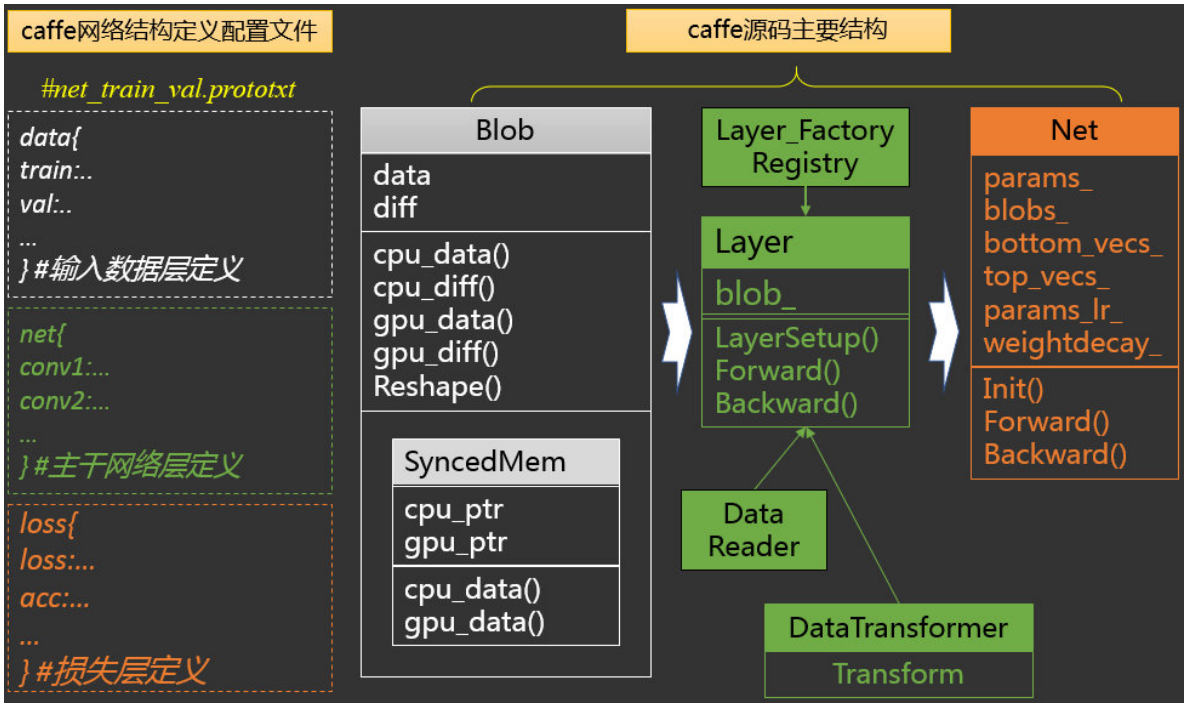
![图44. 机器学习平台发展史18

2.4.2.1 caffe

我们开发的第一个模型训练平台是 caffe。caffe 平台的核心架构如下图 45 所示，caffe 平台现在主要是工业界用的多，现在学术界用的少些。我们开发的 caffe 同款检索平台，具有如下特点：

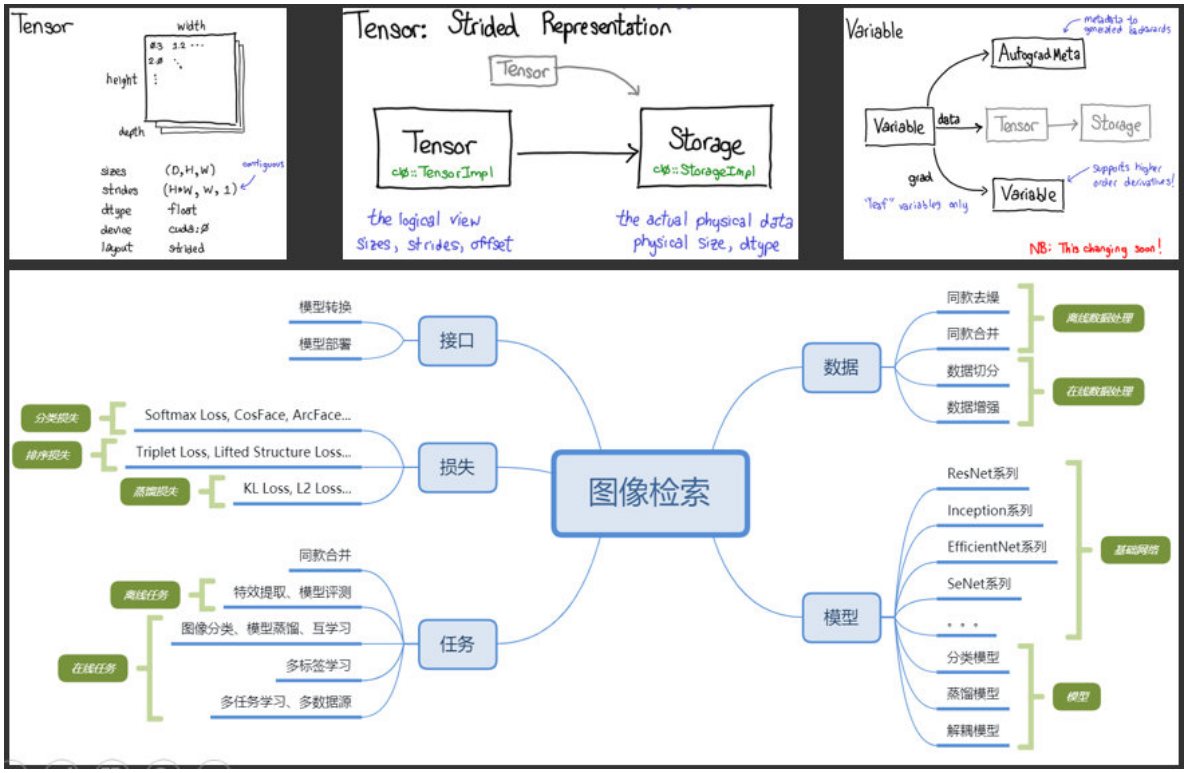
1)支持丰富的数据增广策略；2)支持多类型数据加载；3)支持蒸馏学习/互学习算法；4)支持困难感知模型算法；5)支持排序模型算法；6)支持 batchsize 扩充。

caffe 的优缺点非常明显，其优点有：1)训练快，结果稳定；2)基于 prototxt 快速试验各种多模型/多标签/多数据源任意组合。其缺点有：1)新算法开发慢；2)调试不灵活；3)显存优化不好；4)学术前沿方法更新少。第 4 个缺点是较为致命的，我们无法快速跟进学术前言，因而我们后续决定开发 pytorch 检索平台。



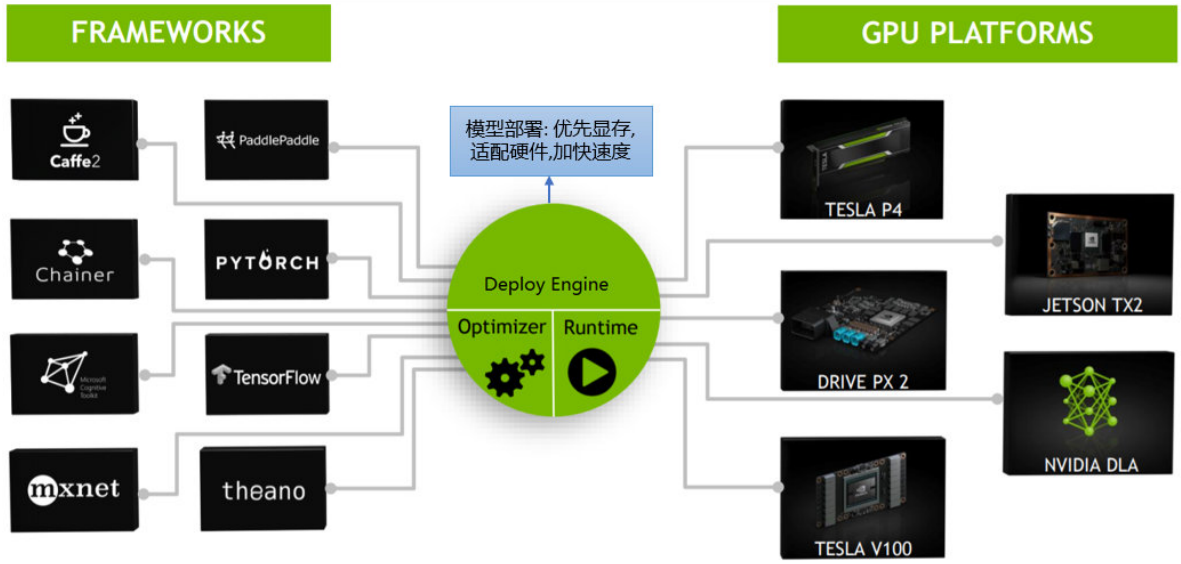
2.4.2.2 pytorch

我们开发的 pytorch 检索架构如下图 46 所示，基本支持 caffe 检索平台的所有特点：1)支持丰富的数据增广策略；2)支持多类型数据加载；3)支持蒸馏学习/互学习算法；4)支持排序模型算法；5)支持更多主流网络 EfficientNet；6)支持数据去噪/合并同款/检索；7)支持混合精度训练。pytorch 优缺点也非常明显，其优点：1)自动求导，算法开发效率高；2)动态图，Python 编程，简单易用；3)Tensorboard 可视化方便；4)Github 资源多，紧跟前沿；5)Pytorch1.3 支持移动端部署。当然 pytorch 也不是完美无缺的，相比 caffe 有其缺点：1)在多任务自由组合不如 caffeprototxt 方便。



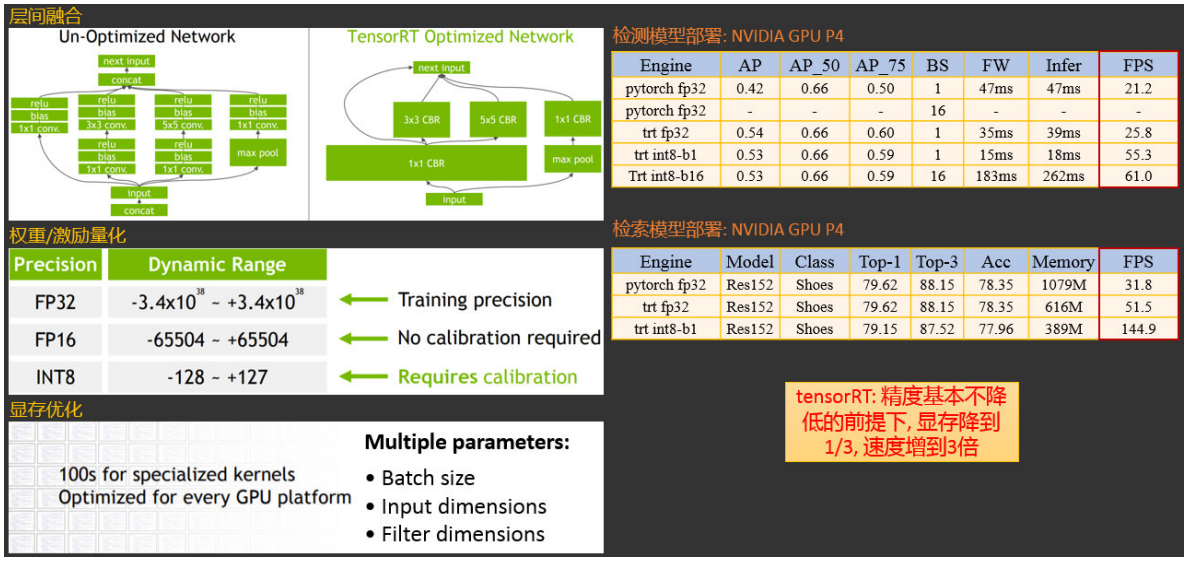
2.4.3 模型部署平台

模型训练我们可以不在乎模型运行时间代价，但是在模型部署时候我们得要充分重视模型的资源占用情况，尽量提升模型的并发能力，如 GPU 部署时候优化显存，适配硬件，加快速度。这里我们重点介绍后台 GPU 部署使用的 tensorRT 和手机端部署使用的 ncnn。



2.4.3.1 模型部署平台：tensorRT

tensorRT 是英伟达开发的部署平台，能够有效的降低模型显存，加速模型前向速度。这里我们不展开细节，大家可以关注下面的检测模型和检索模型，通过 tensorRT 量化加速后，显存和速度都有了巨大的飞跃。



检测模型部署: NVIDIA GPU P4

Engine	AP	AP_50	AP_75	BS	FW	Infer	FPS
pytorch fp32	0.42	0.66	0.50	1	47ms	47ms	21.2
pytorch fp32	-	-	-	16	-	-	-
trt fp32	0.54	0.66	0.60	1	35ms	39ms	25.8
trt int8-b1	0.53	0.66	0.59	1	15ms	18ms	55.3
Trt int8-b16	0.53	0.66	0.59	16	183ms	262ms	61.0

检索模型部署: NVIDIA GPU P4

Engine	Model	Class	Top-1	Top-3	Acc	Memory	FPS
pytorch fp32	Res152	Shoes	79.62	88.15	78.35	1079M	31.8
trt fp32	Res152	Shoes	79.62	88.15	78.35	616M	51.5
trt int8-b1	Res152	Shoes	79.15	87.52	77.96	389M	144.9

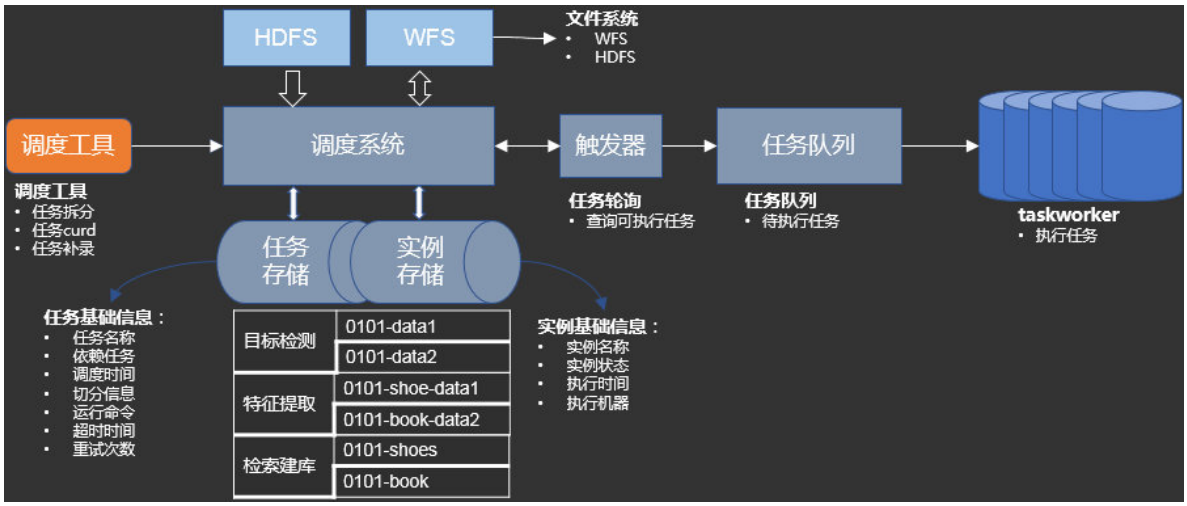
2.4.3.2 模型部署平台：ncnn

移动端部署，我们用腾讯自己开发的 ncnn 架构，其优点如下图 49 左图所示，demo 如右图所示。



2.4.4 任务调度系统平台

任务调动平台由我们的后台大神们开发，主要用于各个任务的有效调用，考虑到我们的检索库是上亿的数据库，需要保证平台具有较好的容错、容灾，以及鲁棒机制。如下图 50 所示，当然这里展示的只是冰山一角，后面等后台大神们在 KM 里给大家详细解释。



三. 扫一扫识物展望

最后，我们对我们的扫一扫识物进行未来展望，还是那句话，我们期待扫一扫识物成为大家的一个生活习惯：扫一扫，知你所看；扫一扫，新生活，新姿势。



参考文献

[1] 公司内部文档

[2] https://blog.csdn.net/Notzuonotdied/article/details/95727107

[3] Learning Deep Features for Discriminative Localization， CVPR16

[4] Weakly Supervised Object Localization with Latent Category Learning， ECCV14

[5] Weakly Supervised Deep Detection Networks， arXiv16

[6] Seed， Expand and Constrain: Three Principles for Weakly-Supervised Image Segmentation， arXiv16

[7] https://scikit-learn.org/stable/modules/clustering.html

[8] Focal Loss for Dense Object Detection， arXiv18

[9][https://zhuanlan.zhihu.com/p/76391405](https://zhuanlan.zhihu.com/p/76391405SphereFace:%20Deep%20Hypersphere%20Embedding%20for%20Face%20Recognition，%20arXiv18)

[10] SphereFace: Deep Hypersphere Embedding for Face Recognition， arXiv18

[11] Large-Margin Softmax Loss for Convolutional Neural Networks， arXiv17

[12] CosFace: Large Margin Cosine Loss for Deep Face Recognition， arXiv18

[13] ArcFace: Additive Angular Margin Loss for Deep Face Recognition， arXiv18

[14] Adaptively Weighted Multi-task Deep Network for Person A!ribute Classification， MM17

[15] Concurrent Spatial and Channel ‘Squeeze & Excitation’ in Fully Convolutional Networks， arXiv18

[16] Hard-Aware Deeply Cascaded Embedding， ICCV17

[17] Re-ranking Person Re-identification with k-reciprocal Encoding， CVPR17

[18] 公司内部文档

揭秘微信「扫一扫」识物为什么这么快?

作者：arlencai，腾讯 WXG 应用研究员

> 微信“扫一扫”识物已上线一段时间，在公司内外均受到极大的关注。相比于行内相关竞品的“拍”，“扫一扫”识物的特点在于“扫”，带来更为便捷的用户体验。“扫”离不开高效的移动端物体检测，本文将为你揭秘。

一、背景

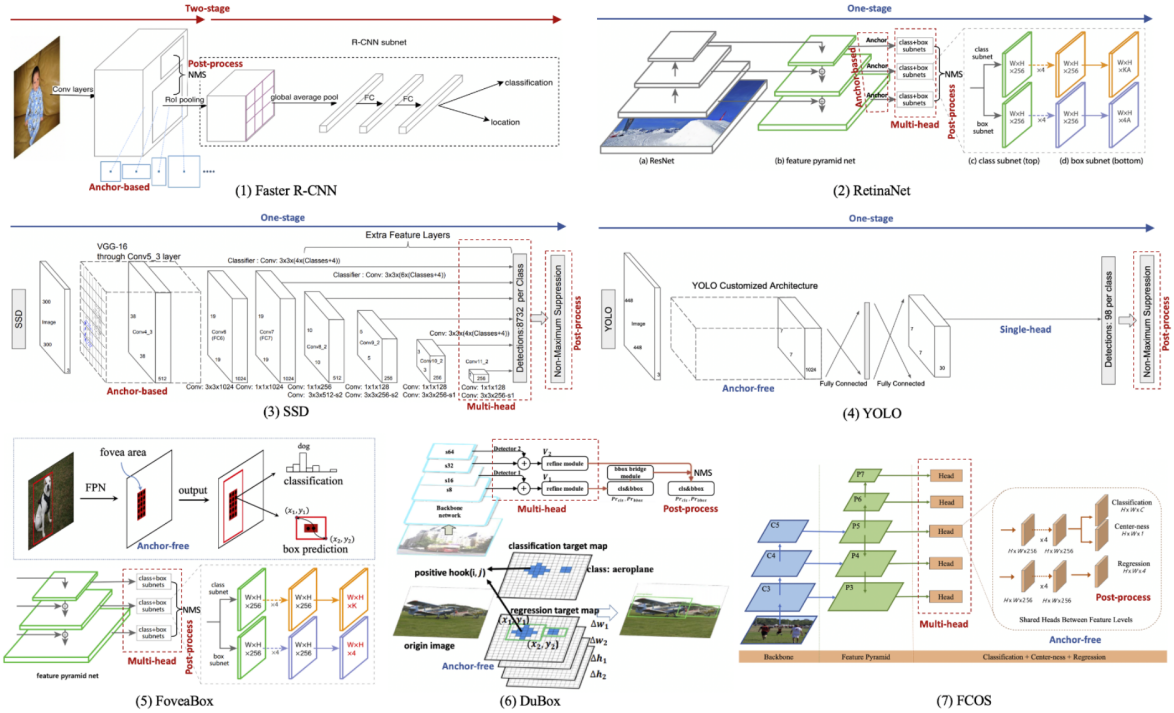
“扫”是“扫一扫”识物的亮点，带来更为便捷的用户体验。相比于“拍”的交互方式，“扫”的难点在于如何自动地选择包含物体的图像帧，这离不开高效的移动端物体检测。

二、问题

“扫一扫”识物是一种面向开放环境的通用物体检测——复杂多样的物体形态要求模型具有强的泛化性，移动端的计算瓶颈又要求模型保持高的实时性。“扫一扫”识物需要一个什么样的移动端检测（Class-wise or Object-ness）呢？Class-wise 检测（即传统意义上的物体检测）的优势在于同时输出物体的位置和类别，然而开放环境下的物体类别很难准确定义和完整覆盖。因此，我们将问题定义为**Object-ness 检测（即主体检测）：只关注是否为物体和物体的位置，并不关心物体的具体类别。**Object-ness 的物体检测对多样化的物体具有更强的普适性，同时大大减轻模型的负担来保证其实时性。这是“扫一扫”识物相比于相关竞品在移动端检测问题上定义的不同。

三、选型

近几年物体检测算法日新月异，面对琳琅满目的检测模型（见图 1），**合适的才是最好的。**



（1）One-stage

从模型的层次结构上，可分为两阶段（two-stage）和单阶段（one-stage）。

（a）Two-stage 检测器以 R-CNN 系列（Fast R-CNN [1]、Faster R-CNN [2]、Mask-RCNN[3]）为代表，其模型的第一阶段输出粗糙的物体候选框（proposal），第二阶段进一步回归物体坐标和分类物体类别。Two-stage 检测器的优势在于：RoIPool 的候选框尺度归一化对小物体具有较好的鲁棒性；进一步的区域（region）分类对于较多类别的检测需求更为友好。

（b）One-stage 检测器以 YOLO 和 SSD 系列（YOLO V1-V3 [4-6]、SSD [7]、RetinaNet[8]）为代表，其特点是全卷积网络（FCN）直接输出物体的坐标和类别，为移动端加速提供了便利。

对于"扫一扫"识物中主体检测的应用场景，小物体和多类别的需求不如实时性来得强烈，因此我们选择 one-stage 的模型结构。

（2）Anchor-free

（a）锚点（anchor）是 R-CNN 系列和 SSD 系列检测方法的特点：在 one-stage 检测器中，通过滑动窗口（slide
window）产生各式各样的 anchor 作为候选框；在 two-stage 检测器中，RPN 从 anchor 中挑选合适的候选框进行第二阶段的分类和回归。Anchor 为检测器提供物体的形状先验，可有效地降低检测任务的复杂度，但经验性的 anchor 参数会极大地影响模型的性能。

（b）无锚点（anchor-free）的检测器随着网络结构（如：FPN[9]、DeformConv [10]）和损失函数（如：Focal Loss [8]、IOU Loss[11]）的发展逐渐焕发出新的生机。其中，尺度鲁棒的网络结构增强模型的表达能力，训练鲁棒的损失函数解决样本的平衡和度量问题。Anchor-free 方法以 YOLOV1-V2 [4-5]及其衍生（DenseBox [12]、DuBox [13]、FoveaBox [14]、FCOS[15]、ConerNet [16]、CenterNet[17]等）为代表。他们抛开候选框的形状先验，直接分类物体的类别和回归物体的坐标。

在"扫一扫"识物的应用场景中，复杂多样的物体形状对 anchor 的设计提出了巨大挑战，因此我们选择 anchor-free 的模型结构。

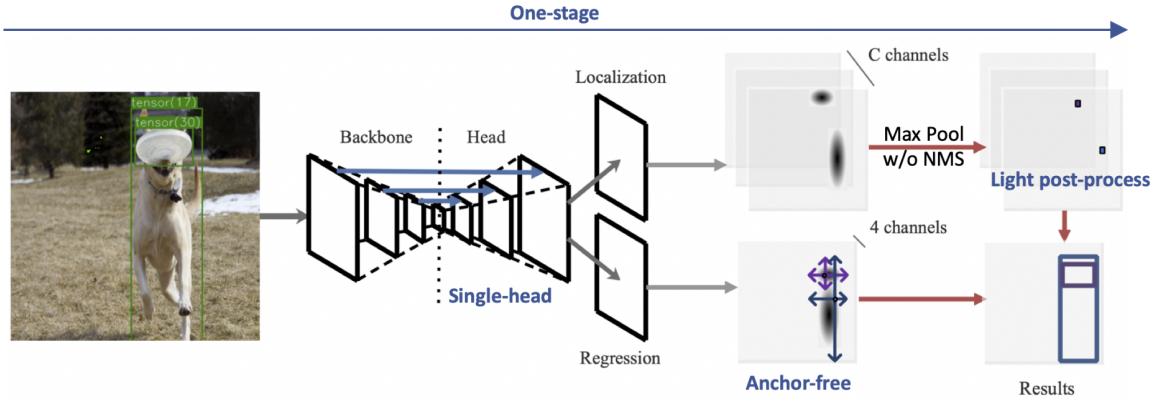
（3）Light-head

近一年来，anchor-free 的检测器日新月异。然而，在移动端的应用场景下，大部分 one-stage 且 anchor-free 的检测器仍存在以下不足：

（a）多输出（Multi-head）：为了增强模型对多尺度物体的检测能力，大部分检测器（如：FoveaBox[14]、DuBox [13]、FCOS[15]）普遍采用多头输出来提高模型的尺度鲁棒性。其中，低层特征满足小物体检测需求，高层特征应对大物体检测。然而，多头输出的网络结构对于移动端加速并不友好。

（b）后处理（Post-process）：为了解决 anchor 先验缺失和 multi-head 结果整合的问题，大部分检测器都需依赖复杂的后处理，如：非极大值抑制（NMS）和各式各样的奇技淫巧（trick），但它们普遍不适合并行化加速。

综上，我们选取**CenterNet**作为"扫一扫"识物的移动端检测模型（见图 2）。*CenterNet 是 one-stage 的 anchor-free 检测方法，single-head 的输出和高斯响应图的回归使其不依赖 NMS 的后处理。*CenterNet 将目标检测问题变成一个标准的关键点估计问题：通过全卷积网络得到中心点的热力图（峰值点即中心点），并预测峰值点对应的物体宽高信息。此外，我们引进了 TTFNet[18]中高斯采样、高斯加权和 GIOU Loss[19]等技术实现 CenterNet 的训练加速，仅需 5 小时即可在 4 块 Tesla P4 下完成 MS-COCO 的训练，这为模型调参和优化节省了大量的时间。

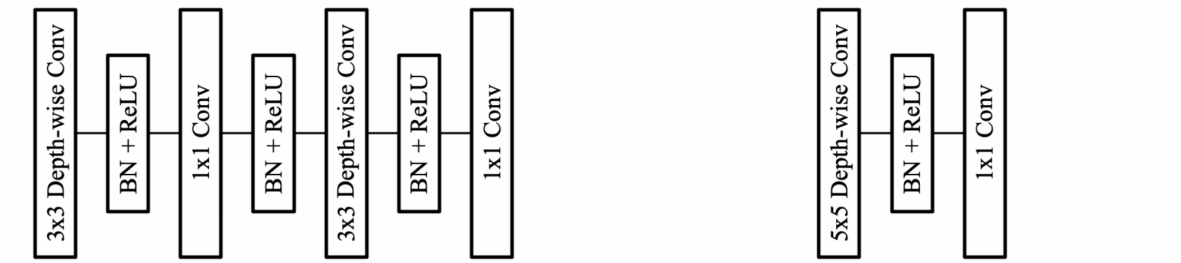


四、优化

针对移动端的检测需求，首先我们将 CenterNet 的骨干网络（backbone）从 ResNet18 更换为对移动设备更为友好的 ShuffleNetV2[20]。然而，仅仅依赖 backbone 带来的效能提升是有限的，对此我们进行针对性的模型优化。

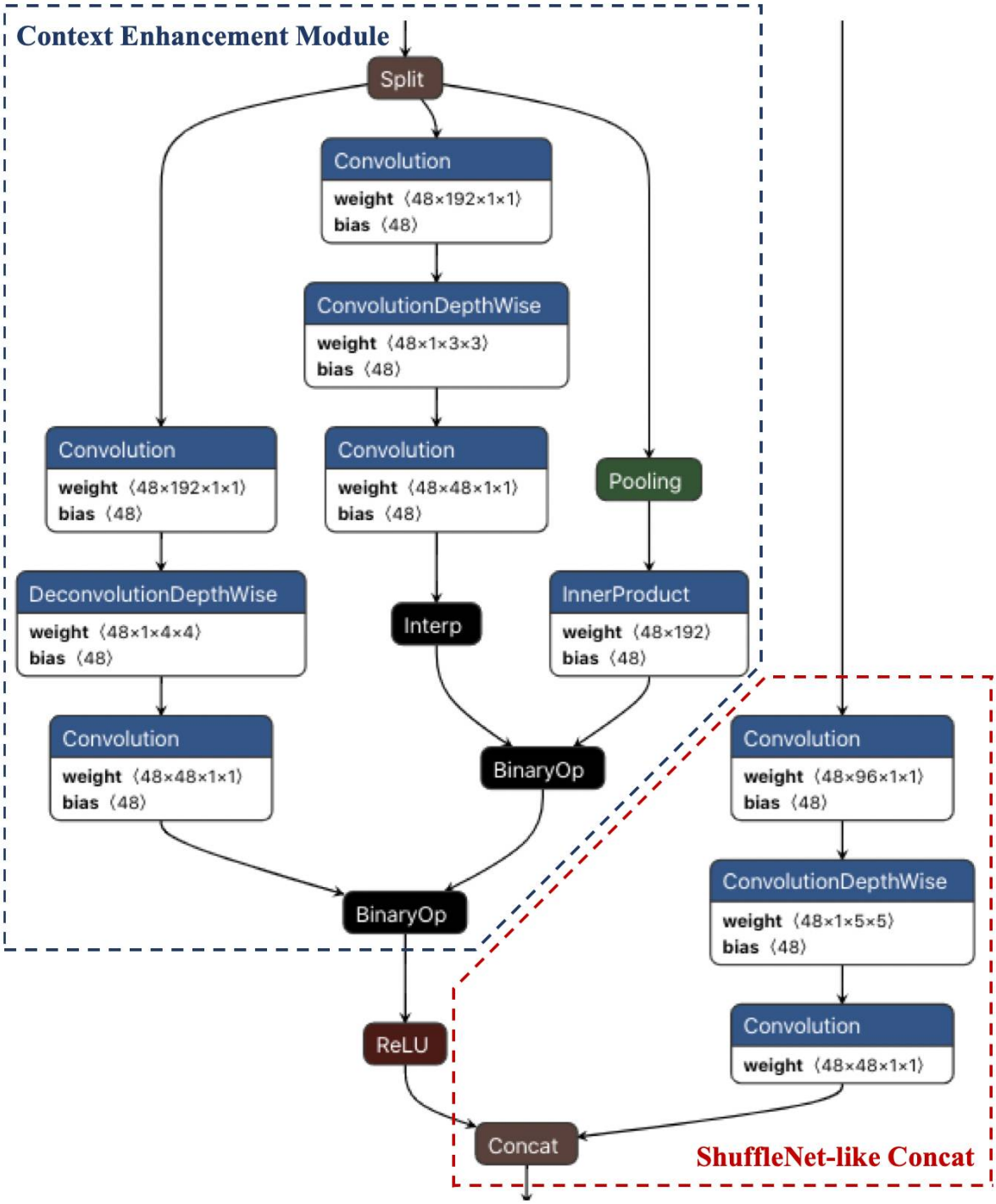
（1）大感受野（Large RF）

从 ResNet 到 ShuffleNetV2 主要影响了模型的深度和感受野。在以热力图回归的 CenterNet 中，模型的感受野显得异常重要。如何在保持网络轻量的前提下提高模型的感受野呢？从 AlexNet 到 VGG，VGG 通过将大尺度的卷积核拆解为多个小尺度的卷积核（1 个 5x5→2 个 3x3）：在相同感受野下，2 个 3x3 卷积的参数量和计算量均只有 1 个 5x5 的 18/25。然而，这在深度（depth-wise）卷积的时代并不适用。在 ShuffleNet 中，5x5 的 depth-wise 卷积获得两倍感受野，仅比 3x3 的 depth-wise 卷积增加极少的计算量（如图 3）。因此，我们将 ShuffleNetV2 中所有的 depth-wise 卷积均替换为 5x5 卷积。因为缺少 ImageNet 预训练的 5x5 模型，我们取巧地将 3x3 的 ShuffleNetV2 预训练模型进行卷积核的零扩边（zero padding），得到 5x5 的大卷积核 ShuffleNetV2。



（2）轻检测头（Light Head）

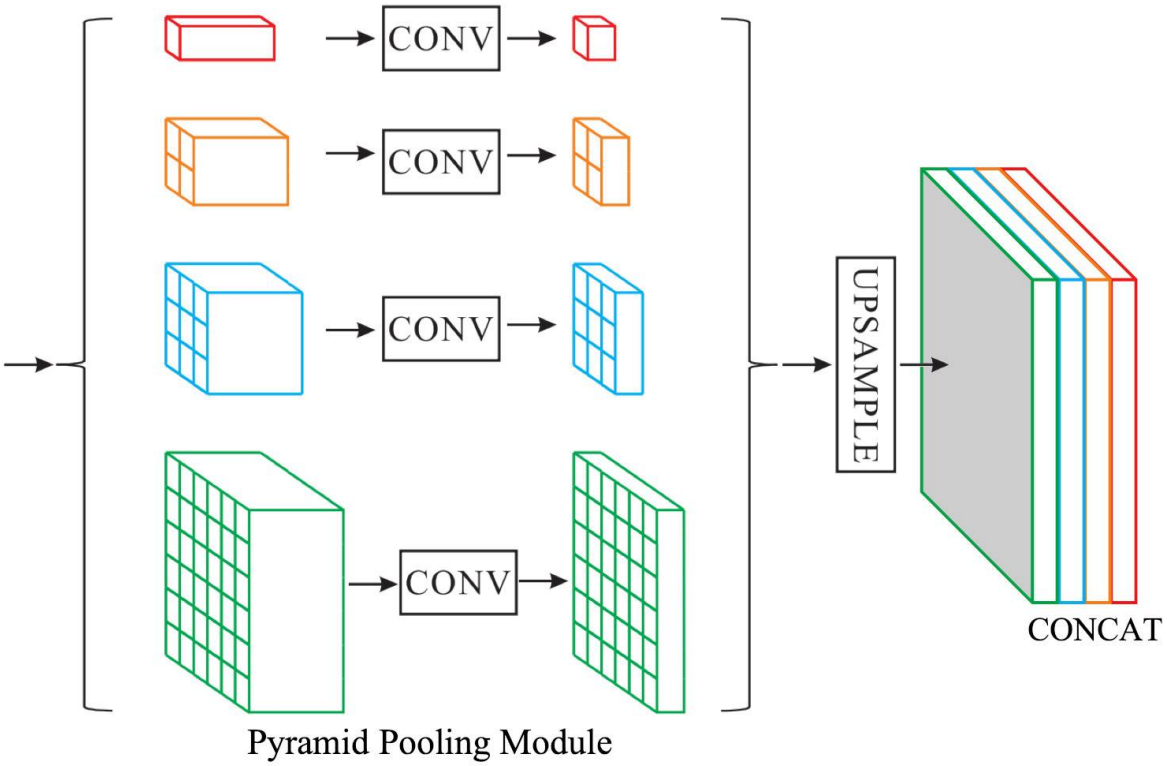
CenterNet 的检测头使用类 U-Net[21]的上采样结构，可有效地融合低层细节信息，从而提高对小物体的检测性能。然而，CenterNet 的检测头并未针对移动端进行优化，因此我们对其进行 ShuffleNet 化改造（见图 4 红框）。首先，将检测头的所有普通 3x3 卷积替换为 5x5 的 depth-wise 卷积，并将可形变卷积（DeformConv）也改造为 depth-wise 的可形变卷积。其次，参照 ShuffleNet 通道压缩的技巧，将 CenterNet 中多层特征的残差融合（residual）改造为通道压缩的连接融合（concat）。通过大感受野（Large RF）和轻检测头（Light Head），优化后的模型在 MS-COCO 数据库在计算量（FLOPs）、参数量（Parameters）和检测性能（mAP）均取得优异的结果，见表 1。



Model	Input	FLOPs	Parameters	mAP	AP_50	AP_75
CenterNet+ResNet18	512x512	17,790 M	14.1 M	25.9	41.3	27.9
SSD300	300x300	35,200 M	34.3 M	25.7	43.9	26.2
YOLOv2	416x416	17,500 M	67.4 M	21.6	44.0	19.2
YOLOv3	320x320	38,970 M	62.3 M	28.2	51.5	-
YOLOv3-Tiny	416x416	5,560 M	12.3 M	-	33.1	-
Light-Head R-CNN (ShuffleNetV2 x1.0)	800×1200	5,650 M	2.30 M	23.7	-	-
SSD+MobileNet	300x300	1,200 M	6.80 M	18.8	-	-
SSDLite+MobileNet	320x320	1,300 M	5.10 M	22.2	-	-
Pelee	304x304	1,290 M	5.98 M	22.4	38.3	22.9
Ours (ShuffleNetV2 x1.0)	**512x512**	**1,380 M**	**1.10 M**	**23.9**	**39.2**	**25.3**

（3）金字塔插值（Pyramid Interpolation Module，PIM）

然而，可形变卷积（DeformConv）对移动端加速并不友好，因此我们需要重新设计 DeformConv 的替代品。DeformConv 可自适应地对多尺度信息进行抽取，在 MS-COCO 中的小物体检测起到巨大作用。“扫一扫”识物对小物体的检测需求并不是非常强烈，DeformConv 更多的是提供多样化的尺度特征。对此，我们借鉴图像分割方法 PSPNet[22]（见图 5）的金字塔池化（Pyramid PoolingModule，PPM），提出了金字塔插值（Pyramid Interpolation Module，PIM）同时实现多尺度特征的融合和特征图的插值（见图 4 蓝框）。PIM 中主要包括三条分支进行 2 倍上采样：空洞解卷积，卷积+上采样，全局平均池化+全连接。其中，“空洞解卷积”对应大尺度特征；“卷积+上采样”对应小尺度特征；“全局平均池化+全连接”对应全局特征。在 ShuffleNetV2 x0.5 的骨干网络下，表 2 对比了各种上采样方法对检测性能的影响，可见 PIM 有效地替代 DeformConv 在“扫一扫”识物中的作用。



Pyramid Pooling Module

Ours (ShuffleNetV2 x0.5)	Input	FLOPs	Parameters	mAP	AP_50	AP_75
w/ DeformConv (Baseline)	320x320	200 M	477.77 K	67.20	88.60	75.90
w/ DeConv	320x320	99 M	198.45 K	62.90	83.80	71.90
w/ PIM	**320x320**	**107 M**	**198.45 K**	**67.60**	**87.80**	**76.00**

五、部署

通过以上优化，我们最终采用表 2 中最优结果作为“扫一扫”识物的移动端检测模型。该模型采用基于 pytorch 框架的 mmdetection 作为训练工具。在移动端部署上，我们采用 ncnn 框架，将 pytorch 模型转换为 onnx 模型再转换为 ncnn 模型，并在转换过程中将参数量化到 16bit。此外，为了进一步减小模型体积和加速，我们将网络中 conv/bn/scale 三个连续的线性操作融合为一个 conv 层，在不影响效果的同时可减少约 5%的参数量，并提速约 5%~10%。最终，**“扫一扫”识物的移动端检测模型仅 436 KB，在 iphone8 的 A11 CPU 上的单帧检测时间仅 15ms。**

六、展望

目前“扫一扫”移动端检测只是开端，移动端物体检测的发展也才刚刚开始。抛开“扫一扫”识物的场景，CenterNet 在通用的物体检测上仍存在以下问题：如何解决类别增加带来的检测头爆炸性增长？可形变卷积（DeformConv）是否存在更通用的替代品？U-Net 式的上采样结构是否可进一步优化？路漫漫其修远兮，在我们后续工作中将针对这些问题进行探索。

参考文献

[1] Girshick, Ross. "Fast R-CNN." international conference on computer vision (2015): 1440-1448.
[2] Ren, Shaoqing, et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." IEEE Transactions on Pattern Analysis and Machine Intelligence 39.6 (2017): 1137-1149.
[3] He, Kaiming, et al. "Mask R-CNN." international conference on computer vision (2017): 2980-2988.
[4] Redmon, Joseph, et al. "You Only Look Once: Unified, Real-Time Object Detection." computer vision and pattern recognition (2016): 779-788.
[5] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." computer vision and pattern recognition (2017): 6517-6525.
[6] Redmon, Joseph, and Ali Farhadi. "YOLOv3: An Incremental Improvement." arXiv: Computer Vision and Pattern Recognition (2018).
[7] Liu, Wei, et al. "SSD: Single Shot MultiBox Detector." european conference on computer vision (2016): 21-37.
[8] Lin, Tsungyi, et al. "Focal Loss for Dense Object Detection." international conference on computer vision (2017): 2999-3007.
[9] Lin, Tsungyi, et al. "Feature Pyramid Networks for Object Detection." computer vision and pattern recognition (2017): 936-944.
[10] Dai, Jifeng, et al. "Deformable Convolutional Networks." international conference on computer vision (2017): 764-773.
[11] Yu, Jiahui, et al. "UnitBox: An Advanced Object Detection Network." acm multimedia (2016): 516-520.
[12] Huang, Lichao, et al. "DenseBox: Unifying Landmark Localization with End to End Object Detection." arXiv: Computer Vision and Pattern Recognition (2015).
[13] Chen, Shuai, et al. "DuBox: No-Prior Box Objection Detection via Residual Dual Scale Detectors." arXiv: Computer Vision and Pattern Recognition (2019).
[14] Kong, Tao, et al. "FoveaBox: Beyond Anchor-based Object Detector." arXiv: Computer Vision and Pattern Recognition (2019).
[15] Tian, Zhi, et al. "FCOS: Fully Convolutional One-Stage Object Detection." international conference on computer vision (2019): 9627-9636.
[16] Law, Hei, and Jia Deng. "CornerNet: Detecting Objects as Paired Keypoints." european conference on computer vision (2019): 765-781.
[17] Zhou, Xingyi, Dequan Wang, and Philipp Krahenbuhl. "Objects as Points." arXiv: Computer Vision and Pattern Recognition (2019).
[18] Liu, Zili, et al. "Training-Time-Friendly Network for Real-Time Object Detection." arXiv: Computer Vision and Pattern Recognition (2019).
[19] Rezatofighi, Hamid, et al. "Generalized Intersection Over Union: A Metric and a Loss for Bounding Box Regression." computer vision and pattern recognition (2019): 658-666.
[20] Ma, Ningning, et al. "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design." european conference on computer vision (2018): 122-138.
[21] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." medical image computing and computer assisted intervention (2015): 234-241.
[22] Zhao, Hengshuang, et al. "Pyramid Scene Parsing Network." computer vision

and pattern recognition (2017): 6230-6239.
[23] Li, Zeming, et al. "Light-Head R-CNN: In Defense of Two-Stage Object Detector." arXiv: Computer Vision and Pattern Recognition (2017).
[24] Wang, Jun, Xiang Li, and Charles X. Ling. "Pelee: A Real-Time Object Detection System on Mobile Devices." neural information processing systems (2018): 1967-1976.

微信看一看数据质量控制介绍

作者：harryzhong，腾讯 WXG 应用研究员

> 在过去的几年里，质量控制作为微信看一看推荐的基石，我们在不断的迭代升级中积累了一定的技术和经验。本文主要跟大家分享一下总体框架的设计思路，以及如何通过平台通用化来解决组合爆炸的问题。文章为了尽可能的给读者描述一个完整质量控制体系的框架，这里既有技术上，架构上考虑，也会有产品上，业务上的考虑。

一、看一看介绍

1.1 看一看的场景

微信作为国内最大用户群体的应用，在人们的生活，工作，学习中提供社交功能的同时，还提供了工具，游戏，购物，支付，内容，搜索，小程序等服务。看一看作为微信的重要组成部分，在原有公众平台资讯阅读的基础上，为用户提供中心化的内容消费平台，旨在提升内容发现效率，优化内容质量，丰富内容种类。

1.2 看一看的质量控制

与其他很多资讯产品上线初的野蛮生长不一样，质量控制一开始就作为看一看上线标准被提出并进行了大量的投入。质量控制的必要性是由于多方面的因素决定的。



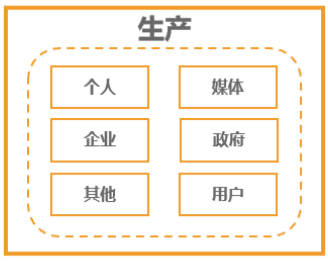
1. （内在属性）一个产品的出身和环境决定了其的调性，微信内在也决定了看一看的调性。在这里，调性这个词更多的是形容内容质量可以被用户接受的底线。用户对微信的期待也提高了这个底线。
2. （外部压力）微信的每一次改动都会受到广泛的关注，在功能上线初期面临着比其他应用更大的舆论压力和监管压力。
3. （黑产对抗）因为用户多，流量大，强大的经济效益引来黑产的参与，使得内容生产的大环境很差，低质量数据充斥各个角落；问题也在不断地的变化，识别控制难度也在持续加大。

二、总体框架

既然质量控制作为推荐系统必不可少的一部分，那么在推荐系统中处质量控制处于什么位置，包含哪些子模块，又是从哪几方面方面进行控制?

首先，我们先看一下内容数据在推荐系统的一生。

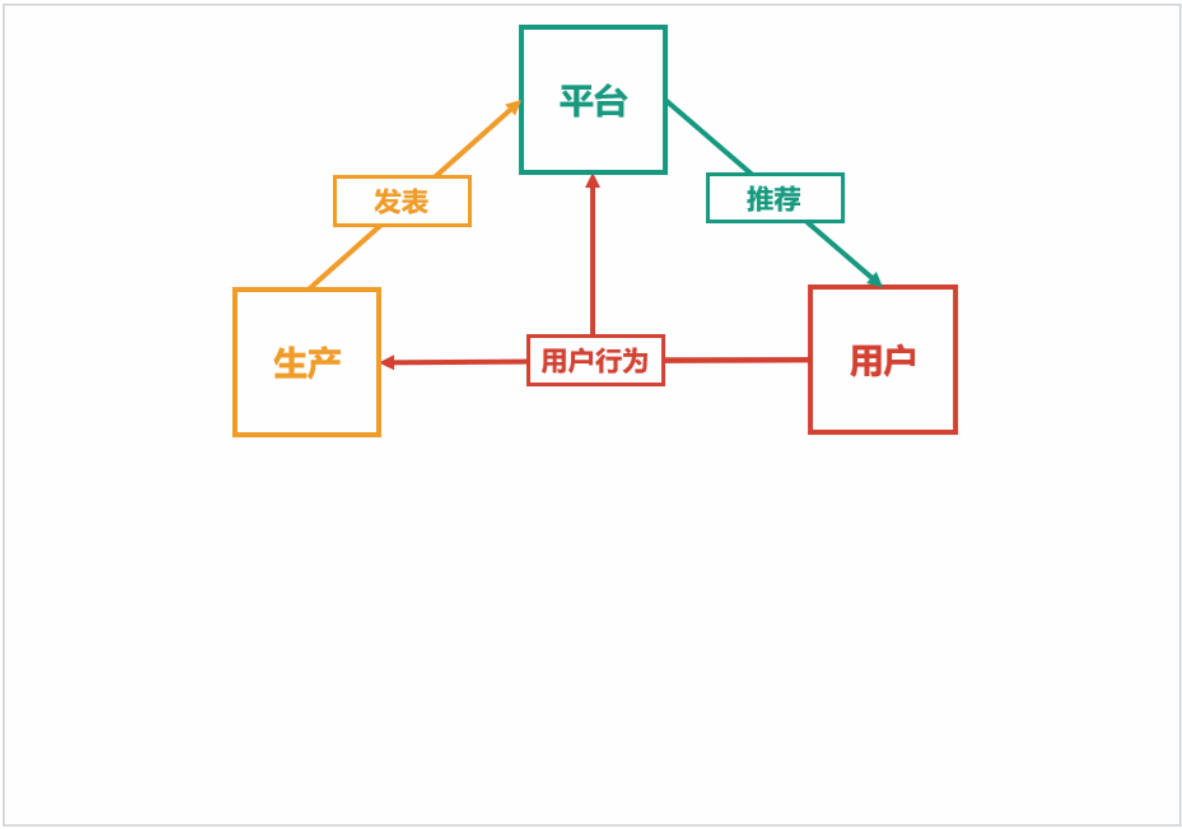
2.1 数据的一生



1. 数据从内容生产方发表产生，由平台收集数据。这里生产方来自四面八方，生产的内容五花八门，质量也是参差不齐。所以，平台将数据接入之后第一件事情就是将这些内容进行格式化和特征提取，方便后面的流程能通用化地处理这些内容数据。
2. 内容经过前面的初步处理后，质量控制模块根据已有的特征对内容进行过滤，主要是过滤不符合平台规则的低质量数据和提供后面排序需要考虑的质量因子。这里面涉及到过滤效率的问题，一般会分为粗过滤还有细过滤。排序推荐模块将根据内容特征还有用户行为对数据进行排序。
3. 最后，平台将内容推荐给用户进行消费，消费的过程中会与内容产生很多交互行为。这些行为数据，可以用于平台的质量控制和推荐排序；也可以反馈给生产者，让生产者能够了解到内容被消费的情况，生产更多符合用户需求的内容。

2.2 多维度控制

数据的一生中，我们可以看到平台都是通过对发表内容本身进行检测来进行质量控制，这显然是不够的。一个控制框架的有效运行不能仅仅在一个地方进行监控，需要全面，多维度共同作用。我们可以抽象一下平台，生产，还有用户。我们会发现其实除了对平台自身检测能力进行提升外，也可以对生产者和用户进行引导，评估。



1. 根据生产者生产内容的质量，还有用户对内容的反馈，平台可以对生产者进行分级分类。对不同等级的生产者，使用不同质量控制策略，如，高质量的账号颁发个免检标签，在质量控制的时候避免误伤。而对于一些已经被检测到有低质量内容的账号进行一定时间的封禁，减少相似问题的爆发，增加长尾问题的召回，提升系统的响应速度。不同类目也是同样的道理，比如高危的内容类（社会民生，养生健康，娱乐）需要加强控制。
2. 而对于用户，我们同样可以根据一下用户的基础数据（地域，年龄，性别）和用户在微信内的一下阅读信息，对用户进行分类。不同用户可以使用不同的控制策略。这样在保证了用户对体验有不同要求的前提下，尽可能地保证推荐系统的效率。

2.3 辅助模块

除了上面说的三个部分之外，还有几个同样非常重要的质量控制辅助模块，这些模块对保证整个质量控制的正常运行至关重要。



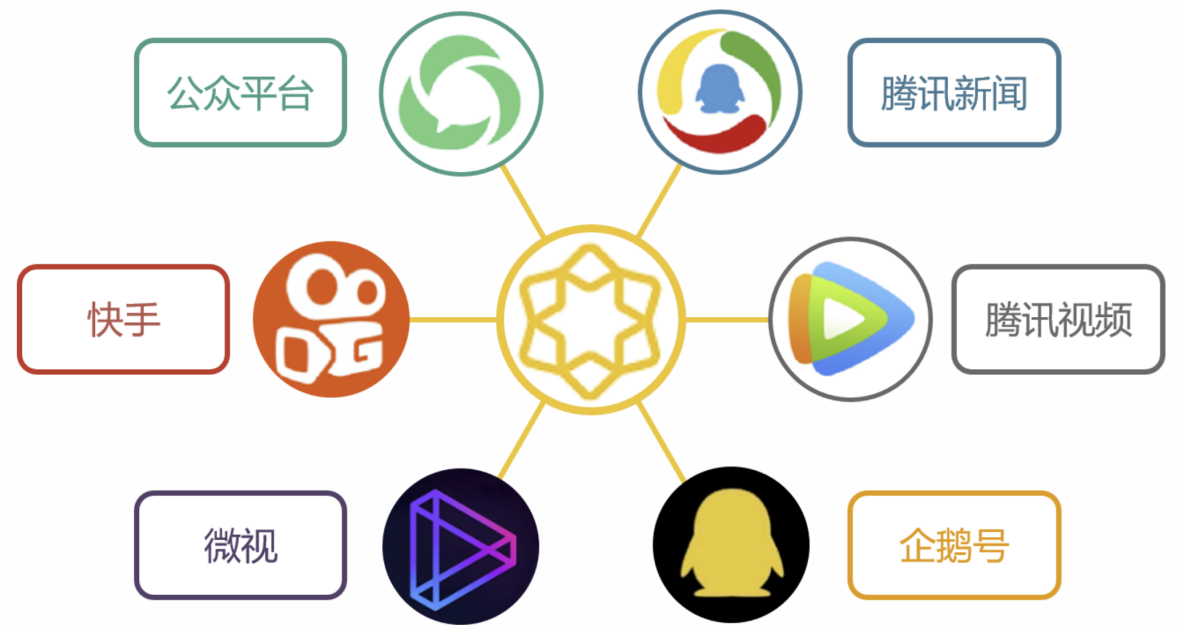
1. 监控系统：监控系统需要对平台，生产，用户的各个方面进行监控，有些数值监控指标可以通过机器持续监控，而有些数据本身的问题则需要人工的轮询与审核，还有就是新问题的发现与归纳汇总。其中最重要的是能够进行全方位覆盖，不同范围，不同领域，有时候问题可能隐藏的茫茫大盘数据中。
2. 干预系统：干预系统是对问题出现后，进行快速处理的手段。这样能够在模型策略迭代较慢的时候暂时压制问题，减少影响范围。干预系统除了生效快之外，还有一个重要的能力是相似内容的查找（去重指纹），如相似标题的封禁和相似图片的封禁。
3. 标注系统：标注系统作为一个提升样本收集效率的平台，不仅能有效提升标注人员的标注效率，进而有效提升模型的迭代效率；而且标注系统方便对问题的收集，总结，归纳，形成各个领域的知识库。

三、数据与质量

前面介绍了质量控制的必要性，也介绍了质量控制的大体框架。那么到底数据是指哪些数据？所谓的质量又指的是哪些质量指标？这些数据跟这些质量指标带给我们什么问题与挑战呢？

3.1 内容数据源

看一看的数据从来源上分包括公众平台，腾讯新闻，腾讯视频，企鹅号，微视，快手等。从内容形式上分有文章，新闻，视频，图片等。样式丰富的数据在满足不同用户的内容需求的同时也加大了监管的难度。



3.2 质量指标

同时，看一看以用户的体验为出发点结合具体业务需要建立了一套质量指标体系。这里指标随着业务的发展不断在增加，含义也在不断的丰富。多维度的指标可以为业务的质量过滤和推荐排序提供更多的个性化的选择。

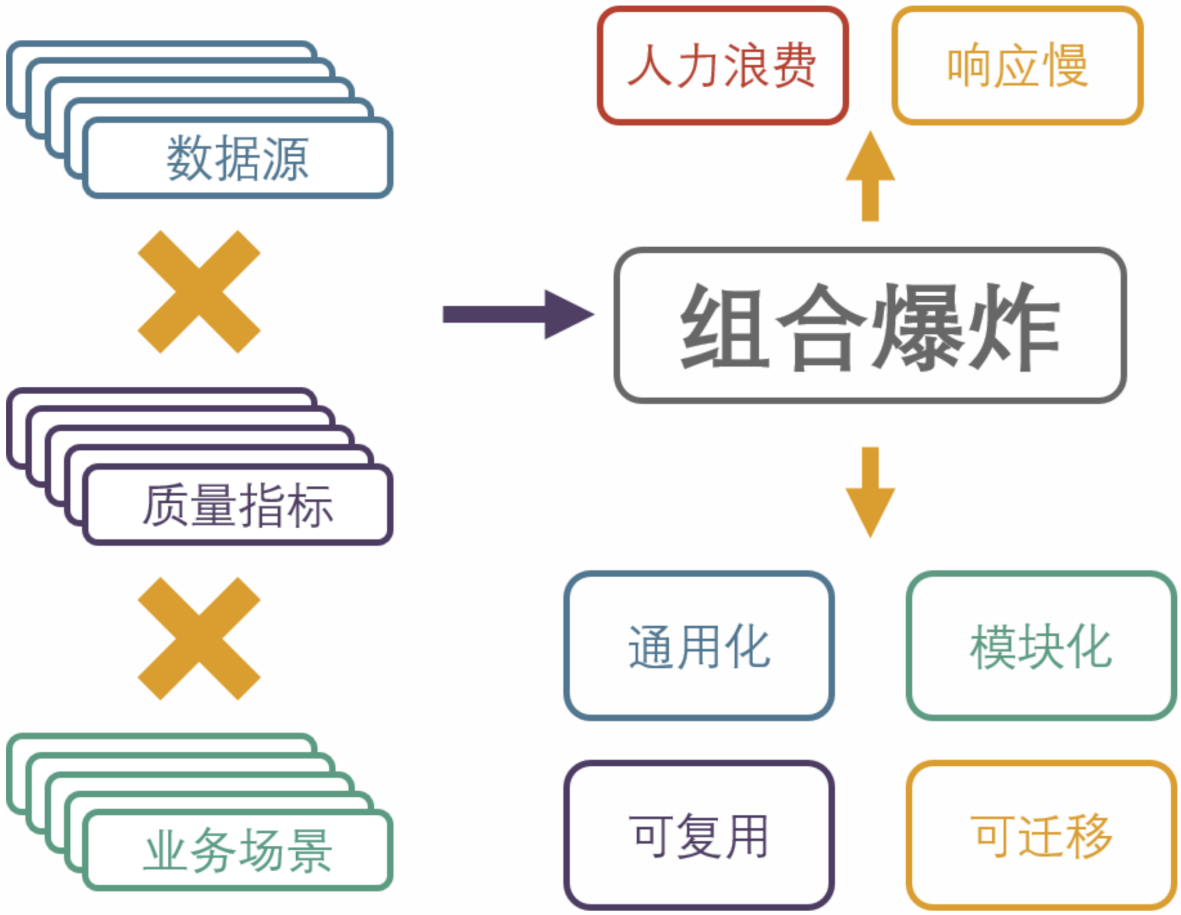


3.3 问题与挑战

那么，我们的问题和挑战是什么呢？我们先来看一下日常工作中经常遇到的一些对话。事情的本质往往隐藏在复杂的表面下。



丰富的数据源，多维度的质量指标，个性化的业务场景带来的其实是**组合爆炸**问题。如果对每一种组合都单独处理，不仅导致大量人力的浪费；同时，也无法对速度要求高的质量控制问题进行快速响应。毫无疑问从体系上进行**通用化，模板化，可复用，可迁移**是我们解决问题的方向。下一节，我们会从一个普通任务的迭代周期出发，分析上面对话对应的迭代过程的那一部分，并对相应的部分进行相应的通用化。



四、体系演进

4.1 规则，特征，样本，模型

下图展示了一个迭代周期里面几个关键的过程。



1.规则制定：规则制定可以说是质量问题解决中最重要的工作，问题分析清楚了，解法也就出来了。这里简单列几个小原则，不进行展开。

- 主要组成：需要对问题进行全面的分析，包括场景数据是什么，问题有多少子类型，问题比例有多大，影响范围有多广，业界是否也有同样的问题，做法是什么。
- 基于需求：这个阶段更应该从需求出发进行问题的分解，尽量减少技术性的考虑。很多时候人会往往会混淆“需要做”和“怎么做”。因为觉得这个东西难做，或者不能做而将这个东西认为不需要做。
- 基于数据：规则的制订需要结合数据进行讨论，不能几个人凭主观想法就定下规则。数据往往会提供很多我们意想不到的冲突情况。
- 衡量标准：规则必须能在人与人之间传递。

2.特征提取：这里涉及如何将业务需求转化为技术需求。根据问题分析的结论，对直接特征进行提取，如标题，正文，封面图，隐藏内容，页面结构，跳转链接，锚文本，账号，统计特征等。

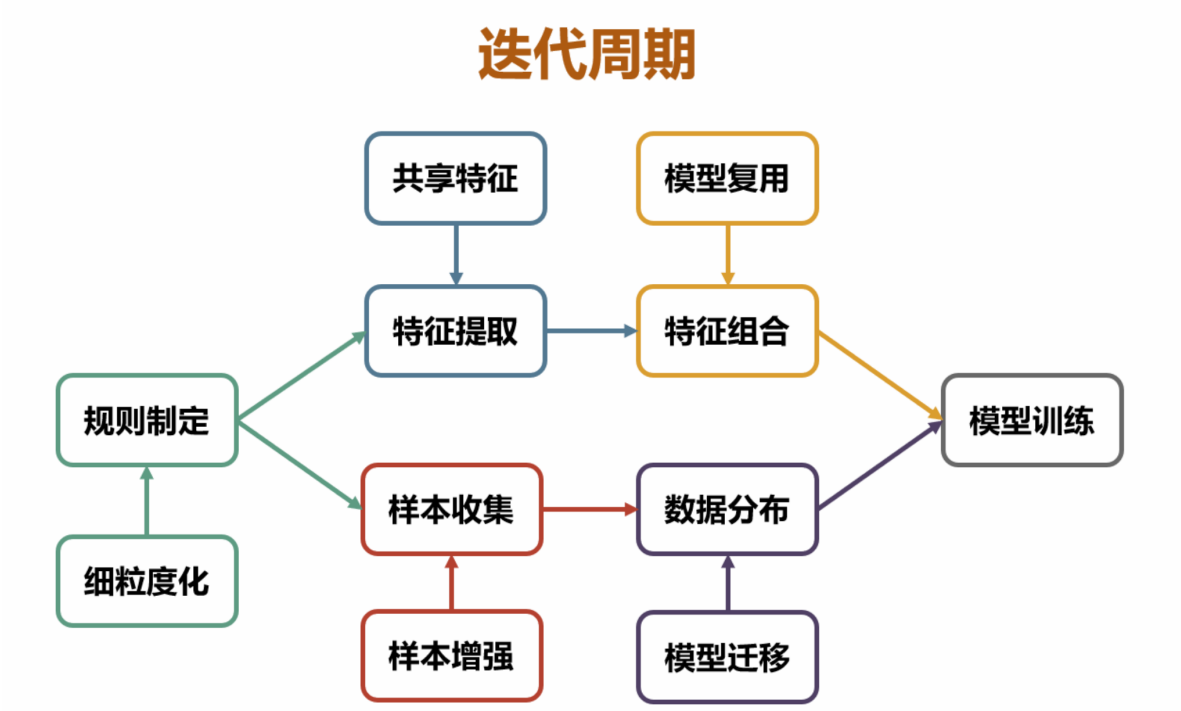
3.样本收集：根据需要从原始数据中收集一定量的正负样本，这些样本可以通过相似样本挖掘，也可以通过人工标注。

4. 模型训练：使用模型对提取的特征进行组合，并对收集的样本分布进行拟合。

- 特征组合：模型将问题需要考虑的特征进行组合。有些问题只需要考虑标题（标题党），有些问题需要考虑文章标题的匹配程度（文不对题），有些问题需要同时考虑文本与图片的结合（低俗色情）；有些问题只需要考虑主题词（广告），有些问题需要考虑上下文语境（假新闻）。
- 数据分布：模型会尽可能贴合样本数据的分布，降低 Loss。有些数据源问题比例高，有些数据源问题比例低。切换数据源的时候，如果样本分布与目标数据源的分布不一致，效果往往有较大的差异。

4.2 通用化

看完上一小节的描述，不知道大家有没有把对话跟迭代周期的过程对应起来。接下来尝试对迭代周期里面的各个部分进行通用化，包括细粒度化，共享特征，样本增强，模型复用，模型迁移。



1.细粒度化：在进行规则制定阶段，要避免指定一个很大的质量问题进行优化，尽量把问题定义成“独立”，“明确”，“与业务无关”的细粒度问题。这样可以减少后面因为业务变化导致的不确定监督问题，提升模型适用性。举个例子：我们之前有个质量问题是黑四类（广告，活动，通知，招聘），标注的时候没有区分，训练的时候也是用一个模型。这样在有新的业务出现的时候突然发现我们不需要黑四类，而是黑三类。还有其他一些不建议的问题类型："标题不规范"，"正文质量差"，"内容不优质"。

2.共享特征：虽然不同数据源的样式不一样，这样需要对不同的数据源通过页面解析进行特征抽取，统计。但是对于同一个数据源，不同的质量问题，往往会用到很多相同的特征。一个数据源抽取后的结果可以保存在结构化的数据结构中方便各个质量模型，还有其他的业务模型使用。如文本的位置，是否有链接，是否隐藏，是否模板内容等。

3.样本增强：通过相似样本挖掘和人工标注容易导致过拟合，标注效率低的问题。这里可以结合数据增强提升模型的鲁棒性，结合主动学习提升标注效率。

4.3 模型复用

在处理不同的质量问题的时候，我们发现有些问题用到的特征，拟合的方法都是相似的。像越来越多平台算法库对基础模型提供支持一样，通过从具体问题中抽象问题，构建通用模型框架可以有效提高我们的工作效率。而构建通用模型框架需要结合的前面提到的共享特征在设计之初就其保证通用性，适配性，可扩展性。

4.3.1 通用模型



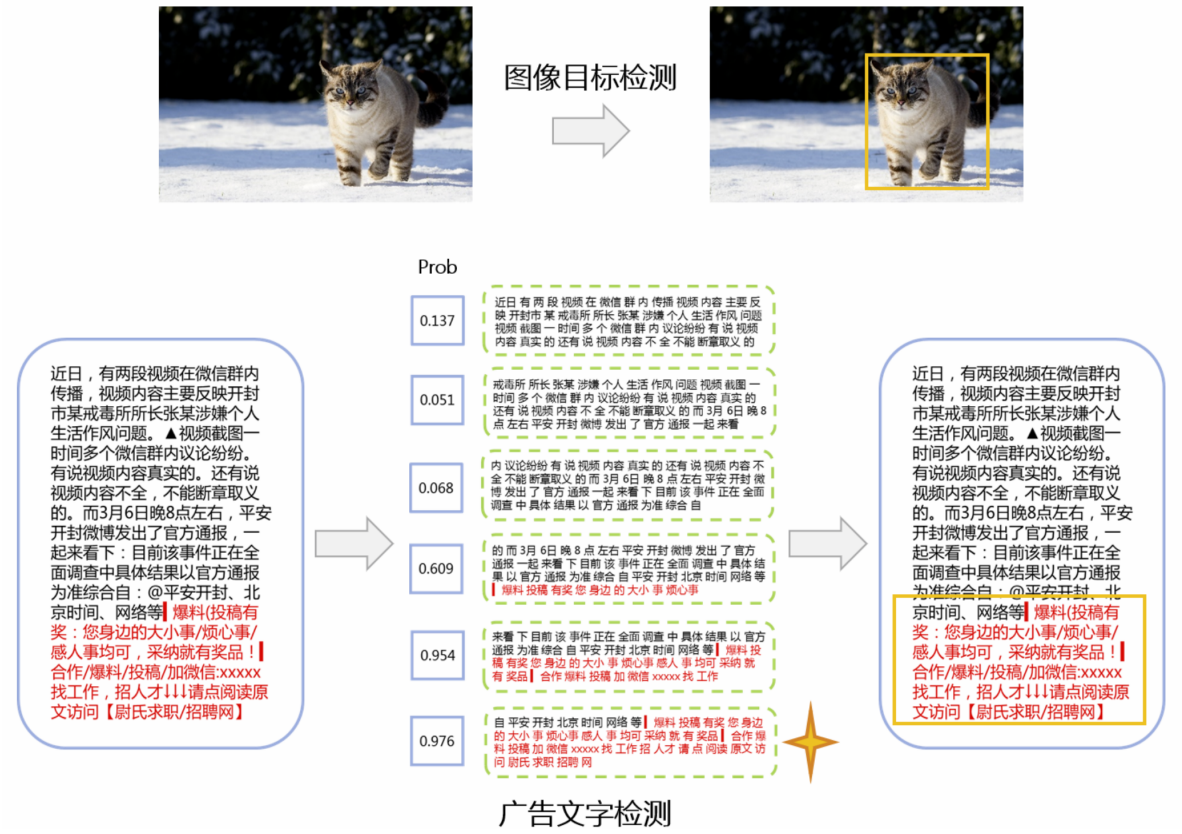
4.3.2 广告模型

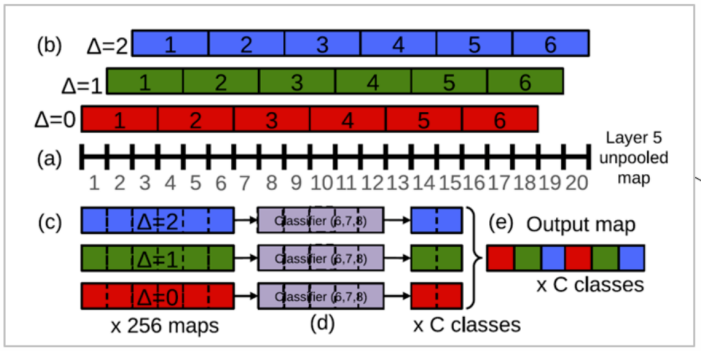
这里介绍一下广告识别模型在构建过程中，涉及的几个通用问题模型。广告模型主要是负责识别文章里面的广告内容。从内容上，分为文字广告，图片广告，图文广告。从位置上分顶部广告，中间广告，底部广告。从篇幅上分主体广告，插播广告。 那么这里可以抽象成 3 个通用的问题：

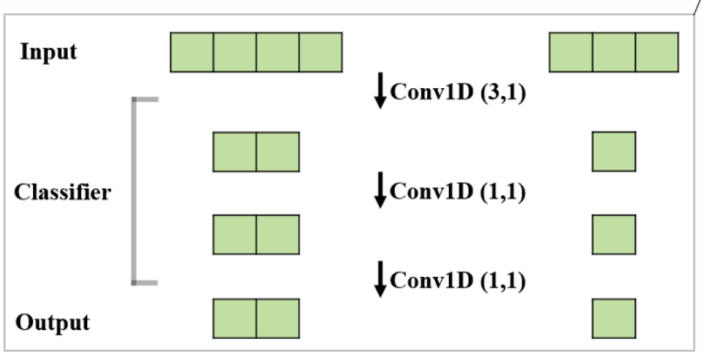
1.问题内容的定位（插播广告）； 2. 多类型特征的融合（图文广告）； 3.文章结构的序列化（顶部，中部，尾部，篇幅）。

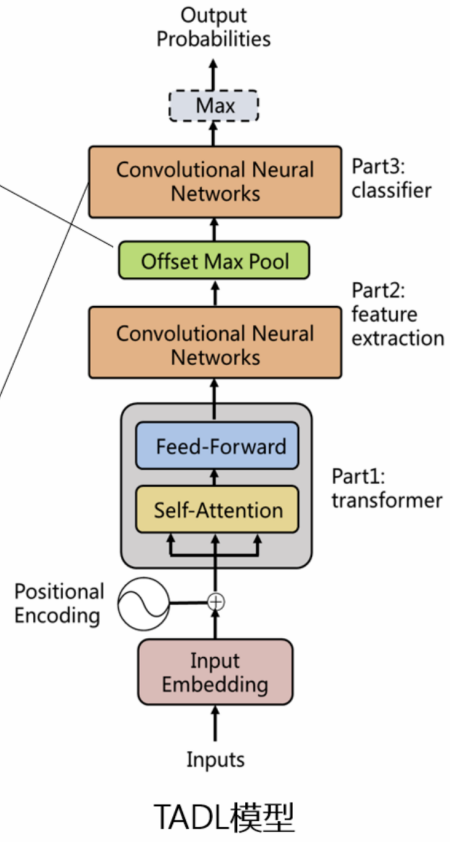


- 问题内容的定位：这类问题与图像中的物体检测相似，都是需要识别相应目标及其对应的位置。据此，我们提供一个全新的用于文本问题检测的框架 TADL。通过滑动窗口的检测方法将大段文本切分成多个小片段，并对每个小片段进行广告概率打分和反推定位，从而在一个模型框架内同时实现了广告文字的识别和定位，并且只需要文章级别的标注就能完成训练。

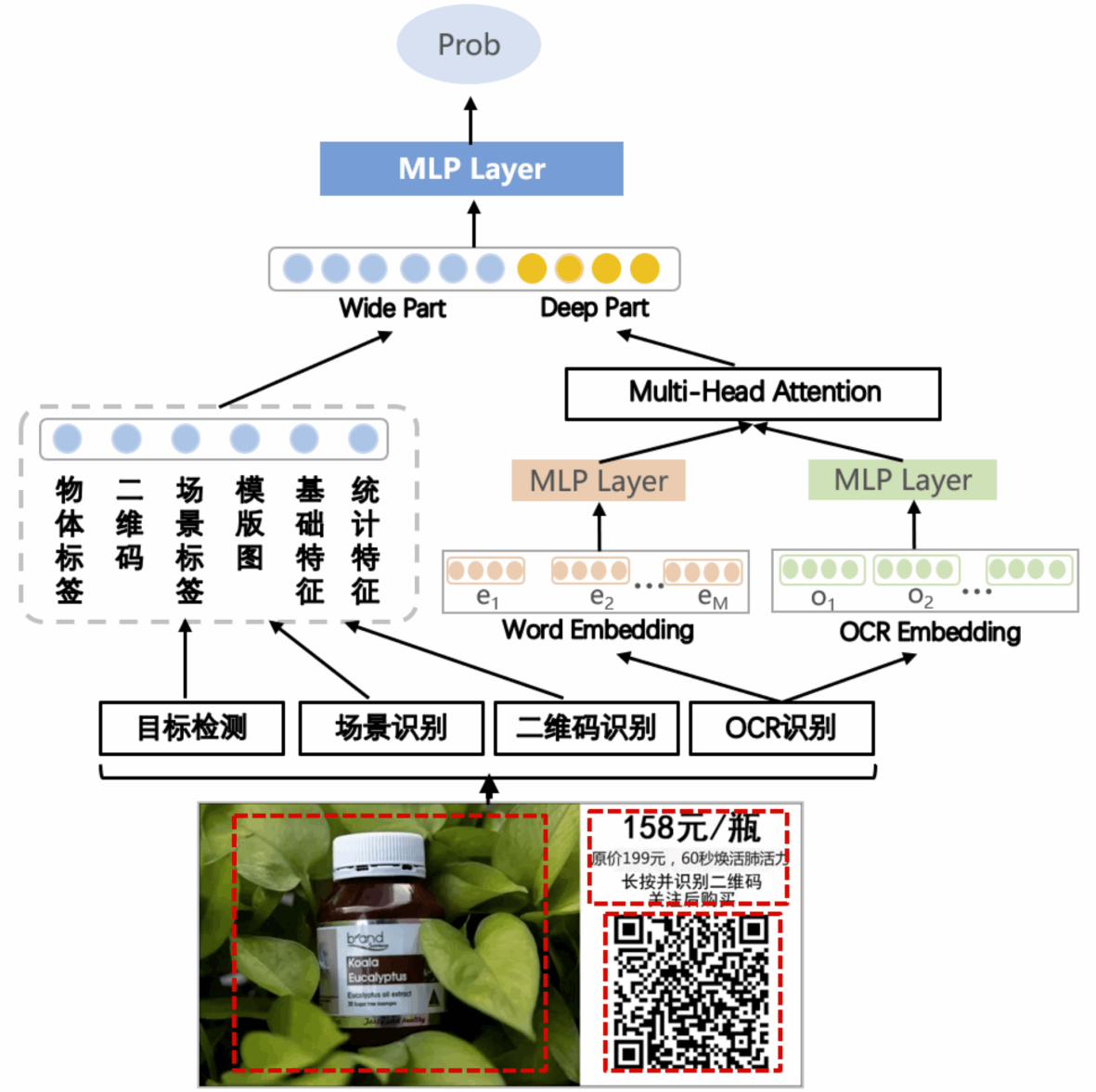




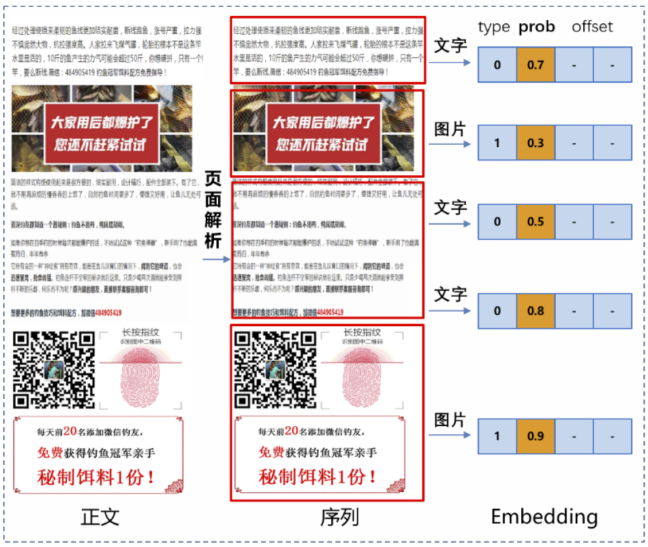


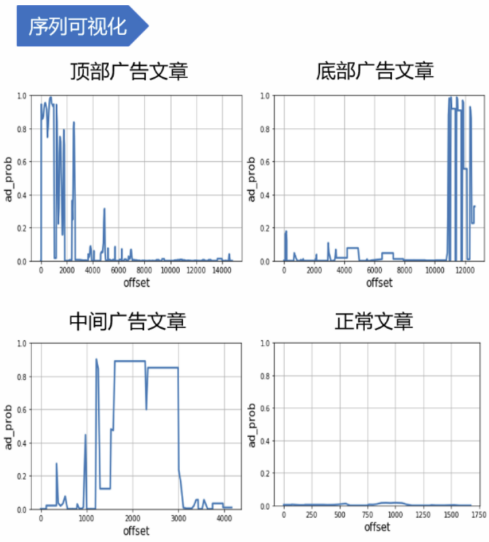


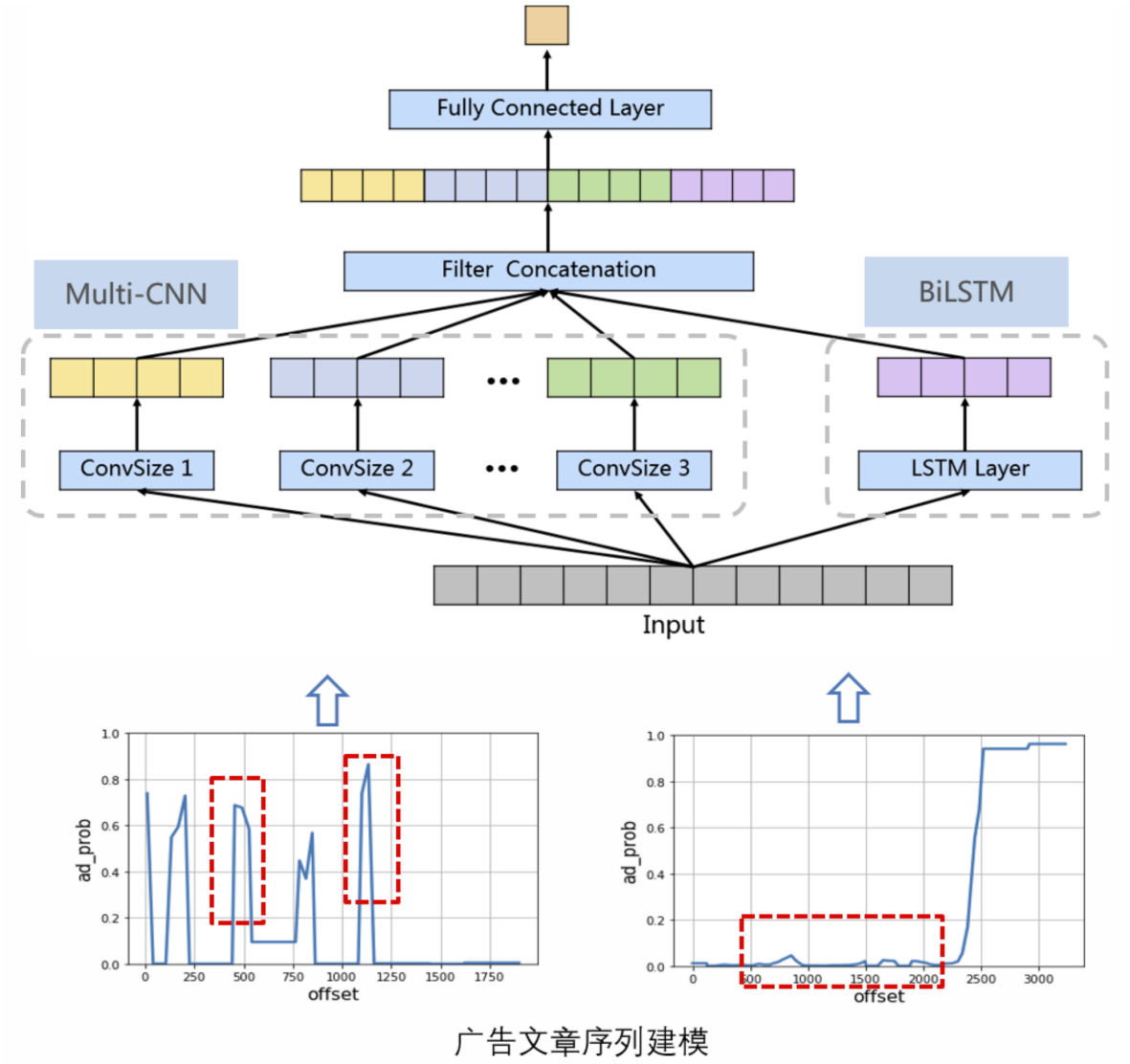
- 多类型特征的融合：图片广告定义广泛，广告的程度需要同时考虑多种类型的特征（文本大小，文本位置，文本语义，图片场景，图片内容）。相对于端到端的需要大规模标注的深度模型，对问题进行分而治之的 wide&deep 模型框架具有更好的解释性，更高的召回。



- 文章结构的序列化：一篇文章是否有广告问题，需要对整篇文章中多模态广告的占比，位置进行综合考虑。将不同广告文章中的文本广告概率和图片广告概率进行序列化的可视化展示，发现通过对有限的序列模式进行识别检测可以用来解决多模态文章分类问题。模型通过 Multi-CNN 检测异常“突起”，通过 BiLSTM 检测概率变化趋势。



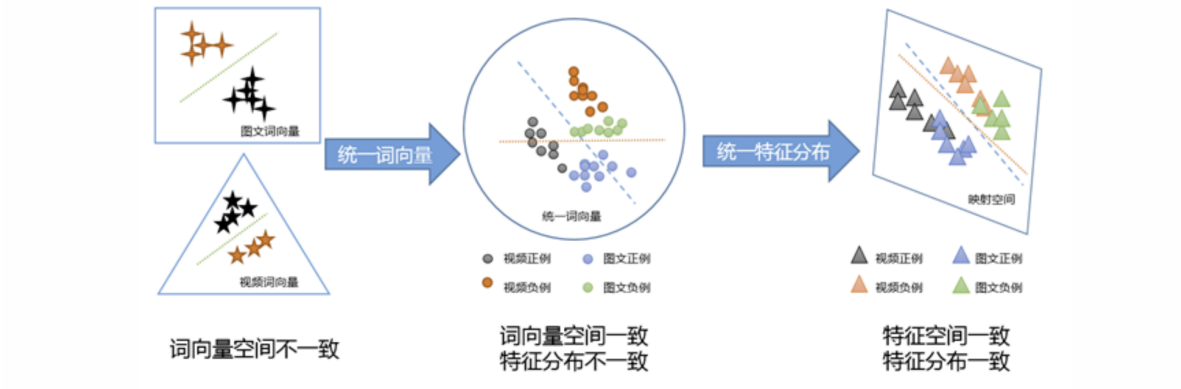




广告文章序列建模

4.4 模型迁移

数据源之间的数据分布一定是会存在差异的。模型迁移主要是想要解决数据分布不一致的问题。这里我们以色情低俗识别为例子，分“词向量”和“特征分布”两个方面来考虑迁移问题。



4.4.1 词向量统一

词向量在不同的场景中会表现不一样的语义，如“下面”在图文和视频标题里面就有不同的语义。多数据源数据训练的词向量因为获得了更多的信息量要好于多个数据源单独训练的向量。到后来基于更大量训练集的 bert 的出现更是极大的丰富了词向量的信息量。

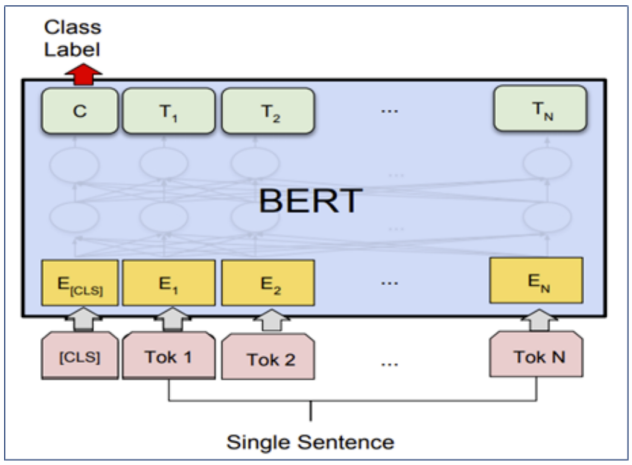
与“下面”最近top5/cos距离			
图文源		视频源	
上面	0.678	上面	0.750
进去	0.576	底下	0.732
仔细	0.566	下边	0.702
嘿嘿	0.538	里边	0.593
难受	0.524	里面	0.586

4.4.2 特征空间统一

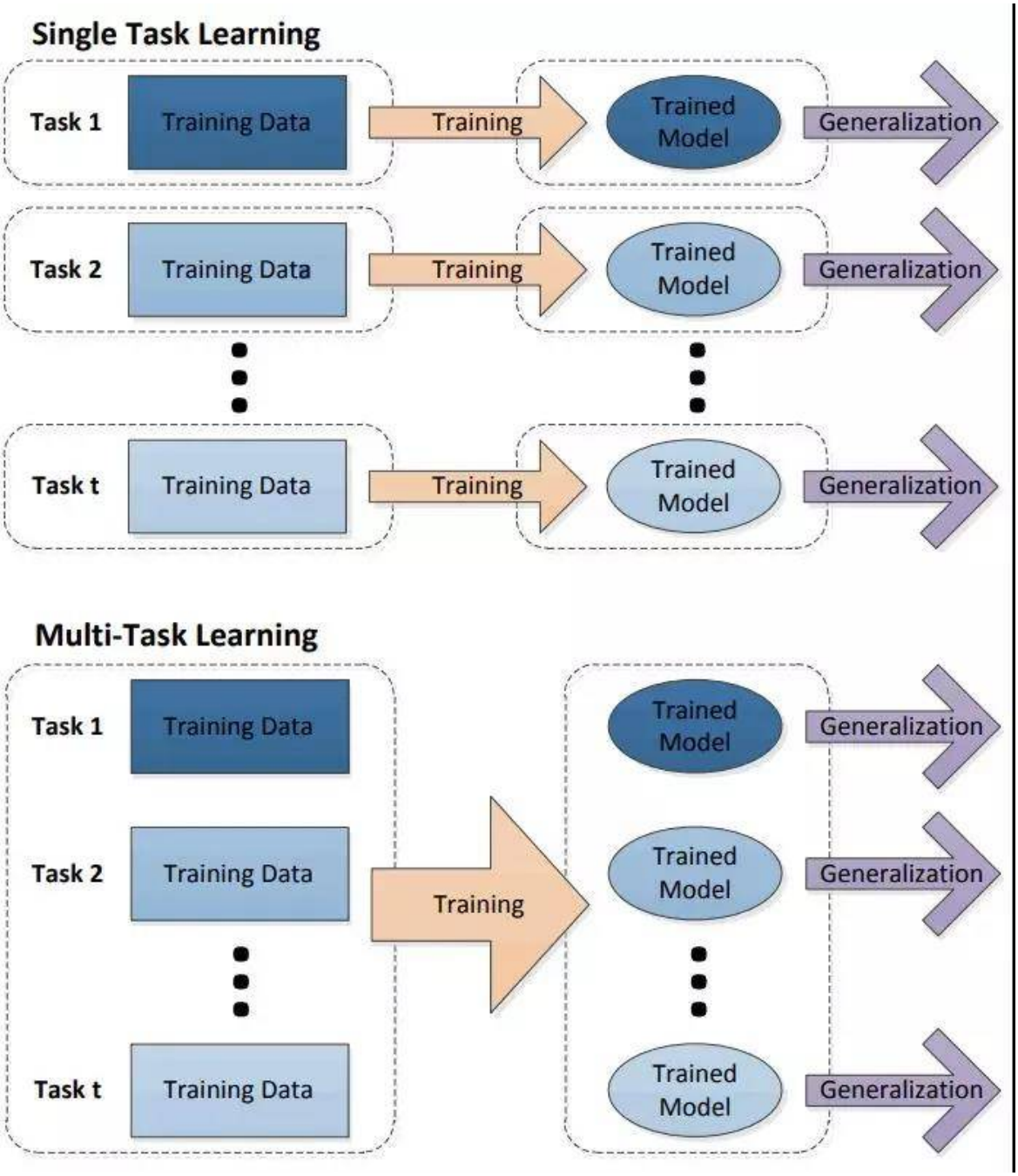
这里面的方法主要有 finetune，多任务学习，还有对抗学习。

top5敏感词/互信息MI值			
图文源		视频源	
美女	0.0085	出轨	0.0146
性感	0.0063	原配	0.0067
男人	0.0054	丈夫	0.0059
啪啪	0.0051	妻子	0.0058
早泄	0.0045	情人	0.0050

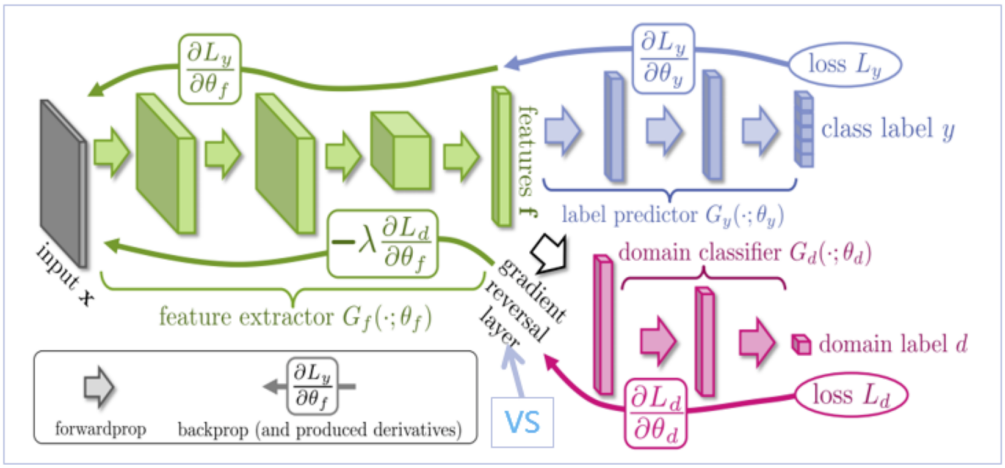
- Finetune 的目的是使用已有模型的特征提取能力，然后目标数据源进行组合调整达到较好的效果。这个模型在源领域数据标注量要比较大，模型框架能够表示足够多的共同特征时比较有效，如 bert。



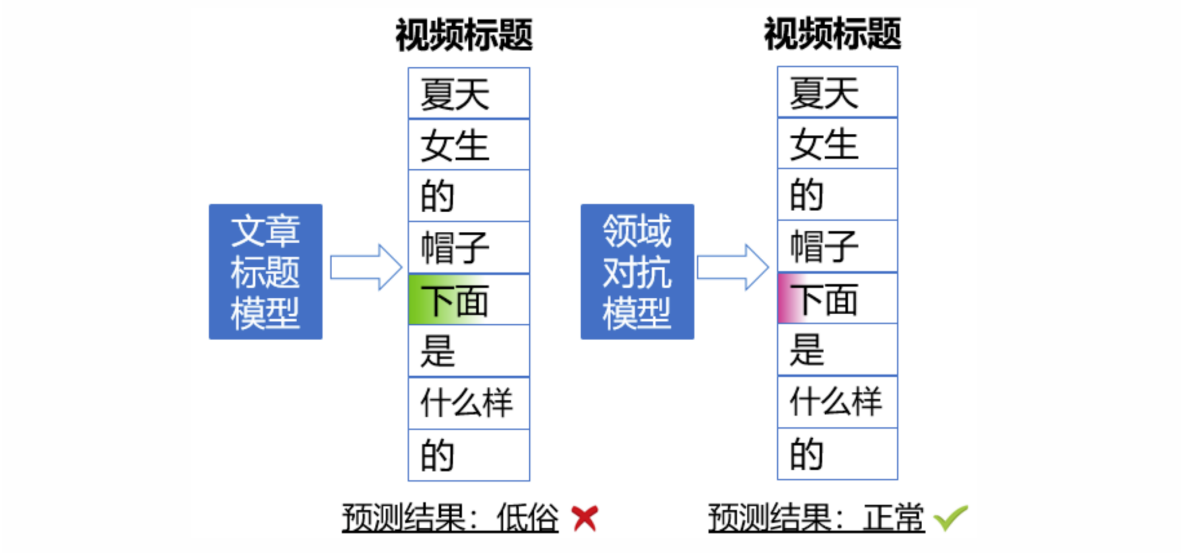
- 多任务学习通过不同任务间建立共享层，学习不同 domain 之间的共享知识，进而实现双赢。缺点是需要大量的目标领域的有标数据。



- 对抗学习充分利用了大量的无标数据消除一些邻域间分布不同的差异。



DANN模型示意图



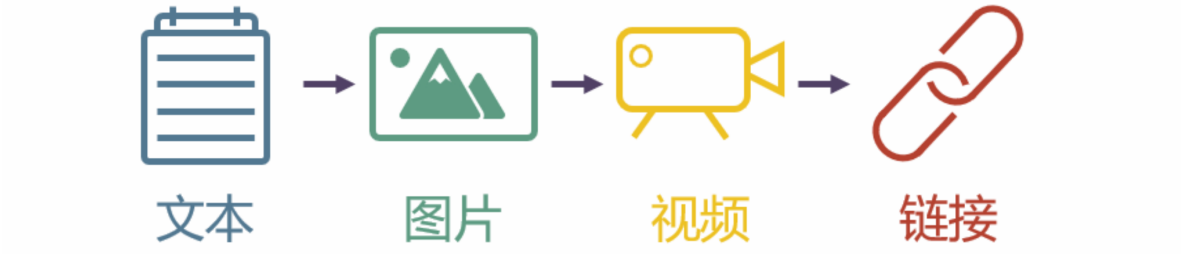
五、持续对抗

5.1 动态数据

前面所提到的多样化的问题，更多的是平台内部因为业务需求主动引入更多的数据和质量指标导致。这些问题相对来说是静态的。在业务上线之前，平台会进行摸底优化，不会有太多的意外。

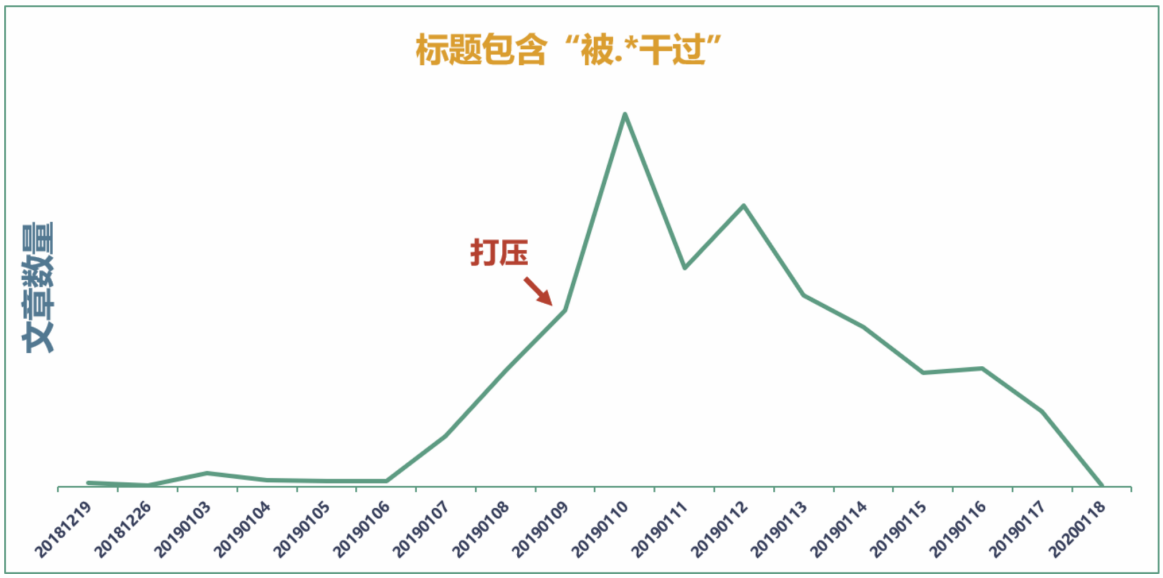
接下来我们讨论数据动态变化的问题。在日常的内容生产过程中，内容生产者根据平台的环境和用户的反馈不断修改内容的样式，类目。这些变化会改变数据的分布，从而导致平台对新数据出现漏召或者误判。其中，黑产的持续不断的有意对抗最为频繁，花样最多，也是我们最为关注的。这些对抗具有形式类目多，迭代周期多的特征。

- **形式类目多**：问题数据类型不断变化演进，从文本，图片，视频一直到链接。类目上也是在多个高危类目上来回切换，不断创新。





- **迭代周期短**：某个特殊的色情标题党样式，在某一时间短获得不错的曝光后，黑产立即进行了大规模的投产，而在受到打压后又迅速的转向其他类型的 case。



5.2 解决体系

由于敌暗我明，完全杜绝几乎是不可能的。原则上是提高黑产的作弊成本，降低平台的维护成本。我们还是从前面提到质量控制体系的三部分入手。

平台：每天都有少量的标注人工对线上数据进行轮询，对有问题的数据进行屏蔽收集。

用户：对大量用户反馈有问题的数据进行限制。

生产：

- 黑名单：对历史上劣迹斑斑的账号进行封禁，同时，对跟这些账号关联紧密的账号进行挖掘封禁。
- 沙盒机制：对新注册的，发文较少，未被平台验证的账号，限制其推广范围。

样本扩散，模型迭代：

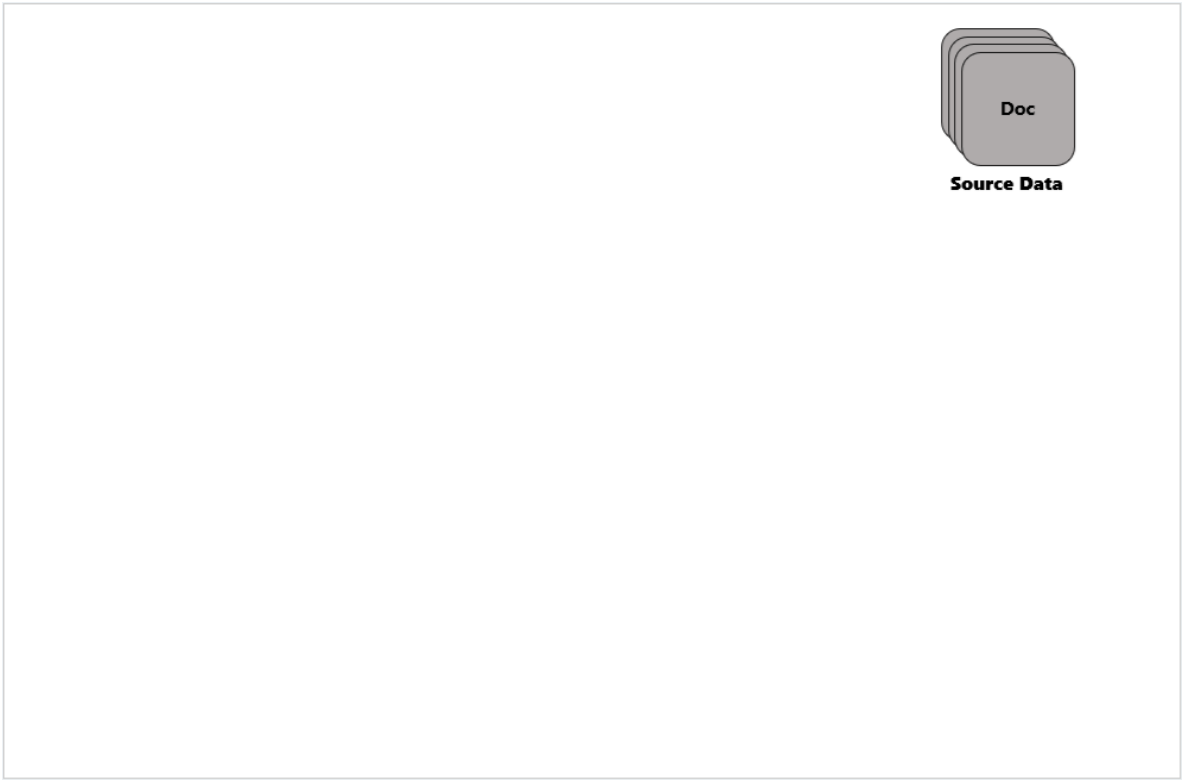
- 用户反馈和平台监控，虽然能发现新的问题，但是泛化性不够。
- 账号打压虽然具有泛化性，但是无法将新问题，新账号发现。
- 为了解决原有体系的不足，我们尝试结合两部分，通过人工监控跟负反馈收集样本进行扩散，然后提供给模型进行迭代提升召回。最后再从账号维度进行泛化打击。这里跟前文提到的样本增强的区别在于样本扩散更多的是对新问题样本的收集。

5.3 扩散迭代

5.3.1 正常流程

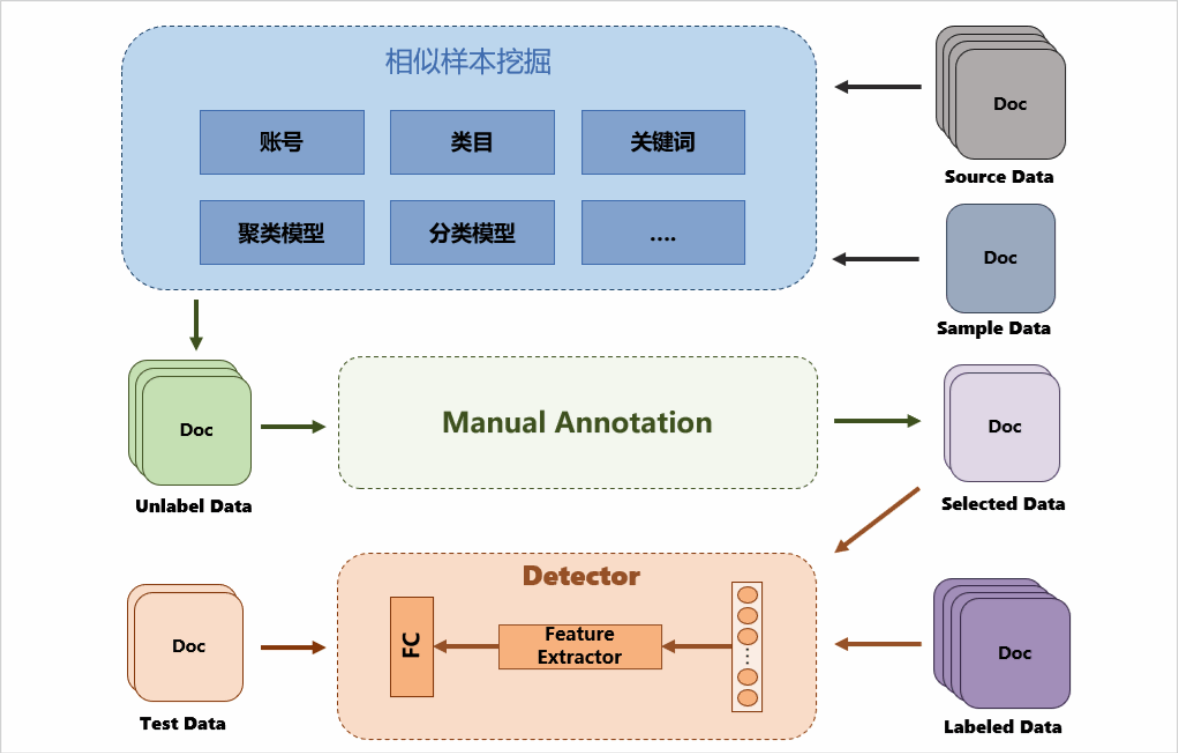
那么怎么进行样本扩散，加快模型的迭代周期呢？下面是一个样本收集周期里面大概需要做些什么事情。

- 首先根据发现的问题数据通过相似度计算，我们在源数据（source data）找到一批跟问题数据（sample data）相似度较高的未标注数据(unlabeled data)。
    1. 账号：一个账号的内容大部分比较相似，同一个黑产内容跟也往往相同。
    2. 类目：对问题比较集中的类目进行数据挖掘。
    3. 关键词：用一些匹配式框定一个范围。
    4. 聚类模型：简单的计算样本内容的文本相似性。
    5. 分类模型：构建简单的分类模型进行初筛。
- 然后对这批未标注的数据进行人工标注，这样可以筛选出一批能有效提升模型对新问题识别效率的标注样本（selected data）。
- 最后将筛选的样本跟原有的标注数据（labeld data）进行合并作为训练样本进行模型训练。训练好模型之后在测试数据（test data）上面进行测试效果。



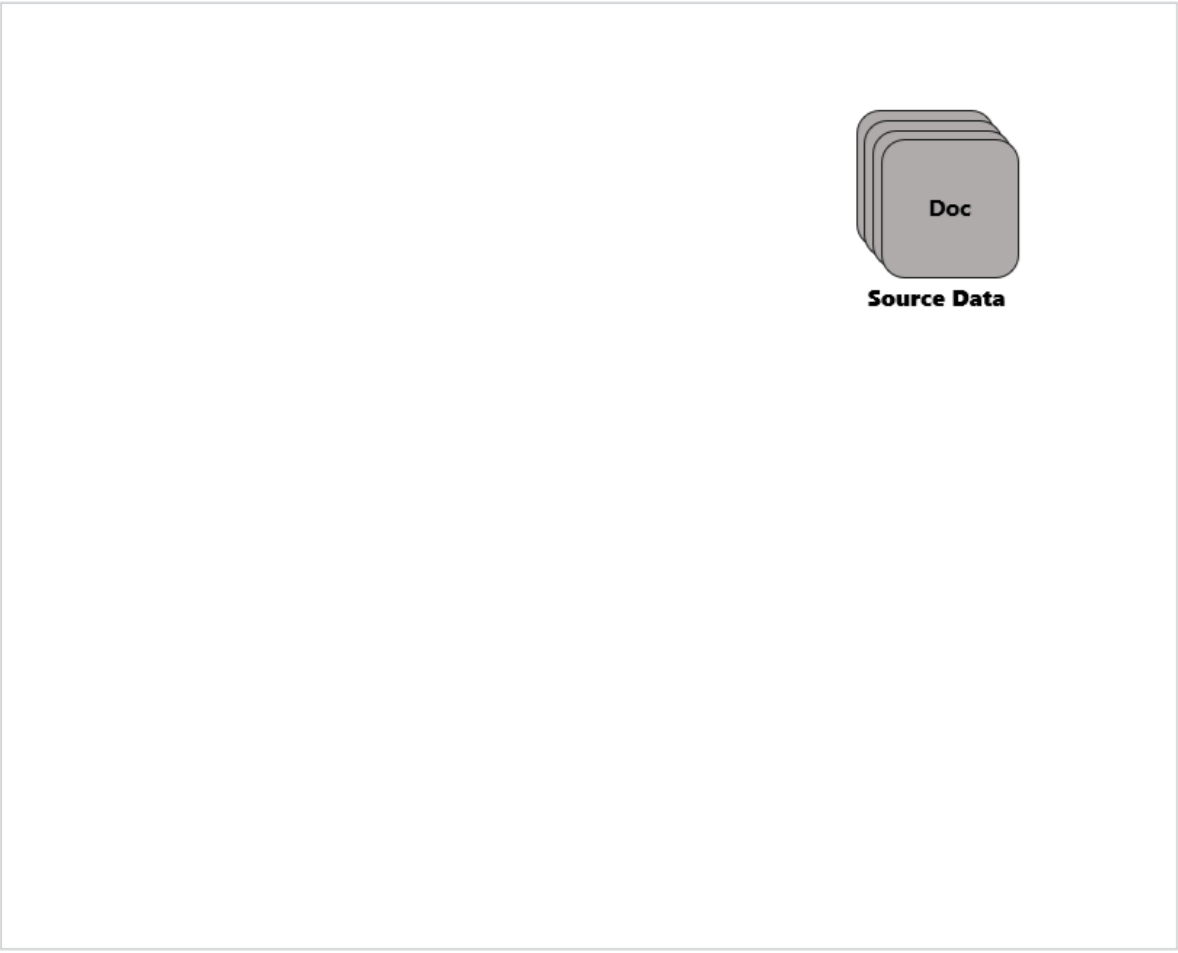
5.3.2 加速流程

工作周期主要的工作量是在两个部分，一是相似样本的挖掘，二是人工标注样本；相似样本挖掘的精度越高，覆盖的样式越全，人工标注的样本量就越少。而这两方面我们可以使用半监督学习跟强化学习的方法来进行优化。



5.3.3 自动化流程

更进一步我们尝试使用强化学习代替整个问题样本的挖掘过程，强化学习选择器从源数据里面选择样本，跟已有标注样本送入判别器训练，然后将判别器在测试数据和问题数据的表现当成 Reward 反馈给强化学习选择器。



六、总结展望

前面讨论了框架内部的通用性，那么这个框架整体是否具有通用性，能否在其他平台适用呢？其实因为各类内容平台在问题类型跟数据类型本身就存在大量的共性；所以看一看的控制框架和技术，在面对如搜一搜，公众平台，朋友圈等其他场景时，依然可以提供相应的经验和能力。近半年来，搜一搜作为用户获取内容重要场景，我们在复用看一看能力的基础上也构建了类似的控制框架。

虽然我们已经掌握了各式各样的能力，能够解决很多问题，但是相比起知道的知识，我们不了解的内容更多，经常会出现让我们意想不到的状况。质量控制，乃至于机器学习的主要矛盾在于用有限的知识处理无限的未知。持续的知识获取意味着自主学习，也意味着持续的与现实交互。

一个平台的质量控制系统就像是人体的免疫组织，当病毒来临，免疫细胞产生抗体，消灭病毒。人类在漫长的进化过程中，构建了复杂的免疫系统，同时，病毒也在不断的进步。越是复杂的系统，出现问题的可能性越高，出现问题后解决问题的成本难度也越大。在构建强健的内部免疫系统的同时，我们同样需要营造良好的外部环境，让更多优质的内容生产者进来，才是解决质量问题之根本。

唯一ID生成算法剖析

作者：cloudoxou，腾讯 WXG 后台开发工程师

在业务开发中，大量场景需要唯一ID来进行标识：用户需要唯一身份标识；商品需要唯一标识；消息需要唯一标识；事件需要唯一标识...等等，都需要全局唯一ID，尤其是分布式场景下。 **常用的分布式唯一ID生成思路** 基本是利用一个长串数字或字符串，将其分割成多个部分，分别记录时间信息、机器/名字信息、随机信息、序列信息等。时间信息部分决定了该策略能使用的时长，机器/名字信息支持了分布式环境下的独自生成唯一ID与识别能力，序列信息保证了事件的顺序记录以及同一时间单位下的并发数，而随机信息则加大了ID整体的不可识别性。

实际上如果现有的方法依然不能满足，我们完全可以依据自身业务和发展需求，来自行决定使用何种策略生成唯一ID。

全文链接：https://mp.weixin.qq.com/s/E3PGP6FDBFUcghYfpe6vsg

比 Bert 体积更小速度更快的 TinyBERT

作者：chenchen&winse

> TinyBERT 是华为不久前提出的一种蒸馏 BERT 的方法，本文梳理了 TinyBERT 的模型结构，探索了其在不同业务上的表现，证明了 TinyBERT 对复杂的语义匹配任务来说是一种行之有效的压缩手段。

一、简介

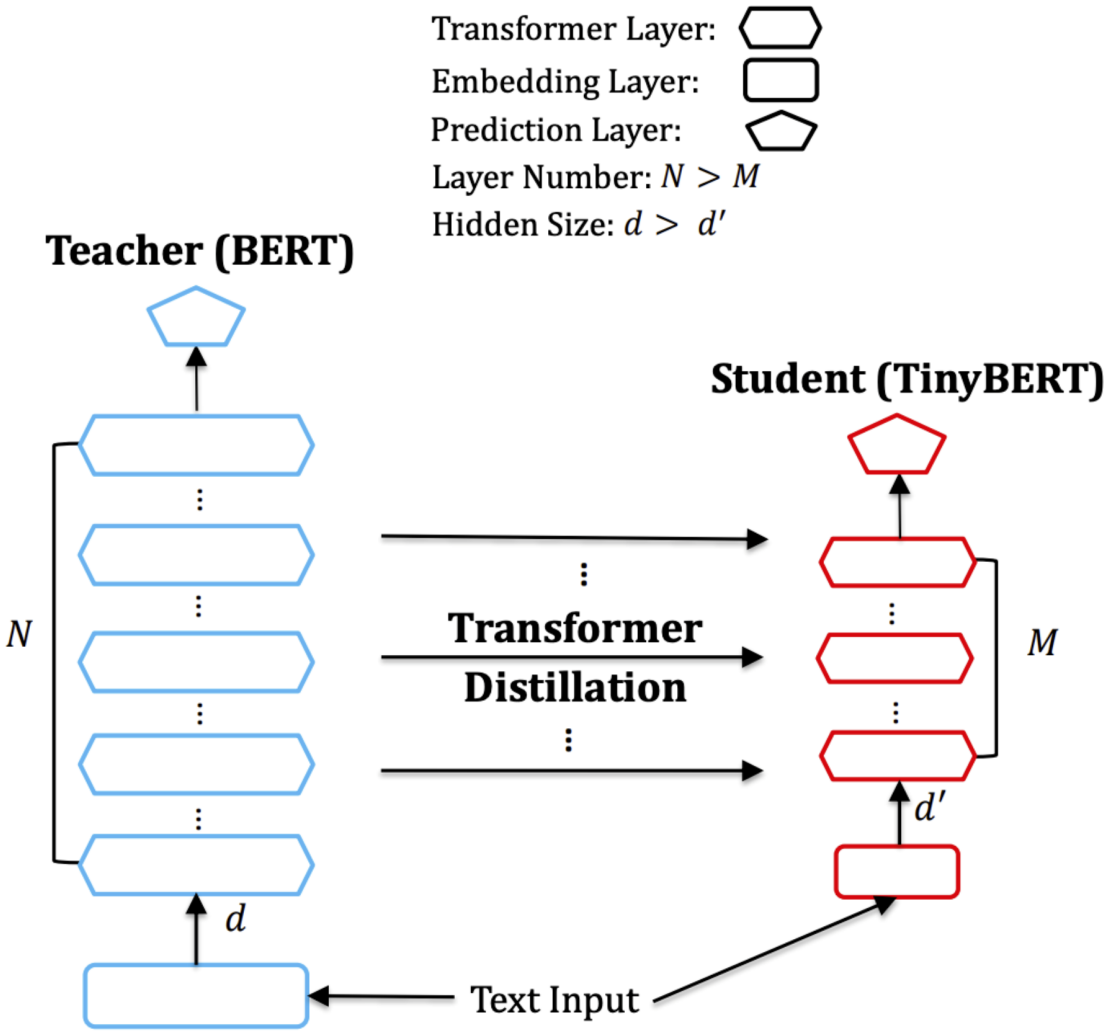
在 NLP 领域，BERT 的强大毫无疑问，但由于模型过于庞大，单个样本计算一次的开销动辄上百毫秒，很难应用到实际生产中。TinyBERT 是华为、华科联合提出的一种为基于 transformer 的模型专门设计的知识蒸馏方法，模型大小不到 BERT 的 1/7，但速度提高了 9 倍，而且性能没有出现明显下降。目前，该论文已经提交机器学习顶会 ICLR 2020。本文复现了 TinyBERT 的结果，证明了 Tiny BERT 在速度提高的同时，对复杂的语义匹配任务，性能没有显著下降。

目前主流的几种蒸馏方法大概分成利用 transformer 结构蒸馏、利用其它简单的结构比如 BiLSTM 等蒸馏。由于 BiLSTM 等结构简单，且一般是用 BERT 最后一层的输出结果进行蒸馏，不能学到 transformer 中间层的信息，对于复杂的语义匹配任务，效果有点不尽人意。

基于 transformer 结构的蒸馏方法目前比较出名的有微软的 BERT-PKD (Patient Knowledge Distillation for BERT)，huggingface 的 DistilBERT，以及本篇文章讲的 TinyBERT。他们的基本思路都是减少 transformer encoding 的层数和 hidden size 大小，实现细节上各有不同，主要差异体现在 loss 的设计上。

二、模型实现细节

TinyBERT 的结构如下图：



整个 TinyBERT 的 loss 设计分为三部分：

1. Embedding-layer Distillation

$$\mathcal{L}_{\text{embd}} = \text{MSE}(\boldsymbol{E}^S \boldsymbol{W}_e, \boldsymbol{E}^T)$$

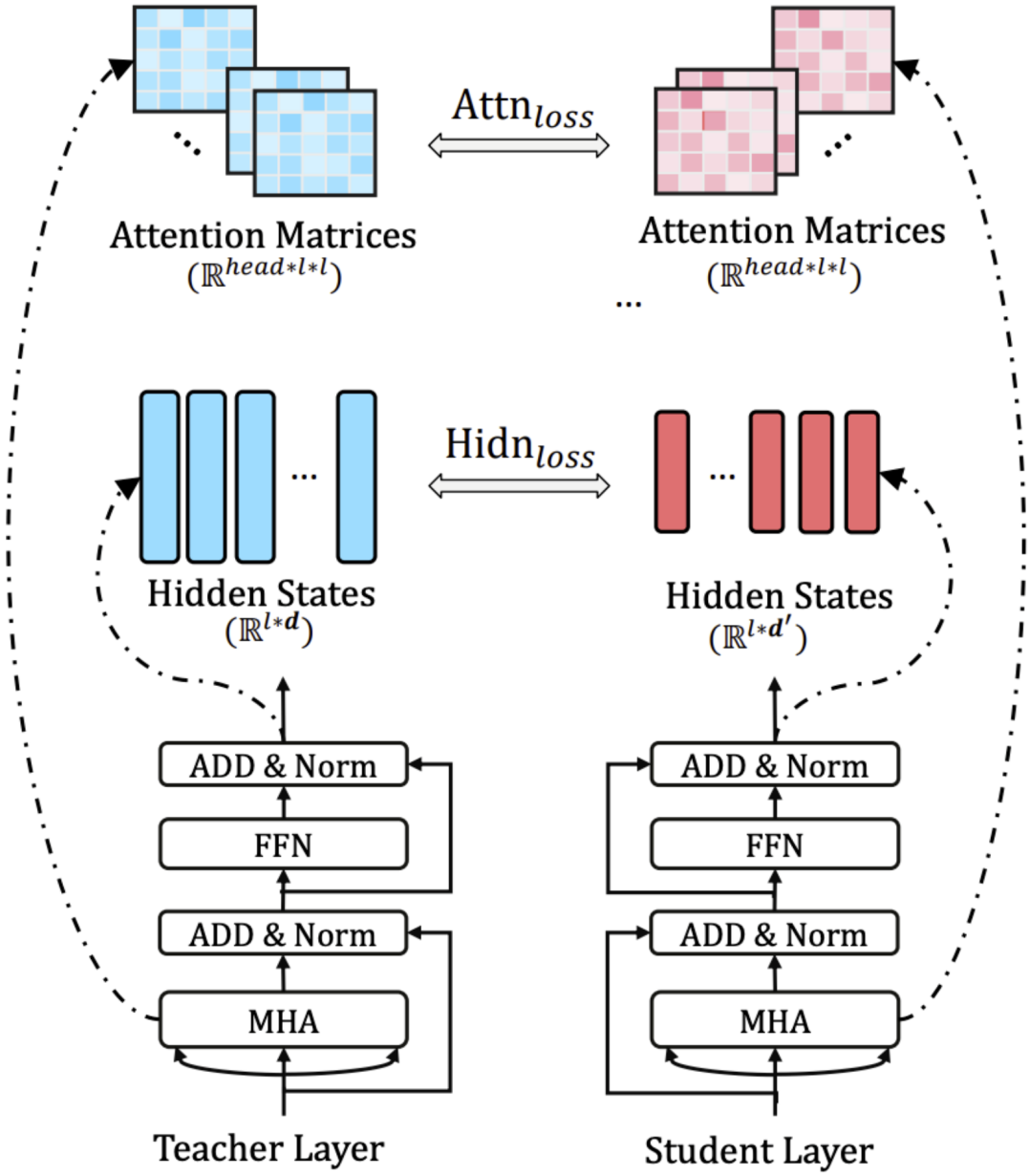
其中:

$$E^S \in R^{l \times d0} \quad , \quad E^T \in R^{l \times d}$$

分别代表 student 网络的 embedding 和 teacher 网络的 embedding. 其中 l 代表 sequence length, d0 代表 student embedding 维度， d 代表 teacher embedding 维度。由于 student 网络的 embedding 层通常较 teacher 会变小以获得更小的模型和加速，所以 We 是一个 d 0×d 维的可训练的线性变换矩阵，把 student 的 embedding 投影到 teacher embedding 所在的空间。最后再算 MSE，得到 embedding loss.

2. Transformer-layer Distillation

TinyBERT 的 transformer 蒸馏采用隔 k 层蒸馏的方式。举个例子，teacher BERT 一共有 12 层，若是设置 student BERT 为 4 层，就是每隔 3 层计算一个 transformer loss. 映射函数为 g(m) = 3 * m, m 为 student encoder 层数。具体对应为 student 第 1 层 transformer 对应 teacher 第 3 层，第 2 层对应第 6 层，第 3 层对应第 9 层，第 4 层对应第 12 层。每一层的 transformer loss 又分为两部分组成，attention based distillation 和 hidden states based distillation.



2.1 Attention based loss

$$\mathcal{L}_{\text{attn}} = \frac{1}{h}\sum_{i=1}^{h}\text{MSE}(\boldsymbol{A}_i^S, \boldsymbol{A}_i^T)$$

其中，

$$A_i \in R^{l\times l}$$

h 代表 attention 的头数，l 代表输入长度，

$$A_i^S$$

代表 student 网络第 i 个 attention 头的 attention score 矩阵，

$$A_i^T$$

代表 teacher 网络第 i 个 attention 头的 attention score 矩阵。这个 loss 是受到斯坦福和 Facebook 联合发表的论文，What Does BERT Look At? An Analysis of BERT's Attention 的启发。这篇论文研究了 attention 权重到底学到了什么，实验发现与语义还有语法相关的词比如第一个动词宾语，第一个介词宾语，以及[CLS], [SEP], 逗号等 token，有很高的注意力权重。为了确保这部分信息能被 student 网络学到，TinyBERT 在 loss 设计中加上了 student 和 teacher 的 attention matrix 的 MSE。这样语言知识可以很好的从 teacher BERT 转移到 student BERT.

2.2 hidden states based distillation

$$\mathcal{L}_{\text{hidn}} = \text{MSE}(\boldsymbol{H}^S\boldsymbol{W}_h, \boldsymbol{H}^T)$$

其中，

$$H^S \in R^{l\times d0} \text{ 和 } H^T \in R^{l\times d}$$

分别是 student transformer 和 teacher transformer 的隐层输出。和 embedding loss 同理，

$$W_h \text{ 把 } H^S$$

投影到 Ht 所在的空间。

3. Prediction-Layer Distillation

$$\mathcal{L}_{\text{pred}} = -\texttt{softmax}(\boldsymbol{z}^T) \cdot \texttt{log_softmax}(\boldsymbol{z}^S / t)$$

其中 t 是 temperature value，暂时设为 1.除了模仿中间层的行为外，这一层用来模拟 teacher 网络在 predict 层的表现。具体来说，这一层计算了 teacher 输出的概率分布和 student 输出的概率分布的 softmax 交叉熵。这一层的实现和具体任务相关，我们的两个实验分别采取了 BERT 原生的 masked language model loss + next sentence loss 和单任务的 classification softmax cross-entropy。

另外，值得一提的是 prediction loss 有很多变化。在 TinyBERT 中，这个 loss 是 teacher BERT 预测的概率和 student BERT 预测概率的 softmax 交叉熵，在 BERT-PKD 模型中，这个 loss 是 teacher BERT 和 student BERT 的交叉熵和 student BERT 和 hard target(one-hot)的交叉熵的加权平均。我们在业务中有试过直接用 hard target loss，效果比使用 teacher student softmax 交叉熵下降 5-6 个点。因为 softmax 比 one-hot 编码了更多概率分布的信息。并且实验中，softmax cross-entropy loss 容易发生不收敛的情况，把 softmax 交叉熵改成 MSE, 收敛效果变好，但泛化效果变差。这是因为使用 softmax cross-entropy 需要学到整个概率分布，更难收敛，因为拟合了 teacher BERT 的概率分布，有更强的泛化性。MSE 对极值敏感，收敛的更快，但泛化效果不如前者。

所以总结一下，loss 的计算公式为:

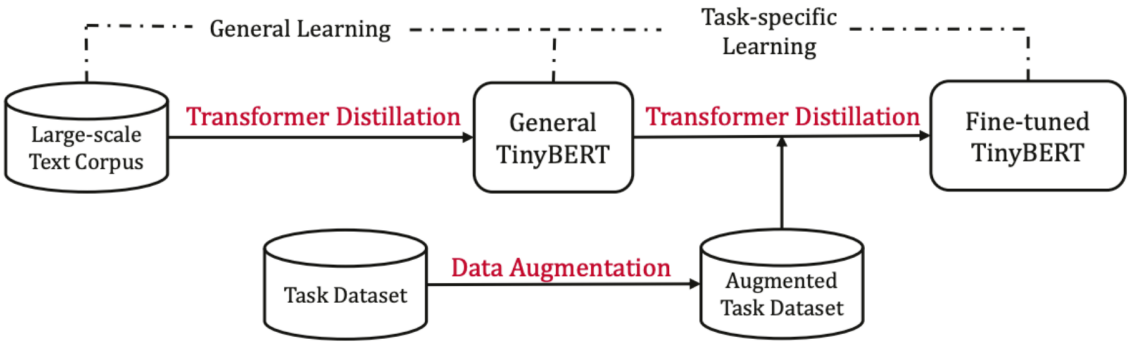
$$\mathcal{L}_{\text{model}} = \sum_{m=0}^{M+1} \lambda_m \mathcal{L}_{\text{layer}}(S_m, T_{g(m)})$$

其中，

$$\mathcal{L}_{\text{layer}}(S_m, T_{g(m)}) = \begin{cases} \mathcal{L}_{\text{embd}}(S_0, T_0), & m = 0 \\ \mathcal{L}_{\text{hidn}}(S_m, T_{g(m)}) + \mathcal{L}_{\text{attn}}(S_m, T_{g(m)}), & M \geq m > 0 \\ \mathcal{L}_{\text{pred}}(S_{M+1}, T_{N+1}), & m = M+1 \end{cases}$$

三、实验

TinyBERT 论文中提出了两阶段学习框架，比较新颖。类似于原生的 BERT 先 pre-train, 根据具体任务再 fine-tine, TinyBERT 先在 general domain 数据集上用未经微调的 BERT 充当教师蒸馏出一个 base 模型，在此基础上，具体任务通过数据增强，利用微调后的 BERT 再进行重新执行蒸馏。



这种两阶段的方法给 TinyBERT 提供了像 BERT 一样的泛化能力。 不过为了快速得到实验结果，并且论文中的控制变量实验显示 general 的蒸馏对各项下游任务的影响较小，我们此次选择直接用 fine-tune 过的 teacher BERT，蒸馏得到 student BERT.

所以我们蒸馏 TinyBERT 的流程是:

1. 制作任务相关数据集;

2. fine-tune teacher BERT;
3. 固定 teacher BERT 参数，蒸馏得到 TinyBERT.

关于实验结果，先上 TinyBERT 论文中的结论：

Table 2: GLUE test results that are scored by the GLUE evaluation server. All models are learned in a single-task manner. "-" means the result is not reported. $\text{BERT}_{\text{SMALL}}$ means directly pretraining a small BERT, which has the same model architecture as TinyBERT, through tasks of *Masked Language Model* (MLM) and *Next Sentence Prediction* (NSP).

System	MNLI-m	MNLI-mm	QQP	SST-2	QNLI	MRPC	RTE	CoLA	STS-B	Average
$\text{BERT}_{\text{BASE}}$ (Google)	84.6	83.4	71.2	93.5	90.5	88.9	66.4	52.1	85.8	79.6
$\text{BERT}_{\text{BASE}}$ (Teacher)	83.9	83.4	71.1	93.4	90.9	87.5	67.0	52.8	85.2	79.5
$\text{BERT}_{\text{SMALL}}$	75.4	74.9	66.5	87.6	84.8	83.2	62.6	19.5	77.1	70.2
Distilled $\text{BiLSTM}_{\text{SOFT}}$	73.0	72.6	68.2	90.7	-	-	-	-	-	-
BERT-PKD	79.9	79.3	70.2	89.4	85.1	82.6	62.3	24.8	79.8	72.6
DistilBERT	78.9	78.0	68.5	91.4	85.2	82.4	54.1	32.8	76.1	71.9
TinyBERT	82.5	81.8	71.3	92.6	87.7	86.4	62.9	43.3	79.9	76.5

Table 3: The model sizes and inference time for baselines and TinyBERT. The number of layers does not include the embedding and prediction layers.

System	Layers	Hidden Size	Feed-forward Size	Model Size	Inference Time
$\text{BERT}_{\text{BASE}}$ (Teacher)	12	768	3072	109M($\times$1.0)	188s($\times$1.0)
Distilled $\text{BiLSTM}_{\text{SOFT}}$	1	300	400	10.1M($\times$10.8)	24.8s($\times$7.6)
BERT-PKD/DistilBERT	4	768	3072	52.2M($\times$2.1)	63.7s($\times$3.0)
TinyBERT	4	312	1200	14.5M($\times$7.5)	19.9s($\times$9.4)

可以看到 TinyBERT 表现优异。在 GLUE 上，相较于完整的 BERT，性能下降 3 个点，但是推理性能却得到了巨大提升，快了 9 倍多。

我们在自己的业务上，也用 TinyBERT 得到了相似的结果。

3.1 文章连贯性特征任务

做这个特征的目的是为了过滤东拼西凑或者机器生成前后没有逻辑的文章。由于语义的复杂性还有语义的转移，这个任务和语义相似度任务略有不同，文章的上下句之间语义会有不同。在这个背景下，实验过 DSSM, Match-Pyramid 等模型，表现效果较差。

由于 BERT 能学到丰富的语义，这个任务目前采用 BERT 的 next sentence 任务较为合适。但是一旦文章很长，原生 BERT 需要算 1 秒甚至更久，这样的速度是不能接受的。

TinyBERT 在不同实验参数下的表现如下：

encoder层数	hidden_size	head数目	intermedia_hidden_size	loss	masked_lm_loss	masked_lm_accuracy	next_sentence_loss	next_sentence_accuracy	变长平均耗时/ms
12	768	12	3072	0.9336338	0.8347494	0.80225	0.098884754	0.96125	94.93
4	312	12	1200	2.4133847	2.2523394	0.56025	0.16104595	0.93875	12.48
3	300	12	1200	4.9365015	4.7348723	0.207	0.20162955	0.9125	8.31
4	768	12	3072	2.897974	2.7335763	0.48	0.16439858	0.93	29.35

可以看到 4 层 encoder 的 TinyBERT 在 next sentence 任务下准确率较 BERT base 准确率下降了不到 3 个点，在 mlm 任务上下降较多。在 CPU 上，TinyBERT 相较于 base 速度获得了将近 8 倍的提升。

3.2 问答 FAQ 任务

业务场景：为用户的 query 匹配最接近的 question，将其 answer 返回，是一个 Query-question 语义匹配任务。

下面是蒸馏到两层 encoder 的 TinyBERT 结果：

		pair	pair	pair	pair	pair	Top1	Top1	Top1	Top1	
	阈值	F1	ACC	AUC	PRE	REC	F1	ACC	PRE	REC	标准
基础bert	0.5	0.781	0.876	0.94	0.842	0.728	0.853	0.855	0.874	0.832	大F1
基础bert	0.63	0.758	0.870	0.94	0.876	0.667	0.844	0.852	0.9	0.795	90准确
tinyBertNoMask1loss2Layer	0.6	0.653	0.794	0.848	0.67	0.637	0.696	0.722	0.693	0.698	大F1
tinyBertNoMask4loss2Layer	**0.5**	**0.724**	0.844	0.905	**0.783**	**0.674**	**0.804**	**0.811**	**0.831**	**0.779**	**大F1**
tinyBertNoMask4loss2Layer	**0.76**	**0.619**	**0.821**	**0.905**	**0.88**	**0.477**	**0.746**	**0.78**	0.902	0.637	**90准确**
XGB-online	0.4	0.663	0.793	0.852	0.656	0.67	0.726	0.747	0.712	0.74	大F1
XGB-online	0.79	0.386	0.761	0.852	0.882	0.247	0.551	0.691	0.901	0.396	90准确

可以看到 ACC 损失 3 个点，AUC 损失 4 个点，取得了不错的效果。

四、总结

我们证明了 TinyBERT 作为一种蒸馏方法，能有效的提取 BERT transformer 结构中丰富的语意信息，在不牺牲性能的情况下，速度能获得 8 到 9 倍的提升。下一步可能会尝试蒸馏一个 general 的 TinyBERT base。

一种全新的点击率建模方案

本文作者： branxu，腾讯 CDG 应用研究员

> 2018 年和 2019 年腾讯算法广告大赛都可以看做推荐系统问题。这类问题最重要的特征是点击率，最大的难点是冷启动。文本结合 2018 年比赛亚军方案和 2019 年比赛冠军方案中的一部分技巧，提出了一种新的点击率建模方案，试图解决一部分冷启动问题。该方案复杂度很低，实现简单，效果好。

问题介绍

推荐系统和广告算法中，对于新用户或者新内容，记录很少，如果我们直接将历史点击率作为特征，会存在问题。比如

1，新用户 A 有 2 条浏览记录，1 次点击，转化率 50%,

2，老用户 B 有 2 条浏览记录，0 次点击，转化率 0%。

A 和 B，只因为 1 次点击，点击率就相差 50%，这不合理。显然，问题出现在 A,B 用户都是新用户，他们的历史数据太少了，历史点击率自然不准。

这就像我告诉同事小明：我王者荣耀贼溜，后羿 100%胜率。实际上，我只打了两盘后羿，其中一盘还是新手教学。同事小明可能会多嘴问一句：你打了几盘后羿啊？但是模型不会，没有专门调整过的模型只会默默接受我后羿 100%胜率的设定，然后给我匹配一堆王者选手。这就是冷启动问题。

解决思路

已经 9102 年了，我们人类星球上的人工智能模型的计算能力还可以，但还是太“老实”，太“傻”。所以，解决上述问题的方法就是：**直接把预测结果告诉模型，别让模型自己去算，去猜。**这显然是句废话，不过翻译成学术语言就不是了：**给模型输入概率，而不是频率。**

所以最好的办法是，利用用户的历史点击率，去计算用户之后点击的概率，再将这个概率输入模型。通过用户 A 的历史点击频率去计算用户 A 之后点击的概率，听起来不错，但又不太可行，因为这里的信息太少了。好在我们有所有数据，**用所有用户的历史点击频率去预测用户 A 之后点击的概率**，似乎有点希望。

贝叶斯平滑

新用户 A 只有两条浏览记录，模型还不够认识用户 A，如何办？如果 A 用户能多几次浏览记录就好了。可是去哪里找那不存在的浏览记录呢？我们可以假定用户 A 和其他所有用户是差不多，用其他用户的历史数据“构造”一些浏览记录，作为新用户 A 的浏览记录。这里“构造”出来的记录，可以理解成先验知识。当我们见过了很多用户之后，即使不认识新用户 A，也会对 A 有个大概的“预期值”。贝叶斯平滑就是这样工作的，它通过“观测”所有用户数据，为新用户确定一个初始预期值，这个预期值就是“先验”。而用户 A 自己真实的行为所产生的“预期值”，被称之为后验。最后我们将先验和后验综合起来，计算一个贝叶斯平滑修正过的点击率。

贝叶斯平滑的推导比较繁杂，也不是本文的重点，有兴趣的话，可以查看：

https://www.jianshu.com/p/c98f3bb48b97

有了贝叶斯平滑，我们可以对点击率进行修正，让历史转化率这个频率值，更加接近用户真实点击的概率。

连续值与深度学习

通过上文，我们可以得到一个贝叶斯平滑后的点击率，那么直接把点击率特征输入深度神经网络，问题不就解决了吗？

只能说，对于大多数普通特征也许可以这样，但是转化率这种强特征，这样做太浪费了。原因如下：

近年来，推荐系统相关的深度网络模型层出不穷：

DeepFM，Wide & Deep，Deep & Cross Network，Attentional Factorization Machine，xDeepFM，Deep Interest Network，AutoInt…

这些模型都有个共同特点：拥有 FM 层或者 Attention 层（Wide & Deep 除外）。FM 层和 Attention 层都能有效进行特征交叉，从而提高了模型精度。FM 层和 Attention 层的输入都是向量，所以这些模型基本都需要让特征先进入嵌入层，变成一个向量，再参与后面的特征自动交叉。

这时候，连续值特征就会很尴尬，他们无法像离散值特征一样进入嵌入层，从而无法参与后面的特征交叉，效果大打折扣。

目前将连续值转化成向量的解决方案主要有以下几种：

第一种方式是对连续值特征做离散化分桶，之后将分桶后的离散值输入到嵌入层到的嵌入向量。分桶本质上就是做四舍五入近似，等距分桶是直接四舍五入，等频分桶是排序后对序做四舍五入，这两种方法会影响精度。因为近似必然会损失信息。

第二种方式也是离散化，不过是有监督的离散化。它借鉴了决策树的思路，枚举所有分割点，找到一组分割点，使分割后的数据组的信息熵增益最大。这里有个比较 trick 的做法：直接用一部分数据训练一个 lightGBM 模型，然后解析模型文件，里面记录了 lightGBM 模型选出来的最优分割点，直接可以用。需要注意的是，有监督的离散化用到了数据的标签，所以可能会带来数据穿越。为了避免这个问

题，建议训练 lightGBM 模型的数据和深度学习的数据不要重合。

第三种方法来自 AutoInt 论文，非常有趣。它先用前面两种方法对连续值特征 Z 做离散化，得到 Z'，之后将 Z'输入嵌入层得到嵌入向量 emd(Z')，最后用嵌入层的输出 emd(Z')再乘以 Z。想法很巧妙：既然离散化后的特征会损失精度，那么就将原始特征再一次输入模型。

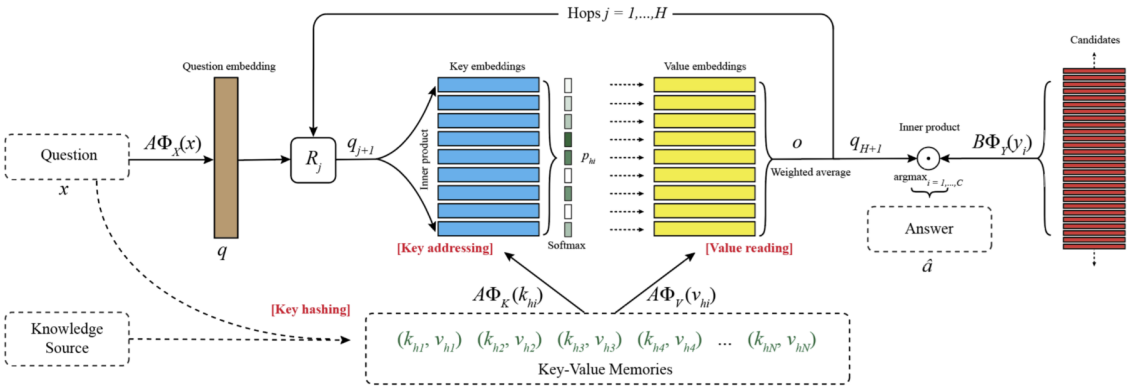
最后一种方法来自 2019 年比赛冠军成员郭大，下文重点介绍。

键值储存网络

该方法灵感来自 NLP 顶会 ACL2016 的论文《Key-Value Memory Networks for Directly Reading Documents》。

文章为深度网络引入了记忆模块，原本是用来解决 QA 问答问题的，不过简单改进后，可以用来将构造续值特征的专属嵌入层。（目前推荐系统很多好的 idea，都来自 NLP 和 CV 领域。所以，学习推荐系统或者广告算法，了解 NLP 和 CV 领域的前沿研究成果也很重要）

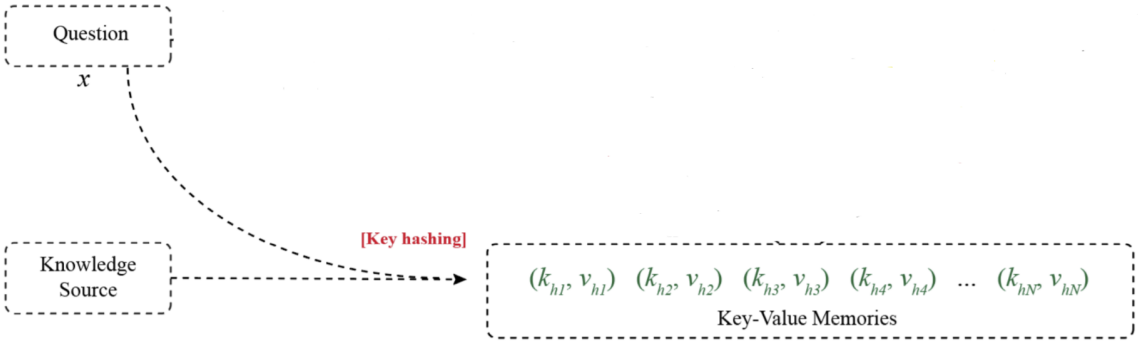
不多说，键值储存网络(Key-Value Memory Networks)结构如下：



键值储存网络与普通网络最大的区别是可以方便的引入先验知识，即图中的 Knowledge Source 模块。该模块相当于一个内嵌在神经网络中的“搜索引擎”，对于输入的任何一个 Question，先在 Knowledge Source 中做一次搜索，然后将搜索结果也作为神经网络的输入。为什么说它是内嵌在神经网络中呢？因为搜索结果与 Question 之间存在一个相似度，这个相似度的计算是依赖神经网络的，它可以享受梯度下降带来的优化。

模型主要分为三个部分：Key hashing,Key addressing 和 Value reading。

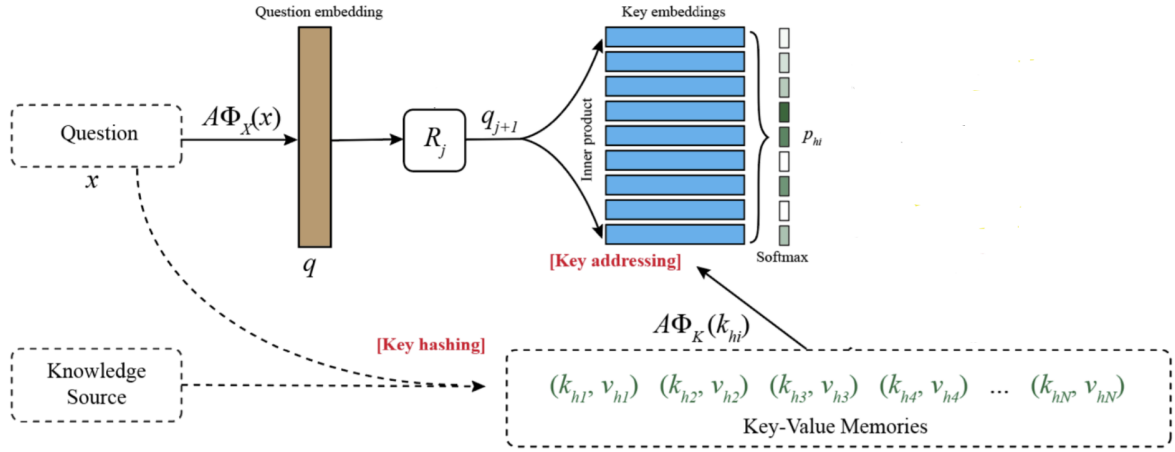
Key hashing



Key hashing 是离线计算好的。它的输入是 Question 和 Knowlege Source。Question 是 QA 问答问题中的提问，比如“如何打开企业微信”。Knowlege Source 是一个类似维基百科的数据库，里面记录了各种词汇，实体和知识。

Key hashing 就是把所有 Question 里面的常用词（出现次数大于某个阈值）挑出来，然后给这些词一个编号，组成一个字典。字典的 key 是这些常用词，value 是常用词编号。

Key addressing

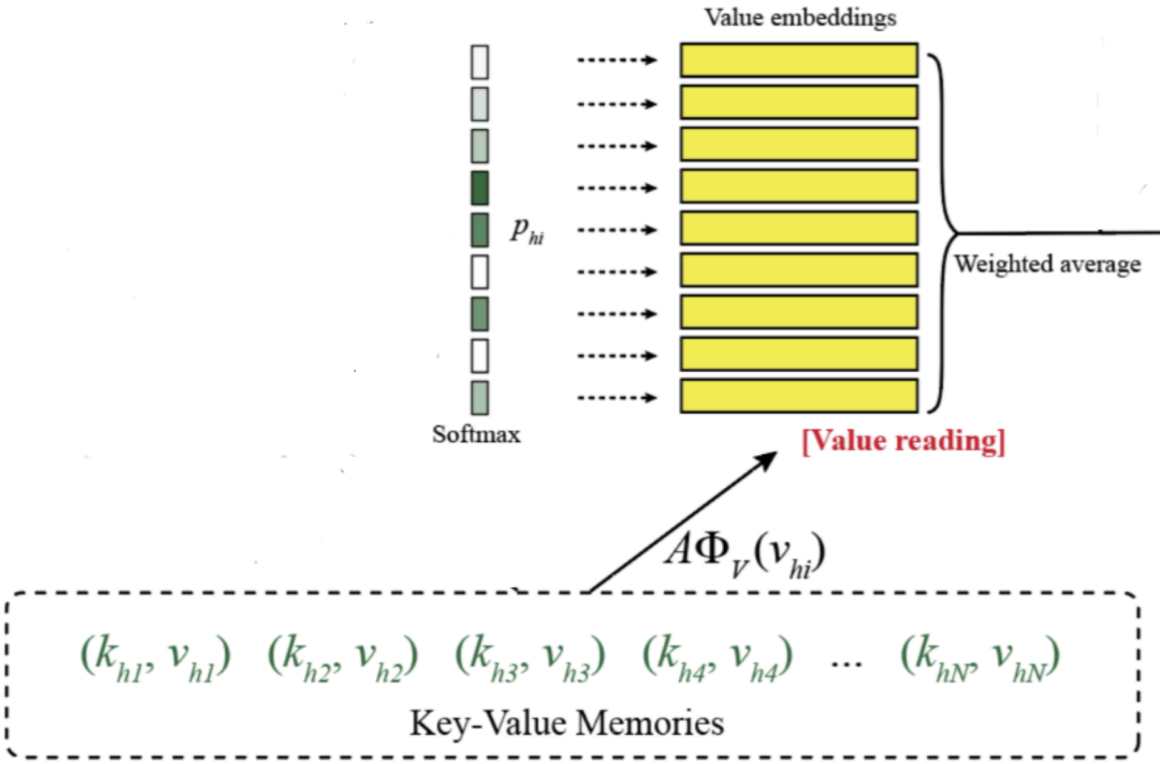


Key addressing 就是去 Knowlege Source 里面寻找 Question 中的词汇和短语。比如找到了："企业","微信","企业微信","如何"等词。Question“如何打开企业微信”会有一个训练出的 Question embedding 值，同时，"企业","微信","企业微信","如何"等词也都有各自的 embedding 值，被称为 Key embeddings。用 Question embedding 分别乘以每一个 Key embeddings，再经过一次 Softmax，就可以得到 Question 与各个 Key 的相似度权重 P。具体公式如下：

$$p_{h_i} = \text{Softmax}\left(A\Phi_X(x) \cdot A\Phi_K(k_{h_i})\right)$$

Key hashing 和 Key addressing 用上述模型解决了一个问题：Question 与 Knowlege Source 中相近词汇的相关性。比如对于 Question"如何打开企业微信"，可以得到一个相似度权重 P 字典{"企业":0.4,"微信":0.2,"企业微信":0.3,"如何":0.1}。

Value reading



value reading 是键值储存网络的核心部分。还记得我们上文有个 key embedding，对应的，Key-Value Memories 还有个 value embedding，它的输入是 Knowlege Source 里面每个词的 id。对 value embedding 以上文的 p 为权重加权求和，便得到我们需要的向量 o。

$$o = \sum_i p_{h_i} A\Phi_V(v_{h_i})$$

优势

和传统的深度神经网络比，键值储存网络可以方便的让先验知识以键值对的方式输入模型(图中的 Key-Value Memories)。这意味着，神经网络的输入值可以直接是多个键值对组成的字典。

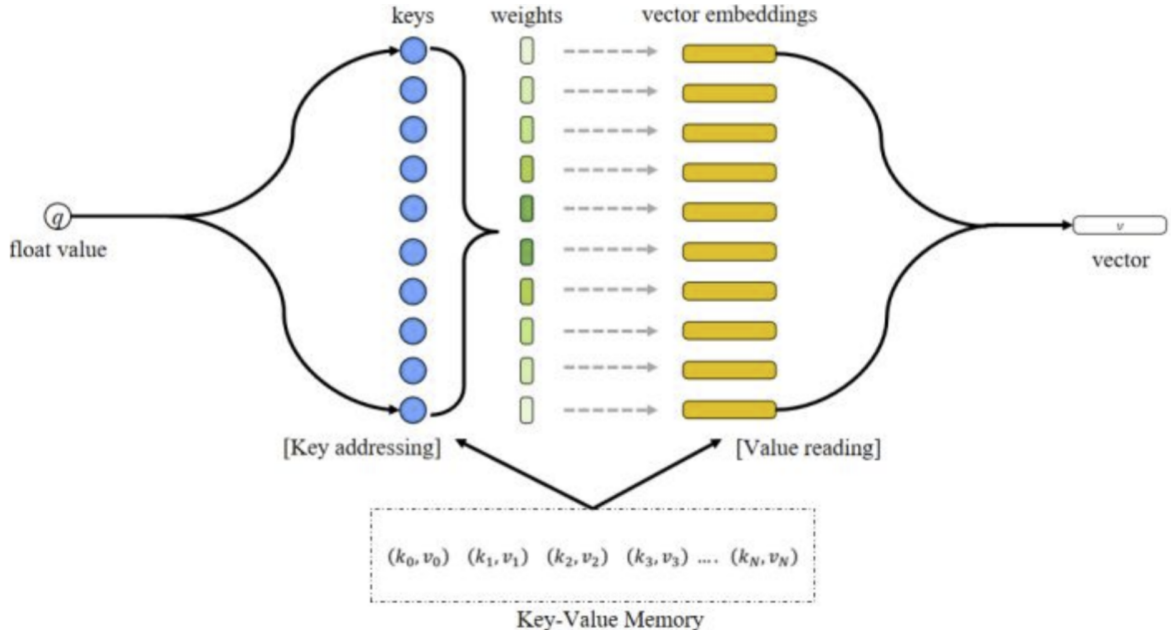
举个例子，传统神经网络只能将数字作为特征，比如:身高(173),体重(90)或者年龄(25)。而键值储存网络可以将兴趣 ({'篮球':0.5,'足球':0.2,'台球':0.1}) 作为特征，直接输入给模型。字典特征的 key 是实体，使用 LabelEncoding 后可以进入嵌入层，value 是其权重。键值储存网络可以方便的将输入的字典特征转化成上文的向量 o。

连续值键值储存网络

回到最开始的问题，我们想找到一个将连续值转成向量的方法，但上文却一直在讲键值储存网络，为什么?

因为键值储存网络实现了字典特征->向量的转换，我们希望的是连续值->向量的转换。所以，借助键值储存网络，只需要再实现连续值->字典特征的转化就大功告成了。连续值->向量很难，但是连续值->字典特征方式很多，易于实现。

假定有了连续值->字典特征的转化，那么总体架构和键值记忆网络基本一致，如下图所示：



连续值->字典特征的转化即图中的 Key-Value Memory,如何实现这部分应当结合具体的业务场景，数据分布。这里先介绍下郭大的做法吧：

1. 将连续值特征缩放至[0,1]区间
2. 在[0,1]区间找 n 等分点，比如 n=6 时，就是(0, 0.2, 0.4, 0.6, 0.8, 1)
3. 依次计算连续值 x 与 n 等分点的距离，比如 x=0.3，n=6，就是(0.3, 0.1, 0.1, 0.3, 0.5,0.7)，之后构造字典特征{0:0.3, 1:0.1, 2:0.1, 3:0.3, 4:0.5, 5:0.7}
4. 对字典特征的 value 取倒数后 softmax，具体相似度公式如下：

$$w_i = softmax(\frac{1}{|q-k_i+10^{-15}|})$$

python 伪代码：{i: softmax(1/(q-i/n+1e-15)) for i in range(n+1)},其中 q 为浮点数特征，n 为等分点个数。这里加上 1e-15 是为了防止 q 正好等于某个等分点时，分母为 0。

郭大的方法将字典特征的 key 定义为[0,1]区间的等分点，之后对浮点数与各等分点的距离做取倒和 softmax 变换。取倒是为了保证浮点数越接近等分点，权重越大。softmax 变换是为了保证所有权重之和为 1。

实践中发现，当 q 与某个等分点较接近时，value 中除该等分点对应的值外，都非常接近 0。这主要是因为 softmax 函数会指数级加大距离间差异。

为了缓解这种情况，我在最近的代码里使用如下相似度公式：

$$w_i = \frac{\left(\frac{1}{q-k_i+10^{-15}}\right)^2}{\sum_{i=1}^{i=n}\left(\frac{1}{q-k_i+10^{-15}}\right)^2}$$

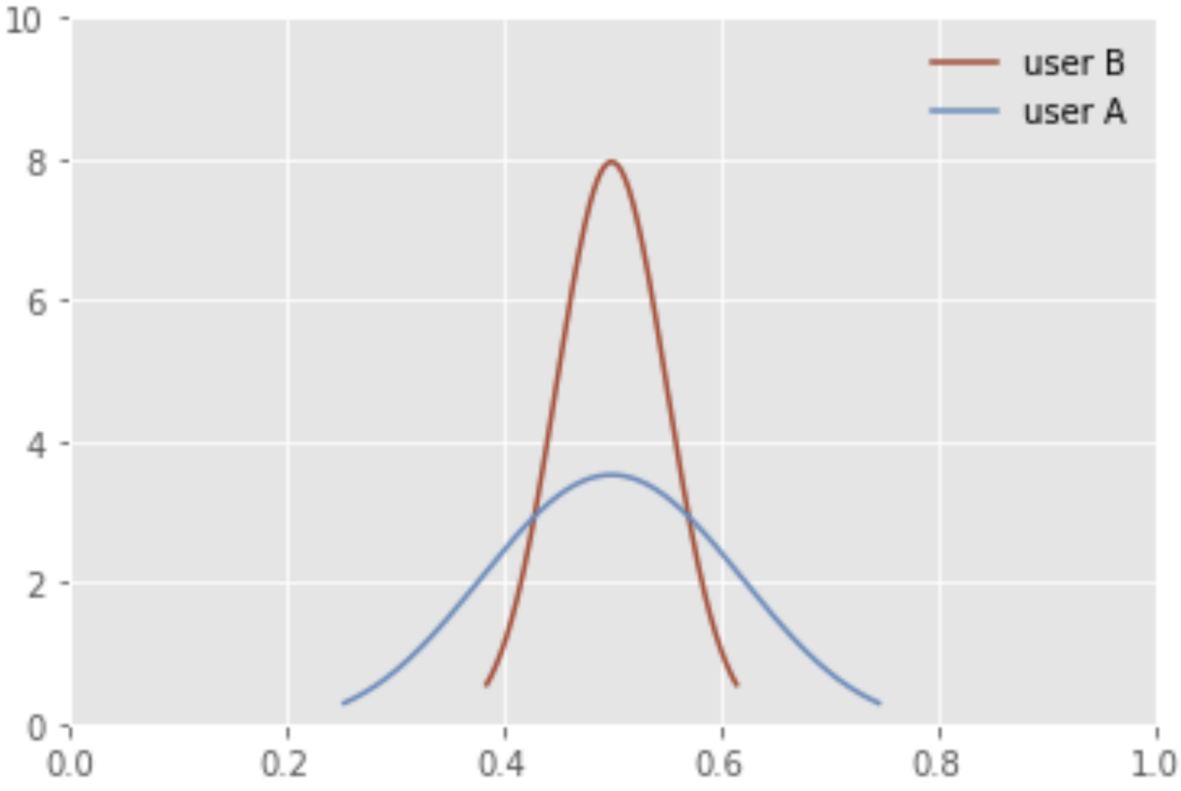
该公式取距离平方反比为权值，之后将权值缩放至总和为 1。用该公式得到的权值比较"分散",可以让模型更好的学习那些冷门分位数的嵌入表示。

概率分布特征

截至目前，文章讲了点击率特征的贝叶斯平滑，以及如何在不损失精度的情况下把浮点数特征（比如点击率特征）输入神经网络。

如果把点击率看成一个普通浮点数，问题已经解决。但是点击率并不普通，点击率可以被认为是用户是否点击广告这个随机变量的期望值。

用户是否点击广告实际上是一个随机变量，点击率就是用这个随机变量的期望值作为特征，去描述它。这样做实际上是用一个值去代表一个复杂的分布，必然会带来信息损失。举个例子，A 用户浏览 20 次，点击 10 次。B 用户浏览 100 次，点击 50 次。A 和 B 的点击率都是 50%，但是他们是否点击广告的概率分布却大不一样：



虽然 AB 两用户点击率都是 50%，但是 B 用户点击次数更多，所以 B 用户的点击率更置信，B 用户的概率分布也更集中。这就体现出点击率特征的弊端，它只能描述概率分布的期望，无法完整描述概率分布。

我们希望完整描述概率分布给模型，我们希望精确区分出点击率很相似但总浏览数差异很大的那群人。这个问题可以被定义为如何向模型描述一个概率分布。

用户是否点击广告的概率分布是连续的，用概率密度函数可以表示。可以对概率密度函数函数进行分段近似，分别统计它在[0,0.1),[0.1,0.2),[0.2,0.3),[0.3,0.4)...区间的平均值，用这些平均值来表示这个分布。形式如下：

{[0,0.1):0.1,[0.1,0.2):0.2,[0.2,0.3):0.4,[0.3,0.4):0.4,...}

该形式其实也是字典特征，它的 key 是区间，value 是点击率这个随机变量落在各区间的概率。如此一来，可以直接将这个字典特征输入键值储存网络。这种方式利用随机变量的概率分布，跳过了连续值->字典特征这一步，直接做随机变量->字典特征，避免了上文中的人工设计相似度公式。

如果构造的特征可以被看做是随机变量，那么就可以利用数学工具得到他的概率分布，概率分布分段近似得到字典特征，最后将字典特征输入键值储存网络。

代码实现与复杂度分析

上文的方法在代码实现上很容易，用途广泛（任何使用了嵌入层的网络都可以用）。

代码主要有四部分：贝叶斯平滑，随机变量->字典特征的转换，浮点数>字典特征的转换和键值储存网络。

相关第三方库

```
import numpy as np
import random
import pandas as pd
import scipy.special as special
from sklearn.model_selection import StratifiedKFold,KFold
from scipy import stats
from collections import OrderedDict, namedtuple
from itertools import chain
from tensorflow.python.keras.initializers import RandomNormal
from tensorflow.python.keras.layers import  Embedding, Input, Flatten
from tensorflow.python.keras.regularizers import l2
```

贝叶斯平滑

```
class BayesianSmoothing(object):
    def __init__(self, alpha, beta):
        self.alpha = alpha
        self.beta = beta

    def sample(self, alpha, beta, num, imp_upperbound):
        sample = np.random.beta(alpha, beta, num)
        I = []
        C = []
        for clk_rt in sample:
            imp = random.random() * imp_upperbound
```

```
            clk = imp * clk_rt
            I.append(int(imp))
            C.append(int(clk))
        return I, C

    def update(self, imps, clks, iter_num, epsilon):
        for i in range(iter_num):
            new_alpha, new_beta = self.__fixed_point_iteration(imps, clks,
self.alpha, self.beta)
            if abs(new_alpha-self.alpha)<epsilon and abs(new_beta-self.beta)
<epsilon:
                break
            self.alpha = new_alpha
            self.beta = new_beta

    def __fixed_point_iteration(self, imps, clks, alpha, beta):
        numerator_alpha = 0.0
        numerator_beta = 0.0
        denominator = 0.0

        for i in range(len(imps)):
            numerator_alpha += (special.digamma(clks[i]+alpha) -
special.digamma(alpha))
            numerator_beta += (special.digamma(imps[i]-clks[i]+beta) -
special.digamma(beta))
            denominator += (special.digamma(imps[i]+alpha+beta) -
special.digamma(alpha+beta))

        return alpha*(numerator_alpha/denominator), beta*
(numerator_beta/denominator)
```

随机变量（beta 分布）->字典特征

```
def beta_ppf(alpha, beta, dim):
    return stats.beta(alpha, beta).ppf([x/(dim+1) for x in range(0,dim+2)])

def beta_prior_feat_2_vec(data, key_col, count_col, sum_col, dim):
    data_simple = data.drop_duplicates([key_col],keep='last')
    bs = BayesianSmoothing(1, 1)
    bs.update(data_simple[count_col].values, data_simple[sum_col].values, 1000,
0.0000000001)
    if np.isnan(bs.alpha) or np.isnan(bs.beta):
        bs.alpha, bs.beta = 0, 0

    data[key_col+'_beta_cdf_value'] = list(
        map(lambda x,y:beta_cdf(x,y,dim), data[sum_col]+bs.alpha,
data[count_col]-data[sum_col]+bs.beta))
    data[key_col + '_beta_ppf_value'] = list(
        map(lambda x,y:beta_ppf(x,y,dim), data[sum_col] + bs.alpha,
data[count_col] - data[sum_col] + bs.beta))
    data[key_col + '_beta_key'] = [np.array([i for i in range(dim)]) for _ in
range(data.shape[0])]
    return data[key_col+'_beta_cdf_value'].values, data[key_col +
'_beta_ppf_value'].values, data[key_col + '_beta_key'].values
```

浮点数->字典特征

```
def numpy_softmax(x):
    orig_shape = x.shape
    if len(x.shape) > 1:
        exp_minmax = lambda x: np.exp(x - np.max(x))
        denom = lambda x: 1.0 / np.sum(x)
        x = np.apply_along_axis(exp_minmax,1,x)
        denominator = np.apply_along_axis(denom,1,x)
        if len(denominator.shape) == 1:
            denominator = denominator.reshape((denominator.shape[0],1))
        x = x * denominator
    else:
        x_max = np.max(x)
        x = x - x_max
        numerator = np.exp(x)
        denominator =  1.0 / np.sum(numerator)
        x = numerator.dot(denominator)
    assert x.shape == orig_shape
    return x

def float2vec(float_feat, bar_num = 20, method = 'gravitation'):
    float_feat = (float_feat-np.min(float_feat))*1.0 / np.max(float_feat-
np.min(float_feat))
    key_array = np.array([[i*1.0/(bar_num + 1) for i in range(bar_num + 1)]] *
len(float_feat))
    value_array = None
    if method == 'gravitation':
        value_array = 1/(np.abs(key_array - float_feat[:,None] + 0.00001))**2
        value_array = value_array/np.sum(value_array,axis=1, keepdims=True)
    if method == 'sofmax':
        value_array = 1 / np.abs(key_array - float_feat[:, None] + 0.00001)
        value_array = numpy_softmax(value_array)
    return key_array,value_array
```

网络结构

```
def get_varlen_multiply_list(embedding_dict, features,
varlen_sparse_feature_columns_name_dict):
    multiply_vec_list = []
    print(embedding_dict)
    for key_feature in varlen_sparse_feature_columns_name_dict:
        for value_feature in
varlen_sparse_feature_columns_name_dict[key_feature]:
            key_feature_length_name = key_feature.name + '_seq_length'
            if isinstance(value_feature, VarLenSparseFeat):
                value_input = embedding_dict[value_feature.name]
            elif isinstance(value_feature, DenseFeat):
                value_input = features[value_feature.name]
            else:
```

```
                raise TypeError("Invalid feature column
type,got",type(value_feature))
            if key_feature_length_name in features:
                varlen_vec = SequenceMultiplyLayer(supports_masking=False)(
                    [embedding_dict[key_feature.name],
features[key_feature_length_name], value_input])
                vec = SequencePoolingLayer('sum', supports_masking=False)(
                    [varlen_vec, features[key_feature_length_name]])
            else:
                varlen_vec = SequenceMultiplyLayer(supports_masking=True)(
                    [embedding_dict[key_feature.name], value_input])
                vec = SequencePoolingLayer('sum', supports_masking=True)(
varlen_vec)
            multiply_vec_list.append(vec)
    return multiply_vec_list

class SequenceMultiplyLayer(Layer):

    def __init__(self, supports_masking, **kwargs):
        super(SequenceMultiplyLayer, self).__init__(**kwargs)
        self.supports_masking = supports_masking

    def build(self, input_shape):
        if not self.supports_masking:
            self.seq_len_max = int(input_shape[0][1])
        super(SequenceMultiplyLayer, self).build(
            input_shape)  # Be sure to call this somewhere!

    def call(self, input_list, mask=None, **kwargs):
        if self.supports_masking:
            if mask is None:
                raise ValueError(
                    "When supports_masking=True,input must support masking")
            key_input, value_input = input_list
            mask = tf.cast(mask[0], tf.float32)
            mask = tf.expand_dims(mask, axis=2)
        else:
            key_input, key_length_input, value_input = input_list
            mask = tf.sequence_mask(key_length_input,
                                    self.seq_len_max, dtype=tf.float32)
            mask = tf.transpose(mask, (0, 2, 1))

        embedding_size = key_input.shape[-1]
        mask = tf.tile(mask, [1, 1, embedding_size])
        key_input *= mask
        if len(tf.shape(value_input)) == 2:
            value_input = tf.expand_dims(value_input, axis=2)
            value_input = tf.tile(value_input, [1, 1, embedding_size])
        return tf.multiply(key_input,value_input)

    def compute_output_shape(self, input_shape):
        return input_shape[0]

    def compute_mask(self, inputs, mask):
        if self.supports_masking:
            return mask[0]
        else:
            return None
```

```
    def get_config(self, ):
        config = {'supports_masking': self.supports_masking}
        base_config = super(SequenceMultiplyLayer, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))
```

上述代码除贝叶斯平滑转载自网络外，均为原创。

改进后的键值网络与连续值离散化后接入嵌入层的方法相比，没有增加训练参数，只是多做了一次向量加权求和，多增加了一些权重的输入。另一方面，改进后的键值网络中，分位数或者概率区间个数是可以人工调整的，当分位数或者概率区间个数为 1 时，该方法就退化成离散化后接入嵌入层。

一文了解推荐领域最新工作

本文作者：腾讯微信支付算法研究员

介绍

工业界通用推荐系统包括两个阶段的流程，匹配（match）和排序（rank）。在匹配的过程中主要是根据用户的一些兴趣点，找到可能感兴趣的潜在商品集。由于整个商品集的海量性，对实时访问的用户去计算他对全部商品的感兴趣程度是不实际的，所以需要预先根据一些兴趣、特征策略等来寻找用户可能感兴趣的潜在商品集，在此基础上，再根据特定的模型算法来进行商品集兴趣分深层排序，效果指标往往通过点击率（Click Trough Rate）、转化率、时长等来量化，所以 rank 阶段的主要目的就在于预测一个用户在他感兴趣的商品中的 CTR 等，并且根据预测分值的大小进行排序，最终返回排序作为推荐系统的推荐结果。

本文将介绍三篇重点分别在 ranking，matching 和 feature enginerring 上的文章，借此来展示业界推荐系统方面的一些最新工作进展。

1. Ranking:

Behavior Sequence Transformer for E-commerce Recommendation in Alibaba

这篇文章主要关注在 rank 阶段的改进。以往在预测 CTR 场景下常见的深度学习方法是基于嵌入层与多层前馈神经网络的结合，将大量的低阶原始特征通过简单地嵌入层进行降维，然后降维后的低维特征再作为全连接网络的输入，最终通过网络输出对 CTR 点击率的概率预测（由于输出结果是一个点击概率，所以取值范围落在区间[0,1]上）。具有代表性的工作参考 WDL(wide and deep learning by Google)和 DIN(Deep Interest Neworks by Alibaba)。

但是上述方法存在一个比较明显的问题，直接输入原始特征实际上缺失了很多高阶的特征信息，比如高阶的交互项、序列信息等。这类特征在推荐场景下往往比较重要，比如年龄特征与性别特征的交互项可以刻画不同年龄层与不同性别的购买需求差异性。另一点，一个客户在购买了手机后很有可能会继续浏览手机配件等商品，就需要一些行为序列特征去捕捉这些信息。因此，从这些实例中可以获知，高阶的特征在进行点击率预测的任务十分重要。对比前面提到的两种方法，WDL 仅仅是简单地拼接 raw features，DIN 考虑了通过注意力机制来刻画商品与用户的历史兴趣的多峰表征，二者都没有考虑行为的序列特征。

为了解决这方面的问题，作者参考了在 NLP 领域机器翻译任务 transformer 模块。由于在机器翻译中，语言的序列信息是十分关键的，而 transformer 可以取得很好的效果，所以认为如果把客户的购买历史看做是一串文字，那么每种商品之间的序列相关性的信息也可以通过 transformer 捕捉到。

该论文整体网络结构如下，主要包含三个部分：嵌入层，转换层，前馈网络层。

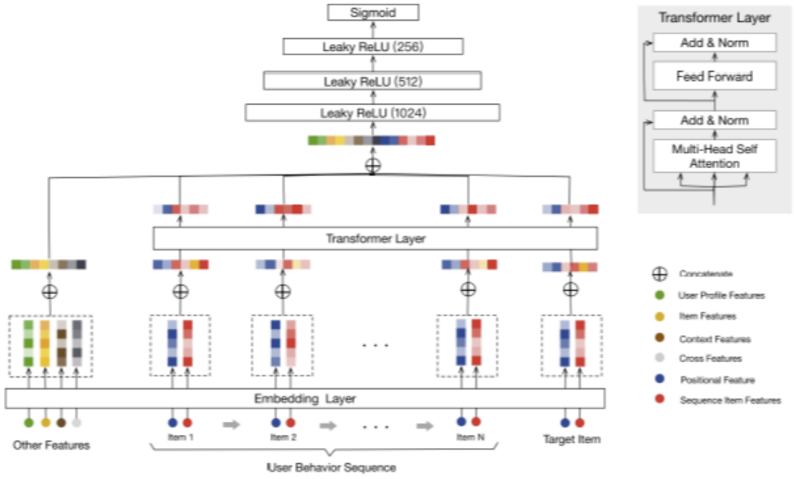


Figure 1: The overview architecture of the proposed BST. BST takes as input the user's behavior sequence, including the target item, and "Other Features". It firstly embeds these input features as low-dimensional vectors. To better capture the relations among the items in the behavior sequence, the transformer layer is used to learn deeper representation for each item in the sequence. Then by concatenating the embeddings of Other Features and the output of the transformer layer, the three-layer MLPs are used to learn the interactions of the hidden features, and sigmoid function is used to generate the final output. Note that the "Positional Features" are incorporated into "Sequence Item Features".

嵌入层的主要目的仅仅在于将高维的特征映射到相对低维的空间中，通过一个嵌入矩阵$W_o\in \mathbb{R}^{|D| \times d_o}$，其中$d_o$就是嵌入空间的维度。需要输入到嵌入层的特征主要有一下几个部分：包含序列信息的 item 特征和其他特征。其他特征的内容如下表格所示：

User	Item	Context	Cross
gender	category_id	match_type	age * item_id
age	shop_id	display position	os * item_id
city	tag	page No.	gender * category_id
...	...	...	...

序列信息的 item 特征作者又分为了两个部分，分别是 Sequence Item Features 和 Position Features。其中 Sequence Item Features 主要是商品的 item_id 和 category_id 这两个可以表征商品信息的特征，原因是考虑到提取序列信息的计算复杂性，输入全部的 item 特征效率很低。Position Feature 实际上是通过 postional embedding 的变换来获取的，在"Attention is all your need"中有提及。本文中考虑一个客户的浏览历史，时间间隔最能够体现浏览历史中的位置信息。

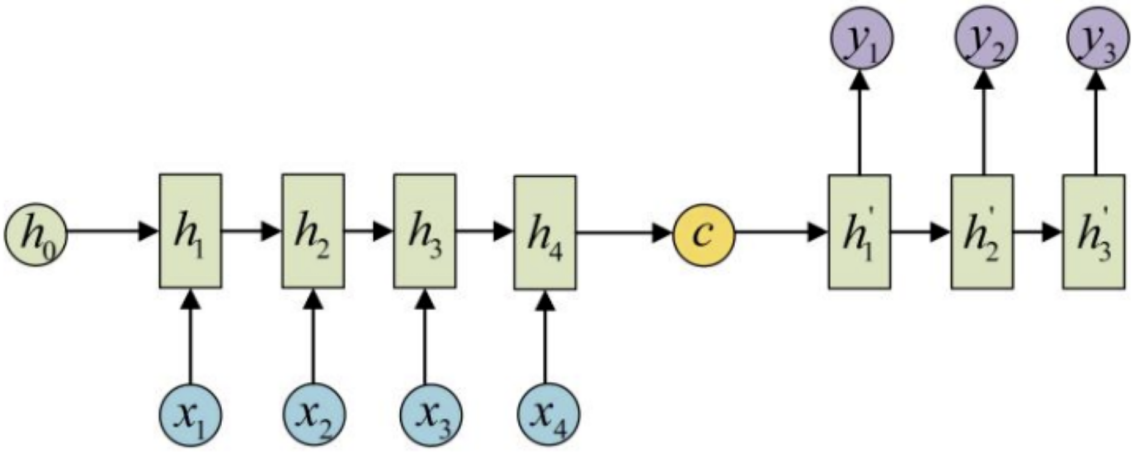
因此本文定义如下 position embedding：pos(v_i) = t(v_t) - t(v_i),其中 t(v_t) 代表的是推荐系统的推荐时间点，t(v_i)代表用户点击第 i 件商品的时间点。构造序列特征之后，通过$W_V \in \mathbb{R}^{|V|\times d_V}$进行序列特征的嵌入，$|V|$代表浏览历史的长度，$|d_v|$是嵌入空间的维度。

转换层：

Multi-Head Self-Attention(多头注意力模型)是 Transformer 的核心单元，理解 MH Attention 机制对于 Transformer 层原理掌握特别重要，并且在 Encoder 和 Decoder 中都有用到。MH Self-attention 最新应用可以替代一些基于 RNN 行为序列模型在推荐场景落地。另外在下文即将介绍的 AutoInt 中也是引入了 Multi-head Self-attention 核心思想，所以这里花一定篇幅对此进行原理讲解。

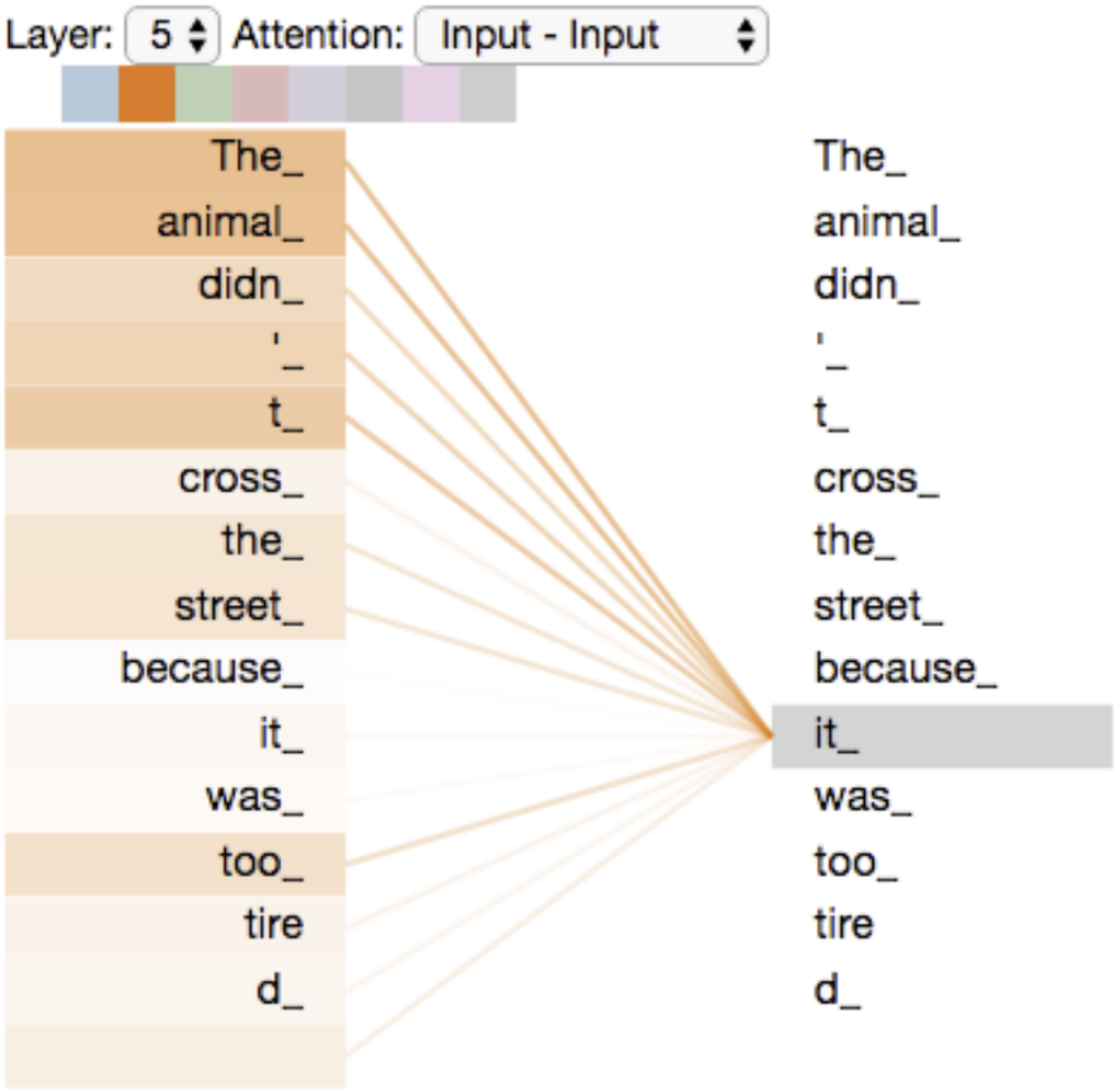
先从一个机器翻译的小例子来看：
"The animal didn't cross the street because it was too tired"
单词"it"表示什么呢，是"animal"还是"street"，对机器算法来说识别这个不是一件简单的事情，算法在处理每个词的时候需要知道上下文信息。

较早一代 NLP 算法中 RNN、LSTM、SRU 等序列模型可以处理这种场景，基本做法是将句子分词，然后每个词转化为对应的词向量序列，经过 RNN 算法来处理词序列信息，在翻译这种场景仅用 RNN 很难做到高水准，这样出现了 RNN 的变种：Encoder-Decoder，也叫 Seq2Seq。如下图所示，在 Encoder 阶段，输入数据编码成一个上下文语义向量 c（语义向量 c 可以有多种表达方式，最简单的方法就是把 Encoder 的最后一个隐状态赋值给 c），Encoder 是一个 RNN。Decoder 是同样的 RNN（也可以不一样的结构），拿到 c 之后，对其进行解码操作，c 当做之前的初始状态 h0 输入到 Decoder 中，每一次得到一个最有可能的翻译结果，然后让所有单词的 cross entropy 达到最小。

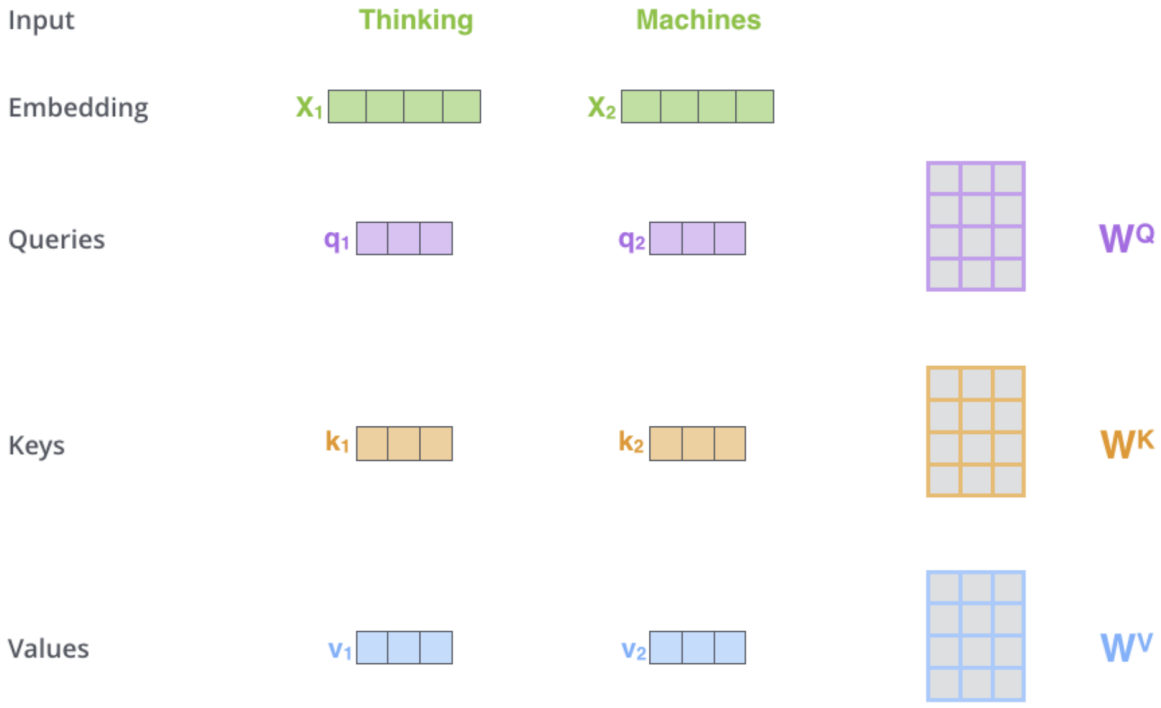


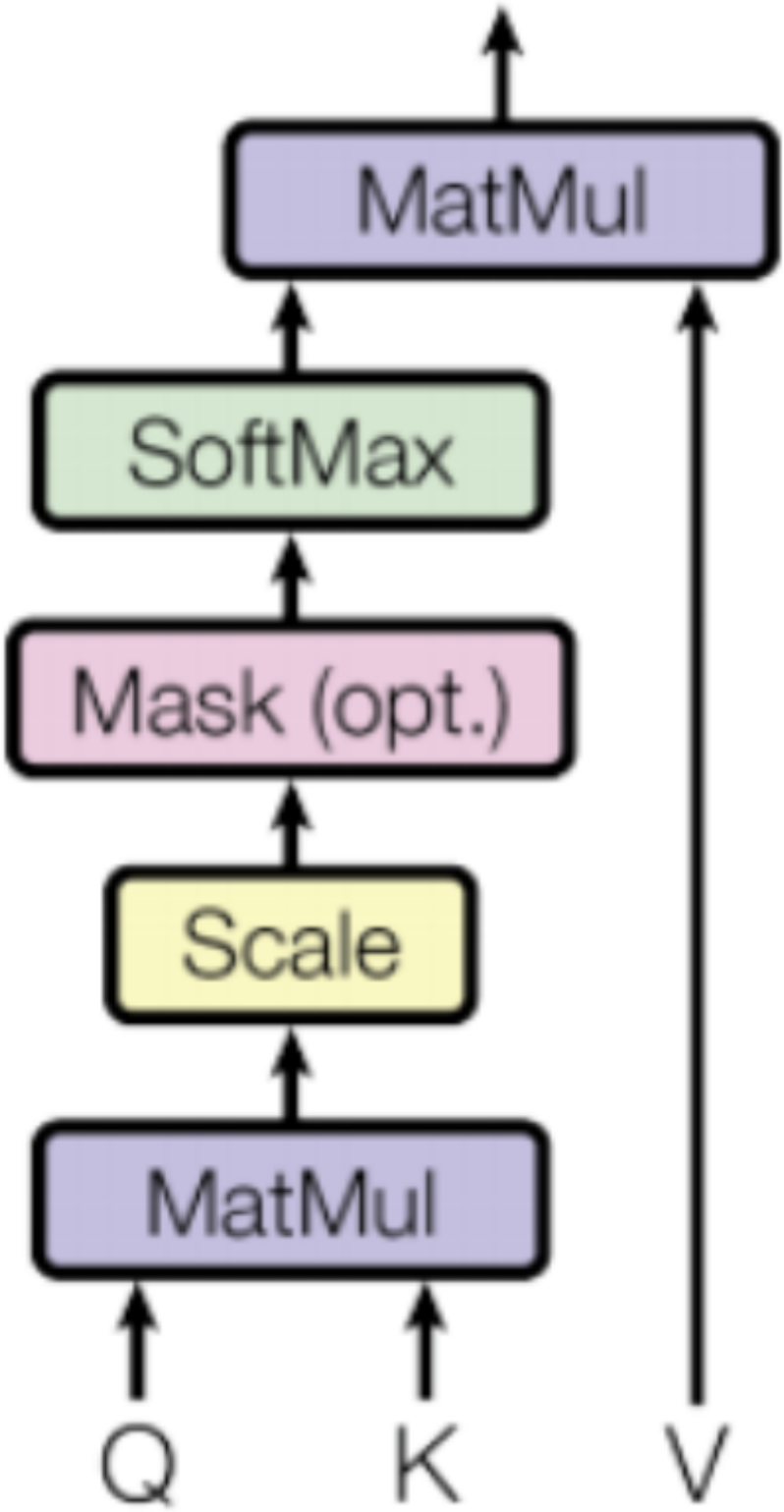
RNN、LSTM 这种类型算法自然可以很好的处理序列信息，结构相对比较简单，本质上都是递归处理结构，缺点是无法做到并行，训练速度比较慢，另外它也是一种马尔科夫决策过程，无法很好的学习全局结构信息。那有没有改进的算法呢？Attention,对"Attention is All you need"，这也是谷歌 17 年的一篇经典论文，后面在此基础上也诞生了各种优秀论文。
接着看上面翻译的小例子，这里的"it"表示"animal"还是"street"呢，联系上下文，就知道 it 很大概率指的是 animal。

下图直观地展示了 self-attention 机制，计算每个单词与其他单词之间的关联，这里用 attention score 来表示关联度，处理"it"时"the"、"animal"就有比较高的 attention score。这些 score 在 self-attention 中就是权重的概念，对输入 vector 加权然后喂入前馈神经网络中，得到新的表示，这样可以很好的获取上下文信息。



Self-attention layer：一般的注意力层定义如下：





attention 计算：$Attention(Q, K, V) = softmax(\frac{QK}{\sqrt{d}})V$Q 是查询，K 是键，V 是数值。光从上面图中理解起来比较抽象，结合原论文以及代码，就比较容易理解了，先看看核心部分代码：

```
def scaled_dot_product_attention(q, k, v, mask):
  """
  参数:
    q: 请求的形状 == (..., seq_len_q, depth)
    k: 主键的形状 == (..., seq_len_k, depth)
    v: 数值的形状 == (..., seq_len_v, depth_v)
          (..., seq_len_q, seq_len_k)。默认为None。
  返回值:
    输出，注意力权重
```

```
  """
  matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_k)
  # 缩放 matmul_qk
  dk = tf.cast(tf.shape(k)[-1], tf.float32)
  scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

  # softmax 在最后一个轴（seq_len_k）上归一化，因此分数
  # 相加等于1。
  attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)  # (...,
seq_len_q, seq_len_k)
  output = tf.matmul(attention_weights, v)  # (..., seq_len_q, depth_v)
  return output, attention_weights

class MultiHeadAttention(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads):
    super(MultiHeadAttention, self).__init__()
    self.num_heads = num_heads
    self.d_model = d_model
    assert d_model % self.num_heads == 0

    self.depth = d_model // self.num_heads
    self.wq = tf.keras.layers.Dense(d_model)
    self.wk = tf.keras.layers.Dense(d_model)
    self.wv = tf.keras.layers.Dense(d_model)
    self.dense = tf.keras.layers.Dense(d_model)

  def split_heads(self, x, batch_size):
    """分拆最后一个维度到 (num_heads, depth).
    转置结果使得形状为 (batch_size, num_heads, seq_len, depth)
    """
    x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(x, perm=[0, 2, 1, 3])

  def call(self, v, k, q, mask):
    batch_size = tf.shape(q)[0]
    q = self.wq(q)  # (batch_size, seq_len, d_model)
    k = self.wk(k)  # (batch_size, seq_len, d_model)
    v = self.wv(v)  # (batch_size, seq_len, d_model)
    q = self.split_heads(q, batch_size)  # (batch_size, num_heads, seq_len_q,
depth)
    k = self.split_heads(k, batch_size)  # (batch_size, num_heads, seq_len_k,
depth)
    v = self.split_heads(v, batch_size)  # (batch_size, num_heads, seq_len_v,
depth)
    # scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
    # attention_weights.shape == (batch_size, num_heads, seq_len_q, seq_len_k)
    scaled_attention, attention_weights = scaled_dot_product_attention(
        q, k, v, mask)

    scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])  #
(batch_size, seq_len_q, num_heads, depth)

    concat_attention = tf.reshape(scaled_attention,
                                  (batch_size, -1, self.d_model))  #
(batch_size, seq_len_q, d_model)

    output = self.dense(concat_attention)  # (batch_size, seq_len_q, d_model)
```

```
    return output, attention_weights
```

代码是 Multi-head attention 类的构建方式，下面会提到 Multi-head attention 是什么，这里先介绍 self-attention 中的 attention score 怎么计算的，Q、K、V 怎么来的。其实是从输入 embeding 经过线性层变换得到了，有三个权重矩阵，WQ、WK、WV，输入 X 与这三个权重矩阵相乘得到，其实表示的意义大致相同，只是生成了不同的表达形式，也是神经网络里面一些思想，很难去说清楚为什么要这样做。WQ、WK、WV 这三个权重矩阵其实也是模型需要训练的参数矩阵。Attention score 生成大概可以分为以下几个步骤：

1、输入词生成 embedding 向量，经过线性层变换得到对应 Q、K、V 向量。

2、计算输入词 Query Vector 与其他相关词 Key Vector 点积，再对 score 进行归一化，再通过 softmax 来获取权重（attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)）。

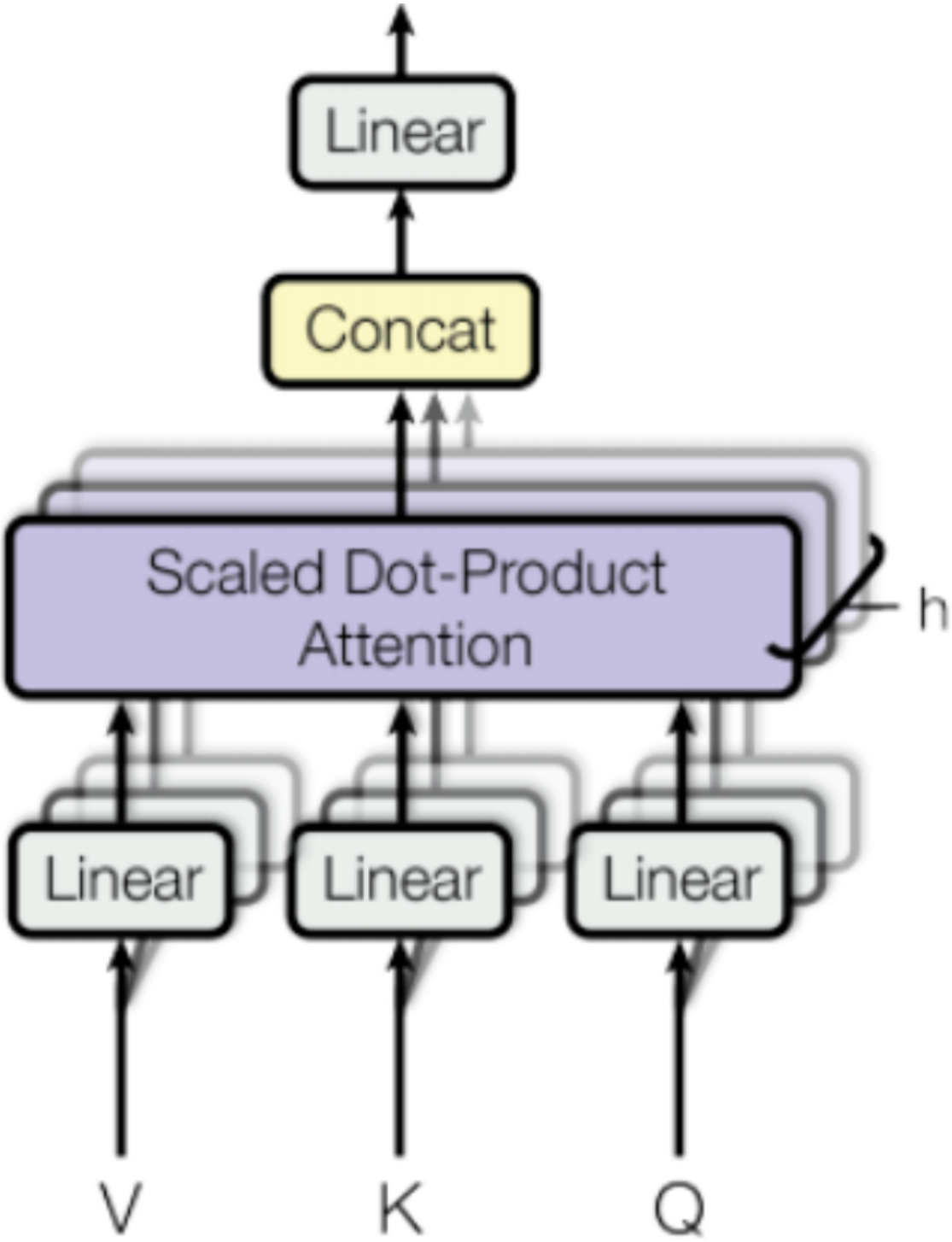
3、将 softmax 结果与 Value Vector 相乘得到加权的权重向量 attention score(output = tf.matmul(attention_weights, v))

投影矩阵$W^Q, W^K, W^V \in \mathbb{R}^{d\times d}$，$E$是一个共用的嵌入矩阵。再结合一层全连接神经网络$FFN(\cdot)$的映射来加强非线性形式，最终的 self_attention 模块的形式为：

$S' = LayerNorm(S + Dropout(MH(S))),$

$F = LayerNorm(S' + Dropout(LeakyReLU(S'W^{(1)} + b^{(1)})W^{(2)}+b^{(2)}))$

再来看看 multi-head attention：



$S = MH(E) = Concat(head_1,...,head_h)W^H$

简单来说，multi-head attention 就是多个 self-attention 的集成，为什么要这么做呢，论文中说是将模型分为多个头，形成多个子空间，这样可以让模型从不同角度去学习多方面不同的信息，再去融合，具体是不是真的可以让模型学习"不同子空间的特征"，这里还没有深入作研究。其实通过增加模型 layer 是不是也可以学习到更多信息，当然相对 multi-head 来说性能肯定比不上，因为 multi-head 是并行的，这也是论文中提到的一个优势，那论文里面提到的 multi-head 其他优点可以增加网络空间，怎么去考量，其实也存在一定玄学在里面。

前馈网络层：

将其他特征与转换器的输出拼接后输入三层前馈神经网络，目的是为了处理其他特征的有效信息。由于 CTR 问题是一个预测点击概率的问题，所以作者选择一个 sigmoid 函数作为输出层，损失函数的选择也是标准的交叉熵损失：
$L = -\frac{1}{N} \sum_{(x,y) \in \mathcal{D}}(ylog p(x) + (1-y)log(1 - p(x))$

模型评估：

本文选择 WDL 和 DIN 作为对比模型，构建了一个 WDL+seq 的基准模型，基准模型的序列信息是通过简单地将历史信息直接进行嵌入得到的。本文的网络结构和 WDL 的区别在于转换层的搭建，与 DIN 区别则是在处理序列信息上思路不同。

作者选择 AUC 来作为模型评估的度量，并且也考量了 A/B 测试的 CTR 结果和推荐系统的响应时间(RT response time)。

Table 4: Offline AUCs and online CTR gains of different methods. Online CTR gain is relative to the control group.

Methods	Offline AUC	Online CTR Gain	Average RT(ms)
WDL	0.7734	-	13
WDL(+Seq)	0.7846	+3.03%	14
DIN	0.7866	+4.55%	16
BST($b = 1$)	**0.7894**	+7.57%	20
BST($b = 2$)	0.7885	-	-
BST($b = 3$)	0.7823	-	-

从上述的结果可以看出，文章提出的 BST 网络具有更好的效果。

BST 的主要工作在于改进 raw feature 的重新组合问题以及进行高阶特征提取，在推荐系统中针对特征工程的工作也有很多。有一类基于 WDL 的工作，主要考虑在于将 item 的特征或者用户的特征进行交互项的构建，比如 DeepFM, Cross networks 等，大多借鉴了 Factorization Machines。这类方法的问题主要在于特征的构建效率比较低，而且高阶交互项没有办法提取。

因此另一类基于深度学习的方法通过复杂网络结构来提取高阶特征，这类方法的问题在于获取到的特征解释性较差，一些任务场景中需要特征的解释性与实际意义来提升工作效率。同时考虑了特征的解释性与高阶复杂性的结果，AutoInt 有相对较好的效果，但是也受制于任务种类和应用场景。AutoInt 也是基于 multihear self-attention 机制，重点考虑对于用户信息、商品信息这类原始特征的处理（下文有介绍），AutoInt 实际上可以作用在"other features"中，来得到更高质量的信息。

但本文依旧是选择手动来构建"other features"中的 cross features 交互部分。AutoInt 的缺陷在于无法提取序列信息。DIN 确实也考虑利用用户行为序列的信息，但是处理办法与本文不同，DIN 是通过注意力机制来刻画当前推荐商品与历史商品之间的多峰表征，而本文则是利用注意力机制构建转换器将序列数据进行提取与嵌入。

总而言之，本文通过对序列数据特征的合理构建与利用，可以明显地提升推荐效果，淘宝将其投入在实际的生产场景中，也取得了较好的线上效果。

2. Matching:

Multi-Interest Network with Dynamic Routing for Recommendation at Tmall

本工作的主要目的在于在 matching 阶段进行特征工程的工作，实现提取更加结构化，使模型能够得到更多信息增益的特征。获取更高质量的特征就可以进而促进整个推荐系统的效果改善。

Matching 阶段需要获取客户感兴趣的商品，或者客户对某种商品感兴趣的概率。所以在这个阶段，如何量化客户的"兴趣"更重要。多数文章的做法一般是把用户的一些 profile 特征或者 history 信息简单地作为一个单向量输入。这种做法忽略了存在多种兴趣的可能性：首先，有未利用的信息，在对整个模型而言会带来相应的信息损失；其次，用一个向量来代表所有不同兴趣本身就比较难，等价于把每个用户所有不同的兴趣压缩在一起，因此将用户的所有兴趣信息混合在一起会导致在 matching 阶段匹配到不准确的商品类型或者商品个体。而且文中选择对用户的每个不同类型兴趣都会独立计算，这样能够保证 matching 阶段的待推荐商品的准确度。

文中提出了先利用类似聚类的思想，先在用户的历史信息中分析提取客户的多兴趣的形式，并且每个用户的总兴趣数允许是不同的。通过构建的多兴趣网络(MIND)可以实现这个目标，利用胶囊网络对用户行为的 embeded features 再提取就可以得到较好的兴趣类。基于这些兴趣类，就可以在整个商品池中通过近似最近邻查找的方式找出该客户最可能感兴趣的 N 个商品作为 candidates，随后放入 ranking 阶段进行推荐方面的预测。

网络结构如下图所示：

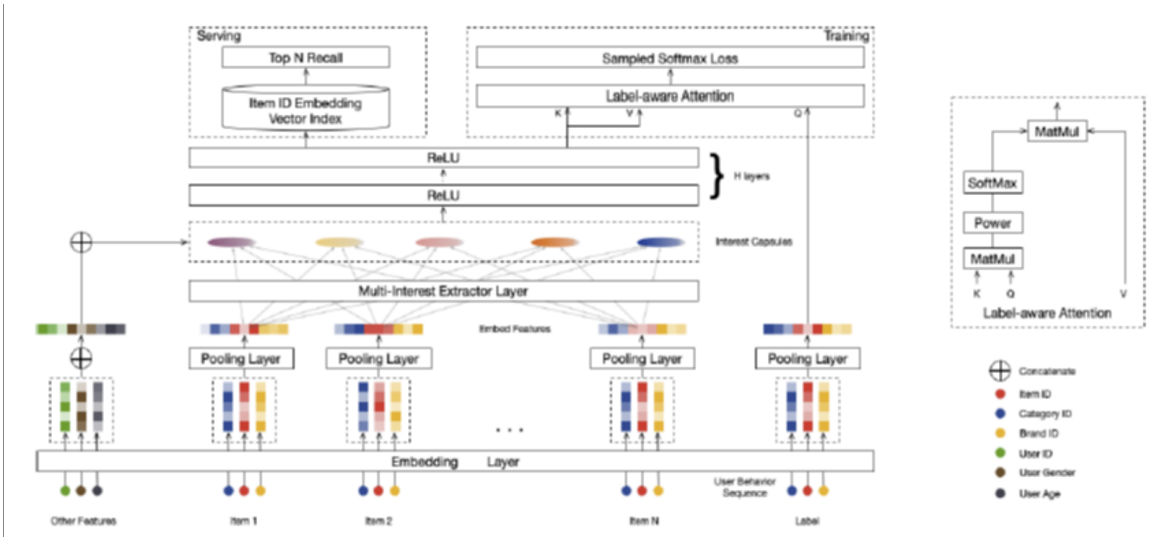


Figure 2: Overview of MIND. MIND takes user behaviors with user profile features as inputs, and outputs user representation vectors for item retrieval in the matching stage of recommendation. Id features from input layer are transformed into embeddings through the embedding layer, and embeddings of each item are further averaged by a pooling layer. User behavior embeddings are fed into the multi-interest extractor layer, which produces interest capsules. By concatenating interest capsules with user profile embedding and transforming the concatenated capsules by several ReLU layers, user representation vectors are obtained. During training, an extra label-aware attention layer is introduced to guide the training process. At serving, the multiple user representation vectors are used to retrieve items through an approximate nearest neighbor lookup approach.

整体网络结构就是通过将商品集合中的特征通过嵌入，映射到低维的特征空间，再将不同商品的嵌入特征连接起来输入到多兴趣提取层，通过学习胶囊网络中的参数，得到较好的模型后，再将胶囊网络提取的用户行为信息与用户的画像信息进行连接输入给全连接层作为最后的特征整合。
在训练阶段，有几个主要的与其他工作不同之处：

1. 通过构建合适的 label-attention 层来加强 capsules 网络的类内聚合性，原因是利用了商品类别作为查询，兴趣胶囊作为键值，如下图所示。输出的形式如下：

$v_u = attention(e_i, V_u, V_u) = V_u softmax(pow(V^T_ue_i, p))$

其中当 p 为 0 是，代表 capsule 选择最均匀的特征信息；如果 p 逐渐增大，那么 capsule 会选择相似度高的信息。如果 p 无穷大，那么最终就会变为一个最大 Attention 方法，并且会忽略其他的值。文中就是选择这种最大 attention 的方法。

2. 在训练过程中的模型参数初始化过程中，为了防止每个 capsules 的训练出很类似的效果，在动态路由算法的基础上，做了小量的修改和调整。本文中，作者选择希望训练得到的胶囊落在同一个向量空间中，但是希望不同的 bilinear mapping 能够将兴趣映射到不同的向量空间中，所以选择 routing logit 为：$b_{ij} = u_jSe_i$。e_i代表商品 i 的嵌入特征，u_j代表兴趣的向量。
3. 由于 Bilinear 矩阵是在各对胶囊之间共享的，为了避免矩阵训练得到单一结果，所以应该避免直接将所有权重初始化为 0，而应该从一个随机分布中抽样产生。本文选择从正态分布中产生这些随

机初始值。

4. 因为不同用户所用的兴趣胶囊个数可能是不同的，所以文中允许了动态的兴趣个数来提高网络的可塑性与准确性。$K'_u = max(1, min(K, log_2(I_u|)))$，由于多数人们的兴趣胶囊比较少量，这种可变个数的兴趣胶囊设定可以节省很多资源。

整个 B2I Dynamic Routing 伪代码如下：

Algorithm 1 B2I Dynamic Routing.

Input: behavior embeddings $\left\{\overrightarrow{e}_i, i \in I_u\right\}$, iteration times r, number of interest capsules K

Output: interest capsules $\left\{\overrightarrow{u}_j, j = 1, ..., K'_u\right\}$

1: calculate adaptive number of interest capsules K'_u by (9)
2: for all behavior capsule i and interest capsule j: initialize $b_{ij} \sim \mathcal{N}(0, \sigma^2)$.
3: **for** $k \leftarrow 1, r$ **do**
4: for all behavior capsule i: $w_{ij} \leftarrow softmax(b_{ij})$
5: for all interest capsule j: $\overrightarrow{z}_j = \sum_{i \in I_u} w_{ij} S \overrightarrow{e}_i$
6: for all interest capsule j: $\overrightarrow{u}_j \leftarrow squash(\overrightarrow{z}_j)$
7: for all behavior capsule i and interest capsule j: $b_{ij} \leftarrow b_{ij} + \overrightarrow{u}_j^T S \overrightarrow{e}_i$
8: **end for**
9: **return** $\left\{\overrightarrow{u}_j, j = 1, ..., K'_u\right\}$

在模型最终训练完成之后，整个 MIND 网络拿掉 label-aware 层就可以当做一个用户行为的表征映射。在运行阶段，用户的行为序列以及用户的画像特征全部输入到f_{user}中，就可以得到多个向量特征。这些特征进一步用来选出前 N 个备选商品，用于推荐系统的下一阶段的 ranking 工作。

与其他工作的联系与区别：

1. YouTube DNN：两种方法都采用了 deep learning 的优势，通过深度学习可以获取较好的高阶特征表示。但是 YouTube 仅仅使用一个向量来表示用户行为，本文中的 MIND 使用多个向量来表示一个用户。所以 MIND 可以看做 DNN 的一种泛化模型。
2. DIN：二者都利用到了用户的不同的性却特征。DIN 仅仅使用了商品层面上的注意力机制，简单来说就是对商品 ID 的一些相似度计算；而本文利用了动态路由算法产生的兴趣网络来提取兴趣层面的信息。DIN 将 ranking 阶段融合在一起，而 MIND 实际上不是端到端的，他通过将两部分拆解开来，为整个系统提供了更好的稳健性。

模型评估：

Dataset	HP	Metric	WALS	YouTube DNN	MaxMF-K-interest	MIND-1-interest	MIND-K-interest
Amazon Books	$K = 3$ $d = 36$	HR@10	0.0144 (-37.66%)	0.0231	0.0285 (+23.38%)	0.0273 (+18.18%)	**0.0309 (+33.77%)**
		HR@50	0.0553 (-25.87%)	0.0746	0.0862 (+15.55%)	0.0978 (+31.10%)	**0.1101 (+47.59%)**
		HR@100	0.0907 (-20.65%)	0.1143	0.1304 (+14.09%)	0.1459 (+27.65%)	**0.1631 (+42.69%)**
TmallData	$K = 5$ $d = 64$	HR@10	0.0372 (-36.84%)	0.0589	0.0628 (+6.62%)	0.0720 (+22.24%)	**0.0972 (+65.03%)**
		HR@50	0.0831 (-33.84%)	0.1256	0.1820 (+44.90%)	0.1512 (+20.38%)	**0.2080 (+65.60%)**
		HR@100	0.1126 (-31.67%)	0.1648	0.2567 (+55.76%)	0.1930 (+17.11%)	**0.2699 (+63.77%)**

CTR 预测：

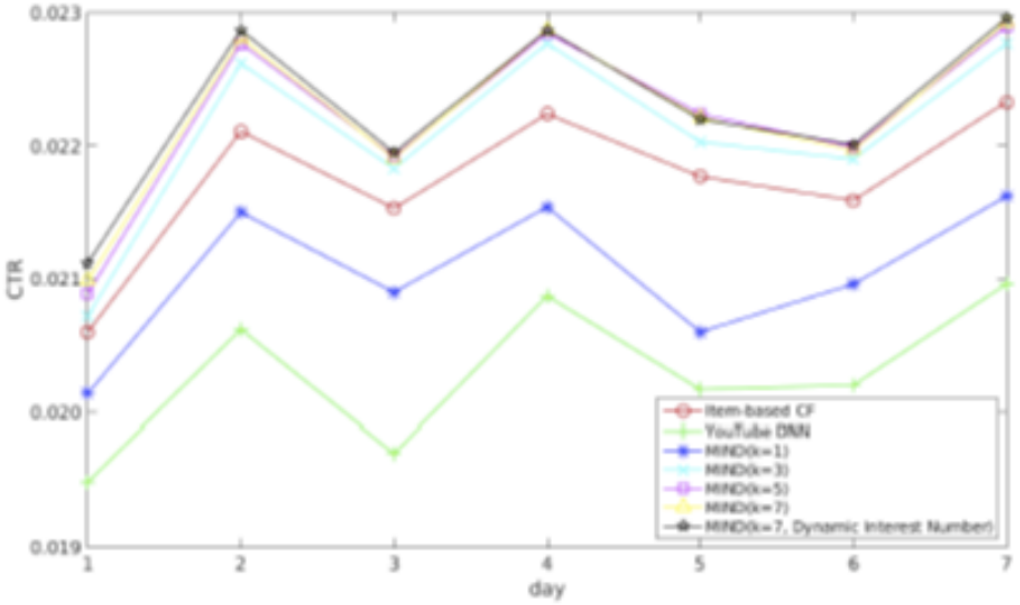


Figure 4: Online CTRs in a week. MIND with 5 ~7 interests performs best in all comparing methods. MIND significantly beats the two baseline methods, item-based CF and YouTube DNN.

胶囊网络的兴趣提取示例：

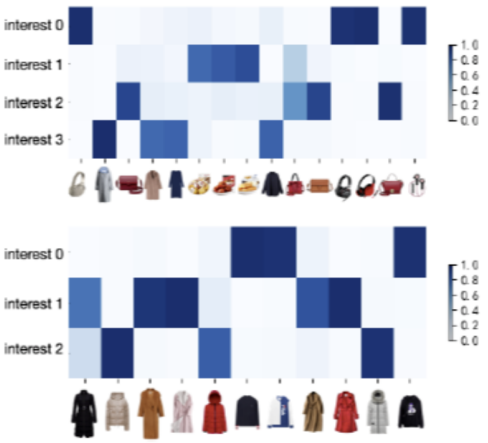


Figure 5: Heatmap of coupling coefficients for two users. Each class of behaviors has the max coupling coefficients on the corresponding interest. User C (upper) and user D (below) have different granularity of interests.



Figure 6: The distribution of items recalled by each interest corresponding to the user behaviors exampled on the left. Each interest is demonstrated by one axis, of which the coordinate is the similarity between items and interests. The size of the point is proportional to the number of the items with the specific similarity.

从第一个表格我们可以看出，所有与 Multi-interest 结合的模型对比起原始模型或者其他模型都得到了更好的效果。利用多兴趣网络确实可以得到不同客户的多种兴趣表征，能够筛选出更加有效的待推荐商品，从而提升推荐系统的准确度。在天猫的数据上，这种提升更加明显，因为大多数的人在购物时候的兴趣是由多种类型的。对于不同数据之间客户群体兴趣类别个数的比较，可以根据每个数据最好的超参数 K 的比较结果来确定。

在 CTR 预测上，MIND 也有明显的效果提升。

对于多兴趣特征的提取有效性，可以从图 5，图 6 中分析。图 5 中，上下两块代表了两个 user，每一行代表针对每个 user 训练得到的 interest，横轴上代表的是 user 的挑选商品行为，可以很明显的看出每种 Interest 所关联的商品类型是高度相近的，这反映了 interest capsules 得到的 multiple interest 都具有较好的聚合性。图 6 则表示在系统服务阶段，根据第一个 user 的 interest 来进行待推荐商品的

选择。可以看到其中的商品相似度是很高的，最后一行代表的是 Youtube DNN 的商品相似度，对比发现，multi-interest 分出的多种兴趣效果十分明显。

3. Feature Engineering：

AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks

在整个推荐系统中，特征工程方面最常遇到两个问题：

1. 输入特征的维度过高而且稀疏性很强；
2. 有效的点击率预测都要基于高度准确的交叉特征，人为地构造交叉特征是一件十分耗时耗力的操作。

因此，寻找一种高效的特征提取方法是提升推荐系统表现效果的关键步骤。本文利用了深度学习的抽象特征提取的优势，通过 Multi-head attention 层进行高阶的特征组合。这种方法对于数值型特征与分类特征都很适用。

之前的工作也有考虑过交叉特征的文章，有的将全部的特征进行外积展开得到一组非常高稀疏性的特征，这类特征在进行模型训练的过程中十分容易造成过拟合的现象。寻找有意义的高阶交叉特征一般都是需要有经验的人进行人为构建，但是穷举出所有潜在的可能性是很难的，所以需要借助深度学习类的方法来进行有效的特征的低维表示学习。

深度学习类的方法虽然可以提取出高阶的特征，但是也有很大的局限性:

1. 全连接神经网络已经被证实过在学习交互特征的过程中效率很低；
2. 对提取出来的特征很难具有解释性；

本文提出的这种 AutoInt 方法就是针对上述两个问题对深度学习网络进行调整加以改进。

首先将数值变量与分类变量都进行低维嵌入，得到一个低维空间的向量表达，也是为了解决分类变量与数值变量之间没有良定义的内积这类算子的问题。随后将低维嵌入的特征向量输入到本文提出的 Interacting Layer，目的是为了得到具有交互性的新的特征。在每一层的 Interacting Layer 里，每一个特征都会通过 multi-head attention 机制和其他特征进行变换组合，这就是产生交叉特征的原理。而且由于 multi-head attention 可以把一个特征向量映射到多个不同的空间中，因此在经过 multi-head attention 变换后的特征的组合方式也是具有多种形式的。在经过一层的 Interacting Lyaer，特征经过了一阶的交叉组合，多次叠加 Interacting Layer 就可以得到高阶复杂的特征交叉组合。对于特征的解释性，由于注意力机制模块主要是利用变量之间的相似度，因此高度相似的特征进行组合时很容易得到较好的解释。

整个网络的结构如下：

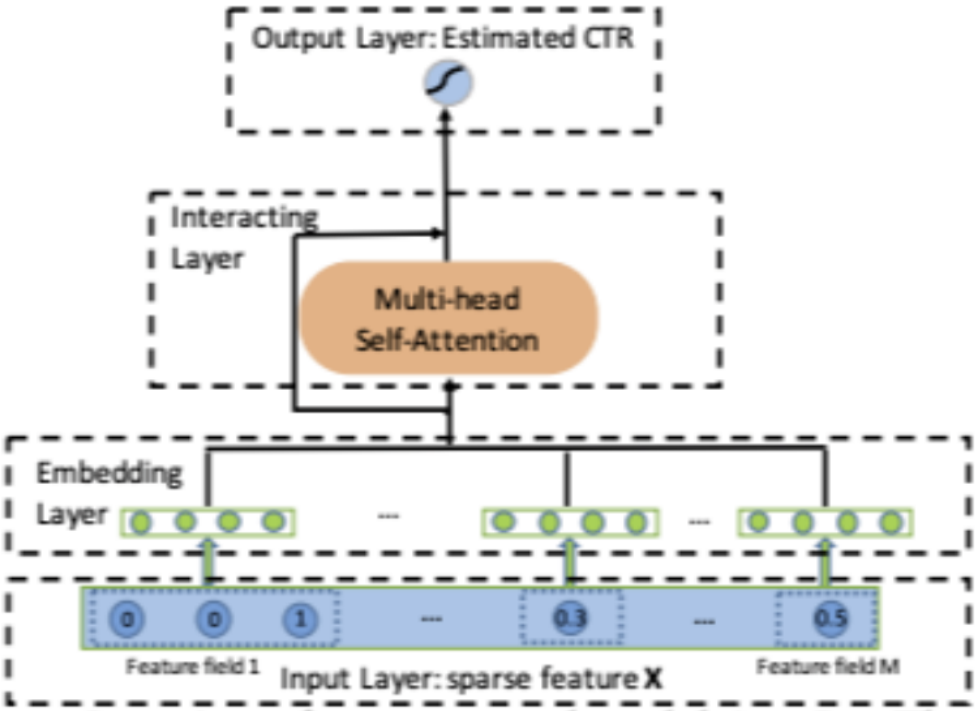


Figure 1: Overview of our proposed model AutoInt. The details of embedding layer and interacting layer are illustrated in Figure 2 and Figure 3 respectively.

本文主要的核心模块 Interacting Layer 的结构如下：

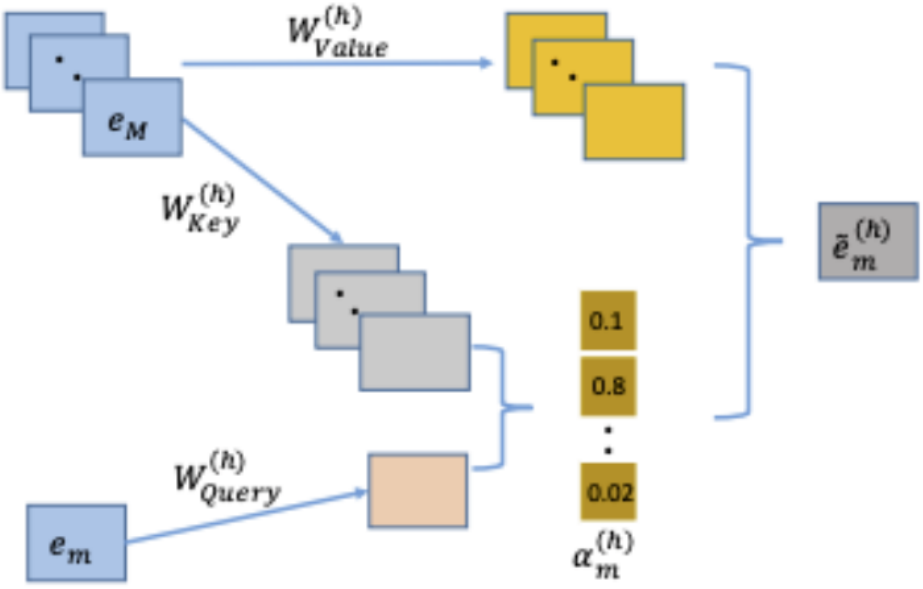


Figure 3: The architecture of interacting layer. Combinatorial features are conditioned on attention weights, i.e., $\alpha_m^{(h)}$.

实际上是以关注的某个特征作为 query，其他的特征向量作为 key, value，并且允许多种不同类型的相似度计算从而得到不同的 attention value，这也是 multi-head 的主要特点。

$$\alpha_{m,k}^{(h)} = \frac{\exp(\psi^{(h)}(e_m, e_k))}{\sum_{l=1}^{M} \exp(\psi^{(h)}(e_m, e_l))},$$

$$\psi^{(h)}(e_m, e_k) = \langle W_{Query}^{(h)} e_m, W_{Key}^{(h)} e_k \rangle,$$

最后把所有的e向量拼接起来作为学习到的组合特征，最后经过一个 ReLU 变换与之前的 feature 重新结合即可得到 Interacting Layer 的输出：

$$e_m^{Res} = ReLU(\tilde{e}_m + W_{Res} e_m),$$

最终的预测输出为：

$$\hat{y} = \sigma(w^T(e_1^{Res} \oplus e_2^{Res} \oplus \cdots \oplus e_M^{Res}) + b),$$

$$\hat{y} = \sigma(w^T(e_1^{Res} \oplus e_2^{Res} \oplus \cdots \oplus e_M^{Res}) + b),$$

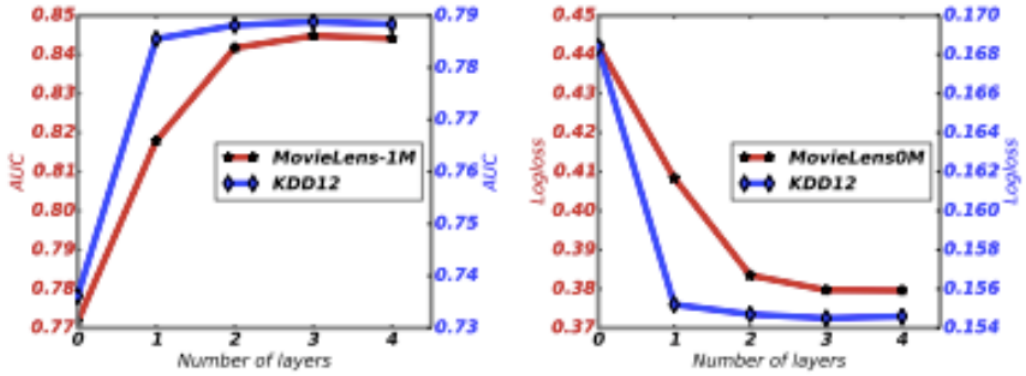
由于本文的主要贡献在于，更加快速有效的高阶特征提取工作以及特征的可解释性，所以从这两个角度来看本文方法的效果。

Table 2: Effectiveness Comparison of Different Algorithms. We highlight that our proposed model almost outperforms all baselines across four data sets and both metrics. Further analysis is provided in Section 5.2.

Model Class	Model	Criteo		Avazu		KDD12		MovieLens-1M	
		AUC	Logloss	AUC	Logloss	AUC	Logloss	AUC	Logloss
First-order	LR	0.7820	0.4695	0.7560	0.3964	0.7361	0.1684	0.7716	0.4424
Second-order	FM [26]	0.7836	0.4700	0.7706	0.3856	0.7759	0.1573	0.8252	0.3998
	AFM[40]	0.7938	0.4584	0.7718	0.3854	0.7659	0.1591	0.8227	0.4048
High-order	DeepCrossing [32]	0.8009	0.4513	0.7643	0.3889	0.7715	0.1591	0.8448	0.3814
	NFM [13]	0.7957	0.4562	0.7708	0.3864	0.7515	0.1631	0.8357	0.3883
	CrossNet [38]	0.7907	0.4591	0.7667	0.3868	0.7773	0.1572	0.7968	0.4266
	CIN [19]	0.8009	0.4517	**0.7758**	0.3829	0.7799	0.1566	0.8286	0.4108
	HOFM [5]	0.8005	0.4508	0.7701	0.3854	0.7707	0.1586	0.8304	0.4013
	AutoInt (ours)	**0.8061****	**0.4455****	0.7752	**0.3824**	**0.7883****	**0.1546****	**0.8456***	**0.3797****

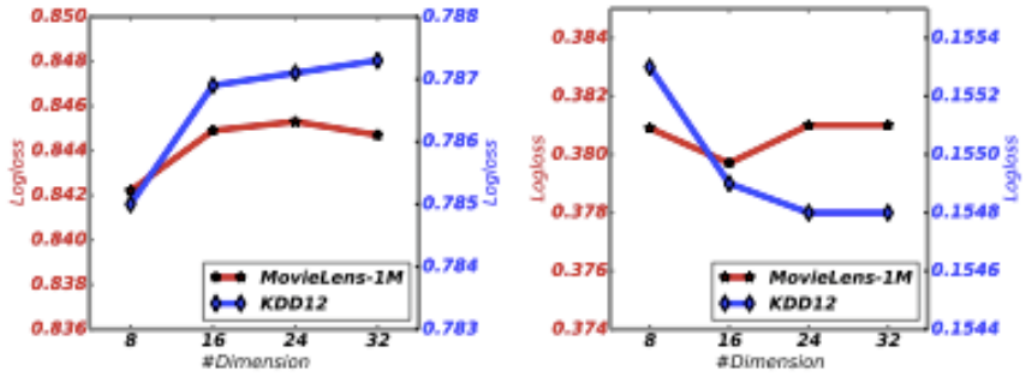
AutoInt outperforms the strongest baseline w.r.t. Criteo, KDD12 and MovieLens-1M data at the: ** 0.01 and * 0.05 level, unpaired t-test.

通过表格可以看出 AutoInt 在大多数场景下都有更好的效果，对于 Interacting Layer 的层数选取也是依据实验结果得到的启发式结论：



(a) AUC (b) Logloss

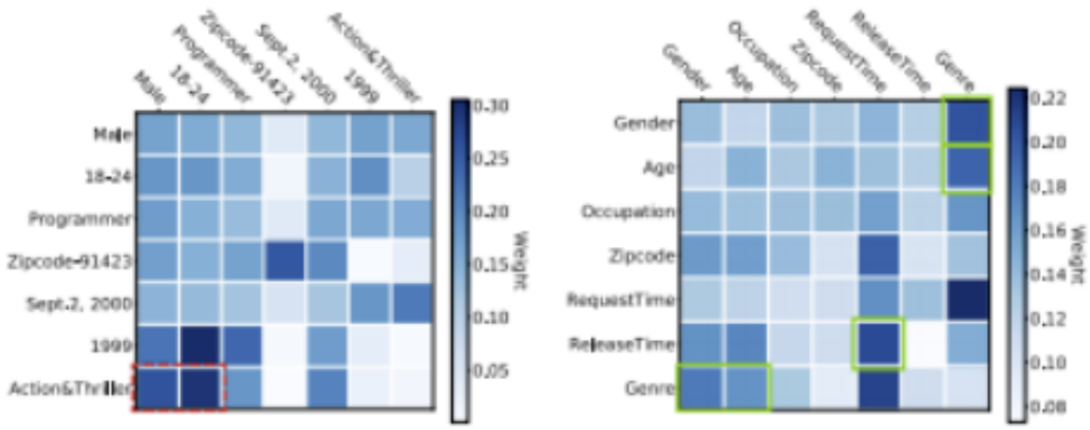
Figure 5: Performance w.r.t. the number of interacting layers. Results on Criteo and Avazu data sets are similar and hence omitted.



(a) AUC (b) Logloss

Figure 6: Performance w.r.t. number of embedding dimensions. Results on Criteo and Avazu data sets are similar and hence omitted.

当 Interacting Layer 过多的时候不仅会增加模型复杂度，模型的预测效果也会受到影响。最后是模型的特征解释：



(a) Label=1, Predicted CTR=0.89 (b) Overall feature interactions

Figure 7: Heat maps of attention weights for both case- and global-level feature interactions on MovieLens-1M. The axises represent feature fields <*Gender, Age, Occupation, Zipcode, RequestTime, RealeaseTime, Genre*>. We highlight some learned combinatorial features in rectangles.

在这个数据任务当中，Interacting Layer 确实捕捉到了有意义的组合特征：<Gender = Male, Age = [18 - 24], MovieGenre = Action&Triller>，左图中红色方框部分。右图我们给出了数据本身的相关性的热力图，从中可以发现，注意力机制基本能够将高度相关的特征进行组合并且给以较高的权重，这样的话一层 Interacting Layer 之后输出的特征实际上就具有了较好的解释性。

总结

以上这些是今年来在推荐系统方面比较有意义的工作，任何模型的效果都强依赖数据、特征、样本质量。论文思想实际上也是来源于人的本身行为习惯生活经验等，例如购物时候行为序列的重要性等。通过思考如何让计算机模拟或者关注人们在日常行为的一些关键模式，得到特征的有效转变和映射，让整个模型能够有更好的表现。所谓网络的构建不同也无非在于不同应用场景，特征的利用方式不同。

另外推荐领域一些最新论文其实也是参考了 NLP 一些新的思路，例如上面介绍的 BST、autoint 都引入了 Transformer 的核心模块 Multi-head Self-attention，在此基础上做各种变换，实现各种玩法。看懂论文需要先彻底熟悉 Multi-head Self-attention 的原理，可以结合源论文与代码去看，用开源数据集去跑流程，打印中间结构，熟悉论文思想的同时也可以提升工程实现能力。

从各自的出发点来看，三篇文章都可以相互弥补：

对于 MIND 而言，虽然已经考量了顾客在浏览商品的时候会有固定类别的兴趣偏好，但是仅仅考虑了历史行为中的聚类信息而没有利用行为的序列或者趋势信息，这种序列信息往往在下一步的对未来行为的推荐中会更有作用。另外文中也提及了，这种聚类的方式还有待改进，通过结合其他方法能够提升类别的聚合性，进一步提升类别的准确程度。而 BST 本身恰恰就是利用原始特征在 ranking 模块通过网络结构来获取序列行为信息。但 BST 本身的劣势在于，如果原始特征维度很大，人工的构建交叉特征很不现实。

此时，AutoInt 就可以出面帮助解决对于 raw features，如何快速高效构建 cross features 的问题。通过 AutoInt 构建出来的特征是否在其他的分类器或者模型上依旧能够带来提升，这是有待进一步实验研究的问题。另外，如何针对推荐系统的冷启动问题或者不同客户群体进行推荐的策略也可以基于以上工作加以改进。最后，这些光鲜亮丽的论文思想听上去很是高大上，在实际场景下真正复现论文里面的效果还是比较难的，需要结合数据场景作各种优化，模型结构优化，还要具备生产环境下复杂模型上线的能力，这些都是需要努力去攻克的。

参考

- AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks
- Multi-Interest Network with Dynamic Routing for Recommendation at Tmall
- Behavior Sequence Transformer for E-commerce Recommendation in Alibaba, Alibaba
- Deep Interest Network for Click-Through Rate Prediction
- Deep Neural Network for YouTube Recommendation
- https://github.com/tensorflow/docs/tree/master/site/zh-cn/tutorials

超越 BERT 模型的ELECTRA代码解读

作者：feelingzhou，腾讯 WXG 应用研究员

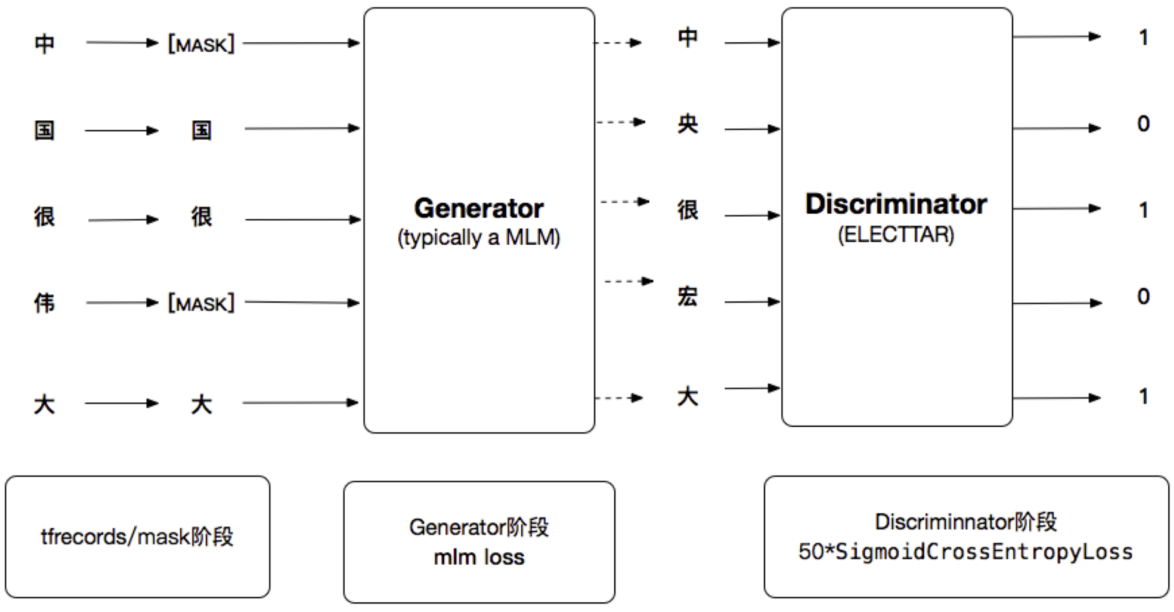
1、概述

在2019年11月份，NLP大神Manning联合谷歌做的ELECTRA一经发布，迅速火爆整个NLP圈，其中ELECTRA-small模型参数量仅为 BERT-base模型的1/10，性能却依然能与BERT、RoBERTa等模型相媲美，得益于ELECTRA模型的巧妙构思LOSS,在2020年3月份Google对代码做了开源，下面针对Google放出的ELECTRA做代码做解读，希望通过此文章大家能在自己文本数据、行为序列数据训练一个较好的预训练模型，在业务上提升价值。

2、ELECTRA模型

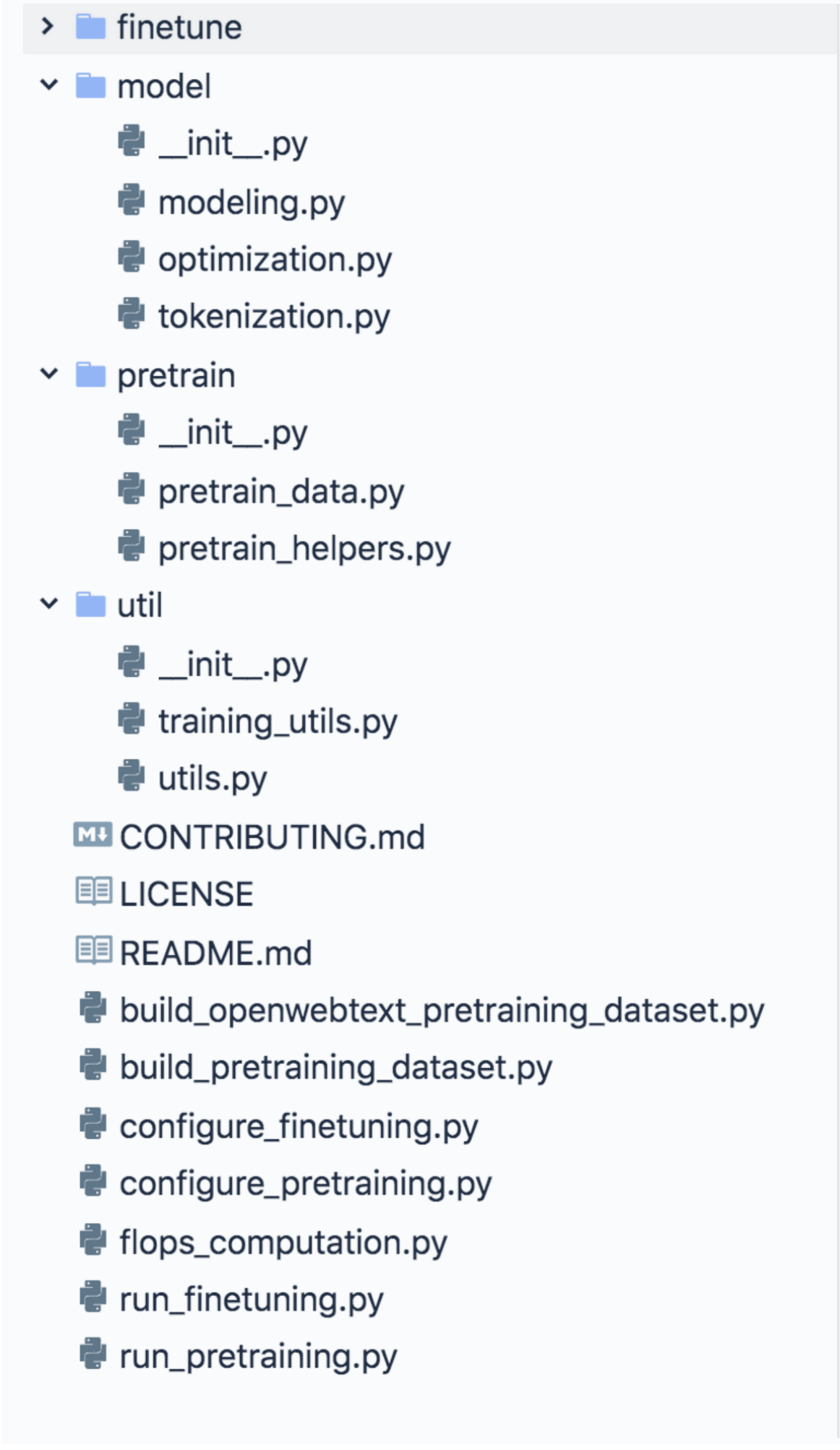
2、1 总体框架

ELECTRA模型(BASE版本)本质是换一种方法来训练BERT模型的参数；BERT模型主要是利用MLM的思想来训练参数，直接把需要预测的词给挖掉了，挖了15%的比例。由于每次训练是一段话中15%的token，导致模型收敛更新较慢，需要的语料也比较庞大。同时为了兼顾处理阅读理解这样的任务，模型加入了NSP，是个二分类任务，判断上下两句是不是互为上下句；而ELECTRA模型主要借助于图像领域gan的思想，利用生成器和判别器思想，如下图所；ELECTRA的预训练可以分为两部分，生成器部分仍然是MLM，结构与BERT类似，利用这个模型对挖掉的15%的词进行预测，并将其进行替换，若替换的词不是原词，则打上被替换的标签，语句的其他词则打上没有替换的标签，判别器部分是训练一个判别模型对所有位置的词进行替换识别，此时预测模型转换成了一个二分类模型。这个转换可以带来效率的提升，对所有位置的词进行预测，收敛速度会快的多，损失函数是利用生成器部分的损失和判别器的损失函数以一个比例数(官方代码是50)相加。



2、2 代码框架

2020年3月份Google开源了ELECTRA模型代码，见代码连接，其主要代码框架如下：


finetune
model
__init__.py
modeling.py
optimization.py
tokenization.py
pretrain
__init__.py
pretrain_data.py
pretrain_helpers.py
util
__init__.py
training_utils.py
utils.py
CONTRIBUTING.md
LICENSE
README.md
build_openwebtext_pretraining_dataset.py
build_pretraining_dataset.py
configure_finetuning.py
configure_pretraining.py
flops_computation.py
run_finetuning.py
run_pretraining.py

下面对该代码框架做一一说明：

finetune： 该文件夹下面的代码主要是对已经训练好的ELECTRA模型做微调的代码例子，如文本分类、NER识别、阅读理解等任务，这个和BERT的任务一致，在这里不做过多累赘。

model/modeling.py： 该文件主要是bert模型的实现逻辑以及bert模型的配置读取代码，在ELECTRA模型的预训练阶段生成和判别阶段做调用，另外在做各种finetuneing任务会调用。

model/optimization.py： 该文件主要是对优化器的实现，主要是对AdamWeightDecay的实现，可以自己加lamb等优化方法的实现。

model/tokenization.py： 该文件主要是WordPiece分词器的实现，可以对英文、中文分词，在将文本转化为tfrecord的时候会用的到。

pretrain/pretrain_data.py: 该文件是主要作用是对ELECTRA模型在pretraining对tfrecords文件读取、collections.namedtuple更新上的一些逻辑实现。

pretrain_helpers.py： 该文件是pretraining阶段核心功能实现，主要实现逻辑对序列做动态随机**mask**以及对已经mask的序列做**unmask**。

util/training_utils.py： 该文件主要在训练阶段实现了一个Hook，在训练阶段为了打印更多日志信息。

util/utils.py： 该文件主要是一些基础方法，如增删文件夹、序列化反序列化、读取配置文件等。

build_openwebtext_pretraining_dataset.py、build_pretraining_dataset.py： 这两个文件功能类似，但是用到的数据源不一样，主要是把文本文件转化为tfrecord文件，tfrecord文件的key包括**input_ids**、**input_mask**、**segment_ids**，生成的tfrecord文件不像bert预训练需要的文件那样，不需要再生成 **masked_lm_positions**,**masked_lm_ids**,**masked_lm_weights** ，这几个key会在模型pretraining阶段自动生成，与此同时mask也是随机动态的，类似于RoBerta，不像BERT那样固定。里面写了个多线程加速；对于大型的文件，还是使用spark将文本转化较适宜。

configure_finetuning.py： finetuneing阶段的一些超参数配置，google这次放出的代码参数并没有使用tf.flags。

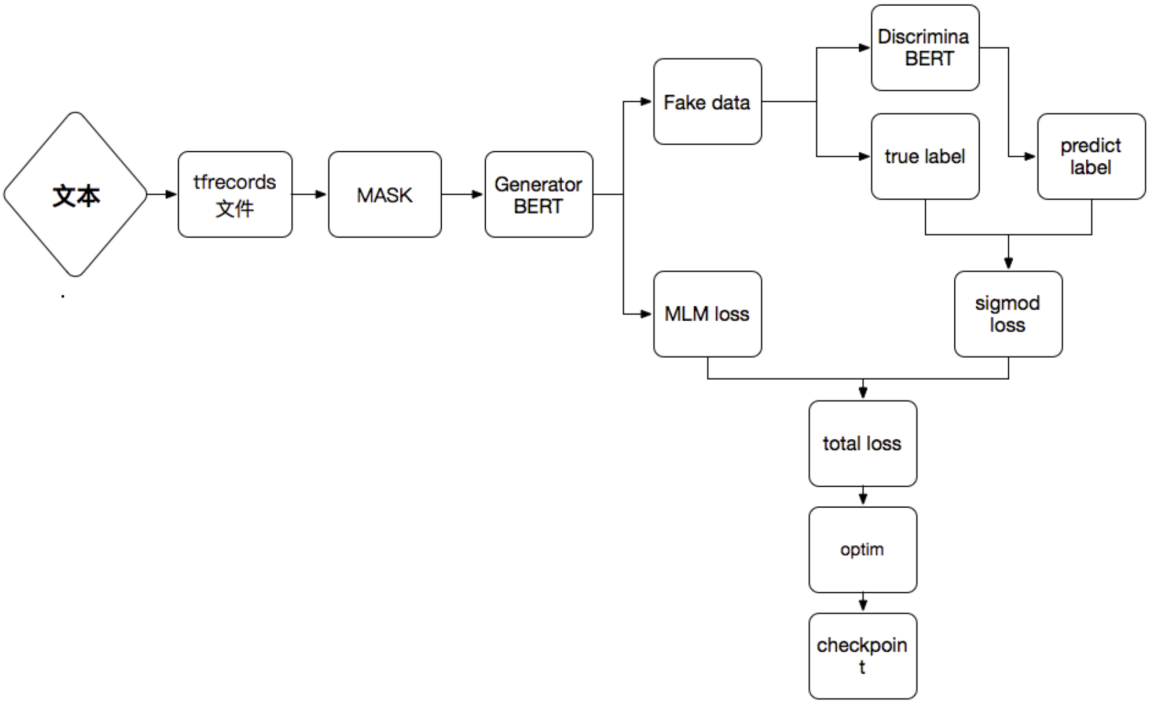
configure_pretraining.py： pretraining阶段的一些超参数配置，google这次放出的代码参数并没有使用tf.flags。

run_finetuning.py： 模型做finetuning逻辑，加载已经训练好的ELECTRA模型做微调。

run_pretraining.py： 模型pretraining逻辑，也是ELECTRA模型最核心的逻辑，下面会加以详细说明。

2、3 pretraining阶段

ELECTRA模型pretraining阶段是最核心的逻辑，代码是在run_pretraining.py里面，下面会加以详细说明，整体阶段理解绘制了一张图，逻辑见如下图：



2.3.1 主方法入口

在run_pretraining.py文件中主方法有三个必须的参数见：

```
def main():
  parser = argparse.ArgumentParser(description=__doc__)
  parser.add_argument("--data-dir", required=True,
                      help="Location of data files (model weights, etc).")
  parser.add_argument("--model-name", required=True,
                      help="The name of the model being fine-tuned.")
  parser.add_argument("--hparams", default="{}",
                      help="JSON dict of model hyperparameters.")
  args = parser.parse_args()
  if args.hparams.endswith(".json"):
    hparams = utils.load_json(args.hparams)
  else:
    hparams = json.loads(args.hparams)
  tf.logging.set_verbosity(tf.logging.ERROR)
  train_or_eval(configure_pretraining.PretrainingConfig(
      args.model_name, args.data_dir, **hparams))
```

三个必须参数见如下：
--data-dir： 表示tfrecord文件的地址，一般是以pretrain_data.tfrecord-0-of*这种格式，调用build_pretraining_dataset.py文件生成，默认生成1000个tfrecord文件，数目可以自己改，此外需要注意的是切词需要制定vocab.txt，训练中文的模型词典指定BERT模型那个vocab.txt词典即可，同理用于英文的模型训练。

--model-name： 表示预训练模型的名字一般是electar，可以自己设定。

--hparams：，一般是个json文件，可以传递自己的参数进去，比如你要训练的模型是small、base、big等模型，还有vocab_size，一般中文是21128，英文是30522。还有模型训练是否是测试状态等参数，一般我训练中文模型hparams参数是的config.json是:

```
{
  "model_size": "base",
  "vocab_size": 21128
}
```

详细的参数可以去看configure_pretraining.py，一般你传进去的参数进去会更新里面的超参数。

程序入口训练模型：

```
train_or_eval(configure_pretraining.PretrainingConfig(
      args.model_name, args.data_dir, **hparams))
```

还有一个入口，只see.run()一次，用于测试，见如下：

```
train_one_step(configure_pretraining.PretrainingConfig(
       args.model_name, args.data_dir, **hparams))
```

2.3.2 数据mask

训练模型主要是代码是**PretrainingModel**类的定义，在PretrainingModel里面
程序首先对输入的tfrecord文件做随机mask，

```
# Mask the input
    masked_inputs = pretrain_helpers.mask(
        config, pretrain_data.features_to_inputs(features), config.mask_prob)
```

用于生成含有masked_lm_positions,masked_lm_ids,masked_lm_weights等key的tfrecord文件，随机MASK实现的主要逻辑是调用pretrain_helpers.mask()来实现，其中用到了随机生成多项分布的函数tf.random.categorical，这个函数目的是随机获取masked_lm_positions、masked_lm_weights，再根据masked_lm_positions调用 tf.gather_nd做索引截取来获取masked_lm_ids。

2.3.3 Generator BERT

数据获取之后往下一步走就是生成Generator BERT阶段的模型，调用方法见如下：

```
 generator = self._build_transformer(
          masked_inputs, is_training,
         bert_config=get_generator_config(config, self._bert_config),
          embedding_size=(None if config.untied_generator_embeddings
                          else embedding_size),
          untied_embeddings=config.untied_generator_embeddings,
          name="generator")
```

这里主要用于Generator阶段BERT模型生成，同时生成MLM loss 和 Fake data，其中Fake data非常核心。
MLM loss生成见代码，和BERT的逻辑几乎一样：

```
 def _get_masked_lm_output(self, inputs: pretrain_data.Inputs, model):
    """Masked language modeling softmax layer."""
    masked_lm_weights = inputs.masked_lm_weights
    with tf.variable_scope("generator_predictions"):
      if self._config.uniform_generator:
        logits = tf.zeros(self._bert_config.vocab_size)
        logits_tiled = tf.zeros(
```

```
            modeling.get_shape_list(inputs.masked_lm_ids) +
            [self._bert_config.vocab_size])
        logits_tiled += tf.reshape(logits, [1, 1, self._bert_config.vocab_size])
        logits = logits_tiled
      else:
        relevant_hidden = pretrain_helpers.gather_positions(
            model.get_sequence_output(), inputs.masked_lm_positions)
        hidden = tf.layers.dense(
            relevant_hidden,
            units=modeling.get_shape_list(model.get_embedding_table())[-1],
            activation=modeling.get_activation(self._bert_config.hidden_act),
            kernel_initializer=modeling.create_initializer(
                self._bert_config.initializer_range))
        hidden = modeling.layer_norm(hidden)
        output_bias = tf.get_variable(
            "output_bias",
            shape=[self._bert_config.vocab_size],
            initializer=tf.zeros_initializer())
        logits = tf.matmul(hidden, model.get_embedding_table(),
                           transpose_b=True)
        logits = tf.nn.bias_add(logits, output_bias)

      oh_labels = tf.one_hot(
          inputs.masked_lm_ids, depth=self._bert_config.vocab_size,
          dtype=tf.float32)

      probs = tf.nn.softmax(logits)
      log_probs = tf.nn.log_softmax(logits)
      label_log_probs = -tf.reduce_sum(log_probs * oh_labels, axis=-1)

      numerator = tf.reduce_sum(inputs.masked_lm_weights * label_log_probs)
      denominator = tf.reduce_sum(masked_lm_weights) + 1e-6
      loss = numerator / denominator
      preds = tf.argmax(log_probs, axis=-1, output_type=tf.int32)

      MLMOutput = collections.namedtuple(
          "MLMOutput", ["logits", "probs", "loss", "per_example_loss", "preds"])
      return MLMOutput(
          logits=logits, probs=probs, per_example_loss=label_log_probs,
          loss=loss, preds=preds)
```

Fake data数据生成逻辑见下面代码，这里调用了unmask函数和上面提到的mask函数作用相反，把原来input_ids随机mask的函数还原回去生成一个input_ids_new，再利用判别模型生成的logit取最大索引去还原原来被mask调的input_ids,生成一个updated_input_ids，判断input_ids_new和updated_input_ids是否相等，生成true label

```
  def _get_fake_data(self, inputs, mlm_logits):
    """Sample from the generator to create corrupted input."""
    inputs = pretrain_helpers.unmask(inputs)
    disallow = tf.one_hot(
        inputs.masked_lm_ids, depth=self._bert_config.vocab_size,
        dtype=tf.float32) if self._config.disallow_correct else None
    sampled_tokens = tf.stop_gradient(pretrain_helpers.sample_from_softmax(
        mlm_logits / self._config.temperature, disallow=disallow))
    sampled_tokids = tf.argmax(sampled_tokens, -1, output_type=tf.int32)
    updated_input_ids, masked = pretrain_helpers.scatter_update(
        inputs.input_ids, sampled_tokids, inputs.masked_lm_positions)
```

```
    labels = masked * (1 - tf.cast(
        tf.equal(updated_input_ids, inputs.input_ids), tf.int32))
    updated_inputs = pretrain_data.get_updated_inputs(
        inputs, input_ids=updated_input_ids)
    FakedData = collections.namedtuple("FakedData", [
        "inputs", "is_fake_tokens", "sampled_tokens"])
    return FakedData(inputs=updated_inputs, is_fake_tokens=labels,
                     sampled_tokens=sampled_tokens)
```

2.3.4 Discrimina BERT

利用上一步生成的Fake data，作为 Discrimina BERT的输入，见代码：

```
    if config.electra_objective:
      discriminator = self._build_transformer(
          fake_data.inputs, is_training, reuse=not config.untied_generator,
          embedding_size=embedding_size)
      disc_output = self._get_discriminator_output(
          fake_data.inputs, discriminator, fake_data.is_fake_tokens)
```

获取二分类的损失函数，代码见：

```
def _get_discriminator_output(self, inputs, discriminator, labels):
   """Discriminator binary classifier."""
   with tf.variable_scope("discriminator_predictions"):
     hidden = tf.layers.dense(
         discriminator.get_sequence_output(),
         units=self._bert_config.hidden_size,
         activation=modeling.get_activation(self._bert_config.hidden_act),
         kernel_initializer=modeling.create_initializer(
             self._bert_config.initializer_range))
     logits = tf.squeeze(tf.layers.dense(hidden, units=1), -1)
     weights = tf.cast(inputs.input_mask, tf.float32)
     labelsf = tf.cast(labels, tf.float32)
     losses = tf.nn.sigmoid_cross_entropy_with_logits(
         logits=logits, labels=labelsf) * weights
     per_example_loss = (tf.reduce_sum(losses, axis=-1) /
                         (1e-6 + tf.reduce_sum(weights, axis=-1)))
     loss = tf.reduce_sum(losses) / (1e-6 + tf.reduce_sum(weights))
     probs = tf.nn.sigmoid(logits)
     preds = tf.cast(tf.round((tf.sign(logits) + 1) / 2), tf.int32)
     DiscOutput = collections.namedtuple(
         "DiscOutput", ["loss", "per_example_loss", "probs", "preds",
                        "labels"])
```

2.3.5 总的损失函数

上面一步骤求出了disc_output.loss也就是sigmod的loss，代码见：

```
self.total_loss = config.gen_weight * mlm_output.loss
self.total_loss += config.disc_weight * disc_output.loss
```

这里config.gen_weight=1以及config.disc_weight=50，这里sigmod的损失函数设置为50，作者也没给明确的答复。

2.3.6 模型优化以及checkpoint

上面一步已经求出了总的损失函数，下一步则是做模型优化训练以及做checkpoint，程序入口在train_or_eval()里面，代码见:

```
is_per_host = tf.estimator.tpu.InputPipelineConfig.PER_HOST_V2
  tpu_cluster_resolver = None
  if config.use_tpu and config.tpu_name:
    tpu_cluster_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(
        config.tpu_name, zone=config.tpu_zone, project=config.gcp_project)
  tpu_config = tf.estimator.tpu.TPUConfig(
      iterations_per_loop=config.iterations_per_loop,
      num_shards=(config.num_tpu_cores if config.do_train else
                  config.num_tpu_cores),
      tpu_job_name=config.tpu_job_name,
      per_host_input_for_training=is_per_host)
  run_config = tf.estimator.tpu.RunConfig(
      cluster=tpu_cluster_resolver,
      model_dir=config.model_dir,
      save_checkpoints_steps=config.save_checkpoints_steps,
      tpu_config=tpu_config)
  model_fn = model_fn_builder(config=config)
  estimator = tf.estimator.tpu.TPUEstimator(
      use_tpu=config.use_tpu,
      model_fn=model_fn,
      config=run_config,
      train_batch_size=config.train_batch_size,
      eval_batch_size=config.eval_batch_size)
```

可以看到google开源的代码主要是TPU的一些钩子，改成GPU也比较简单，在BERT里面就有GPU相关的钩子，下面就会讲到。

2、4 finetuning阶段

ELECTRA finetuning阶段给出了不少例子，也比较简单，在finetuneing文件下面，这里不做过多的说明，和bert类似，唯一要改的就是把TPU相关的设置改为GPU、CPU即可。

2、5 序列训练改进

上面代码主要存在两个问题，**第一个**是TPU设置的问题，并不是人人都是土豪，还是要适配GPU的训练，**第二个**就是假如我想训练一个vocab比较大的序列模型，上面模型是训练不动的，loss方面改为负采样的形式。

2.5.1 TPU改GPU训练

```
 global_step = tf.train.get_or_create_global_step()
            optimizer =
optimization.AdamWeightDecayOptimizer(learning_rate=learning_rate)
            train_op = optimizer.apply_gradients(zip(grads, tvars), global_step)

            update_global_step = tf.assign(global_step, global_step + 1,
name='update_global_step')
            output_spec=tf.estimator.EstimatorSpec(
              mode=mode,
              predictions=probabilities,
              loss=total_loss,
              train_op=tf.group(train_op, update_global_step))
```

```
 run_config =
tf.estimator.RunConfig(model_dir=FLAGS.modelpath,save_checkpoints_steps=500)

    bert_config = modeling.BertConfig.from_json_file(FLAGS.bert_config_file)

    model_fn = model_fn_builder(
      bert_config=bert_config,
      num_labels=FLAGS.n_class,
      is_training=True,
      init_checkpoint=FLAGS.init_checkpoint,
      learning_rate=FLAGS.learning_rate,
      use_one_hot_embeddings=False)

    estimator = tf.estimator.Estimator(model_fn=model_fn,
             config=run_config)
    total_files=glob.glob("/tf*")
    random.shuffle(total_files)
    eval_files=total_files.pop()
    input_fn_train=lambda:input_fn(total_files,FLAGS.batch_size,num_epochs=N)
    input_fn_eval=lambda:input_fn(eval_files,FLAGS.batch_size,is_training=False)


    train_spec = tf.estimator.TrainSpec(input_fn=input_fn_train,
max_steps=10000)
    eval_spec = tf.estimator.EvalSpec(input_fn=input_fn_eval,steps=None,
start_delay_secs=30, throttle_secs=30)

    tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

2.5.2 负采样改造

主要是对mlm loss做改造：

```
def get_masked_lm_output(bert_config, input_tensor, output_weights, positions,
                         label_ids, label_weights):

  with tf.variable_scope("cls/predictions"):
    with tf.variable_scope("transform"):
      input_tensor = tf.layers.dense(
          input_tensor,
          units=bert_config.hidden_size,
          activation=modeling.get_activation(bert_config.hidden_act),
          kernel_initializer=modeling.create_initializer(
              bert_config.initializer_range))
      input_tensor = modeling.layer_norm(input_tensor) #batch*10*embeding_size

    # The output weights are the same as the input embeddings, but there is
    # an output-only bias for each token.
    output_bias = tf.get_variable(
        "output_bias",
        shape=[bert_config.vocab_size],
        initializer=tf.zeros_initializer())
    label_ids = tf.reshape(label_ids, [-1,1])
    label_weights = tf.reshape(label_weights, [-1])
    per_example_loss = tf.nn.sampled_softmax_loss(
```

```
        weights=output_weights,
        biases=output_bias,
        labels=label_ids,
        inputs=input_tensor,
        num_sampled=N,
        num_classes=bert_config.vocab_size)

    numerator = tf.reduce_sum(label_weights*per_example_loss )
    denominator = tf.reduce_sum(label_weights) + 1e-5
    loss = numerator / denominator
  return (loss, per_example_loss)
```

3、总结

前段时间挺忙，有比较多的新idea出来没来的及看，上周末花了一天时间看了下electra源码，并做记录，也看到不少团队做了一些中文electra预训练模型，虽然electra没有达到state of the art，和roberta差距可以忽视，但是这种训练方式这是一个很棒的idea，其收敛速度是其他以bert为基础为改造的模型不能比的，在序列建模就有非常重大的研究意义，欢迎一起交流。

4、参考文献

[1]https://openreview.net/forum?id=r1xMH1BtvB
[2]https://zhuanlan.zhihu.com/p/104956125

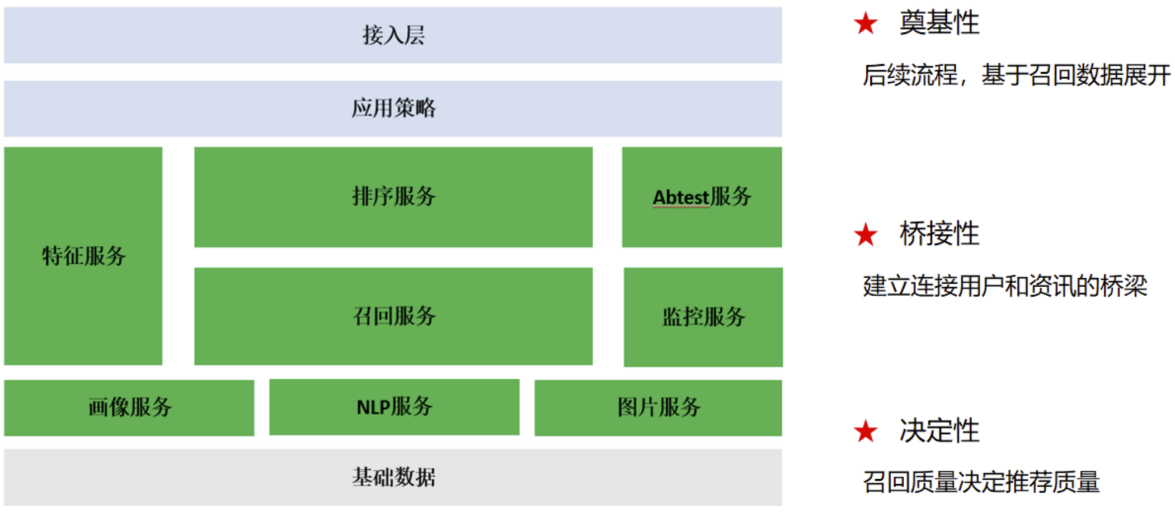
召回算法实践总结

作者：minwxwang，腾讯 PCG 应用研究员

近两年来，在移动腾讯网落地了许多召回算法，绝大多数对业务指标带来了不小的提升，趁着假期闲宅无事，泡壶好茶，倚窗听雨，顺便做点总结。一方面希望得到大神们的指点，另一方面也希望帮助到刚好有需要的同学，如果能起到抛砖引玉的作用更是荣幸之至。

召回在推荐系统中的地位

由于我个人主要是做召回算法的，首先免不了自吹自擂一番。以移动腾讯网（xw.qq.com）新闻推荐系统为例，在整个推荐系统中召回具有奠基性，桥接性，决定性。



什么是召回

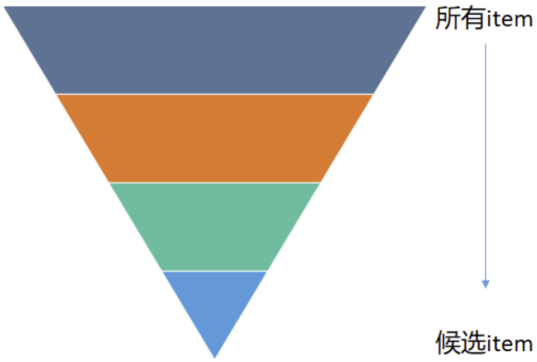
召回系统，本质上是个信息漏斗，负责快速从海量信息中筛选出有价值的信息，缩小排序算法的搜素范围（解决了信息过载的问题）；也负责将多路召回的数据，进行融合（相当于一个信息融合器，解决了单路召回特征单一，信息量小，多样性差的问题），得到一个精简的候选集。



召回算法之初体验

如果用一副通俗易懂的图来解释召回算法做了哪些事情的话，就是下面这幅图。

①去芜存菁，缩小搜索范围



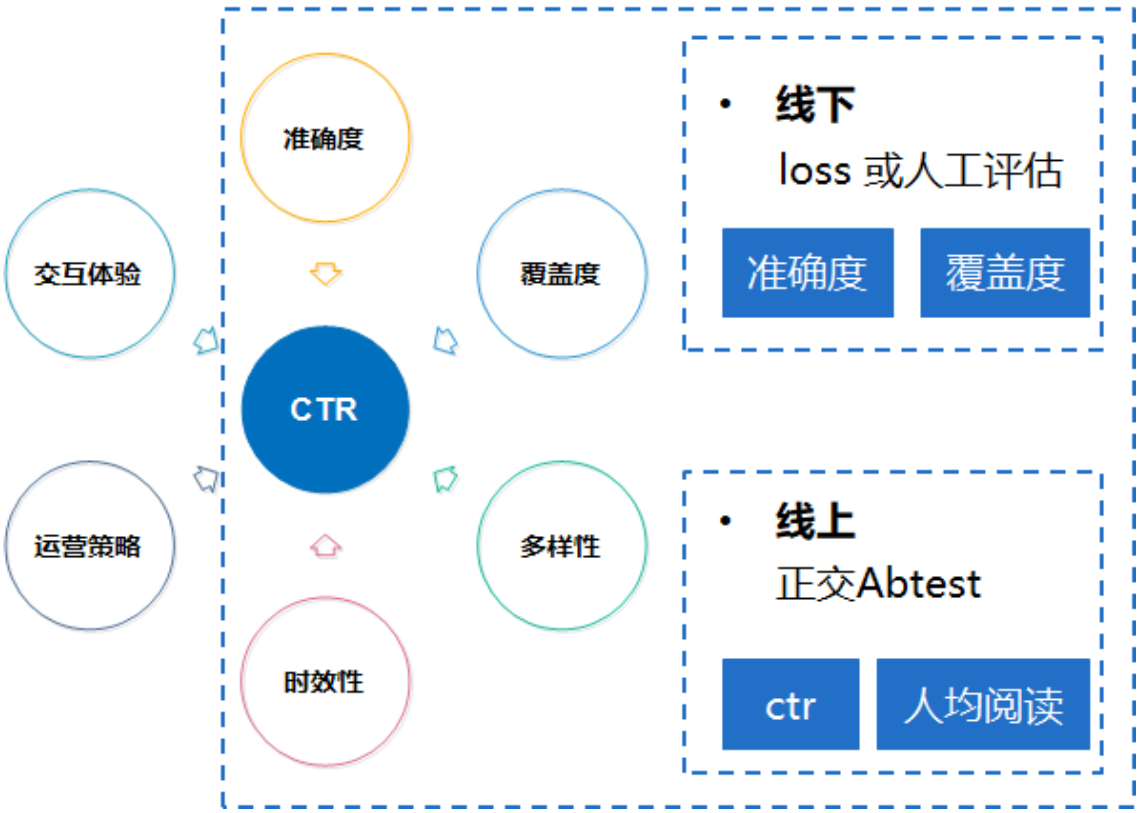
②以偏窥全，通过局部拟合整体



怎么衡量一个召回系统的好坏

召回算法的衡量虽然有一些通用的衡量指标，比如准确度，覆盖度
F1等，但是在实际应用中，一般会和自己的应用场景和业务指标挂钩，通过一些更贴合实际的指标去衡量，这里暂以移动腾讯网为例。

■ 召回的效果评估



已经落地的召回算法

我们的应用场景主要是图文推荐（详情页相关推荐和列表页个性化推荐，目前也在做视频推荐和图文视频混推），

应用场景

■ 图文推荐



推荐页卡

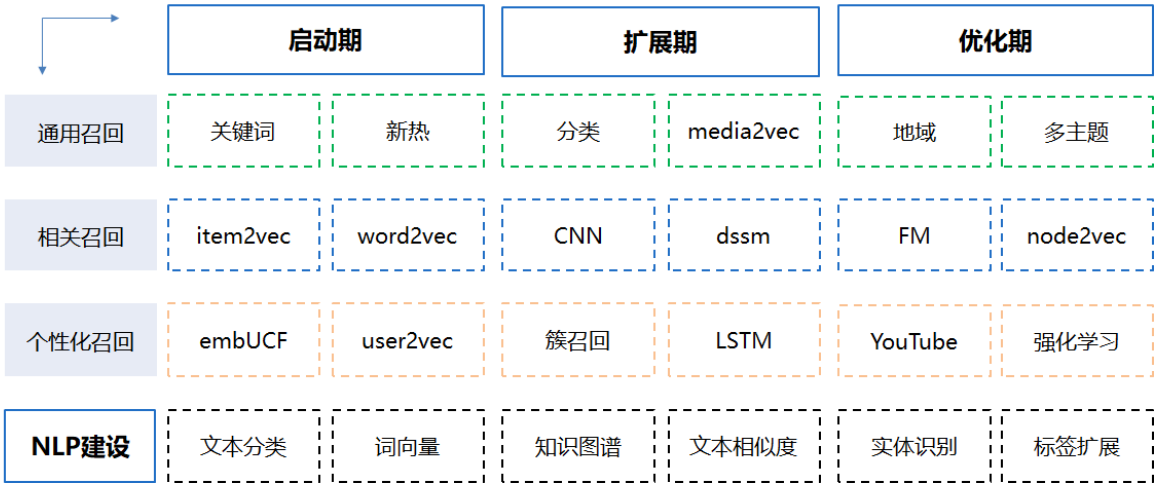
- 流量入口
- 强个性化

底层页精彩推荐

- 二次消费
- 内容强相关

由于是新闻推荐，而且是从零到一实现整个推荐系统，因此我们的推荐和NLP一开始就密切的联系在了一起，并且一直伴随着我们的持续优化过程。从开始到现在，按照时间顺序，依次实现了如下算法：

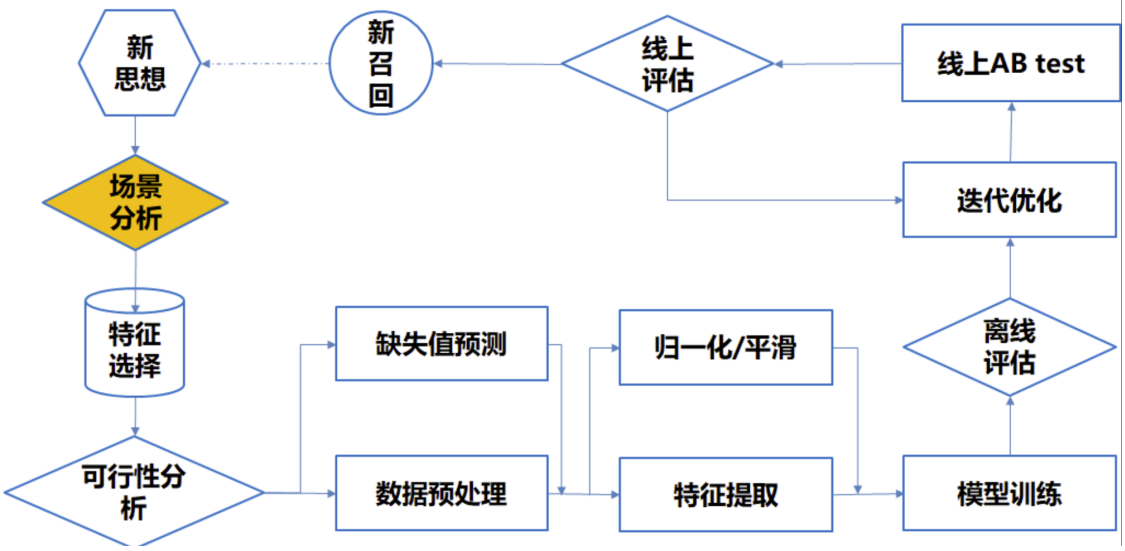
多路召回演进



召回算法的缘起缘灭

一路有效的召回算法从想法的产生，到最后上线并起作用大致要历经一下各态

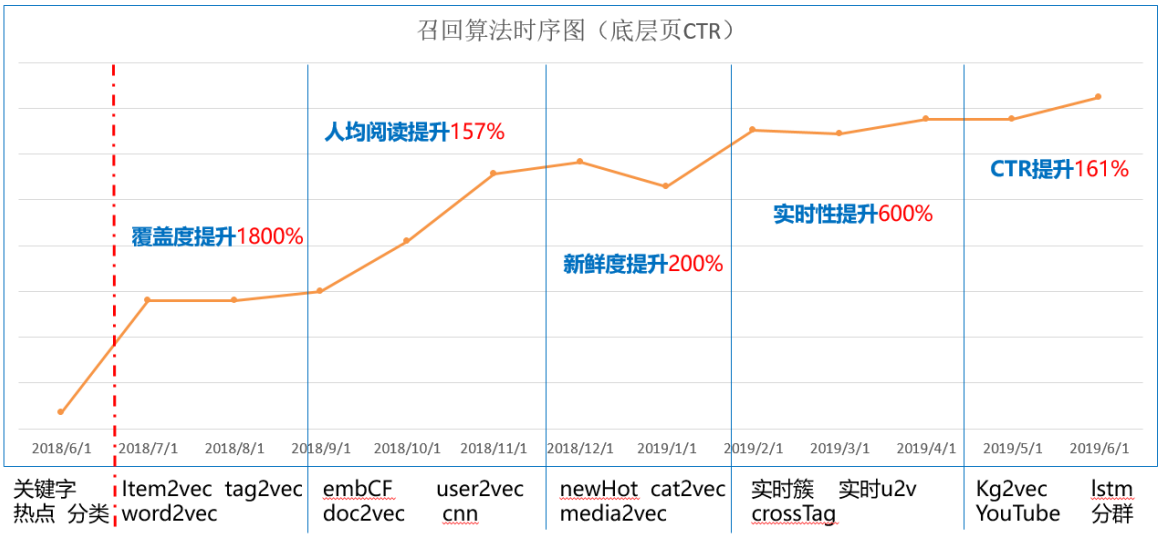
召回算法数据处理流程



算法之美

召回算法，为移动腾讯网业务指标的持续提升发挥了重要作用，以下为分阶段的提升效果（在实际应用时各个阶段并非完全串行的，也存在一些并行和交叉的情况，为了更加清晰的表达，做了一下并不完全合理的划分）

召回算法带动业务指标提升



结束语

由于我们是图文类型的新闻推荐，NLP和召回算法相伴而生，在做召回的过程中也完成了一些基本的NLP基础建设，具体如下：

NLP基础建设小结



在做召回算法的过程中，个人也总结了一些经验：个人认为贴合业务场景的可落地的创新想法，比盲目追求paper
上的模型更有效，自己能够全流程控制的可解释的模型比盲目使用一些黑盒模型要好，再就是如果能跨行业融汇贯通，把生物学，进化论，自动控制，哲学上的一些理论，活学活用到推荐领域也会起到意想不到的作用。



各大语言

Go 为什么这么“快”

作者：joellwang，腾讯 CSIG 后台开发工程师

> 本文主要介绍了 Go 程序为了实现极高的并发性能，其内部调度器的实现架构（G-P-M 模型），以及为了最大限度利用计算资源，Go 调度器是如何处理线程阻塞的场景。

怎么让我们的系统更快

随着信息技术的迅速发展，单台服务器处理能力越来越强，迫使编程模式由从前的串行模式升级到并发模型。

并发模型包含 IO 多路复用、多进程以及多线程，这几种模型都各有优劣，现代复杂的高并发架构大多是几种模型协同使用，不同场景应用不同模型，扬长避短，发挥服务器的最大性能。

而**多线程，因为其轻量和易用**，成为并发编程中使用频率最高的并发模型，包括后衍生的协程等其他子产品，也都基于它。

并发 ≠ 并行

并发 (concurrency) 和 并行 (parallelism) 是不同的。

在单个 CPU 核上，线程通过时间片或者让出控制权来实现任务切换，达到 "同时" 运行多个任务的目的，这就是所谓的并发。但实际上任何时刻都只有一个任务被执行，其他任务通过某种算法来排队。

多核 CPU 可以让同一进程内的 "多个线程" 做到真正意义上的同时运行，这才是并行。

进程、线程、协程

进程：进程是系统进行资源分配的基本单位，有独立的内存空间。

线程：线程是 CPU 调度和分派的基本单位，线程依附于进程存在，每个线程会共享父进程的资源。

协程：**协程是一种用户态的轻量级线程，**协程的调度完全由用户控制，协程间切换只需要保存任务的上下文，没有内核的开销。

线程上下文切换

由于中断处理，多任务处理，用户态切换等原因会导致 CPU 从一个线程切换到另一个线程，切换过程需要保存当前进程的状态并恢复另一个进程的状态。

上下文切换的代价是高昂的，因为在核心上交换线程会花费很多时间。上下文切换的延迟取决于不同的因素，大概在在 50 到 100 纳秒之间。考虑到硬件平均在每个核心上每纳秒执行 12 条指令，那么一次上下文切换可能会花费 600 到 1200 条指令的延迟时间。实际上，上下文切换占用了大量程序执行指令的时间。

如果存在**跨核上下文切换**（Cross-Core Context Switch），可能会导致 CPU 缓存失效（CPU 从缓存访问数据的成本大约 3 到 40 个时钟周期，从主存访问数据的成本大约 100 到 300 个时钟周期），这种场景的切换成本会更加昂贵。

Golang 为并发而生

Golang 从 2009 年正式发布以来，依靠其极高运行速度和高效的开发效率，迅速占据市场份额。
Golang 从语言级别支持并发，通过轻量级协程 Goroutine 来实现程序并发运行。

Goroutine 非常轻量，主要体现在以下两个方面：

上下文切换代价小： Goroutine 上下文切换只涉及到三个寄存器（PC / SP / DX）的值修改；而对比线程的上下文切换则需要涉及模式切换（从用户态切换到内核态）、以及 16 个寄存器、PC、SP...等寄存器的刷新；

内存占用少：线程栈空间通常是 2M，Goroutine 栈空间最小 2K；

Golang 程序中可以轻松支持**10w 级别**的 Goroutine 运行，而线程数量达到 1k 时，内存占用就已经达到 2G。

Go 调度器实现机制

Go 程序通过调度器来调度**Goroutine 在内核线程上执行，**但是 G

- *Goroutine*并不直接绑定 OS 线程 M - *Machine*运行，而是由 Goroutine Scheduler 中的 P - *Processor* （逻辑处理器）来作获取内核线程资源的『中介』。

Go 调度器模型我们通常叫做**G-P-M 模型**，他包括 4 个重要结构，分别是**G、P、M、Sched**：

G:Goroutine，每个 Goroutine 对应一个 G 结构体，G 存储 Goroutine 的运行堆栈、状态以及任务函数，可重用。

G 并非执行体，每个 G 需要绑定到 P 才能被调度执行。

P:
Processor，表示逻辑处理器，对 G 来说，P 相当于 CPU 核，G 只有绑定到 P 才能被调度。对 M 来说，P 提供了相关的执行环境(Context)，如内存分配状态(mcache)，任务队列(G)等。

P 的数量决定了系统内最大可并行的 G 的数量（前提：物理 CPU 核数 >= P 的数量）。

P 的数量由用户设置的 GoMAXPROCS 决定，但是不论 GoMAXPROCS 设置为多大，P 的数量最大为 256。

M:
Machine，OS 内核线程抽象，代表着真正执行计算的资源，在绑定有效的 P 后，进入 schedule 循环；而 schedule 循环的机制大致是从 Global 队列、P 的 Local 队列以及 wait 队列中获取。

M 的数量是不定的，由 Go
Runtime 调整，为了防止创建过多 OS 线程导致系统调度不过来，目前默认最大限制为 10000 个。

M 并不保留 G 状态，这是 G 可以跨 M 调度的基础。

Sched：Go 调度器，它维护有存储 M 和 G 的队列以及调度器的一些状态信息等。

调度器循环的机制大致是从各种队列、P 的本地队列中获取 G，切换到 G 的执行栈上并执行 G 的函数，调用 Goexit 做清理工作并回到 M，如此反复。

理解 M、P、G 三者的关系，可以通过经典的地鼠推车搬砖的模型来说明其三者关系：



地鼠(Gopher)的工作任务是：工地上有若干砖头，地鼠**借助小车**把砖头运送到火种上去烧制。**M 就可以看作图中的地鼠，P 就是小车，G 就是小车里装的砖。**

弄清楚了它们三者的关系，下面我们就开始重点聊地鼠是如何在搬运砖块的。

Processor（P）：

根据用户设置的 **GoMAXPROCS** 值来创建一批小车(P)。

Goroutine(G)：

通过 Go 关键字就是用来创建一个 Goroutine，也就相当于制造一块砖(G)，然后将这块砖(G)放入当前这辆小车(P)中。

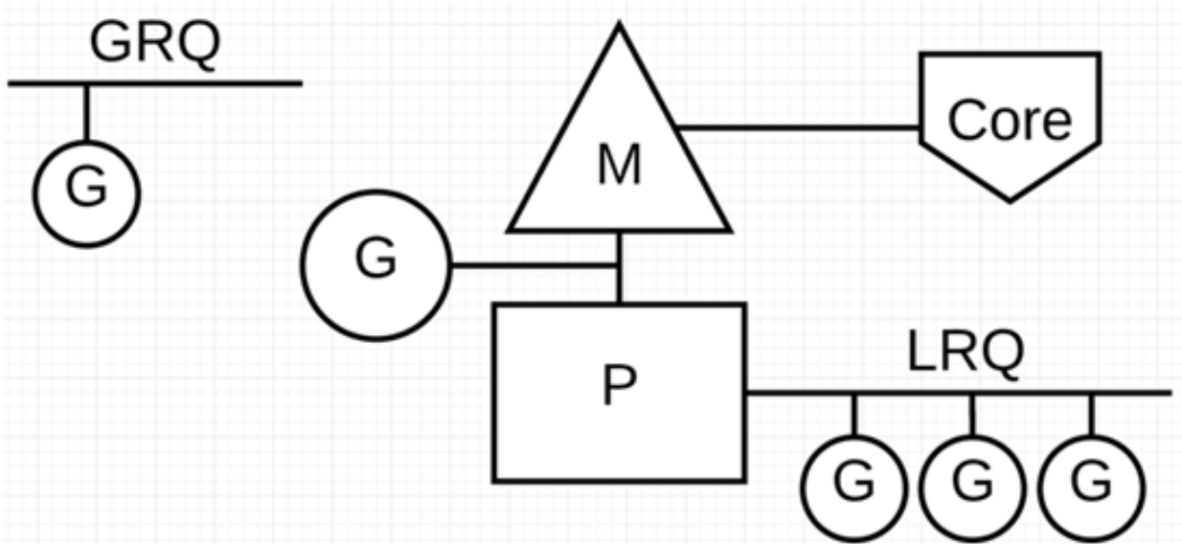
Machine (M)：

地鼠(M)不能通过外部创建出来，只能砖(G)太多了，地鼠(M)又太少了，实在忙不过来，**刚好还有空闲的小车(P)没有使用**，那就从别处再借些地鼠(M)过来直到把小车(P)用完为止。

这里有一个地鼠(M)不够用，从别处借地鼠(M)的过程，这个过程就是创建一个内核线程(M)。

需要注意的是：地鼠(M) 如果没有小车(P)是没办法运砖的，**小车(P)的数量决定了能够干活的地鼠(M)数量**，在 Go 程序里面对应的是活动线程数；

在 Go 程序里我们通过下面的图示来展示 G-P-M 模型：



P 代表可以“并行”运行的逻辑处理器，每个 P 都被分配到一个系统线程 M，G 代表 Go 协程。

Go 调度器中有两个不同的运行队列：**全局运行队列(GRQ)和本地运行队列(LRQ)。**

每个 P 都有一个 LRQ，用于管理分配给在 P 的上下文中执行的 Goroutines，这些 Goroutine 轮流被和 P 绑定的 M 进行上下文切换。GRQ 适用于尚未分配给 P 的 Goroutines。

从上图可以看出，G 的数量可以远远大于 M 的数量，换句话说，Go 程序可以利用少量的内核级线程来支撑大量 Goroutine 的并发。多个 Goroutine 通过用户级别的上下文切换来共享内核线程 M 的计算资源，但对于操作系统来说并没有线程上下文切换产生的性能损耗。

为了更加充分利用线程的计算资源，Go 调度器采取了以下几种调度策略：

任务窃取（work-stealing）

我们知道，现实情况有的 Goroutine 运行的快，有的慢，那么势必肯定会带来的问题就是，忙的忙死，闲的闲死，Go 肯定不允许摸鱼的 P 存在，势必要充分利用好计算资源。

为了提高 Go 并行处理能力，调高整体处理效率，当每个 P 之间的 G 任务不均衡时，调度器允许从 GRQ，或者其他 P 的 LRQ 中获取 G 执行。

减少阻塞

如果正在执行的 Goroutine 阻塞了线程 M 怎么办？P 上 LRQ 中的 Goroutine 会获取不到调度么？

在 Go 里面阻塞主要分为一下 4 种场景：

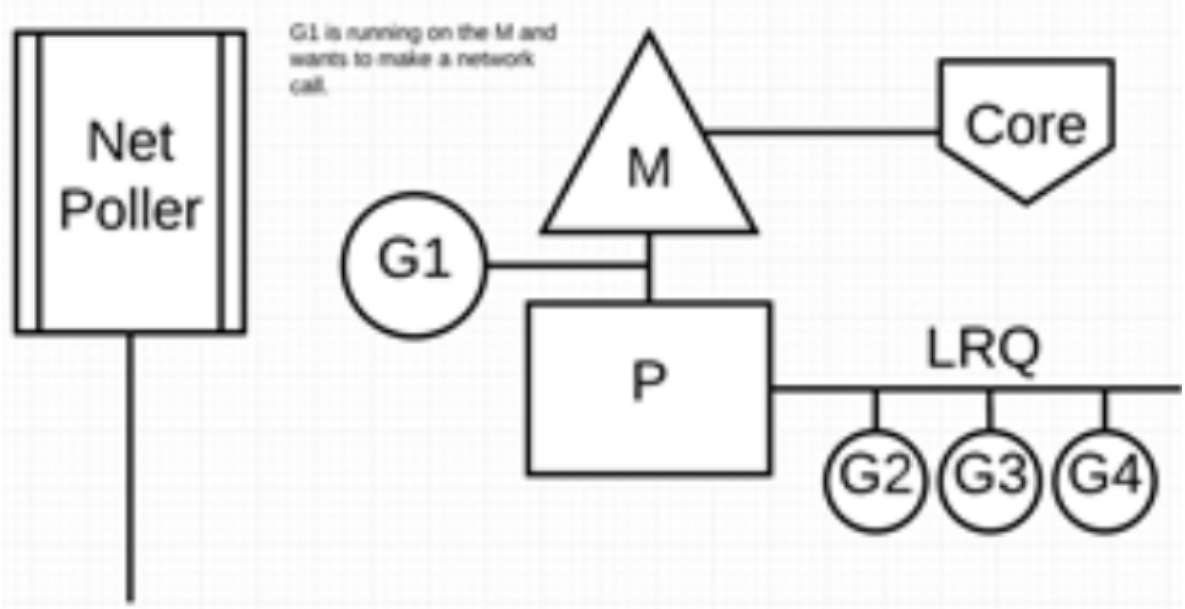
场景 1：由于原子、互斥量或通道操作调用导致 Goroutine 阻塞，调度器将把当前阻塞的 Goroutine 切换出去，重新调度 LRQ 上的其他 Goroutine；

场景 2：由于网络请求和 IO 操作导致 Goroutine 阻塞，这种阻塞的情况下，我们的 G 和 M 又会怎么做呢?

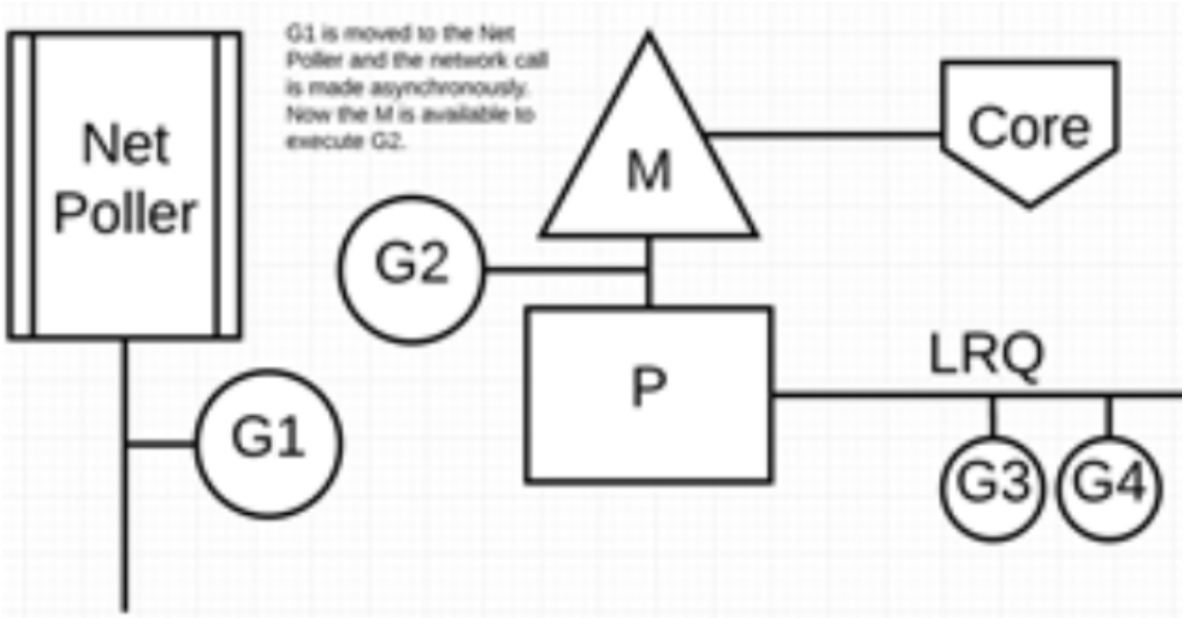
Go 程序提供了**网络轮询器（NetPoller）**来处理网络请求和 IO 操作的问题，其后台通过 kqueue（MacOS），epoll（Linux）或 iocp（Windows）来实现 IO 多路复用。

通过使用 NetPoller 进行网络系统调用，调度器可以防止 Goroutine 在进行这些系统调用时阻塞 M。这可以让 M 执行 P 的 LRQ 中其他的 Goroutines，而不需要创建新的 M。有助于减少操作系统上的调度负载。

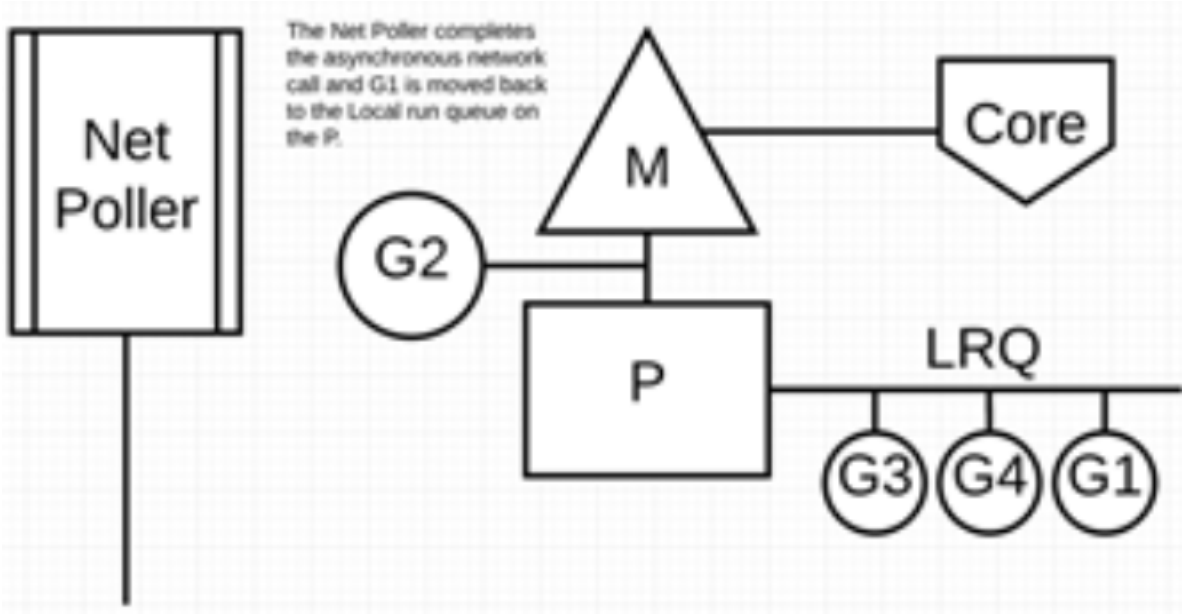
下图展示它的工作原理：G1 正在 M 上执行，还有 3 个 Goroutine 在 LRQ 上等待执行。网络轮询器空闲着，什么都没干。



接下来，G1 想要进行网络系统调用，因此它被移动到网络轮询器并且处理异步网络系统调用。然后，M 可以从
LRQ 执行另外的 Goroutine。此时，G2 就被上下文切换到 M 上了。



最后，异步网络系统调用由网络轮询器完成，G1 被移回到 P 的 LRQ 中。一旦 G1 可以在 M 上进行上下文切换，它负责的 Go 相关代码就可以再次执行。这里的最大优势是，执行网络系统调用不需要额外的 M。网络轮询器使用系统线程，它时刻处理一个有效的事件循环。

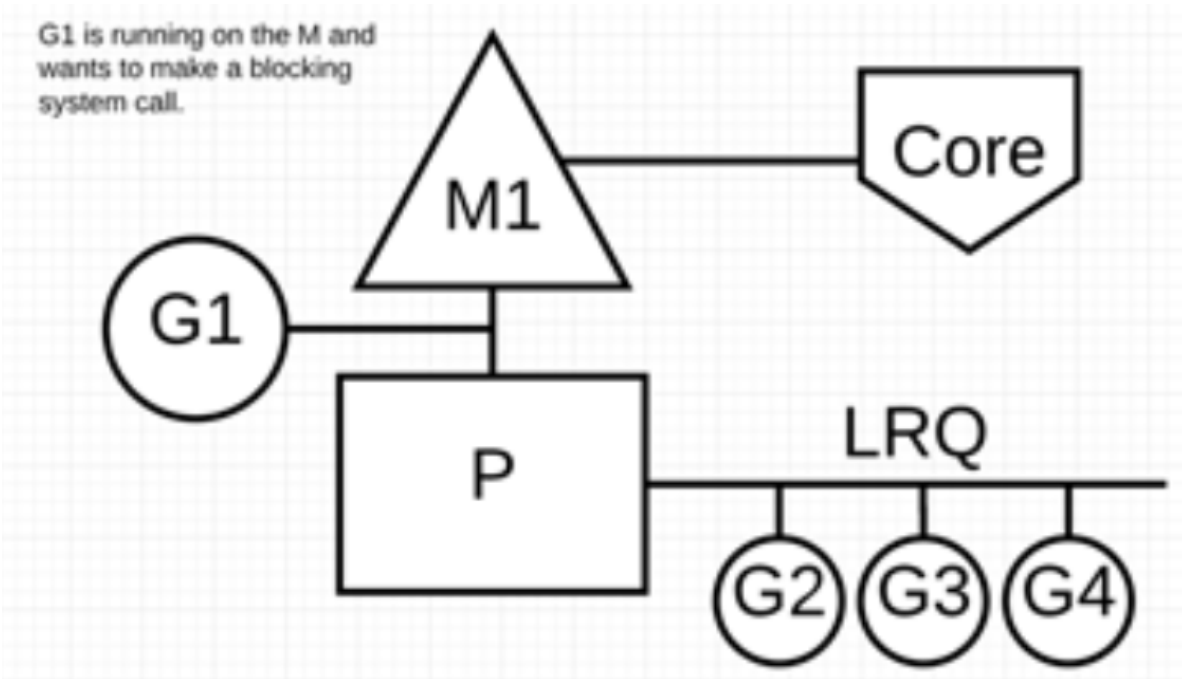


这种调用方式看起来很复杂，值得庆幸的是，**Go 语言将该"复杂性"隐藏在 Runtime 中**：Go 开发者无需关注 socket 是否是 non-block 的，也无需亲自注册文件描述符的回调，只需在每个连接对应的 Goroutine 中以"block I/O"的方式对待 socket 处理即可，**实现了 goroutine-per-connection 简单的网络编程模式**（但是大量的 Goroutine 也会带来额外的问题，比如栈内存增加和调度器负担加重）。

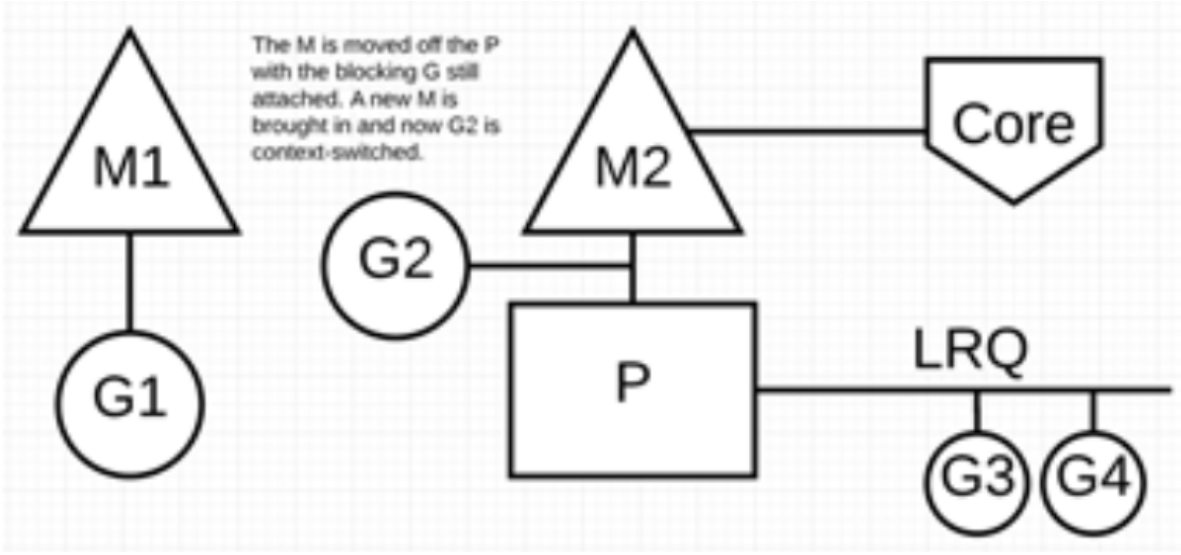
用户层眼中看到的 Goroutine 中的"block socket"，实际上是通过 Go runtime 中的 netpoller 通过 Non-block socket +
I/O 多路复用机制"模拟"出来的。Go 中的 net 库正是按照这方式实现的。

场景 3：当调用一些系统方法的时候，如果系统方法调用的时候发生阻塞，这种情况下，网络轮询器（NetPoller）无法使用，而进行系统调用的 Goroutine 将阻塞当前 M。

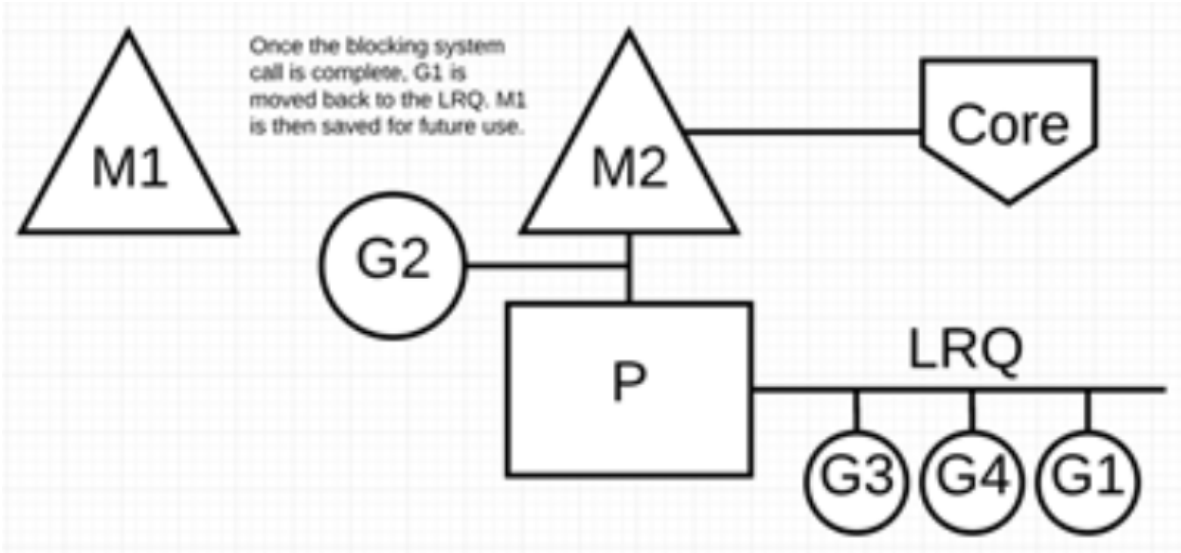
让我们来看看同步系统调用（如文件 I/O）会导致 M 阻塞的情况：G1 将进行同步系统调用以阻塞 M1。



调度器介入后：识别出 G1 已导致 M1 阻塞，此时，调度器将 M1 与 P 分离，同时也将 G1 带走。然后调度器引入新的 M2 来服务 P。此时，可以从 LRQ 中选择 G2 并在 M2 上进行上下文切换。



阻塞的系统调用完成后：G1 可以移回 LRQ 并再次由 P 执行。如果这种情况再次发生，M1 将被放在旁边以备将来重复使用。



场景 4：如果在 Goroutine 去执行一个 sleep 操作，导致 M 被阻塞了。

Go 程序后台有一个监控线程 sysmon，它监控那些长时间运行的 G 任务然后设置可以强占的标识符，别的 Goroutine 就可以抢先进来执行。

只要下次这个 Goroutine 进行函数调用，那么就会被强占，同时也会保护现场，然后重新放入 P 的本地队列里面等待下次执行。

小结

本文主要从 Go 调度器架构层面上介绍了 G-P-M 模型，通过该模型怎样实现少量内核线程支撑大量 Goroutine 的并发运行。以及通过 NetPoller、sysmon 等帮助 Go 程序减少线程阻塞，充分利用已有的计算资源，从而最大限度提高 Go 程序的运行效率。

参考文档：

https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part1.html

https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html

https://www.ardanlabs.com/blog/2018/12/scheduling-in-go-part3.html

https://segmentfault.com/a/1190000016038785

https://segmentfault.com/a/1190000016611742

https://segmentfault.com/a/1190000017333717

https://segmentfault.com/a/1190000015352983

https://segmentfault.com/a/1190000015464889

https://www.cnblogs.com/lxmhhy/p/6041001.html

https://www.cnblogs.com/mokafamily/p/9975980.html

https://studyGolang.com/articles/9211

https://www.zhihu.com/question/20862617

https://codeburst.io/why-Goroutines-are-not-lightweight-threads-7c460c1f155f

https://blog.csdn.net/tiandyoin/article/details/76556702

https://www.jianshu.com/p/cc3c0fefee43

https://www.jianshu.com/p/a315224886d2

Golang 性能优化实战

作者： trumanyan，腾讯 CSIG 后台开发工程师

项目背景

网关服务作为统一接入服务，是大部分服务的统一入口。为了避免成功瓶颈，需要对其进行尽可能地优化。因此，特别总结一下 golang 后台服务性能优化的方式，并对网关服务进行优化。

技术背景：

- 基于 tarsgo 框架的 http 接入服务，下游服务使用 tarsgo 协议进行交互

性能指标

网关服务本身没有业务逻辑处理，仅作为统一入口进行请求转发，因此我们主要关注下列指标

- 吞吐量：每秒钟可以处理的请求数
- 响应时间：从客户端发出请求，到收到回包的总耗时

定位瓶颈

一般后台服务的瓶颈主要为 CPU，内存，IO 操作中的一个或多个。若这三者的负载都不高，但系统吞吐量低，基本就是代码逻辑出问题了。

在代码正常运行的情况下，我们要针对某个方面的高负载进行优化，才能提高系统的性能。golang 可通过 benchmark 加 pprof 来定位具体的性能瓶颈。

benchmark 简介

```
go test -v gate_test.go -run=none -bench=. -benchtime=3s -cpuprofile cpu.prof -
memprofile mem.prof
```

- -run 知道单次测试，一般用于代码逻辑验证
- -bench=. 执行所有 Benchmark，也可以通过用例函数名来指定部分测试用例
- -benchtime 指定测试执行时长
- -cpuprofile 输出 cpu 的 pprof 信息文件
- -memprofile 输出 heap 的 pprof 信息文件。
- -blockprofile 阻塞分析，记录 goroutine 阻塞等待同步（包括定时器通道）的位置
- -mutexprofile 互斥锁分析，报告互斥锁的竞争情况

benchmark 测试用例常用函数

- b.ReportAllocs() 输出单次循环使用的内存数量和对象 allocs 信息
- b.RunParallel() 使用协程并发测试
- b.SetBytes(n int64) 设置单次循环使用的内存数量

pprof 简介

生成方式

- **runtime/pprof**: 手动调用如`runtime.StartCPUProfile`或者`runtime.StopCPUProfile`等 API 来生成和写入采样文件，灵活性高。主要用于本地测试。
- **net/http/pprof**: 通过 http 服务获取 Profile 采样文件，简单易用，适用于对应用程序的整体监控。通过 runtime/pprof 实现。主要用于服务器端测试。
- **go test**: 通过`go test -bench . -cpuprofile cpuprofile.out`生成采样文件，主要用于本地基准测试。可用于重点测试某些函数。

查看方式

- go tool pprof [options][binary] ...
  - --text 纯文本
  - --web 生成 svg 并用浏览器打开（如果 svg 的默认打开方式是浏览器）
  - --svg 只生成 svg
  - --list funcname 筛选出正则匹配 funcname 的函数的信息
  - **-http=":port" 直接本地浏览器打开 profile 查看（包括 top，graph，火焰图等）**
- go tool pprof -base profile1 profile2
  - 对比查看 2 个 profile，一般用于代码修改前后对比，定位差异点。
- 通过命令行方式查看 profile 时，可以在命令行对话中，使用下列命令，查看相关信息
  - topN [-cum] 查看前 N 个数据：
    - ```
      flat  flat%   sum%        cum   cum%
       5.95s 27.56% 27.56%      5.95s 27.56%  runtime.usleep
       4.97s 23.02% 50.58%      5.08s 23.53%  sync.(*RWMutex).RLock
       4.46s 20.66% 71.24%      4.46s 20.66%  sync.(*RWMutex).RUnlock
       2.69s 12.46% 83.70%      2.69s 12.46%  runtime.pthread_cond_wait
       1.50s  6.95% 90.64%      1.50s  6.95%  runtime.pthread_cond_signal
      ```
      - flat: 采样时，该函数正在运行的次数*采样频率(10ms)，即得到估算的函数运行”采样时间”。这里不包括函数等待子函数返回。
      - flat%: flat / 总采样时间值
      - sum%: 前面所有行的 flat% 的累加值，如第三行 sum% = 71.24% = 27.56% + 50.58%
      - cum: 采样时，该函数出现在调用堆栈的采样时间，包括函数等待子函数返回。因此 flat <= cum
      - cum%: cum / 总采样时间值
      ```

- list funcname 查看某个函数的详细信息，可以明确具体的资源（cpu，内存等）是由哪一行触发的。

pprof 接入 tarsgo

1. 服务中 main 方法插入代码

```
cfg := tars.GetServerConfig()
profMux := &tars.TarsHttpMux{}
profMux.HandleFunc("/debug/pprof/", pprof.Index)
profMux.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
profMux.HandleFunc("/debug/pprof/profile", pprof.Profile)
profMux.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
profMux.HandleFunc("/debug/pprof/trace", pprof.Trace)
tars.AddHttpServant(profMux, cfg.App+"."+cfg.Server+".ProfObj")
```

2. taf 管理平台中，添加 servant：ProfObj （名字可自己修改）
3. 发布服务

查看 tasrgo 服务的 pprof

1. 保证开发机能直接访问到 tarsgo 节点部署的 ip 和 port。
2. 查看 profile(http 地址中的 ip,port 为 ProfObj 的 ip 和 port)

```
# 下载cpu profile
go tool pprof http://ip:port/debug/pprof/profile?seconds=120 # 等待120s，不带此参数时等待30s

# 下载heap profile
go tool pprof http://ip:port/debug/pprof/heap

# 下载goroutine profile
go tool pprof http://ip:port/debug/pprof/goroutine

# 下载block profile
go tool pprof http://ip:port/debug/pprof/block

# 下载mutex profile
go tool pprof http://ip:port/debug/pprof/mutex

# 下载20秒的trace记录（遇到棘手问题时，查看trace会比较容易定位）
 curl http://100.97.1.35:10078/debug/pprof/trace?seconds=20 > trace.out
 go tool trace trace.out 查看
```

3. 直接在终端中通过 pprof 命令查看
4. sz 上面命令执行时出现的`Saved profile in /root/pprof/pprof.binary.alloc_objects.xxxxxxx.xxxx.pb.gz`到本地
5. 在本地环境，执行`go tool pprof -http=":8081" pprof.binary.alloc_objects.xxxxxxx.xxxx.pb.gz`即可直接通过http://localhost:8081页面查看。包括topN，火焰图信息等,会更方便一点。

GC Trace

golang 具备 GC 功能，而 GC 是最容易被忽视的性能影响因素。尤其是在本地使用 benchmark 测试时，由于时间较短，占用内存较少。往往不会触发 GC。而一旦线上出现 GC 问题，又不太好定位。目前常用的定位方式有两种：

本地 gctrace

- 在执行程序前加 `GODEBUG=gctrace=1`，每次 gc 时会输出一行如下内容

```
gc 1 @0.001s 11%: 0.007+1.5+0.004 ms clock, 0.089+1.5/2.8/0.27+0.054 ms cpu,
4->4->3 MB, 5 MB goal, 12 P
scvg: inuse: 4, idle: 57, sys: 62, released: 57, consumed: 4 (MB)
```

  - gc 1 表示第 1 次 gc
  - @0.001s 表示程序执行的总时间
  - 11% 表示垃圾回收时间占用总的运行时间百分比
  - 0.007+1.5+0.004 ms clock 表示工作线程完成 GC 的 stop-the-world,sweeping,marking 和 waiting 的时间
  - 0.089+1.5/2.8/0.27+0.054 ms cpu 垃圾回收占用 cpu 时间
  - 4->4->3 MB 表示堆的大小，gc 后堆的大小，存活堆的大小
  - 5 MB goal 整体堆的大小
  - 12 P 使用的处理器数量
  - scvg: inuse: 4, idle: 57, sys: 62, released: 57, consumed: 4 (MB) 表示系统内存回收信息
    - inuse：使用了多少 M 内存
    - idle：剩下要清除的内存
    - sys：系统映射的内存
    - released：释放的系统内存
    - consumed：申请的系统内存
  - 采用图形化的方式查看：https://github.com/davecheney/gcvis

    `GODEBUG=gctrace=1 go test -v *.go -bench=. -run=none -benchtime 3m |& gcvis`

    - 也通过日志转为图形化：

```
GODEBUG=gctrace=1 godoc -index -http=:6060 2> stderr.log
cat stderr.log | gcvis
```

线上 trace

在线上业务中添加**net/http/pprof**后，可通过下列命令采集 20 秒的 trace 信息

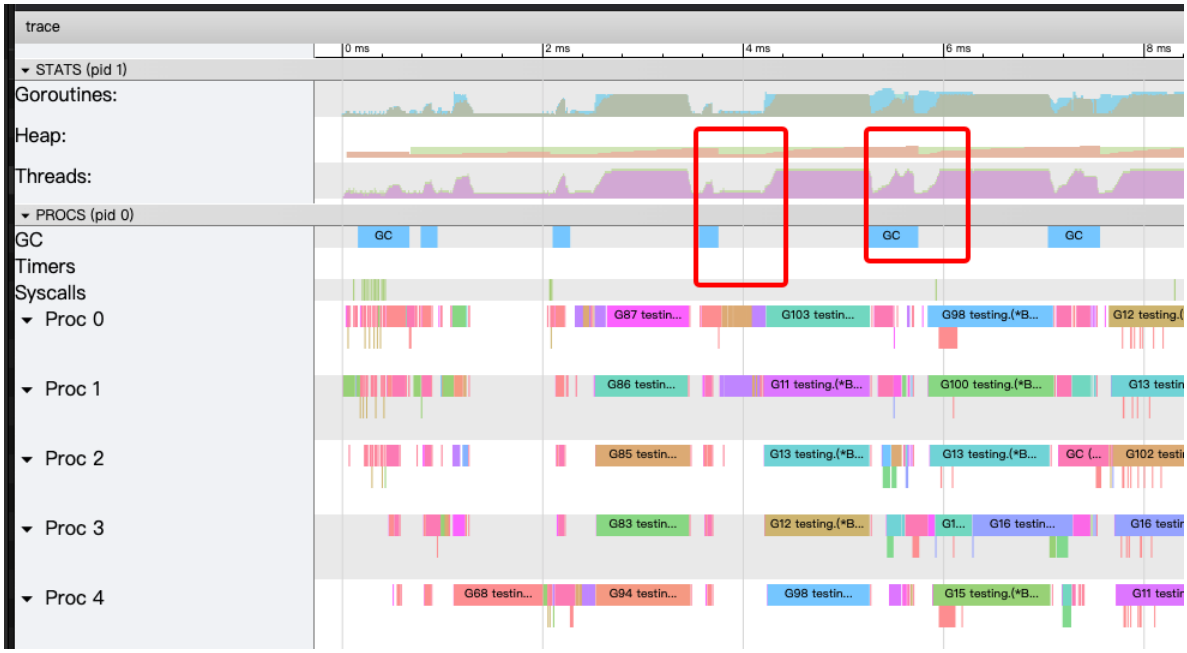
```
curl http://ip:port/debug/pprof/trace?seconds=20 > trace.out
```

再通过`go tool trace trace.out`即可在本地浏览器中查看 trace 信息。

View trace
Goroutine analysis
Network blocking profile (⬇)
Synchronization blocking profile (⬇)
Syscall blocking profile (⬇)
Scheduler latency profile (⬇)
User-defined tasks
User-defined regions
Minimum mutator utilization

- View trace：查看跟踪
- Goroutine analysis：Goroutine 分析
- Network blocking profile：网络阻塞概况
- Synchronization blocking profile：同步阻塞概况
- Syscall blocking profile：系统调用阻塞概况
- Scheduler latency profile：调度延迟概况
- User defined tasks：用户自定义任务
- User defined regions：用户自定义区域
- Minimum mutator utilization：最低 Mutator 利用率

GC 相关的信息可以在 View trace 中看到

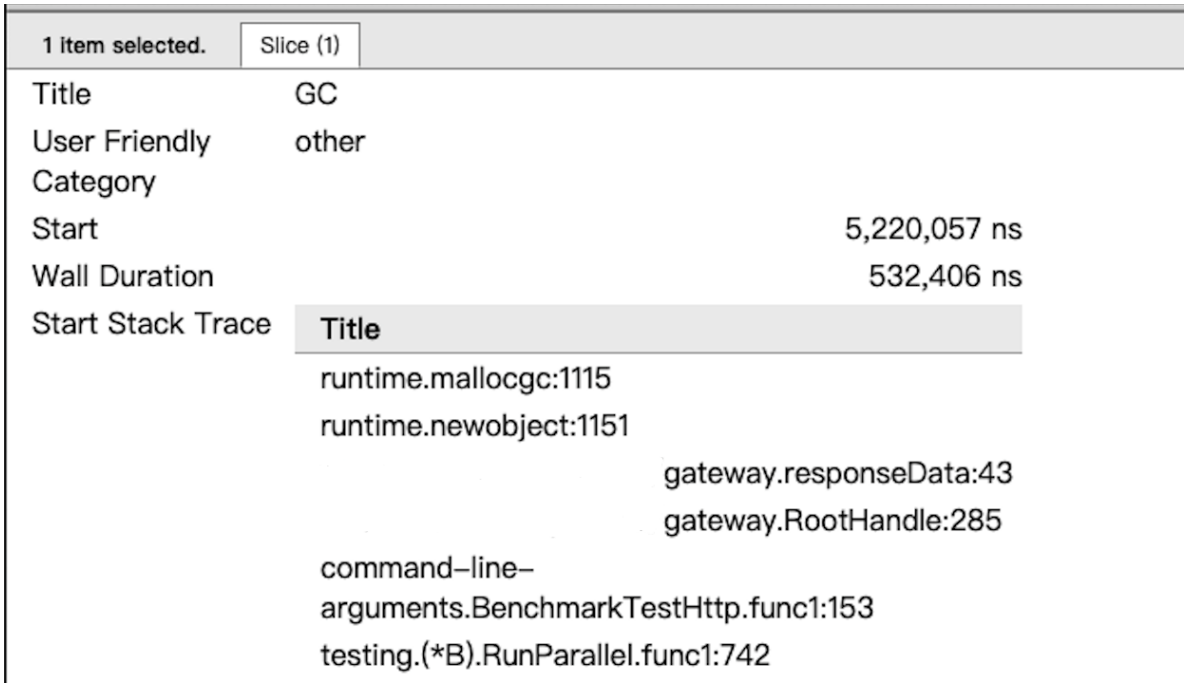


可通过点击 heap 的色块区域，查看 heap 信息。

2 items selected. | Counter Samples (2)

Counter	Series	Time	Value
Heap	NextGC	6.645042999999999	662096
Heap	Allocated	6.645042999999999	3532208

点击 GC 对应行的蓝色色块，查看 GC 耗时及相关回收信息。



通过这两个信息就可以确认是否存在 GC 问题，以及造成高 GC 的可能原因。

使用问题

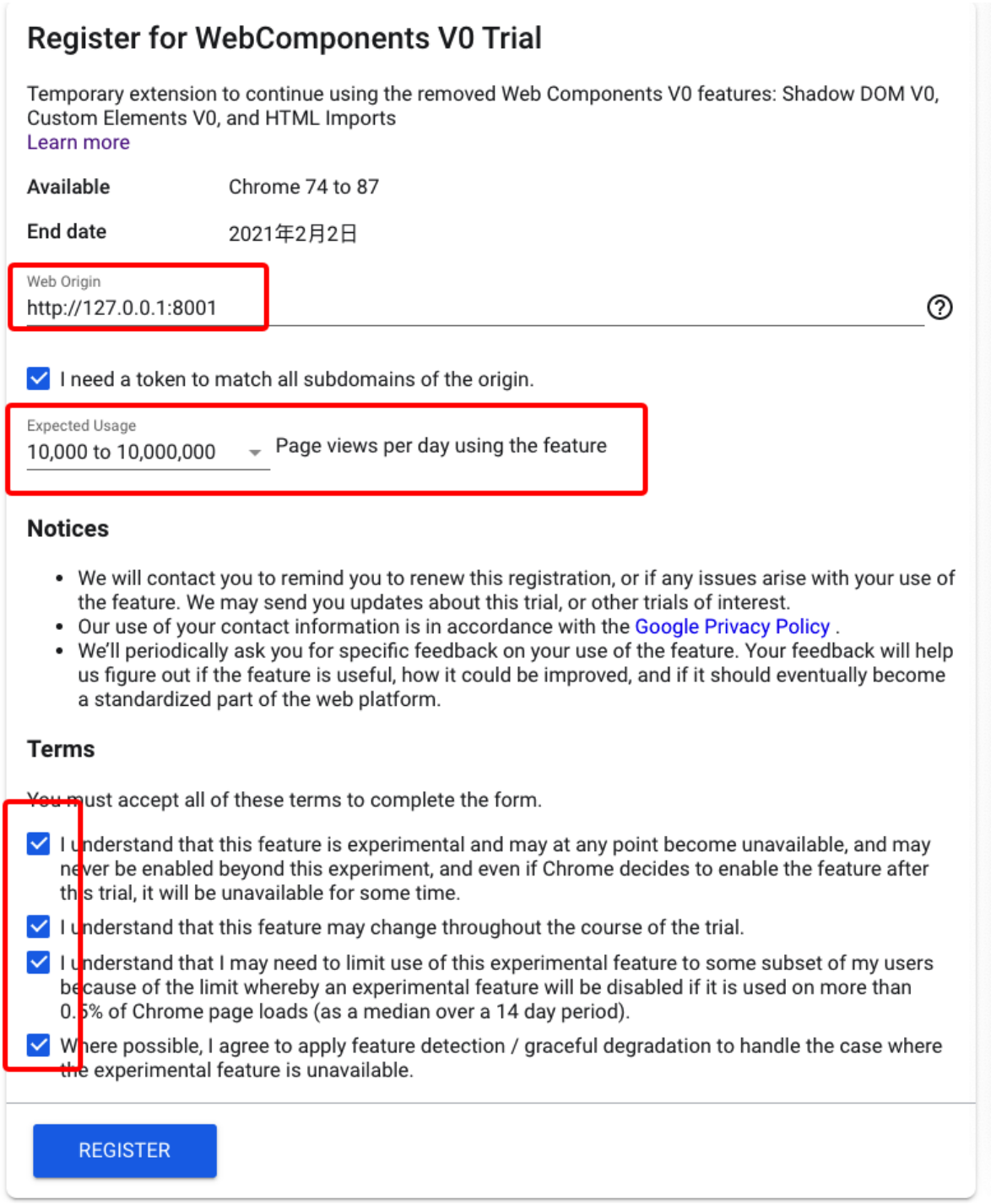
trace 的展示仅支持 chrome 浏览器。但是目前常用的 chrome 浏览器屏蔽了 go tool trace 使用的 HTML import 功能。即打开"view trace"时，会出现一片空白。并可以在 console 中看到警告信息：

```
HTML Imports is deprecated and has now been removed as of M80. See
https://www.chromestatus.com/features/5144752345317376 and
https://developers.google.com/web/updates/2019/07/web-components-time-to-upgrade
for more details.
```

解决办法

申请 token

- https://developers.chrome.com/origintrials/#/register_trial/2431943798780067841 然后登录
- web origin 处填写 http://localhost:8001 端口只能是 8000 - 8003，支持 http 和 https。（也可以填写 127.0.0.1:8001,依赖于你浏览器中显示的地址，否则对不上的话，还要手动改一下)



- 点击注册后即可看到 token

修改 trace.go

- 编辑${GOROOT}/src/cmd/trace/trace.go 文件，在文件中找到 templTrace 然后在 <head> 标签的下一行添加 `<meta http-equiv="origin-trial" content="你复制的token">`

重新编译 go

- ${GOROOT}/src 目录，执行 ./all.bash
- 若提示：`ERROR: Cannot find go1.4\bin\go Set GOROOT_BOOTSTRAP to a working Go tree >= Go 1.4` 则需要先安装一个 go1.4 的版本，再通过它来编译 go。（下载链接https://dl.google.com/go/go1.4-bootstrap-20171003.tar.gz） 在 go1.4/src 下执行./make.bash. 指定 GOROOT_BOOTSTRAP 为 go1.4 的根目录。然后就可以重新编译 go

查看 trace

```
go tool trace -http=localhost:8001 trace.out
```

若打开 view trace 还是空白,则检查一下浏览器地址栏中的地址，是否与注册时的一样。即注册用的 localhost 或 127.0.0.1 则地址栏中也要一样。

常见性能瓶颈

业务逻辑

出现无效甚至降低性能的逻辑。常见的有

- 逻辑重复：相同的操作在不同的位置做了多次或循环跳出的条件设置不当。
- 资源未复用：内存频繁申请和释放，数据库链接频繁建立和销毁等。
- 无效代码。

存储

未选择恰当的存储方式，常见的有：

- 临时数据存放到数据库中，导致频繁读写数据库。
- 将复杂的树状结构的数据用 SQL 数据库存储，出现大量冗余列，并且在读写时要进行拆解和拼接。
- 数据库表设计不当，无法有效利用索引查询，导致查询操作耗时高甚至出现大量慢查询。
- 热点数据未使用缓存，导致数据库负载过高，响应速度下降。

并发处理

并发操作的问题主要出现在资源竞争上，常见的有：

- 死锁/活锁导致大量阻塞，性能严重下降。
- 资源竞争激烈：大量的线程或协程抢夺一个锁。
- 临界区过大：将不必要的操作也放入临界区，导致锁的释放速度过慢，引起其他线程或协程阻塞。

golang 部分细节简介

在优化之前，我们需要对 golang 的实现细节有一个简单的了解，才能明白哪些地方有问题，哪些地方可以优化，以及怎么优化。以下内容的详细讲解建议查阅网上优秀的 blog。对语言的底层实现机制最好有个基本的了解，否则有时候掉到坑里都不知道为啥。

协程调度

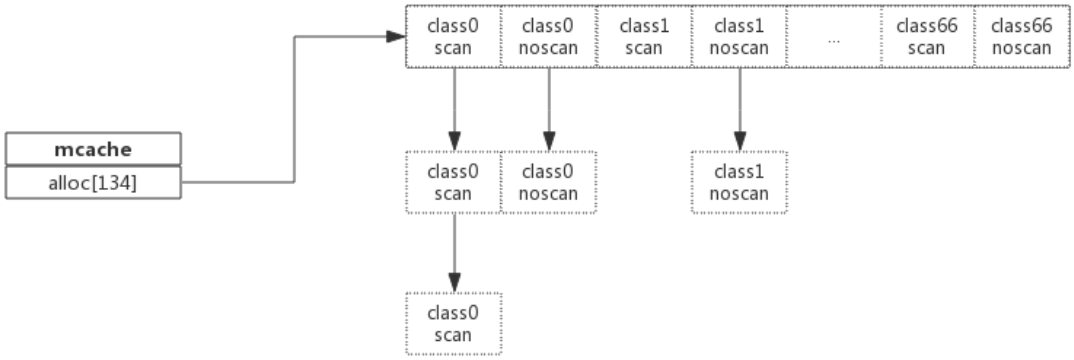
Golang 调度是非抢占式多任务处理，由协程主动交出控制权。遇到如下条件时，才有可能交出控制权

- I/O,select
- channel
- 等待锁
- 函数调用（是一个切换的机会，是否会切换由调度器决定）
- runtime.Gosched()

因此，若存在较长时间的 for 循环处理，并且循环内没有上述逻辑时，会阻塞住其他的协程调度。在实际编码中一定要注意。

内存管理

Go 为每个逻辑处理器（P）提供了一个称为**mcache**的本地内存线程缓存。每个 mcache 中持有 67 个级别的 mspan。每个 msapn 又包含两种：scan（包含指针的对象）和 noscan（不包含指针的对象）。在进行垃圾收集时，**GC 无需遍历 noscan 对象。**

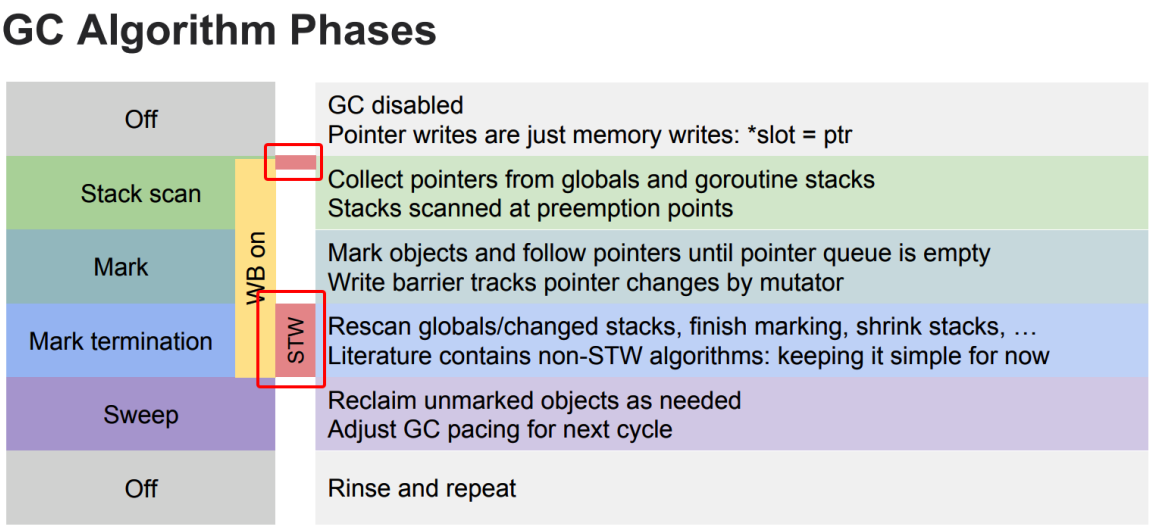


GC 处理

GC 的工作就是确定哪些内存可以释放，它是通过扫描内存查找内存分配的指针来完成这个工作的。GC 触发时机：

- 到达堆阈值：默认情况下，它将在堆大小加倍时运行，可通过 GOGC 来设定更高阈值（不建议变更此配置）
- 到达时间阈值：每两分钟会强制启动一次 GC 循环

为啥要注意 GC，是因为 GC 时出现 2 次 Stop the world，即停止所有协程，进行扫描操作。若是 GC 耗时高，则会严重影响服务器性能。



变量逃逸

> 注意，golang 中的栈是跟函数绑定的，函数结束时栈被回收。

变量内存回收：

- 如果分配在栈中，则函数执行结束可自动将内存回收；
- 如果分配在堆中，则函数执行结束可交给 GC（垃圾回收）处理；

而变量逃逸就意味着增加了堆中的对象个数，影响 GC 耗时。一般要尽量避免逃逸。

逃逸分析不变性：

1. 指向栈对象的指针不能存在于堆中；
2. 指向栈对象的指针不能在栈对象回收后存活；

在逃逸分析过程中，凡是发现出现违反上述约定的变量，就将其移到堆中。

逃逸常见的情况：

- 指针逃逸：返回局部变量的地址（不变性 2）
- 栈空间不足
- 动态类型逃逸：如 fmt.Sprintf,json.Marshel 等接受变量为...interface{}函数的调用，会导致传入的变量逃逸。
- 闭包引用

包含指针类型的底层结构

string

```
type StringHeader struct {
    Data uintptr
    Len  int
}
```

slice

```
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

map

```
type hmap struct {
    count     int
    flags     uint8
    B         uint8
    noverflow uint16
    hash0     uint32
    buckets    unsafe.Pointer
    oldbuckets unsafe.Pointer
    nevacuate  uintptr
    extra *mapextra
}
```

这些是常见会包含指针的对象。尤其是 string，在后台应用中大量出现。并经常会作为 map 的 key 或 value。若数据量较大时，就会引发 GC 耗时上升。同时，我们可以注意到 string 和 slice 非常相似，从某种意义上说它们之间是可以直接互相转换的。这就可以避免 string 和[]byte 之间类型转换时，进行内存拷贝

类型转换优化

```
func String(b []byte) string {
    return *(*string)(unsafe.Pointer(&b))
}
func Str2Bytes(s string) []byte {
    x := (*[2]uintptr)(unsafe.Pointer(&s))
    h := [3]uintptr{x[0], x[1], x[1]}
    return *(*[]byte)(unsafe.Pointer(&h))
}
```

性能测试方式

本地测试

将服务处理的核心逻辑，使用 go test 的 benchmark 加 pprof 来测试。建议上线前，就对整个业务逻辑的性能进行测试，提前优化瓶颈。

线上测试

一般 http 服务可以通过常见的测试工具进行压测，如 wrk，locust 等。taf 服务则需要我们自己编写一些测试脚本。同时，要注意的是，压测的目的是定位出服务的最佳性能，而不是盲目的高并发请求测试。因此，一般需要逐步提升并发请求数量，来定位出服务的最佳性能点。

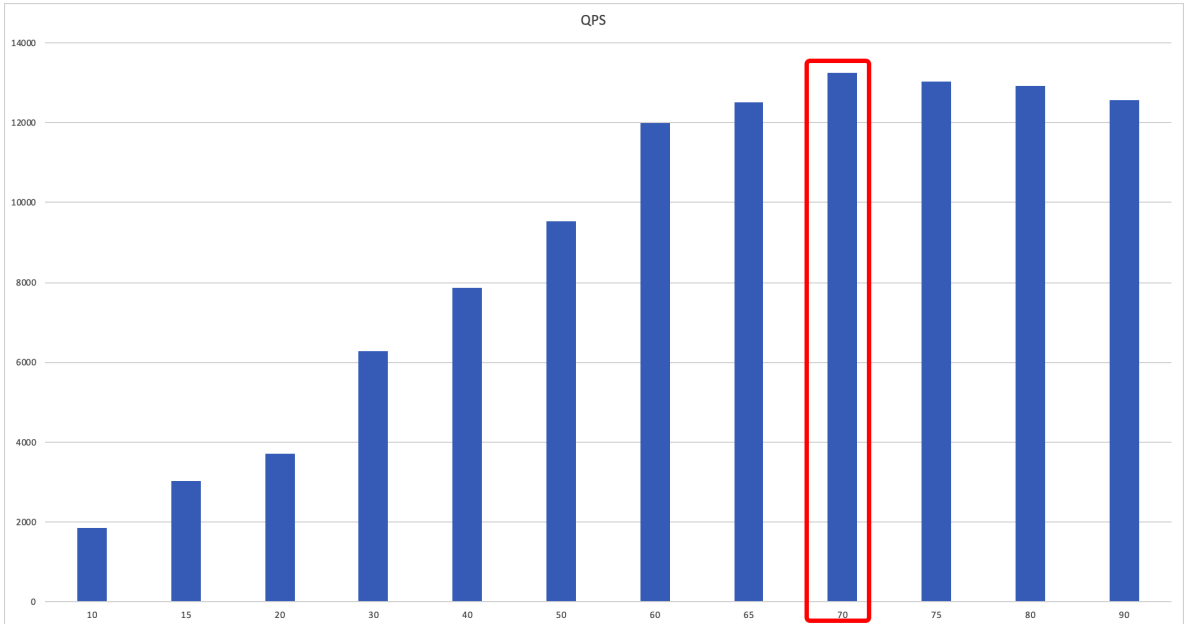
> 注意：由于 taf 平台具备扩容功能，因此为了更准确的测试，我们应该在测试前关闭要测试节点的自动扩容。

实际项目优化

为了避免影响后端服务，也为了避免后端服务影响网关自身。因此，我们需要在压测前，将对后端服务的调用屏蔽。

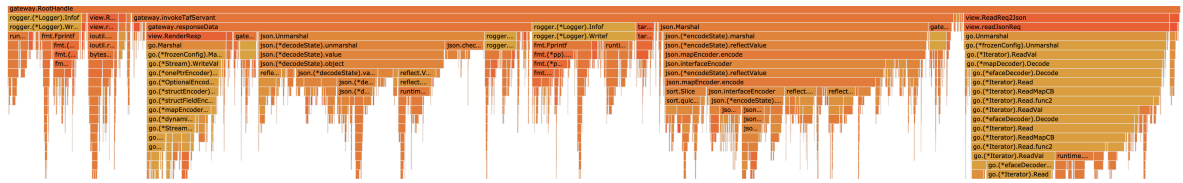
QPS 现状

首先看下当前业务的性能指标，使用 wrk 压测网关服务



可以看出，在总链接数为 70 的时候，QPS 最高，为 13245。

火焰图



根据火焰图我们定位出 cpu 占比较高的几个方法为：

- json.Marshal
- json.Unmarshal
- rogger.Infof

为了方便测试，将代码改为本地运行，并通过 benchmark 的方式来对比修改前后的差异。

> 由于正式环境使用的 golang 版本为 1.12，因此本地测试时，也要使用同样的版本。

benchmark

```
Benchmark       50000000          3669 ns/op        4601 B/op         73
allocs/op
```

查看 cpu 和 memory 的 profile，发现健康度上报的数据结构填充占比较高。这部分逻辑基于 tars 框架实现。暂时忽略，为避免影响其他测试，先注释掉。再看看 benchmark。

```
Benchmark         500000          3146 ns/op        2069 B/op         55
allocs/op
```

优化策略

JSON 优化

先查看 json 解析的部分，看看是否有优化空间

请求处理

```
//RootHandle view.ReadReq2Json readJsonReq 中进行json解析
type GatewayReqBody struct {
    Header  GatewayReqBodyHeader   `json:"header"`
    Payload map[string]interface{} `json:"payload"`
}
func readJsonReq(data []byte, req *model.GatewayReqBody) error {
    dataMap := make(map[string]interface{})
    err := jsoniter.Unmarshal(data, &dataMap)
    ...
  headerMap, ok := header.(map[string]interface{})
  businessName, ok := headerMap["businessName"]
  qua, ok := headerMap["qua"]
  sessionId, ok := headerMap["sessionId"]
  ...
  payload, ok := dataMap["payload"]
  req.Payload, ok = payload.(map[string]interface{})
}
```

这个函数本质上将 data 解析为 model.GatewayReqBody 类型的结构体。但是这里却存在 2 个问题

1. 使用了复杂的解析方式，先将 data 解析为 map，再通过每个字段的名字来取值，并进行类型转换。
2. Req.Playload 解析为一个 map。但又未使用。我们看看后面这个 payload 是用来做啥。确认是否为无效代码。

```
func invokeTafServant(resp http.ResponseWriter, gatewayHttpReq
*model.GatewayHttpReq) {
    ...
  payloadBytes, err := json.Marshal(gatewayHttpReq.ReqBody.Payload)
    if err == nil {
        commonReq.Payload = string(payloadBytes)
    } else {
        responseData(gatewayHttpReq, StatusInternalServerError, "封装json异常",
"", resp)
        return
    }
  ...
 }
```

后续的使用中，我们可以看到，又将这个 payload 转为 string。因此，我们可以确定，上面的 json 解析是没有意义，同时也会浪费资源（payload 数据量一般不小）。

优化方法

- golang 自带的 json 解析性能较低，这里我们可以替换为`github.com/json-iterator`来提升性能
- 在 golang 中，遇到不需要解析的 json 数据，可以将其类型声明为`json.RawMessage`. 即，可以将上述 2 个方法优化为

```
type GatewayReqBody struct {
    Header  GatewayReqBodyHeader `json:"header"`
    Payload json.RawMessage      `json:"payload"`
}
func readJsonReq(data []byte, req *model.GatewayReqBody) error {
    err := jsoniter.Unmarshal(data, req)
    if err != nil {
        return jsonParseErr
    }
    for k, v := range req.Header.Qua {
        req.Header.Qua[k] = v
        if len(req.Header.QuaStr) == 0 {
            req.Header.QuaStr = k + "=" + v
        } else {
            req.Header.QuaStr += "&" + k + "=" + v
        }
    }
    return nil
}
```

```
func invokeTafServant(resp http.ResponseWriter, gatewayHttpReq
*model.GatewayHttpReq) {
    commonReq.Payload = string(gatewayHttpReq.ReqBody.Payload)
}
```

- 这里注意！出现了 string 和[]byte 之间的类型转换.为了避免内存拷贝，这里将 string()改为上面的类型转换优化中所定义的转换函数，即`commonReq.Payload = encode.String(gatewayHttpReq.ReqBody.Payload)`

回包处理

```
type GatewayRespBody struct {
    Header  GatewayRespBodyHeader  `json:"header"`
    Payload map[string]interface{} `json:"payload"`
}

func responseData(gatewayReq *model.GatewayHttpReq, code int32, message string,
payload string, resp http.ResponseWriter) {
    jsonPayload := make(map[string]interface{})

    if len(payload) != 0 {
        err := json.Unmarshal([]byte(payload), &jsonPayload)
        if err != nil {
            ...
        }
    }

    body := model.GatewayRespBody{
        Header: model.GatewayRespBodyHeader{
            Code:    code,
            Message: message,
        },
        Payload: jsonPayload,
    }
  data, err := view.RenderResp("json", &body)
  ...
  resp.WriteHeader(http.StatusOK)
    resp.Write(data)
}
```

同样的，这里的 jsonPayload，也是出现了不必要的 json 解析。我们可以改为

```
type GatewayRespBody struct {
    Header  GatewayRespBodyHeader  `json:"header"`
    Payload json.RawMessage `json:"payload"`
}

body := model.GatewayRespBody{
        Header: model.GatewayRespBodyHeader{
            Code:    code,
            Message: message,
        },
        Payload: encode.Str2Bytes(payload),
    }
```

然后在 view.RenderResp 方法中

```
func RenderResp(format string, resp interface{}) ([]byte, error) {
    if "json" == format {
        return jsoniter.Marshal(resp)
    }
    return nil, errors.New("format error")
}
```

benchmark

```
Benchmark         500000          3326 ns/op        2842 B/op         50
allocs/op
```

虽然对象 alloc 减少了，但单次操作内存使用增加了，且性能下降了。这就有点奇怪了。我们来对比一下 2 个情况下的 pprof。

逃逸分析及处理

go tool pprof -base

- cpu 差异

```
   flat  flat%   sum%        cum   cum%
     0.09s  1.17%  1.17%      0.40s  5.20%  runtime.mallocgc
     0.01s  0.13%  1.30%      0.35s  4.55%  /vendor/github.com/json-
iterator/go.(*Iterator).readObjectStart
         0     0%  1.30%      0.35s  4.55%  /vendor/github.com/json-
iterator/go.(*twoFieldsStructDecoder).Decode
```

- mem 差异

```
      flat  flat%   sum%        cum   cum%
  478.96MB 20.33% 20.33%   279.94MB 11.88%  gateway.RootHandle
         0     0% 20.33%   279.94MB 11.88%  command-line-
arguments.BenchmarkTestHttp.func1
         0     0% 20.33%   279.94MB 11.88%  testing.(*B).RunParallel.func1
```

可以看出 RootHandle 多了 478.96M 的内存使用。 通过 list RootHandle 对比 2 个情况下的内存使用。发现修改后的 RootHandle 中多出了这一行：`475.46MB 475.46MB 158: gatewayHttpReq := model.GatewayHttpReq{}` 这一般意味着变量 gatewayHttpReq 出现了逃逸。

- go build -gcflags "-m -m" gateway/*.go

```
gateway/logic.go:270:26: &gatewayHttpReq escapes to heap
```

可以看到确实出现了逃逸。这个对应的代码为`err = view.ReadReq2Json(&gatewayHttpReq)`，而造成逃逸的本质是因为上面改动了函数 readJsonReq（动态类型逃逸，即函数参数为 interface 类型，无法在编译时确定具体类型的）

```
func readJsonReq(data []byte, req *model.GatewayReqBody) error {
    err := jsoniter.Unmarshal(data, req)
    ...
}
```

因此，这里需要特殊处理一下，改为

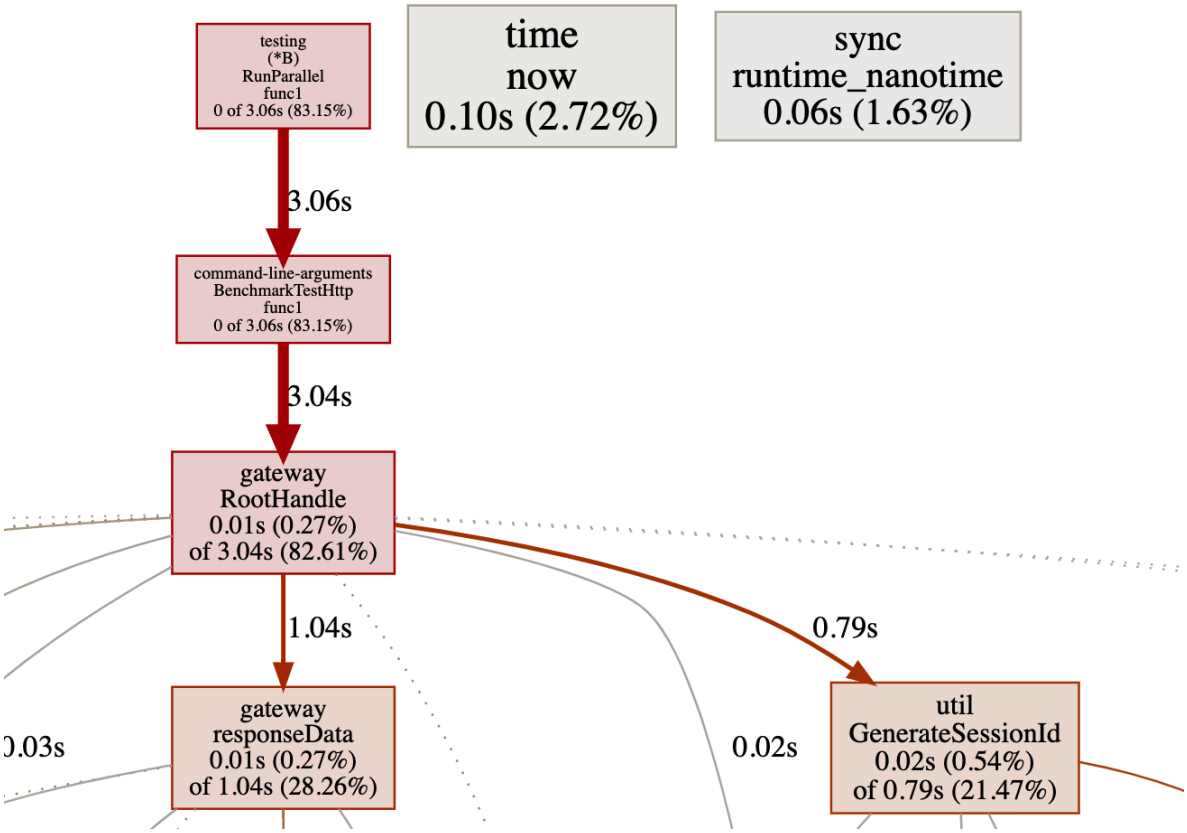
```
func readJsonReq(data []byte, req *model.GatewayReqBody) error {
    var tmp model.GatewayReqBody
    err := jsoniter.Unmarshal(data, &tmp)
    ...
}
```

benchmark

```
Benchmark          500000          2994 ns/op        1892 B/op          50
allocs/op
```

可以看到堆内存使用明显下降。性能也提升了。再看一下 pprof，寻找下个瓶颈。

cpu profile



抛开 responseData(他内部主要是日志打印占比高），占比较高的为 util.GenerateSessionId，先来看看这个怎么优化。

随机字符串生成

```
var letterRunes =
[]rune("0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")
func RandStringRunes(n int) string {
    b := make([]rune, n)
    for i := range b {
        b[i] = letterRunes[rand.Intn(len(letterRunes))]
    }
    return string(b)
}
```

目前的生成方式使用的类型是 rune，但其实用 byte 就够了。另外，letterRunes 是 62 个字符，即最大需要 6 位的 index 就可以遍历完成了。而随机数获取的是 63 位。即每个随机数，其实可以产生 10 个随机字符。而不用每个字符都获取一次随机数。所以我们改为

```
const (
    letterBytes   =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
    letterIdxBits = 6
    letterIdxMask = 1<<letterIdxBits - 1
    letterIdxMax  = 63 / letterIdxBits
```

```
)
func RandStringRunes(n int) string {
    b := make([]byte, n)
    for i, cache, remain := n-1, rand.Int63(), letterIdxMax; i >= 0; {
        if remain == 0 {
            cache, remain = rand.Int63(), letterIdxMax
        }
        if idx := int(cache & letterIdxMask); idx < len(letterBytes) {
            b[i] = letterBytes[idx]
            i--
        }
        cache >>= letterIdxBits
        remain--
    }
    return string(b)
}
```

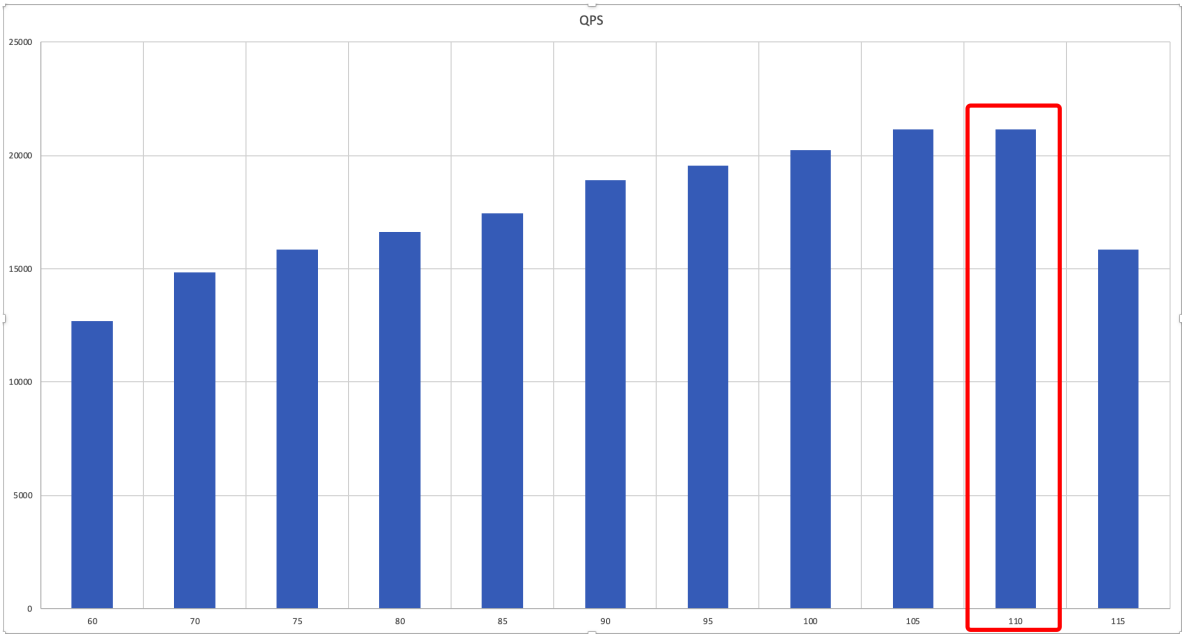
benchmark

```
Benchmark        1000000          1487 ns/op        1843 B/op          50
allocs/op
```

类型转换及字符串拼接

一般情况下，都会说将 string 和[]byte 的转换改为 unsafe；以及在字符串拼接时，用 byte.Buffer 代替 fmt.Sprintf。但是网关这里的情况比较特殊，字符串的操作基本集中在打印日志的操作。而 tars 的日志打印本身就是通过 byte.Buffer 拼接的。所以这可以避免。另外，由于日志打印量大，使用 unsafe 转换[]byte 为 string 带来的收益，往往会因为逃逸从而影响 GC，反正会影响性能。因此，不同的场景下，不能简单的套用一些优化方法。需要通过压测及结果分析来判断具体的优化策略。

优化结果



可以看到优化后，最大链接数为 110，最高 QPS 为**21153.35**。对比之前的**13245**，大约提升 60%。

后续

从 pprof 中可以看到日志打印，远程日志，健康上报等信息占用较多 cpu 资源，且导致多个数据逃逸（尤其是日志打印）。过多的日志基本等于没有日志。后续可考虑裁剪日志，仅保留出错时的上下文信息。

总结

- 性能查看工具 pprof,trace 及压测工具 wrk 或其他压测工具的使用要比较了解。
- 代码逻辑层面的走读非常重要，要尽量避免无效逻辑。
- 对于 golang 自身库存在缺陷的，可以寻找第三方库或自己改造。
- golang 版本尽量更新，这次的测试是在 golang1.12 下进行的。而 go1.13 甚至 go1.14 在很多地方进行了改进。比如 fmt.Sprintf，sync.Pool 等。替换成新版本应该能进一步提升性能。
- 本地 benchmark 结果不等于线上运行结果。尤其是在使用缓存来提高处理速度时，要考虑 GC 的影响。
- 传参数或返回值时，尽量按 golang 的设计哲学，少用指针，多用值对象，避免引起过多的变量逃逸，导致 GC 耗时暴涨。struct 的大小一般在 2K 以下的拷贝传值，比使用指针要快（可针对不同的机器压测，判断各自的阈值)。
- 值类型在满足需要的情况下，越小越好。能用 int8，就不要用 int64。
- 资源尽量复用,在 golang1.13 以上，可以考虑使用 sync.Pool 缓存会重复申请的内存或对象。或者自己使用并管理大块内存，用来存储小对象，避免 GC 影响（如本地缓存的场景)。

万字长文深入浅出 Golang Runtime

本文作者：yifhao，腾讯 PCG 开发工程师

介绍

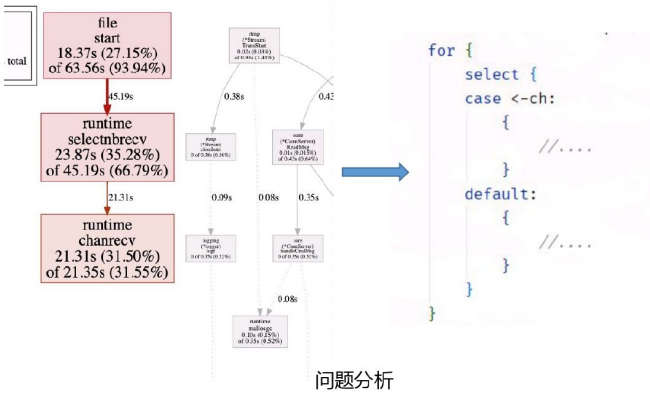
本文基于 2019.02 发布的 go 1.12 linux amd64 版本, 主要介绍了 Runtime 实现的一点原理和细节, 对大家容易错或者网络上很多错误的地方做一些梳理:

1. Golang Runtime 是个什么? Golang Runtime 的发展历程, 每个版本的改进
2. Go 调度: 协程结构体, 上下文切换, 调度队列, 大致调度流程, 同步执行流又不阻塞线程的网络实现等
3. Go 内存: 内存结构, mspan 结构, 全景图及分配策略等
4. Go GC: Golang GC 停顿大致的一个发展历程, 三色标记实现的一些细节, 写屏障, 三色状态, 扫描及元信息, 1.12 版本相对 1.5 版本的改进点, GC Pacer 等
5. 实践: 观察调度, GC 信息, 一些优化的方式, 几点问题排查的思路, 几个有意思的问题排查
6. 总结: 贯穿 Runtime 的思想总结

序

为什么去了解Runtime?

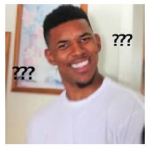
1. 解决疑难杂症&&优化
2. 好奇心
3. 技术深度的一种



问题分析

运行时到底是个什么东西?

Go的调度为什么说是轻量的?
Go调度都发生了啥?
Go的网络和锁会不会阻塞线程?
什么时候会阻塞线程?



Go的对象在内存中是怎样的?
Go的内存分配是怎样的?
栈的内存是怎么分配的?

GC是怎样的?
GC怎么帮我们回收对象?
Go的GC会不会漏掉对象或者回收还在用的对象?
Go GC什么时候开始?
Go GC啥时候结束?
Go GC会不会太慢, 跟不上内存分配的速度?
Go GC会不会暂停我们的应用? 暂停多久? 影不影响我的请求?
....

为什么去了解 runtime 呢?

1. 可以解决一些棘手的问题: 在写这个 PPT 的时候, 就有一位朋友在群里发了个 pprof 图, 说同事写的代码有问题, CPU 利用率很高., 找不出来问题在哪, 我看了下 pprof 图, 说让他找找是不是有这样用 select 的, 一查的确是的. 平时也帮同事解决了一些和并发, 调度, GC 有关的问题
2. 好奇心: 大家写久了 go, 惊叹于它的简洁, 高性能外, 必然对它是怎么实现的有很多好奇. 协程怎么实现, GC 怎么能并发, 对象在内存里是怎么存在的? 等等
3. 技术深度的一种

Runtime简介及历程
调度
内存与GC
实践
问答交流

1. 本次讨论基于最新2019.2发布的Go 1.12 Linux amd64版本
2. PPT中一些参数专门为 Linux amd64版本特化
3. 不同版本有差异, 流程基本类似
4. PPT中的图和文字只限于表达主要流程, 忽略一些分支和异常处理, 实际上源码复杂很多
5. 水平和精力有限, PPT中有错误, 欢迎指出. 代表不了公司其他程序员水平.
6. 有两份PPT, 一份是分享的, 较为精简, 一份是完整的, 均会放出 https://github.com/Frank-Hust/share
7. 有Go相关的问题分享后也可与我线下或微信讨论

希望本次分享和交流, 能够对Golang Runtime发展, 调度原理, 内存分配机制,
GC流程有个大概的认识, 能够解决对Go底层的一些疑惑.
相互讨论, 相互学习!

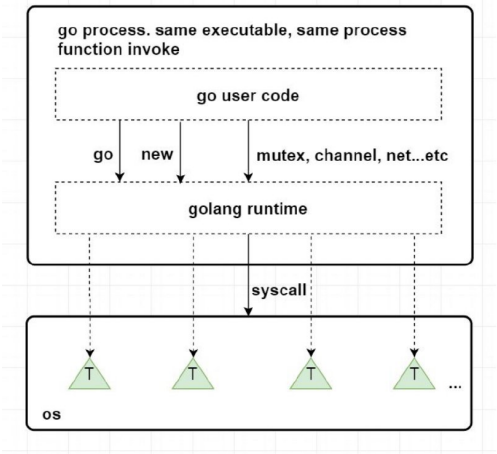
Runtime 简介及发展

Runtime 简介

Golang Runtime简介

Golang Runtime是go语言运行所需要的基础设施.

1. 协程调度, 内存分配, GC;
2. 操作系统及CPU相关的操作的封装(信号处理, 系统调用, 寄存器操作, 原子操作等), CGO;
3. pprof, trace, race检测的支持;
4. map, channel, string等内置类型及反射的实现.



1. 与Java, Python不同, Go并没有虚拟机的概念, Runtime也直接被编译成native code.
2. Go的Runtime与用户代码一起打包在一个可执行文件中
3. 用户代码与Runtime代码在执行的时候并没有明显的界限, 都是函数调用
4. go对系统调用的指令进行了封装, 可不依赖于glibc
5. 一些go的关键字被编译器编译成runtime包下的函数.

关键字	函数
go	newproc
new	newobject
make	makeslice, makechan, makemap, makemap_small...
	gcStart
<- ->	chansend1, chanrecv1
等等	

go 的 runtime 代码在 go sdk 的 runtime 目录下,主要有所述的 4 块功能.

提到 runtime, 大家可能会想起 java, python 的 runtime. 不过 go 和这两者不太一样, java, python 的 runtime 是虚拟机, 而 go 的 runtime 和用户代码一起编译到一个可执行文件中.

用户代码和 runtime 代码除了代码组织上有界限外, 运行的时候并没有明显的界限. 如上所示, 一些常用的关键字被编译成 runtime 包下的一些函数调用.

Runtime 版本历史

Runtime发展历程

版本	发布时间	改进特征	GC STW时间
v1.0	2012/3	调度GM模型, GC STW	百ms级别-秒级
v1.1	2013/5	**调度G-P-M模型**	同上
v1.2	2013/12	实现合作式的抢占	同上
v1.3	2014/6	GC实现Mark STW, Sweep 并行.栈扩容由split stack改为复制方式的continus stack. 添加sync.Pool	百m-几百ms级别
v1.4	2014/12	Runtime移除大部分C代码; 实现准确式GC. 引入写屏障, 为1.5的并发GC做准备.	同上
v1.5	2015/8	Runtime完全移除C代码, 实现了Go的自举. **GC 并发标记清除, 三色标记法**; GOMAXPROCS默认为CPU核数, go tool trace引入	10ms级别
v1.6	2016/2	1.5中一些与并发GC不协调的地方更改. 集中式的GC协调协程, 改为状态机实现	5ms级别
v1.7	2016/8	GC时由mark栈收缩改为并发, 引入dense bitmap, SSA引入	ms级
v1.8	2017/2	**hybrid write barrier**, 消除re-scanning stack, **GC进入sub ms.** defer和cgo调用开销减少一半	sub ms(18GB堆)
V1.9	2017/8	保留用于debug的rescan stack代码移除, runtime.GC, debug.SetGCPercent, and debug.FreeOSMemory等触发STW GC改为并发GC	基本同上
V1.10	2018/2	不再限制最大GOMAXPROCS(Go 1.9为1024), LockOSThread的线程在运行的G结束后可以释放.	基本同上
V1.11	2018/8	连续的arena改为稀疏索引的方式	基本同上
V1.12	2019/2	Mark Termination流程优化	Sub ms, 但几乎减少一半

注: GC STW时间与堆大小, 机器性能, 应用分配偏好, 对象数量均有关. 较早的版本来自网络上的数据. 1.4-1.9数据来源于twitter工程师. 这里是以较大的堆测试, 数据仅供参考. 普通应用的情况好于上述的数值.

左边标粗的是一些更新比较大的版本. 右边的 GC STW 仅供参考.

调度

简介及发展

调度

内存

GC

实践

调度简述

Golang调度简述

- PMG模型, M:N调度模型.
- 调度在计算机中是分配工作所需资源的方法. linux的调度为CPU找到可运行的线程. 而Go的调度是为M(线程)找到P(内存, 执行票据)和可运行的G.
- 轻量级协程G, 栈初始2KB, 调度不涉及系统调用.
- 用户函数调用前会检查栈空间是否足够, 不够的话, 会进行栈扩容.
- 用户代码中的协程同步造成的阻塞, 仅仅是切换协程, 而不阻塞线程.
- 网络操作封装了epoll, 为NonBlocking模式, 未ready, 切换协程, 不阻塞线程.
- 每个p均有local runq, 大多数时间仅与local runq无锁交互. 实现work stealing.
- 用户协程无优先级, 基本遵循FIFO.
- 目前(1.12), go支持协作的抢占调度, 还不支持非协作的抢占调度.

goroutine 实现

Goroutine

Process -> Thread(LWP, lightweight process) -> Goroutine (一种lightweight userspace thread)
不断共享, 不断减少切换成本

Go实现有栈协程
代表协程这种执行流的结构体
保护和恢复上下文的函数
运行队列
编译器将go关键字编译为生成一个协程结构体, 并放入运行队列
解决网络IO阻塞问题
协程级别的同步结构
调度: findrunnable
...

注:Goroutine的形式只是协程的一种形式

我们去看调度的一个进化, 从进程到线程再到协程, 其实是一个不断共享, 不断减少切换成本的过程. go 实现的协程为有栈协程, go 协程的用法和线程的用法基本类似. 很多人会疑问, 协程到底是个什么东西? 用户态的调度感觉很陌生, 很抽象, 到底是个什么东西?

我觉得要理解调度, 要理解两个概念: 运行和阻塞. 特别是在协程中, 这两个概念不容易被正确理解. 我们理解概念时往往会代入自身感受, 觉得线程或协程运行就是像我们吭哧吭哧的处理事情, 线程或协程阻塞就是做事情时我们需要等待其他人. 然后就在这等着了. 要是其他人搞好了, 那我们就继续做当前的事.

其实主体对象搞错了.正确的理解应该是我们处理事情时就像 CPU, 而不是像线程或者协程. 假如我当前在写某个服务, 发现依赖别人的函数还没有 ready, 那就把写服务这件事放一边. 点开企业微信, 我去和产品沟通一些问题了. 我和产品沟通了一会后, 检查一下, 发现别人已经把依赖的函数提交了, 然后我就最小化企业微信, 切到 IDE, 继续写服务 A 了.

对操作系统有过一些了解, 知道 linux 下的线程其实是 task_struct 结构, 线程其实并不是真正运行的实体, 线程只是代表一个执行流和其状态.真正运行驱动流程往前的其实是 CPU. CPU 在时钟的驱动下, 根据 PC 寄存器从程序中取指令和操作数, 从 RAM 中取数据, 进行计算, 处理, 跳转, 驱动执行流往前. CPU 并不关注处理的是线程还是协程, 只需要设置 PC 寄存器, 设置栈指针等(这些称为上下文), 那么 CPU 就可以欢快的运行这个线程或者这个协程了.

线程的运行, 其实是被运行.其阻塞, 其实是切换出调度队列, 不再去调度执行这个执行流. 其他执行流满足其条件, 便会把被移出调度队列的执行流重新放回调度队列.协程同理, 协程其实也是一个数据结构, 记录了要运行什么函数, 运行到哪里了.
go 在用户态实现调度, 所以 go 要有代表协程这种执行流的结构体, 也要有保存和恢复上下文的函数, 运行队列. 理解了阻塞的真正含义, 也就知道能够比较容易理解, 为什么 go 的锁, channel 这些不阻塞线程.

对于实现的同步执行流效果, 又不阻塞线程的网络, 接下来也会介绍.

协程结构体和切换函数

协程结构体和切换函数

```
type g struct {
    goid   int64  //协程的id
    status uint32 //协程状态
    stack  struct {
        lo uintptr //该协程拥有的栈低位
        hi uintptr //该协程拥有的栈高位
    }
    sched     gobuf; //切换时保存的上下文信息
    startfunc uintptr //程序地址
}

type gobuf struct {
    sp uintptr //栈指针位置
    pc uintptr //运行到的程序位置
}
```

代表执行流(协程)的结构体

```
TEXT runtime·mcall(SB), NOSPLIT, $0-8
    MOVQ    fn+0(FP), DI

    get_tls(CX)
    MOVQ    g(CX), AX   //从tls中获取g
    MOVQ    0(SP), BX   //PC
    MOVQ    BX, (g_sched+gobuf_pc)(AX) //保存PC到sched pc
    LEAQ    fn+0(FP), BX
    MOVQ    BX, (g_sched+gobuf_sp)(AX) //保存SP到sched sp
    MOVQ    AX, (g_sched+gobuf_g)(AX) //保存g到sched g
    MOVQ    BP, (g_sched+gobuf_bp)(AX)//保存BP到sched bp
```

切换时保存上下文

```
TEXT runtime·gogo(SB), NOSPLIT, $16-8
    MOVQ    buf+0(FP), BX
    MOVQ    gobuf_g(BX), DX     //DX保存了gobuf结构
    MOVQ    0(DX), CX       // make sure g != nil
    get_tls(CX)
    MOVQ    DX, g(CX)
    MOVQ    gobuf_sp(BX), SP    //恢复SP
    MOVQ    gobuf_ret(BX), AX
    MOVQ    gobuf_ctxt(BX), DX
    MOVQ    gobuf_bp(BX), BP
    MOVQ    gobuf_pc(BX), BX //恢复PC
    JMP BX //跳到对应地址
```

重新调度时恢复上下文

我们 go 一个 func 时一般这样写

```
go func1(arg1 type1,arg2 type2){....}(a1,a2)
```

一个协程代表了一个执行流, 执行流有需要执行的函数(对应上面的 func1), 有函数的入参(a1, a2), 有当前执行流的状态和进度(对应 CPU 的 PC 寄存器和 SP 寄存器), 当然也需要有保存状态的地方, 用于执行流恢复.

真正代表协程的是 runtime.g 结构体. 每个 go func 都会编译成 runtime.newproc 函数, 最终有一个 runtime.g 对象放入调度队列. 上面的 func1 函数的指针设置在 runtime.g 的 startfunc 字段, 参数会在 newproc 函数里拷贝到 stack 中, sched 用于保存协程切换时的 pc 位置和栈位置.

协程切换出去和恢复回来需要保存上下文, 恢复上下文, 这些由以下两个汇编函数实现. 以上就能实现协程这种执行流, 并能进行切换和恢复.(上图中的 struct 和函数都做了精简)

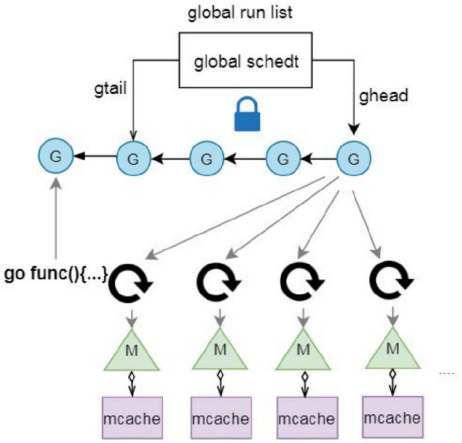
GM 模型及 GPM 模型

GM模型

一开始, 实现一个简单一点的, 一个全局队列放待运行的g.
新生成G, 阻塞的G变为待运行, M寻找可运行的G等操作都在全局队列中操作, 需要加线程级别的锁.

- 调度锁问题. 单一的全局调度锁(Sched.Lock)和集中的状态, 导致伸缩性下降.
- G传递问题. 在工作线程M之间需要经常传递runnable的G, 会加大调度延迟, 并带来额外的性能损耗.
- Per-M的内存问题. 类似TCMalloc结构的内存结构, 每个M都需要memory cache和其他类型的cache(比如stack alloc), 然而实际上只有M在运行Go代码时才需要这些Per-M Cache, 阻塞在系统调用的M并不需要这些cache. 正在运行Go代码的M与进行系统调用的M的比例可能高达1:100, 这造成了很大的内存消耗.

等



是不是可以给运行的M加个本地队列?
是不是可以剥夺阻塞的M的mcache给其他M使用?

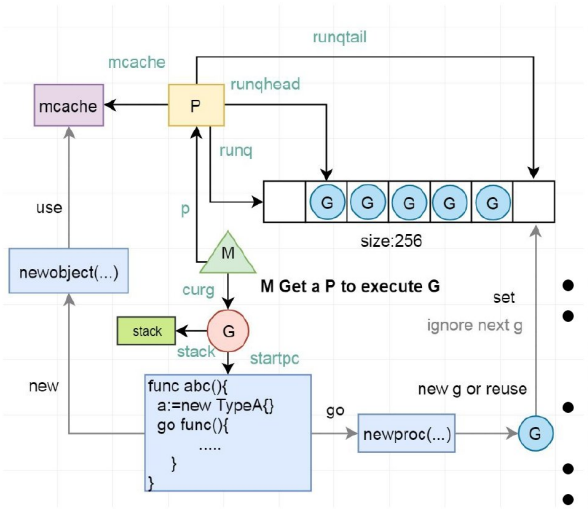
有了协程的这种执行流形式, 那待运行的协程放在哪呢?在 Go1.0 的时候:

1. 调度队列 schedt 是全局的, 对该队列的操作均需要竞争同一把锁, 导致伸缩性不好.
2. 新生成的协程也会放入全局的队列, 大概率是被其他 m(可以理解为底层线程的一个表示)运行了, 内存亲和性不好. 当前协程 A 新生成了协程 B, 然后协程 A 比较大概率会结束或者阻塞, 这样 m 直接去执行协程 B, 内存的亲和性也会好很多.
3. 因为 mcache 与 m 绑定, 在一些应用中(比如文件操作或其他可能会阻塞线程的系统调用比较多), m 的个数可能会远超过活跃的 m 个数, 导致比较大的内存浪费.

那是不是可以给 m 分配一个队列, 把阻塞的 m 的 mcache 给执行 go 代码的 m 使用? Go 1.1 及以后就是这样做的.

GPM模型

Golang 1.1中调度为GPM模型. 通过引入逻辑Processer P来解决GM模型的几个问题.



	数据结构	数量	意义
G Goroutine	runtime.g 运行的函数指针, stack, 上下文等	每次都go func都代表一个G, 无限制	代表一个用户代码执行流
P Processor	runtime.p per-P的cache, runq和free g等	默认为机器核数. 可通过GOMAXPROCS环境变量调整.	表示执行所需的资源
M Machine	runtime.m 对应一个由clone创建的线程	比P多, 一般不会多太多. 最大1万个	代表执行者, 底层线程

- mcache从M中移到P中.
- 不再是单独的全局runq. 每个P拥有自己的runq. 新的g放入自己的runq. 满了后再批量放入全局runq中. 优先从自己的runq获取g执行
- 实现work stealing, 当某个P的runq中没有可运行G时, 可以从全局获取, 从其他P获取
- 当G因为网络或者锁切换, 那么G和M分离, M通过调度执行新的G
- 当M因为系统调用阻塞或cgo运行一段时间后, sysmon协程会将P与M分离. 由其他的M来结合P进行调度.

再 1.1 中调度模型更改为 GPM 模型, 引入逻辑 Process 的概念, 表示执行 Go 代码所需要的资源, 同时也是执行 Go 代码的最大的并行度.

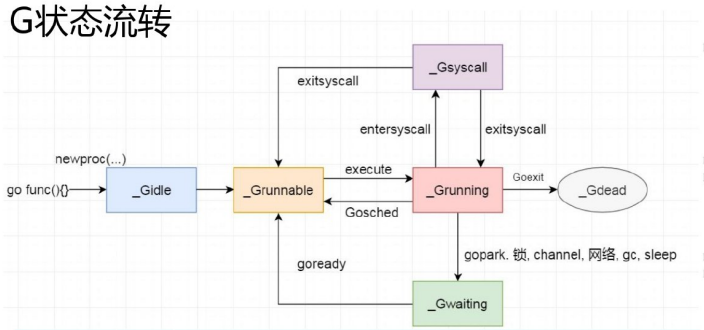
这个概念可能很多人不知道怎么理解. P 涉及到几点, 队列和 mcache, 还有 P 的个数的选取.

首先为什么把全局队列打散, 以及 mcache 为什么跟随 P, 这个在 GM 模型那一页就讲的比较清楚了.然后为什么 P 的个数默认是 CPU 核数: Go 尽量提升性能, 那么在一个 n 核机器上, 如何能够最大利用 CPU 性能呢? 当然是同时有 n 个线程在并行运行中, 把 CPU 喂饱, 即所有核上一直都有代码在运行.

在 go 里面, 一个协程运行到阻塞系统调用, 那么这个协程和运行它的线程 m, 自然是不再需要 CPU 的, 也不需要分配 go 层面的内存. 只有一直在并行运行的 go 代码才需要这些资源, 即同时有 n 个 go 协程在并行执行, 那么就能最大的利用 CPU, 这个时候需要的 P 的个数就是 CPU 核数. (注意并行和并发的区别)

协程状态及流转

G状态流转



G状态	值	说明
Gidle	0	刚刚被分配, 还没有初始化
Grunnabel	1	表示在runqueue上, 还没有被运行
Grunning	2	go协程可能在执行go代码, 不在runqueue上, 与M, P已绑定
Gsyscall	3	go协程在执行系统调用, 没执行go代码, 没有在runqueue上, 只与M绑定
Gwaiting	4	go协程被阻塞(IO, GC, chan阻塞, 锁等). 不在runqueue上, 但是一定在某个地方, 比如channel中, 锁排队中等.
Gdead	6	协程现在没有在使用, 也许执行完, 或者在free list中, 或者正在被初始化. 可能有stack或者没有
Gcopystack	8	栈正在复制, 此时没有go代码, 也不在runqueue上
Gscan	0x1000	与runnable, running, syscall, waiting等状态结合, 表示GC正在扫描这个G的栈

```go
go func() { //_Grunnable
    println( args...: "hello world")  //_Grunning
    runtime.Gosched() //_Grunnable

    time.Sleep(time.Second) // _Gwaiting, reason:sleep

    //not get lock immediately
    m.Lock() // _Gwaiting, reason:semacquire
    //_Grunning
    m.Unlock()
    b:=make([]byte,100)

    //read from network
    _, _ = conn.Read(b) //_Gwaiting, reason:IO wait
    //...

    _, _ = ioutil.ReadFile( filename: "yifhao.txt") //_Gsyscall

    //_Grunning
    funcA()
    //_Gdead
}()
}

func funcA(){
    //GC happened
    //_Gwaiting, reason: -> _Gscan
    //GC end
    //_Grunning
    //...dosomething
}
```

协程的状态其实和线程状态类似,状态转换和发生状态转换的时机如图所示. 还是需要注意: 协程只是一个执行流, 并不是运行实体.

调度

调度

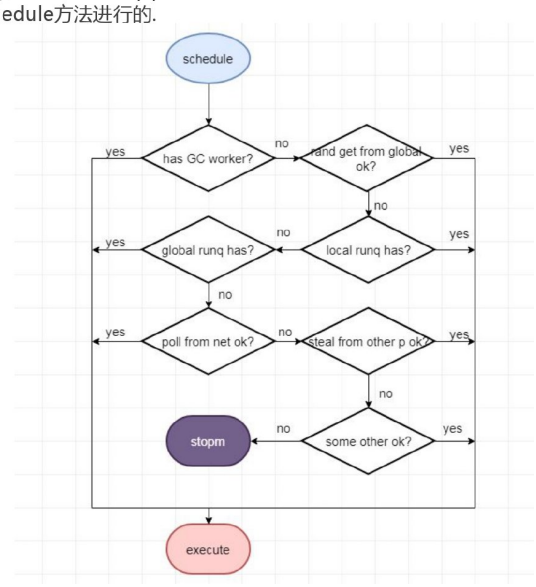
golang调度的职责就是为需要执行的Go代码(G)寻找执行者(M)以及执行的准许和资源(P).
并没有一个调度器的实体, 调度是需要发生调度时由m执行runtime.schedule方法进行的.

调度时机:

- channel, mutex等sync操作发生了协程阻塞
- time.sleep
- 网络操作暂时未ready
- gc
- 主动yield
- 运行过久或系统调用过久
- 等等

调度流程:
实际调度代码复杂很多.

如果有分配到gc mark的工作需要做gc mark.
local runq有就运行local的,
没有再看全局的runq是否有,
再看能否从net中poll出来,
从其他P steal一部分过来.
....
实在没有就stopm



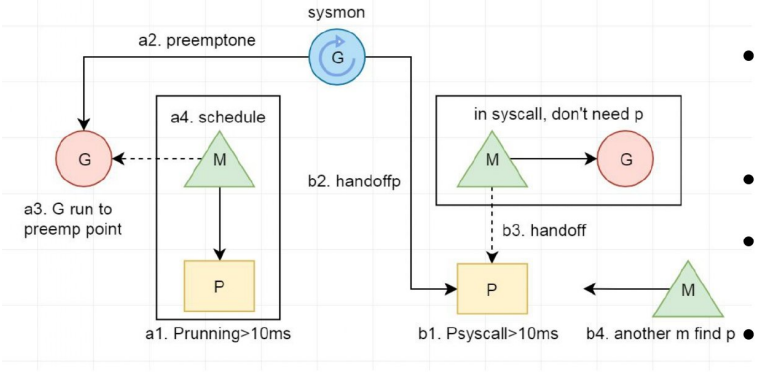
并没有一个一直在运行调度的调度器实体. 当一个协程切换出去或新生成的 m, go 的运行时从 stw 中恢复等情况时, 那么接下来就需要发生调度. go 的调度是通过线程(m)执行 runtime.schedule 函数来完成的.

sysmon 协程

sysmon协程

P的数量影响了同时运行go代码的协程数. 如果P被占用很久, 就会影响调度.
sysmon协程的一个功能就是进行抢占.

sysmon协程是在go runtime初始化之后, 执行用户编写的代码之前, 由runtime启动的不与任何P绑定, 直接由一个M执行的协程. 类似于linux中的执行一些系统任务的内核线程.
可认为是10ms执行一次. (初始运行间隔为20us, sysmon运行1ms后逐渐翻倍, 最终每10ms运行一次. 如果有发生过抢占成功, 则又恢复成初始20us的运行间隔, 如此循环)



- 每sysmon tick进行一次netpoll(在STW结束,和M执行查找可运行的G时也会执行netpoll)获取fd事件, 将与之相关的G放入全局runqueue
- 每次sysmon运行都执行一次抢占, 如果某个P的G执行超过1个sysmon tick, 则执行抢占. 正在执行系统调用的话, 将P与M脱离(handoffp); 正在执行Go代码, 则通知抢占(preemptone).
- 每2分钟如果没有执行过GC, 则通知gchelper协程执行一次GC
- 如果开启schdule trace的debug信息(例如GODEBUG=schedtrace=5000,scheddetail=1), 则按照给定的间隔打印调度信息
- 每5分钟归还GC后不再使用的span给操作系统(scavenge)

在 linux 内核中有一些执行定时任务的线程, 比如定时写回脏页的 pdflush, 定期回收内存的 kswapd0, 以及每个 cpu 上都有一个负责负载均衡的 migration 线程等.在 go 运行时中也有类似的协程, sysmon. 功能比较多: 定时从 netpoll 中获取 ready 的协程, 进行抢占, 定时 GC,打印调度信息,归还内存等定时任务.

协作式抢占

协作式抢占

retake()调用preemptone()将被抢占的G的stackguard0设为stackPreempt,
被设置抢占标记的G进行下一次函数调用时, 检查栈空间失败. 进而触发morestack()（汇编代码，位于asm_XXX.s中）然后进行一连串的函数调用，主要的调用过程如下：

morestack()（汇编代码）-> newstack() -> gopreempt_m() -> goschedImpl() -> schedule()

go 目前(1.12)还没有实现非协作的抢占. 基本流程是 sysmon 协程标记某个协程运行过久, 需要切换出去, 该协程在运行函数时会检查栈标记, 然后进行切换.

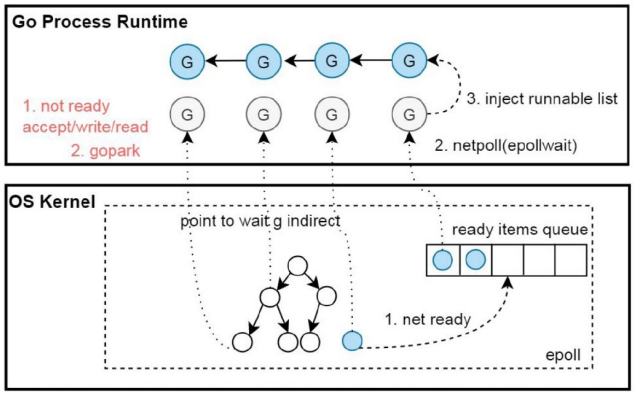
同步执行流不阻塞线程的网络的实现

网络

JavaScript网络操作是异步非阻塞的, 通过事件循环, 回调对应的函数.
一些状态机模式的框架, 每次网络操作都有一个新的状态.
代码执行流被打散.

用户态的协程: 结合 epoll, nonblock模式的fd操作;
网络操作未ready时的切换协程和ready后把相关协程添加到待运行队列. 网络操作达到既不阻塞线程, 又是同步执行流的效果.

1. 封装epoll, 有网络操作时会epollcreate一个epfd.
2. 所有网络fd均通过fcntl设置为NONBLOCK模式, 以边缘触发模式放入epoll节点中.
3. 对网络fd执行Accept(syscall.accept4), Read(syscall.read), Write(syscall.write)操作时, 相关操作未ready, 则系统调用会立即返回EAGAIN; 使用gopark切换该协程
4. 在不同的时机, 通过epollwait来获取ready的epollevents, 通过其中data指针可获取对应的g, 将其置为待运行状态, 添加到runq



go 写后台最舒服的就是能够以同步写代码的方式操作网络, 但是网络操作不阻塞线程.主要是结合了非阻塞的 fd, epoll 以及协程的切换和恢复.linux 提供了网络 fd 的非阻塞模式, 对于没有 ready 的非阻塞 fd 执行网络操作时, linux 内核不阻塞线程, 会直接返回 EAGAIN, 这个时候将协程状态设置为 wait, 然后 m 去调度其他协程.

go 在初始化一个网络 fd 的时候, 就会把这个 fd 使用 epollctl 加入到全局的 epoll 节点中. 同时放入 epoll 中的还有 polldesc 的指针.

```
func netpollopen(fd uintptr, pd *pollDesc) int32 {
    var ev epollevent
    ev.events = _EPOLLIN | _EPOLLOUT | _EPOLLRDHUP | _EPOLLET
    *(**pollDesc)(unsafe.Pointer(&ev.data)) = pd
    return -epollctl(epfd, _EPOLL_CTL_ADD, int32(fd), &ev)
}
```

在 sysmon 中, schedule 函数中, start the world 中等情况下, 会执行 netpoll 调用 epollwait 系统调用, 把 ready 的网络事件从 epoll 中取出来, 每个网络事件可以通过前面传入的 polldesc 获取到阻塞在其上的协程, 以此恢复协程为 runnable.

调度相关结构体

调度相关结构体

```
type g struct {
    stack       stack   //协程拥有的栈内存[stack.lo, stack.hi)
    stackguard0 uintptr //栈扩容点或被抢占
    m              *m      //运行当前g的m的结构体
    sched          gobuf // pc, sp, lr等上下文数据
    param          unsafe.Pointer // passed parameter on wakeup
    atomicstatus   uint32 //协程状态,_Gidle,_Grunnable,_Grunning等
    goid           int64 //协程的id
    waitsince      int64      // 协程开始阻塞的时间
    waitreason     waitReason // 阻塞的原因
    startpc        uintptr         // go func时func的地址
    waiting        *sudog         // 协程在lock中等待的结构
}
```

```
type p struct {
    id          int32
    status      uint32 //P的状态, _Pidle, _Prunning等
    m           muintptr   //非idle时与之关联的m
    mcache      *mcache //per-P的分配cache
    runqhead uint32     //无锁访问, per-P的
    runqtail uint32
    runnext guintptr
    // 运行完状态为Gdead的空闲g, 可用于复用
    gFree struct {
        gList
        n int32
    }
    palloc persistentAlloc // per-P, 用于分配一些runtime里的特殊结构.
    gcw gcWork //per-P的gc相关的work
}
```

图中struct的字段均做了大量省略

```
type m struct {
    g0      *g     //每个m绑定的g, 用于专门执行调度
    procid        uint64 //底层的线程id
    tls           [6]uintptr //传给fs寄存器的线程局部变量
    mstartfn      func()//m启动时的函数, 传给clone
    curg          *g       //在运行go代码时的g
    p             puintptr //在运行go代码时绑定的p
    id            int64
    spinning      bool //m找不到可运行的g, spin
    mcache        *mcache //运行go代码时绑定的p中mcache
    lockedg       guintptr //是否与某个g一直绑定
}
```

```
type schedt struct {
    lock mutex//调度如果需要涉及全局的队列时, 需要获取的锁
    midle        muintptr //空闲的m列表
    pidle      puintptr //空闲的p列表
    nmspinning uint32 //在spin状态的m数量
    runq     gQueue //全局的可运行的队列
    runqsize int32
    gFree struct { //全局空余的g
        lock    mutex
        stack   gList //空余g中有栈的
        noStack gList //空余g中无栈的
        n       int32
    }
    gcwaiting  uint32 // gc is waiting to run
}
```

调度综述

调度综合

- 轻量级的协程, 栈初始2KB, 调度不涉及系统调用.
- 调度在计算机中是分配工作所需资源的方法. linux的调度为CPU找到可运行的线程. 而Go的调度是为M(线程)找到P(内存, 执行票据)和可运行的G.
- 用户函数调用前会检查栈空间是否足够, 不够的话, 会进行*2 栈扩容. 最大栈1G, 超出panic.
- 用户代码中的协程同步造成的阻塞, 仅仅是切换(gopark)协程, 而不阻塞线程, m和p仍结合, 去寻找新的可执行的g.
- 每个P均有local runq, 大多数时间仅与local runq无锁交互. 新生成的g, 放入到local runq中.
- 调度时会随机从全局runq取g. 然后local runq, global runq... 均没有g的话, work stealing从其他P中取.
- sysmon: 对于运行过久的g设置抢占标识; 对于过久syscall的p, 进行m和p的分离. 防止p被占用过久影响调度.
- 封装了epoll, 网络fd会设置成NonBlocking模式, 网络fd的read, write, accept操作, 会以NonBlocking模式操作, 返回EAGAIN则gopark当前协程. 在m调度, sysmon中, gc start the world等阶段均会poll出ready的协程进行运行或者添加到全局runq中
- 目前(1.12), go还不支持非协作的抢占调度, 在密集CPU运算时, 可能会导致调度延迟. 官方已在解决.

未涉及的点

- G状态流转
- 具体调度流程
- 栈扩容
- 合作式抢占
- sysmon
- P状态流转
- M的spin与unspin
- LockOSThread

等

内存分配

内存分配简介

golang内存分配简介

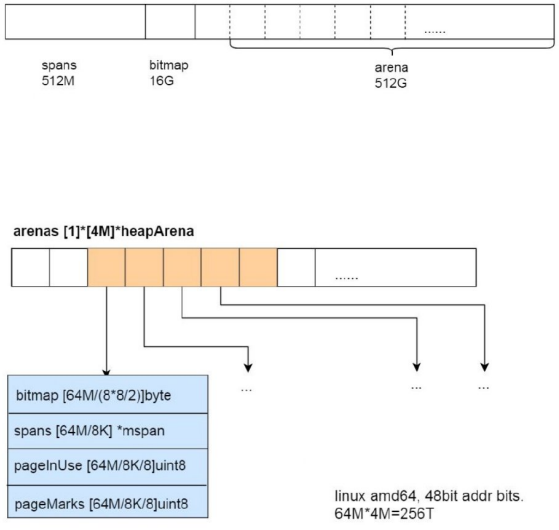
- 类似于TCMalloc的结构
- 使用span机制来减少碎片. 每个span至少为一个页(go中的一个page为8KB). 每一种span用于一个范围的内存分配需求. 比如16-32byte使用分配32byte的span, 112-128使用分配128byte的span.
- 一共有67个size范围, 8byte-32KB, 每个size有两种类型(scan和noscan, 表示分配的对象是否会包含指针)
- 多层次Cache来减少分配的冲突. per-P无锁的mcache, 全局67*2个对应不同size的span的后备mcentral, 全局1个的mheap.
- mheap中以treap的结构维护空闲连续page. 归还内存到heap时, 连续地址会进行合并.
- stack分配也是多层次和多class的.
- 对象由GC进行回收. sysmon会定时把空余的内存归还给操作系统

Go 的分配采用了类似 tcmalloc 的结构.特点: 使用一小块一小块的连续内存页, 进行分配某个范围大小的内存需求. 比如某个连续 8KB 专门用于分配 17-24 字节,以此减少内存碎片. 线程拥有一定的 cache, 可用于无锁分配.

同时 Go 对于 GC 后回收的内存页, 并不是马上归还给操作系统, 而是会延迟归还, 用于满足未来的内存需求.

内存空间结构

golang内存结构



此处讨论的均是linux amd64.

1.10及以前
以下内存并不是初始化时就分配虚拟内存的:

arena的大小为512G，为了方便管理把arena区域划分成一个个的page, 每个page 8KB，一共有512GB/8KB个页

spans区域存放指向span的指针，表示arean中对应的Page所属的span, 所以span区域的大小为(512GB/8KB)*指针大小8byte = 512M

bitmap主要用于GC, 用两个bit表示 arena中一个字的可用状态，所以是(512G/8个字节一个字)*2/8个bit每个字节=16G

1.11及以后:
改成了两阶稀疏索引的方式. 内存可以超过512G, 也可以允许不连续的内存.
mheap中的areans字段是一个指针数组, 每个heapArena管理64M的内存.
bitmap和spans和上面的功能一致.

在 1.10 以前 go 的堆地址空间是线性连续扩展的, 比如在 1.10(linux amd64)中, 最大可扩展到 512GB. 因为 go 在 gc 的时候会根据拿到的指针地址来判断是否位于 go 的 heap 的, 以及找到其对应的 span, 其判断机制需要 gc heap 是连续的. 但是连续扩展有个问题, cgo 中的代码(尤其是 32 位系统上)可能会占用未来会用于 go heap 的内存. 这样在扩展 go heap 时, mmap 出现不连续的地址, 导致运行时 throw.

在 1.11 中, 改用了稀疏索引的方式来管理整体的内存. 可以超过 512G 内存, 也可以允许内存空间扩展时不连续.在全局的 mheap struct 中有个 arenas 二阶数组, 在 linux amd64 上,一阶只有一个 slot, 二阶有 4M 个 slot, 每个 slot 指向一个 heapArena 结构, 每个 heapArena 结构可以管理 64M 内存, 所以在新的版本中, go 可以管理 4M*64M=256TB 内存, 即目前 64 位机器中 48bit 的寻址总线全部 256TB 内存.

span 机制

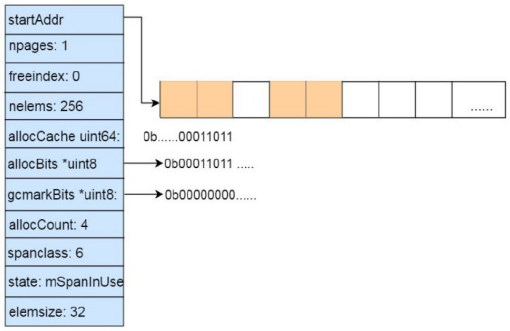
mspan

使用span机制来减少碎片. 每个span至少分配1个page(8KB), 划分成固定大小的slot, 用于分配一定大小范围的内存需求.

runtime/sizeclasses.go

```
class  bytes/obj  bytes/span  objects  tail waste  max waste
    1          8        8192     1024           0     87.50%
    2         16        8192      512           0     43.75%
    3         32        8192      256           0     46.88%
    4         48        8192      170          32     31.52%
    5         64        8192      128           0     23.44%
    6         80        8192      102          32     19.07%
    7         96        8192       85          32     15.95%
    8        112        8192       73          16     13.56%
    9        128        8192       64           0     11.72%
   10        144        8192       56         128     11.82%
   11        160        8192       51          32      9.73%
   12        176        8192       46          96      9.59%
   13        192        8192       42         128      9.25%
   14        208        8192       39          80      8.12%
   15        224        8192       36         128      8.15%
....
....
   58      16384       16384        1           0     12.49%
   59      18432       73728        4           0     11.11%
   60      19072       57344        3         128      3.57%
   61      20480       40960        2           0      6.87%
   62      21760       65536        3         256      6.25%
   63      24576       24576        1           0     11.45%
   64      27264       81920        3         128     10.00%
   65      28672       57344        2           0      4.91%
   66      32768       32768        1           0     12.50%
```

class0表示单独分配一个>32KB对象的span, 有67个size, 每个size两种, 用于分配有指针和无指针对象, 所以有67*2=134个class.



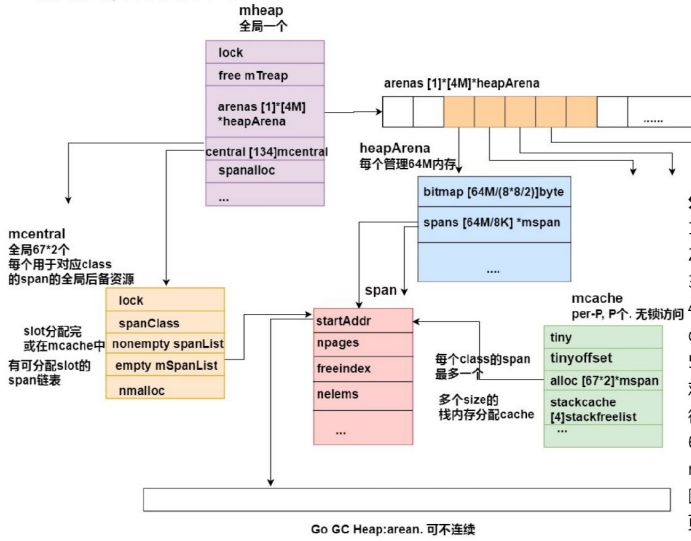
span结构体主要字段如上.

1. 这里表示slot大小为32byte的span, 上一次gc之后, 前8个slot使用如上.
2. freeindex表示 <该位置的都被分配了, >=该位置的可能被分配, 也可能没有. 配合allocCache来寻找. 每次分配后, freeindex设置为分配的slot+1.
3. allocBits表示上一次GC之后哪一些slot被使用了. 0未使用或释放, 1已分配.
4. allocCache表示从freeindex开始的64个slot的分配情况, 1为未分配, 0为分配. 使用ctz(Count Trailing Zeros指令)来找到第一个非0位. 使用完了就从allocBits加载, 取反.
5. 每次gc完之后, sweep阶段, 将allocBits设置为gcmarkBits.

前面提到了 go 的内存分配类似于 tcmalloc, 采用了 span 机制来减少内存碎片. 每个 span 管理 8KB 整数倍的内存, 用于分配一定范围的内存需求.

内存分配全景

全景及分配策略



多层次Cache来减少分配的冲突, 加快分配.
从无锁到粒度较低的锁, 再到全局一个锁, 或系统调用.

分配策略

1. new, make最终调用mallocgc
2. >32KB对象, 直接从mheap中分配, 构成一个span
3. <16byte且无指针(noscan), 使用tiny分配器, 合并分配.
4. <16byte有指针或16byte-32KB, 如果mcache中有对应class的空闲mspan, 则直接从该mspan中分配一个slot.
5. (mcentral.cacheSpan) mcache没有对应的空余span, 则从对应mcentral中申请一个有空余slot的span到mcache中. 再进行分配
6. (mcentral.grow)对应mcentral没有空余span, 则向mheap(mheap_.alloc)中申请一个span, 能sweep出span则返回. 否则看mheap的free mTreap能否分配最大于该size的连续页, 能则分配, 多的页放回 .
7. mheap的free mTreap无可用, 则调用sysAlloc(mmap)向系统申请.
8. 6, 7步中获得的内存构建成span, 返回给mcache, 分配对象.

多层次的分配 Cache, 每个 P 上有一个 mcache, mcache 会为每个 size 最多缓存一个 span, 用于无锁分配. 全局每个 size 的 span 都有一个 mcentral, 锁的粒度相对于全局的 heap 小很多, 每个 mcentral 可以看成是每个 size 的 span 的一个全局后备 cache.

在 gc 完成后, 会把 P 中的 span 都 flush 到 mcentral 中, 用于清扫后再分配. P 有需要 span 时, 从对应 size 的 mcentral 获取. 获取不到再上升到全局的 heap.

几种特殊的分配器

几种特殊分配器

tiny分配器



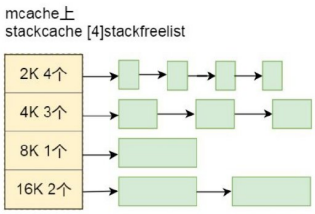
对于小于maxTinySize(16B)无指针对象的内存分配请求.go采取了将小对象合并存储的解决方案.
每个P在本地维护了专门的memory block来存储tiny object, 分配时根据tinyoffset和需要的size及对齐来判断该block内是否可容纳该object, 如果可以, 返回放入的地址.

fixalloc

spanalloc	fixalloc
cachealloc	fixalloc
treapalloc	fixalloc
specialfinalizeralloc	fixalloc
specialprofilealloc	fixalloc

这些都是mheap中的字段.
fixalloc用于分配runtime中固定大小的一些结构, 比如mspan, mcache.
主要思路是一次性分配一大块内存(persistentalloc方法, 使用的是mmap, 不指定地址, 分配的内存不在arena范围中), 每次请求对应结构体的大小, 释放时放在list链表中.

stackcache



stackcache是per-P的. 用于分配groutines的stack. 和普通对象内存一样, 栈分配也有多级和多个层次.
>16K的直接从全局的stackLarge分配.
否则先从p的stackcache中分配, 如果无法分配除, 则需要从全局的stackpool分配出一批stack (stackpoolalloc), 赋给该p的stackcache, 再从local stackcache中分配.

对于很小的对象分配, go 做了个优化, 把小对象合并, 以移动指针的方式分配.对于栈内存有 stackcache 分配, 也有多个层次的分配, 同时 stack 也有多个不同 size. 用于分配 stack 的内存也是位于 go gc heap, 用 mspan 管理, 不过这个 span 的状态和用于分配对象的 mspan 状态不太一样, 为 mSpanManual.

我们可以思考一个问题, go 的对象是分配在 go gc heap 中, 并由 mcache, mspan, mcentral 这些结构管理, 那么 mcache, mspan, mcentral 这些结构又是哪里管理和分配的呢? 肯定不是自己管理自己. 这些都是由特殊的分配 fixalloc 分配的, 每种类型有一个 fixalloc, 大致原理就是通过 mmap 从进程空间获取一小块内存(百 KB 的样子), 然后用来分配这个固定大小的结构.

内存分配综合

Golang内存分配综合

- 类似于TCMalloc的结构
- 使用span机制来减少碎片. 每个span至少为一个页(go中的一个page为8KB). 每一种span用于一个范围的内存分配需求. 比如16-32byte使用分配32byte的span, 112-128使用分配128byte的span.
- 一共有67个size范围, 8byte-32KB, 每个size有两种类型(scan和noscan, 表示分配的对象是否会包含指针)
- 多阶Cache来减少分配的冲突. per-P无锁的mcache, 对应不同size(67*2)的全局mcentral, 全局的mheap.
- go代码分配内存优先从当前p的mcache对应size的span中获取; 有的话, 再从对应size的mcentral中获取一个span; 还没有的话, 从mheap中sweep一个span; sweep不出来, 则从mheap中空闲块找到对应span大小的内存. mheap中如果还没有, 则从系统申请内存. 从无锁到全局1/(67*2)粒度的锁, 再到全局锁, 再到系统调用.
- stack的分配也是多层次和多class的. 减少分配的锁争抢, 减少栈浪费.
- mheap中以treap的结构维护空闲连续page. 归还内存到mheap时, 连续地址会进行合并. (1.11之前采用类似伙伴系统维护<1MB的连续page, treap维护>1MB的连续page)
- 对象由GC进行释放. sysmon会定时把mheap空余的内存归还给操作系统

未涉及的点:
fixalloc和persistentalloc具体流程, scav treap, 逃逸分析

GC

Golang GC 简述

Golang GC简述

- GC是一种自动内存管理方式
- GC三大基础算法: Mark-Sweep, Mark-Sweep-Compact, Mark-Copy. 这三个基础算法GC时需要一直STW(stop the world)
- GC算法是一个综合的考虑: 程序吞吐量, GC吞吐量, STW时间, STW的频率, 压缩, 分配性能, 并发, 伸缩, 调优复杂度等等.
- 当前Go GC特征: 三色标记, 并发标记和清扫, 非分代, 非紧缩, 混合写屏障.
- 1.5之后Go的GC是基于三色标记法的Concurrent Mark-Sweep(并发标记-清扫)算法. 生产上基本没什么问题.
- 1.5之后Golang Runtime中调整最多的就是GC的部分.
- 1.8之后Go的GC STW可以比较稳定在sub ms
- Go的GC并不是完美的

GC 简介

何为GC

对象是否存活是由整体应用其他部分是否对其有引用决定的，是一个全局性的信息，而手动管理内存只能由开发人员根据当前上下文的局部信息来判断.

Garbage Collection (GC)是一种自动管理内存的方式. 支持GC的语言无需手动管理内存，**程序后台自动判断对象是否存活并回收其内存空间，使开发人员从内存管理上解脱出来**.

1959年，GC由 John McCarthy发明，用于简化Lisp中的手动内存管理. 到现在很多语言都提供了GC. 不过GC的原理和基本算法都没有太大的改变.

哪些语言提供GC(基本上90年代后的语言都提供GC):

ML，Haskell，Lisp等大多数函数式编程语言

Python，Ruby，JS等动态类型语言

Smalltalk，Java，C#等面向对象语言

C++

```
for(int i = 0; i < 100; i++)
{
    Person * p = new Person();
    .......
    delete p;
}

//调用者需要自己new Person并负责释放
test(p *Person){
    p->name=" yifhao"
}
```

Go

```
for i:=0;i<100 ;i++  {
      o:=new(Person)
       ......
}

func test() *Person {
      p := new(Person)
      p.Name = "yifhao"
      return p
}
```

注意:

(1)对象被回收时间即GC发生时间是不可知的

(2)垃圾回收不是析构函数.

(3)垃圾回收只与内存有关，不会回收连接资源，文件资源.

(4) GC时自动调用对象的某个方法，来关闭一些资源，是否可行?

GC 并不是个新事物, 使得 GC 大放光彩的是 Java 语言.

GC相关术语

并发和并行: 通常在GC领域中，并发收集器则指垃圾回收的同时应用程序也在执行; 并行收集器指垃圾回收采取多个线程利用多个CPU一起进行GC. 不过一般我们说并发回收器, 就包含了这两层意思.
Safepoint: 安全点(Safepoint)是收集器能够识别出线程执行栈上的所有引用的一点或一段时间.
Stop The World(STW): 某些垃圾回收算法或者某个阶段进行时需要将应用程序完全暂停.
准确式GC和保守式GC: 准确式(precise或exact) GC指GC能够知道一块内存区域是引用还是非引用，如一个64位的区域可以是一个int64整形数字也可以是一个对象引用，当一个对象进行执行时，需要修改指向这个内存的引用的指，非准确式GC也就是保守式GC不能完成这个任务.
Mark: 从Root对象开始扫描, 标记出其引用的对象, 及这些对象引用的对象, 如此循环, 标记所有可达的对象.
Sweep: Sweep清除阶段扫描堆区域, 回收在标记阶段标记为Dead的对象, 通常通过空闲链表(free list)的方式.需要的工作量和堆大小成正比.
Compact: 压缩的方式是将存活对象移动到一起来获得一段连续的空闲空间, 也叫做重定位.这样需要将所有对象的引用指向新的位置.工作量和存活对象量成正比.
Copy: 复制算法将所有存活对象从一个From区域移动到一个To区域, 然后回收From区域. 之后From和To交换角色, 由于From和To是对等的, 所以需要两块大小相同的区域, 而同一时间只能利用其中一块来存放对象.工作量和存活对象量成正比.

GC关心什么

程序吞吐量: 回收算法会在多大程度上拖慢程序? 可以通过GC占用的CPU与其他CPU时间的百分比描述
GC吞吐量: 在给定的CPU时间内，回收器可以回收多少垃圾?
堆内存开销: 回收器最少需要多少额外的内存开销?
停顿时间: 回收器会造成多大的停顿?
停顿频率: 回收器造成的停顿频率是怎样的?
停顿分布: 停顿有时候很长，有时候很短? 还是选择长一点但保持一致的停顿时间?
分配性能: 新内存的分配是快，慢还是无法预测?
压缩: 当堆内存里还有小块碎片化的内存可用时, 回收器是否仍然抛出内存不足（OOM）的错误？如果不是, 那么你是否发现程序越来越慢, 并最终死掉, 尽管仍然还有足够的内存可用？
并发：回收器是如何利用多核机器的？
伸缩：当堆内存变大时, 回收器该如何工作？
调优：回收器的默认使用或在进行调优时, 它的配置有多复杂？
预热时间：回收算法是否会根据已发生的行为进行自我调节？如果是, 需要多长时间？
页释放：回收算法会把未使用的内存释放回给操作系统吗？如果会, 会在什么时候发生？

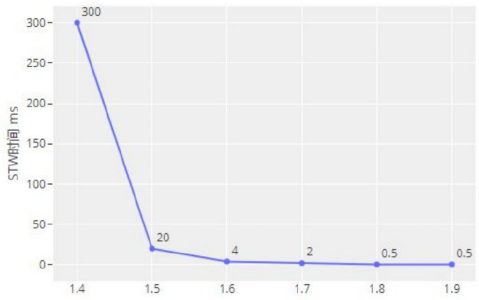
没有十全十美的GC算法, 折衷无处不在

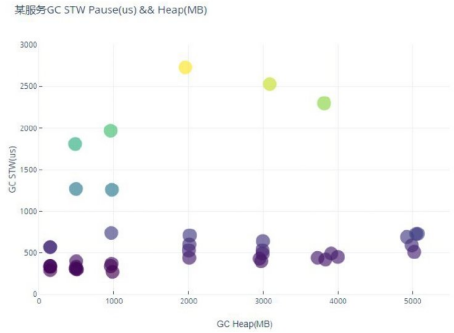
Golang GC 发展

Golang GC发展

Golang早期版本GC可能问题比较多 . 但每一个版本的发布都伴随着 GC 的改进.
1.5版本之后, Go的GC已经能满足大部分大部分生产环境使用要求. 1.8通过hybrid write barrier, 使得STW降到了sub ms.
下面列出一些GC方面比较重大的改动:

版本	发布时间	GC	STW时间(见备注twitter数据)
v1.1	2013/5	STW	百ms-几百ms级别
v1.3	2014/6	Mark STW, Sweep 并行	百ms级别
v1.5	2015/8	三色标记法, 并发标记清除	10ms级别
v1.8	2017/2	hybrid write barrier	sub ms



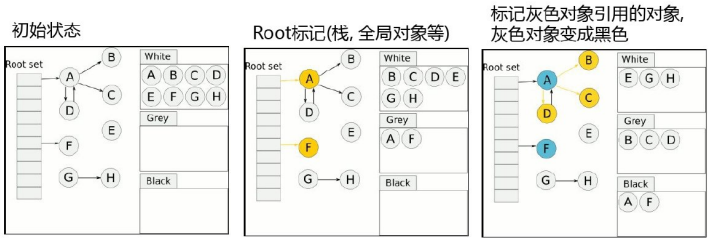


上面是几个比较重要的版本.左图是根据 twitter 工程师的数据绘制的(堆比较大), 从 1.4 的百 ms 级别的停顿到 1.8 以后的小于 1ms.右图是我对线上服务(Go 1.11 编译)测试的一个结果, 是一个批量拉取数据的服务, 大概 3000qps, 服务中发起的 rpc 调用大概在 2w/s. 可以看到大部分情况下 GC 停顿小于 1ms, 偶尔超过一点点.

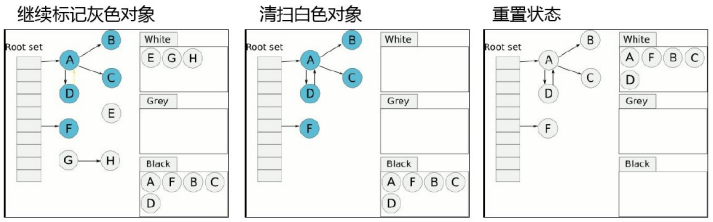
整体来说 golang gc 用起来是很舒心的, 几乎不用你关心.

三色标记

三色标记



1. 有黑白灰三个集合. 初始时所有对象都是白色
2. 从Root对象开始标记, 将所有可达对象标记为灰色
3. 从灰色对象集合取出对象, 将其引用的对象标记为灰色, 放入灰色集合, 并将自己标记为黑色
4. 重复第三步, 直到灰色集合为空, 即所有可达对象都被标记
5. 标记结束后, 不可达的白色对象即为垃圾. 对内存进行迭代清扫, 回收白色对象.
6. 重置GC状态



go和java不同, go的对象在内存中并没有header.

1. 标记和程序并发, 会漏标记对象吗? 如何解决的?
2. 哪里记录了对象的三色标记状态?
3. 标记时, 拿到一个指针, 怎么知道它是哪个对象? 也许是某个对象的内部指针? 这个对象的内存哪些地方代表了它引用的对象呢?

图来自https://en.wikipedia.org/wiki/Tracing_garbage_collection

go 采用的是并发三色标记清除法. 图展示的是一个简单的原理.有几个问题可以思考一下:

- 并发情况下, 会不会漏标记对象?
- 对象的三色状态存放在哪?
- 如何根据一个对象来找到它引用的对象?

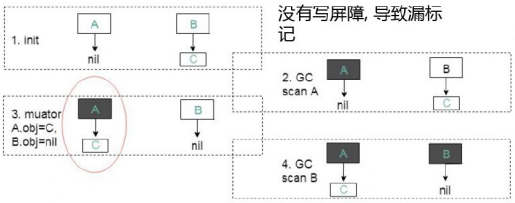
写屏障

Go三色标记实现的一点细节: 写屏障

```
var A Wb
var B Wb

type Wb struct {
    Obj *int
}

func simpleSet(c *int) {
    A.Obj = nil
    B.Obj = c
    //if GC Begin
    //scan A
    A.Obj = c
    B.Obj = nil
    //scan B
}
```



没有写屏障, 导致漏标记

三色标记需要维护(弱)不变性条件: 黑色对象不能引用无法被灰色对象可达的白色对象.
并发标记时, 如果没有做正确性保障措施, 可能会导致漏标记对象, 导致实际上可达的对象被清扫掉.

为了解决这个问题, go使用了写屏障(和内存写屏障不是同一个概念).
写屏障是在写入指针前执行的一小段代码, 用以防止并发标记时指针丢失, 这一小段代码Go是在编译时加入的.
Golang写屏障在mark和marktermination阶段处于开启状态.

左图, 第3步,A.obj=C时, 会将C进行标记, 加入写屏障buf, 最终会flush到待扫描队列, 这样就不会丢失C及C引用的对象.

GC 最基本的就是正确性: 不漏标记对象, 程序还在用的对象都被清除了, 那程序就错误了. 有一点浮动垃圾是允许的.
在并发情况下, 如果没有一些措施来保障, 那可能会有什么问题呢?

看左边的代码和图示, 第 2 步标记完 A 对象, A 又没有引用对象, 那 A 变成黑色对象. 在第 3 步的时候, muator(程序)运行, 把对象 C 从 B 转到了 A, 第 4 步, GC 继续标记, 扫描 B, 此时 B 没有引用对象, 变成了黑色对象. 我们会发现 C 对象被漏标记了.

如何解决这个问题? go 使用了写屏障, 这里的写屏障是指由编译器生成的一小段代码. 在 gc 时对指针操作前执行的一小段代码, 和 CPU 中维护内存一致性的写屏障不太一样哈.所以有了写屏障后, 第 3 步, A.obj=C 时, 会把 C 加入写屏障 buf. 最终还是会被扫描的.

Go三色标记实现的一点细节: 写屏障

```
var A Wb
var B Wb

type Wb struct {
    Obj *int
}

func simpleSet(c *int) {
    A.Obj = nil
    B.Obj = c
    //if GC Begin
    //scan A
    A.Obj = c
    B.Obj = nil
    //scan B
}
```

go build -gcflags "-N -l"
go tool objdump -s 'main\.simpleSet' -S ./main.exe

```
    B.Obj = c
0x44eb9e    488b442410          MOVQ 0x10(SP), AX    参数c
0x44eba3    833d565e080000      CMPL $0x0, runtime.writeBarrier(SB) 判断写屏障是否开启
0x44ebaa    7402                JE 0x44ebae  没开启则直接赋值
0x44ebac    eb52                JMP 0x44ec00 开启了则跳转到这里执行一个汇编函数, 写屏障逻辑
0x44ebae    488905f3c60600      MOVQ AX, main.B(SB)  真正执行B.Obj = c
0x44ebb5    eb00                JMP 0x44ebb7 只是执行下一条
  A.Obj = c
0x44ebb7    488b442410          MOVQ 0x10(SP), AX
    ......
0x44ec00    488d3da1c60600      LEAQ main.B(SB), DI
0x44ec07    e834a0ffff          CALL runtime.gcWriteBarrier(SB)   写屏障逻辑,在asm_amd64p32.s
0x44ec0c    eba8                JMP 0x44ebb7 执行完写屏障, 跳回去
```

栈中指针slot的操作没有写屏障.

Dijkstra写屏障是对被写入的指针进行grey操作, 不能防止指针从heap被隐藏到黑色的栈中, 需要STW重扫描栈.
Yuasa写屏障是对将被覆盖的指针进行grey操作, 不能防止指针从栈被隐藏到黑色的heap对象中, 需要在GC开始时保存栈的快照.

go 1.8写屏障混合了两者, 既不需要GC开始时保存栈快照, 也不需要STW重扫描栈, 原型如下:

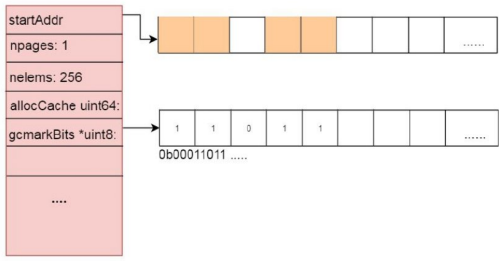
```
writePointer(slot, ptr):
  shade(*slot)
  if current stack is grey:
    shade(ptr)
  *slot = ptr
```

Proposal: Eliminate STW stack re-scanning
https://github.com/golang/proposal/blob/master/design/17503-eliminate-rescan.md

这里感受一下写屏障具体生成的代码. 我们可以看到在写入指针 slot 时, 对写屏障是否开启做了判断, 如果开启了, 会跳转到写屏障函数, 执行加入写屏障 buf 的逻辑. 1.8 中写屏障由 Dijkstra 写屏障改成了混合式写屏障, 使得 GC 停顿达到了 1ms 以下.

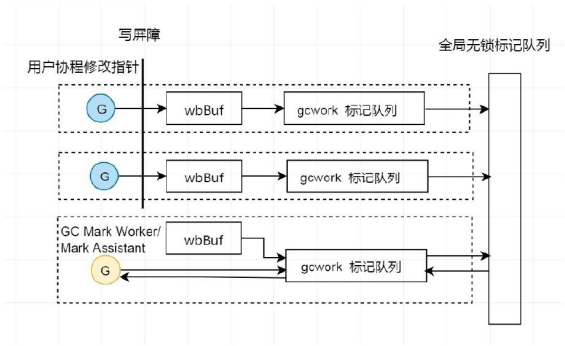
三色状态

Go三色标记实现的一点细节: 三色状态



哪里记录了对象的三色标记状态?

并没有真正的三个集合来分别装三色对象.
前面分析内存的时候, 介绍了go的对象是分配在span中, span里还有一个字段是gcmarkBits, mark阶段里面每个bit代表一个slot已被标记.
白色对象该bit为0, 灰色或黑色为1. (runtime.markBits)
每个p中都有wbBuf和gcw gcWork, 以及全局的workbuf标记队列, 实现生产者-消费者模型, 在这些队列中的指针为灰色对象, 表示已标记, 待扫描.
从队列中出来并把其引用对象入队的为黑色对象, 表示已标记, 已扫描. (runtime.scanobject).



并没有这样一个集合把不同状态对象放到对应集合中. 只是一个逻辑上的意义.

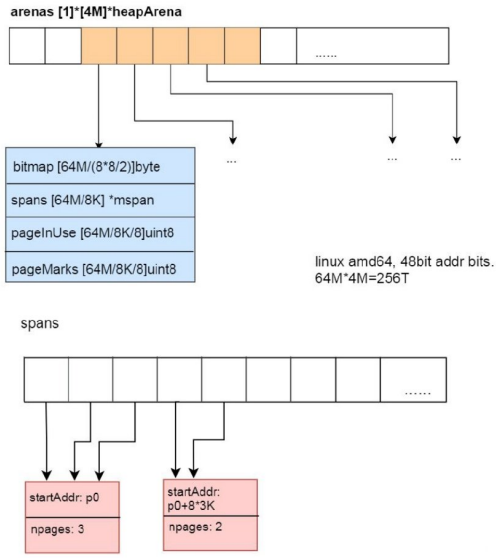
扫描和元信息

Go三色标记实现的一点细节: 扫描与元信息

2. 标记时拿到一个指针p1, 如何知道哪里是其引用的对象?

回到前面所提到的内存结构图. go的gc heap通过右图的arenas进行划分, 每个heapArena管理了64M内存. heapArena存储着pointer, span, bitmap的索引关系.

p1指向的对象所属heapArena: arenas[0][p+constbase/64M]
找到对象所属span: p1%64M/8K就知道了该对象在该heapArena中的页index, 通过spans[index]即可找到其所属的span(runtime.spanOf)
对象首地址: 找到对象所属的span, 根据span的elemsize和span的startAddr, 即可知道其为该span中第几个对象以及其地址(runtime.findObject)
对象的gcmarkBits: 知道了obj在span中属于第几个对象, 即可知道如何设置其gcmarkBits.
对象引用的对象: bitmap每两个bit分别表示某8个字节是否是指针, 以及该字节所属对象后续字节是否还包含指针, 以此知道其引用的对象和减少扫描工作量. 这些bit是分配对象时, 根据对象type信息设置的.



gc 拿到一个指针, 如何把这个指针指向的对象其引用的子对象都加到扫描队列呢? 而且 go 还允许内部指针, 似乎更麻烦了. 我们分析一下, 要知道对象引用的子对象, 从对象开始到对象结尾, 把对象那一块内存上是指针的放到扫描队列就好了. 那我们是不是得知道对象有多大, 从哪开始到哪结束, 同时要知道内存上的 8 个字节, 哪里是指针, 哪里是普通的数据.

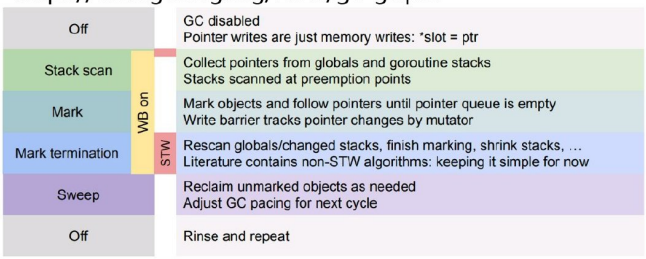
首先 go 的对象是 mspan 管理的, 我们如果能知道对象属于哪个 mspan, 就知道对象多大, 从哪开始, 到哪结束了. 前面我们讲到了 areans 结构, 可以通过指针加上一定得偏移量, 就知道属于哪个 heap arean 64M 块. 再通过对 64M 求余, 结合 spans 数组, 即可知道属于哪个 mspan 了.

结合 heapArean 的 bitmap 和每 8 个字节在 heapArean 中的偏移, 就可知道对象每 8 个字节是指针还是普通数据(这里的 bitmap 是在分配对象时根据 type 信息就设置了, type 信息来源于编译器生成)

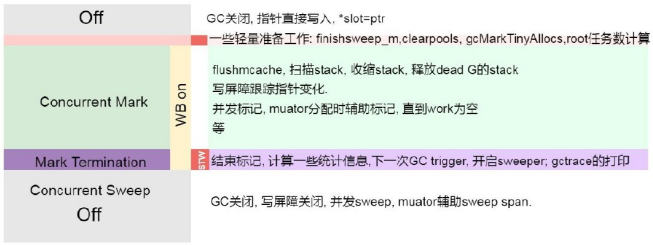
GC 流程

Golang GC流程

Go 1.5 Release : 2015/8 图来自
https://talks.golang.org/2015/go-gc.pdf



Go 1.12 Release: 2019/2 根据源码绘制



GC Trigger
gcTriggerHeap: 分配内存时, 当前已分配内存与上一次GC结束时存活对象的内存达到某个比例时就触发GC.
gcTriggerTime: sysmon检测2min内是否运行过GC, 没运行过则执行GC.
gcTriggerAlways: runtime.GC()强制触发GC.

当然1.5的Mark和Sweep都是Concurrent的, 只不过下图我特地注明了下.

改进点:
Go 1.5是Go转为并发三色标记清除法的版本. 大部分情况下能够将STW控制在10ms以下.

1.5中GC流程及状态的转换由一个协程集中式控制, 容易导致调度延迟.
1.6中采用了分布式检测, 其他协程也可以进行检测并状态转换.

1.5中栈收缩在Mark Termination(STW), 如果协程很多, 会导致STW时间暴增.
1.7将栈收缩移入并发Mark阶段.

1.5只采用了Dijkstra写屏障, 在Mark Termination(STW)阶段需要重新扫描栈, 这个是STW最大的来源.
1.8采用了混合写屏障, 消除了重新扫描栈, STW进入sub ms.
.....

1.12对Mark Terminiation阶段进行了优化

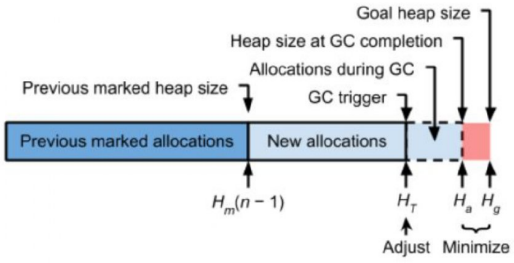
1.5 和 1.12 的 GC 大致流程相同. 上图是 golang 官方的 ppt 里的图, 下图是我根据 1.12 源码绘制的.从最坏可能会有百 ms 的 gc 停顿到能够稳定在 1ms 以下, 这之间 GC 做了很多改进. 右边是我根据官方 issues 整理的一些比较重要的改进. 1.6 的分布式检测, 1.7 将栈收缩放到了并发扫描阶段, 1.8 的混合写屏障, 1.12 更改了 mark termination 检测算法, mcache flush 移除出 mark termination 等等.

Golang GC Pacer

Golang GC Pacer

Go何时开始GC, 何时结束GC? 分配太快怎么办?
Go GC Pacer机制决定这些阈值, GC Pacer基于Go程序大致稳定的假定, 并且有负反馈机制.
Go的GC触发与Java不同, Java是当内存某个具体值时(比如2G)触发GC. 而Go与上一次GC之后存活的对象占用的内存Hm(n-1)成一个比例.

比如Hm(n-1)为1000M, default GOGC=100, 那么大概会在小于但比较接近2000M(比如1850, 1900M) Ht的时候开始GC;
结束GC时, 使得GC堆大小Ha(比如1990, 2010M)趋向于等于目标值Hg 2000M.
GC结束后, 会根据当前数据计算下一次GC触发值Hm(n+1).



https://golang.org/s/go15gcpacing

如何保证GC按时开始?
在GC结束后, 下一次GC开始前, 需要sweep完所有的span.
有一个后台清扫协程, 同时用户协程也需要辅助sweep.
假设有k page的span需要sweep, 那么距离下一次GC开始还有Ht-Hm(n-1)的内存可分配, 那么平均分配1bytes内存需要sweep k/(Ht-Hm(n-1))page的span (会根据sweep进度进行比例更改)

如何保证GC按时结束?
GC在Ht开始, 到达到目标堆大小Hg时, 尽量mark完所有待mark的对象.
除了后台并发的标记协程, 程序分配内存时也需要辅助mark.
从Ht到Hg还有Hg-Ht的内存可分配, 这个时候还有scanWorkExpected的对象需要scan, 那么平均分配1byte内存, 需要辅助mark scanWorkExpected/(Hg-Ht)的对象(会根据mark进度进行比例更改)

下一次GC啥时候开始?
下一次GC的触发值由反馈公式计算. 定性如下:
如果本次GC mark消耗的CPU高于目标值(30%), 则下次GC提前一点.
如果本次GC的Ha高于Hg, 则下次GC提前一点.

大家对并发 GC 除了怎么保证不漏指针有疑问外, 可能还会疑问, 并发 GC 如何保证能够跟得上应用程序的分配速度? 会不会分配太快了, GC 完全跟不上, 然后 OOM?

这个就是 Golang GC Pacer 的作用.

Go 的 GC 是一种比例 GC, 下一次 GC 结束时的堆大小和上一次 GC 存活堆大小成比例. 由 GOGC 控制, 默认 100, 即 2 倍的关系, 200 就是 3 倍, 以此类推.

假如上一次 GC 完成时, 存活对象 1000M, 默认 GOGC 100, 那么下次 GC 会在比较接近但小于 2000M 的时候(比如 1900M)开始, 争取在堆大小达到 2000M 的时候结束. 这之间留有一定的裕度, 会计算待扫描对象大小(根据历史数据计算)与可分配的裕度的比例, 应用程序分配内存根据该比例进行辅助 GC, 如果应用程序分配太快了, 导致 credit 不够, 那么会被阻塞, 直到后台的 mark 跟上来了,该比例会随着 GC 进行不断调整.

GC 结束后, 会根据这一次 GC 的情况来进行负反馈计算, 计算下一次 GC 开始的阈值.

如何保证按时完成 GC 呢? GC 完了后, 所有的 mspan 都需要 sweep, 类似于 GC 的比例, 从 GC 结束到下一次 GC 开始之间有一定的堆分配裕度, 会根据还有多少的内存需要清扫, 来计算分配内存时需要清扫的 span 数这样的一个比例.

未涉及的点

- 栈分配
- fixalloc, tinyalloc
- 逃逸分析
- 内存归还
- 清扫与辅助清扫
- go gc mark任务分配
- 辅助标记
- revise
- mark termination流程
- GC Pacer, trigger计算, goal计算

....

实践与总结

观察调度

观察一下调度

通过godoc来开启一个go进程, 用ab来调用, 加上debug trace观察其状态
GOMAXPROCS=8 GODEBUG=schedtrace=500 godoc -http=:6060 (加上scheddetail=1观察更详细信息)
ab -c 1000 -n 100000 'http://localhost:6060/'

```
SCHED 6978ms: gomaxprocs=8 idleprocs=0 threads=15 spinningthreads=0 idlethreads=3 runqueue=30 [4 25 3 59 0 3 2 17]
SCHED 7280ms: gomaxprocs=8 idleprocs=0 threads=15 spinningthreads=0 idlethreads=3 runqueue=0 [0 0 15 0 24 12 45 0]
SCHED 7584ms: gomaxprocs=8 idleprocs=0 threads=15 spinningthreads=1 idlethreads=3 runqueue=1 [0 0 0 0 5 56 14 0]
SCHED 7887ms: gomaxprocs=8 idleprocs=0 threads=16 spinningthreads=0 idlethreads=3 runqueue=35 [13 6 12 10 7 8 6 6]
SCHED 8189ms: gomaxprocs=8 idleprocs=0 threads=16 spinningthreads=0 idlethreads=3 runqueue=0 [65 0 12 1 3 11 0 2]
SCHED 8492ms: gomaxprocs=8 idleprocs=0 threads=16 spinningthreads=0 idlethreads=3 runqueue=3 [0 0 1 2 43 55 6 7]
SCHED 8800ms: gomaxprocs=8 idleprocs=5 threads=16 spinningthreads=1 idlethreads=8 runqueue=105 [3 6 0 0 0 0 0 0]
SCHED 9102ms: gomaxprocs=8 idleprocs=0 threads=16 spinningthreads=0 idlethreads=4 runqueue=44 [14 9 6 6 1 9 6 6]
SCHED 9406ms: gomaxprocs=8 idleprocs=1 threads=16 spinningthreads=1 idlethreads=5 runqueue=17 [3 4 6 2 4 3 0 0]
SCHED 9708ms: gomaxprocs=8 idleprocs=4 threads=16 spinningthreads=1 idlethreads=7 runqueue=69 [13 11 0 0 12 0 0 0]
```

观察一下调度, 加一些请求. 我们可以看到虽然有 1000 个连接, 但是 go 只用了几个线程就能处理了, 表明 go 的网络的确是由 epoll 管理的. runqueue 表示的是全局队列待运行协程数量, 后面的数字表示每个 P 上的待运行协程数. 可以看到待处理的任务并没有增加, 表示虽然请求很多, 但完全能 hold 住.

同时可以看到, 不同 P 上有的时候可能任务不均衡, 但是一会后, 任务又均衡了, 表示 go 的 work stealing 是有效的.

观察 GC

观察一下GC

这是一个服务通过调用debug.SetGCPercent设置GOGC, 分别是100, 550, 1650时的表现.
GOGC越大, GC频次越低, 但是触发GC的堆内存也越大. 其中具体含义可线下讨论.

```
GOGC=100
gc 8913 @2163.341s 1%: 0.13+14+0.20 ms clock, 1.1+21/22/0+1.6 ms cpu, 147->149->75 MB, 151 MB goal, 8 P
gc 8914 @2163.488s 1%: 0.39+12+0.18 ms clock, 3.1+6.3/20/0+1.5 ms cpu, 147->150->75 MB, 151 MB goal, 8 P
gc 8915 @2163.631s 1%: 0.12+7.2+0.17 ms clock, 0.97+14/14/0+1.4 ms cpu, 147->149->76 MB, 151 MB goal, 8 P
gc 8916 @2163.805s 1%: 0.074+9.9+0.20 ms clock, 0.59+5.7/18/0+1.6 ms cpu, 148->150->75 MB, 152 MB goal, 8 P


GOGC=550
gc 9651 @2366.838s 1%: 0.10+13+0.20 ms clock, 0.82+8.7/23/0+1.6 ms cpu, 507->510->79 MB, 529 MB goal, 8 P
gc 9652 @2367.977s 1%: 1.1+9.0+0.17 ms clock, 8.8+5.4/16/0.65+1.3 ms cpu, 494->496->77 MB, 516 MB goal, 8 P
gc 9653 @2369.093s 1%: 0.21+15+1.6 ms clock, 1.7+12/25/0+13 ms cpu, 483->488->80 MB, 505 MB goal, 8 P
gc 9654 @2370.181s 1%: 0.12+10+0.20 ms clock, 1.0+11/18/0+1.6 ms cpu, 498->501->79 MB, 520 MB goal, 8 P

GOGC=1650
gc 11581 @4300.415s 1%: 0.17+11+0.54 ms clock, 1.3+11/20/0+4.3 ms cpu, 2015->2018->118 MB, 2115 MB goal, 8 P
gc 11582 @4306.141s 1%: 0.29+15+0.32 ms clock, 2.3+12/29/0+2.6 ms cpu, 1983->1987->120 MB, 2081 MB goal, 8 P
gc 11583 @4311.562s 1%: 0.26+18+0.27 ms clock, 2.1+13/30/0+2.1 ms cpu, 2002->2007->120 MB, 2101 MB goal, 8 P
gc 11584 @4316.900s 1%: 0.25+11+0.35 ms clock, 2.0+20/20/0+2.8 ms cpu, 2009->2012->118 MB, 2109 MB goal, 8 P
```

其中一些数据的含义, 在分享的时候没有怎么解释, 不过网上的解释几乎没有能完全解释正确. 我这里敲一下.
其实一般关注堆大小和两个 stw 的 wall time 即可.

gc 8913(第 8913 次 gc) @2163.341s(在程序运行的第 2163s) 1%(gc 所有 work 消耗的历史累计 CPU 比例, 所以其实这个数据没太大意义) 0.13(第一个 stw 的 wall time)+14(并发 mark 的 wall time)+0.20(第二个 stw 的 wall time) ms clock, 1.1(第一个 stw 消耗的 CPU 时间)+21(用户程序辅助扫描消耗的 cpu 时间)/22(分配用于 mark 的 P 消耗的 cpu 时间)/0(空闲的 P 用于 mark 的 cpu 时间)+1.6ms(第 2 个 stw 的 cpu 时间) cpu, 147(gc 开始时的堆大小)->149(gc 结束的堆大小)->75MB(gc 结束时的存活堆大小), 151 MB goal(本次 gc 预计结束的堆大小), 8P(8 个 P).

优化





premature optimization is the root of all evil.

个人建议, 没事不要总想着优化, 好好 curd 就好.

一些优化(写业务时用不到就算了)

GODEBUG, GOMAXPROCS, GOGC

涉及文件, CGO较多的程序, 可以将P增大几个.
runtime.GOMAXPROCS(GOMAXPROCS(0)+1)

协程池的重要性远没有Java, CPP中线程池那么重要.
协程的生成不涉及系统调用, 需要的栈资源也很少. 同时P和全局都做了dead G的缓存. 协程池实现的不好, 反而因为协程池里的一把锁影响了扩展性. 至于并发控制, 保护其他资源, 可以选用其他方式.
什么时候需要协程池?
主要还是隔离减少栈扩容和缩容. 有些场景下栈扩容和缩容消耗CPU(可结合pprof查看morestack)的确比较多. 比如长连接, 大量维持连接的协程可以不用扩容栈, 复杂任务交给任务协程处理, 此类协程的数量比较少.

GOGC=200或更多,: GC Pacer会根据GC情况和GOGC参数来计算gc trigger, 增大GOGC, 可降低GC频率, 注意, 会增加触发GC的堆大小.

sync.Pool: 对于频繁分配的对象, 可以使用sync.Pool, 减少分配频次, 进而降低GC频率 (1.13对sync.Pool进行了优化)

全局缓存对象有大量的key的情况, value少用指针
GC并发Mark需要mark存活的对象, 如果value里指针多, 导致mark消耗的CPU很大, 使用一个struct内嵌数据消除指针.

一点点拷贝胜过传指针: 对象在栈上分配, 减少GC频率.
[]byte和string的magic: 慎用, 仅用在不会修改的地方
slice和map的容量初始化: 减少不断加元素时的扩容
json-iterator替换encoding/json等等

当然还是有一些优化方法的.

一点实践

框架或模板集成gops及默认开启pprof

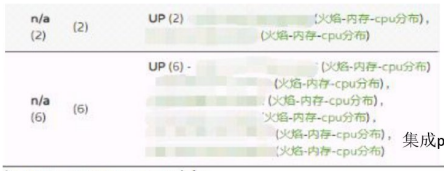
往往有问题才想起没引入pprof，无法查看stack，又需要保留现场.

服务模板代码默认引入一个库开启pprof，集成到服务列表页.
有问题，点一点，通过一个agent，直接获取idc机器上服务的pprof图.

服务模板默认引入一个库封装gops
gops:=golang版(jps + jstack + jstat + jinfo)

https://github.com/google/gops

gops和pprof占用的端口根据服务端口+1，+2.



集成pprof

```
$ gops memstats <pid>
alloc: 22.96MB (24078368 bytes)
total-alloc: 84.27GB (90483014360 bytes)
sys: 70.63MB (74055928 bytes)
mallocs: 1628424898
frees: 1628111117
heap-sys: 62.59MB (65634304 bytes)
heap-idle: 35.53MB (37257216 bytes)
heap-in-use: 27.06MB (28377088 bytes)
heap-released: 32.55MB (34127872 bytes)
heap-objects: 313781
stack-in-use: 1.41MB (1474560 bytes)
stack-sys: 1.41MB (1474560 bytes)
stack-mspan-inuse: 372.80KB (381744 bytes)
stack-mspan-sys: 448.00KB (458752 bytes)
stack-mcache-inuse: 6.78KB (6944 bytes)
stack-mcache-sys: 16.00KB (16384 bytes)
other-sys: 773.62KB (792188 bytes)
gc-sys: 2.50MB (2619392 bytes)
next-gc: when heap-alloc >= 27.80MB (29150848 bytes)
last-gc: 2019-08-10 15:01:05.05691364 +0800 CST
gc-pause-total: 639.073511ms
gc-pause: 17171
num-gc: 11396
enable-gc: true
```
查看runtime内存信息

```
$ gops stack (<pid>|<addr>)
gops stack 85709
goroutine 8 [running]:
runtime/pprof.writeGoroutineStacks(0x13c7bc0, 0xc42000e008, 0xc420ec8520, 0xc420ec8520)
        /Users/jbd/go/src/runtime/pprof/pprof.go:603 +0x79
runtime/pprof.writeGoroutine(0x13c7bc0, 0xc42000e008, 0x2, 0xc428f1c048, 0xc420ec8608)
        /Users/jbd/go/src/runtime/pprof/pprof.go:592 +0x44
runtime/pprof.(*Profile).WriteTo(0x13eeda0, 0x13c7bc0, 0xc42000e008, 0x2, 0xc42000e008, 0x0)
        /Users/jbd/go/src/runtime/pprof/pprof.go:302 +0x3b5
github.com/google/gops/agent.handle(0x13cd560, 0xc42000e008, 0xc420186000, 0x1, 0x1, 0x0, 0x0)
        /Users/jbd/src/github.com/google/gops/agent/agent.go:150 +0x1b3
github.com/google/gops/agent.listen()
        /Users/jbd/src/github.com/google/gops/agent/agent.go:113 +0x2b2
created by github.com/google/gops/agent.Listen
        /Users/jbd/src/github.com/google/gops/agent/agent.go:94 +0x480
```
此图来自官方git
查看go协程栈

我们将 pprof 的开启集成到模板中，并自动选择端口，并集成了 gops 工具，方便查询 runtime 信息，同时在浏览器上可直接点击生成火焰图，pprof 图，非常的方便，也不需要使用者关心.

问题排查的一点思路

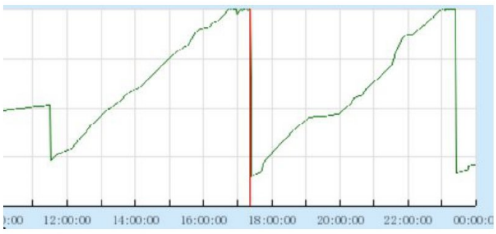
问题排查的一点思路

和Runtime相关的几类问题

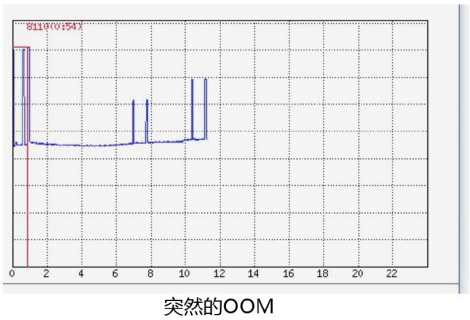
1. 内存慢慢增长OOM: 结合memory inuse_space的pprof和list，加上源码流程即可定位出. 一直把新对象放到全局对象或者长生命周期对象中. 比如长连接，连接池应用或者忘记close http resp body, sql Stmt等.

2. 内存突增OOM: 如果多次分配才OOM，可使用方法1排查. 对于一次就OOM的，比较难抓，可结合go无法分配内存时throw输出的协程栈排查.
比如没有校验参数，调用者填错或恶意，使用传过来的length来进行make([]byte, length)用于编解码

3. 性能问题: 结合火焰图，查看影响性能的热点部分，进行优化: GC频繁，编解码效率低等.



缓慢增长至OOM



突然的OOM

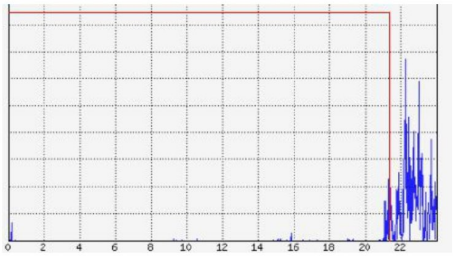
一次有意思的问题排查

一次有趣的排查

一同学找我说自己的服务突然超时很多, 频繁出现页面拉不出数据, 排查了很久, 实在找不出来问题.

1. 查看负载, 内存, 依赖的服务也均正常
2. 框架会捕捉panic, 目前也没有panic发生过
3. cpu的pprof也比较正常
4. 日志打的有点多, 但日志是由业务协程通过channel发给单独一个协程异步打印, 没有channel满的情况.
5. 服务开启了GODEBUG=gctrace=1, 标准输出重定向到一个文件, tail看GC的stw没什么问题, 监控上的GC数据也正常.

看监控发现个很神奇的地方: 内存cache获取>800ms很多. 但是缓存的数据也不多.



cache Get耗时>800ms

负载, 依赖服务都很正常, CPU 利用率也不高, 请求也不多, 就是有很多超时.

一次有趣的排查

关键词: 日志多, gctrace, 重定向

1. Golang gc最后一步gcMarkTermination会判断是否需要打印gctrace信息
此时还是处于STW中
2. 如果是输出console, 纯内存操作, 没影响
3. 早期的服务模板做了重定向操作到文件, 而这个同学的服务现网日志很多, 导致阻塞了gc完成.

GC的STW统计时间反映不出来这个问题

```
// Print gctrace before dropping worldsema. As soon as we drop
// worldsema another cycle could start and smash the stats
// we're trying to print.
if debug.gctrace > 0 {
    util := int(memstats.gc_cpu_fraction * 100)

    var sbuf [24]byte
    printlock()
    print("gc ", memstats.numgc,
        " @", string(itoaDiv(sbuf[:],
uint64(work.tSweepTerm-runtimeInitTime)/1e6, 3)), "s ",
        util, "%: ")
    prev := work.tSweepTerm
    for i, ns := range []int64{work.tMark, work.tMarkTerm, work.tEnd} {
        if i != 0 {
            print("+")
        }
        print(string(fmtNSAsMS(sbuf[:], uint64(ns-prev))))
        prev = ns
    }
    print(" ms clock, ")
    for i, ns := range []int64{sweepTermCpu, gcController.assistTime,
gcController.dedicatedMarkTime + gcController.fractionalMarkTime,
gcController.idleMarkTime, markTermCpu} {
        if i == 2 || i == 3 {
```

该服务在线上打印了 debug 日志, 因为早期的服务模板开启了 gctrace, 框架把 stdout 重定向到一个文件了. 而输出 gctrace 时本来是到 console 的, 输出到文件了, 而磁盘跟不上, 导致 gctrace 日志被阻塞了.

这里更正一下 ppt 中的内容, 并不是因为 gc 没完成而导致其他协程不能运行, 而是后续 gc 无法开启, 导致实质上的 stw.
打印 gc trace 日志时, 已经 start the world 了, 其他协程可以开始运行了. 但是在打印 gctrace 日志时, 还保持着开启 gc 需要的锁, 所以, 打印 gc trace 日志一直没完成, 而 gc 又比较频繁, 比如 0.1s 一次, 这样会导致下一次 gc 开始时无法获取锁, 每一个进入 gc 检查的 g 阻塞, 实际上就造成了 stw.

Runtime 的一点个人总结

Runtime的一点个人总结

思想	作用	示例
并行	减少操作的wall time和阻塞	stw mark -> concurrent mark, stw stack scan, shrink-> 并发mark阶段的逐个g 等
纵向多层次	尽量减少锁竞争和冲突. Per-P无锁 -> 粒度范围比较小的锁->最后才全局和系统调用	调度findrunnable，内存分配mallocgc, stack分配等
横向多个class	找到最适配的, 减少内存浪费和碎片	tinyalloc, 内存分配span机制, 多个class的stack分配等
缓存	减少重新申请	sync.Pool, per-P的mcache, deadg的free list等, 延缓释放归还给mheap的pages
缓冲	放入队列, 减少阻塞, 操作异步化	写屏障的wbBuf, GC标记队列
均衡	负载均衡, 不会因为work太多的而成为瓶颈	调度时从全局runq获取, 从其他P进行work stealing; GC标记工作的本地和全局之间的flush和get.

并行, 纵向多层次, 横向多个 class, 缓存, 缓冲, 均衡.

参考文档

参考文档

描述并发正确性和happens before规则 https://golang.org/ref/mem
Scalable Go Scheduler Design Doc GM模型转为GPM模型
https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit#heading=h.mmq8lm48qfcw
Go Preemptive Scheduler Design Doc Golang支持对运行Go代码的协程进行协作式抢占调度
https://docs.google.com/document/d/1ETuA2IOmnaQ4j81AtTGT40Y4_Jr6_IDASEKg0t0dBR8/edit#heading=h.3pilqarbrc9h
Contiguous stacks 将之前的分裂栈改为连续栈, 大大减少链式栈的创建和destory带来的性能消耗.
https://docs.google.com/document/d/1wAaf1rYoM4S4gtnPh0zOlGzWtrZFQ5suE8qr2sD8uWQ/pub
Golang调度器源码分析 https://colobu.com/2017/05/04/golang-runtime-scheduler/
也谈goroutine调度器 https://tonybai.com/2017/06/23/an-intro-about-goroutine-scheduler/
Golang 内存管理 http://legendtkl.com/2017/04/02/golang-alloc/
runtime: decentralize concurrent sweep termination GC状态检测由集中式改为分布到多个协程
https://go-review.googlesource.com/c/go/+/16391/8
Getting to Go: The Journey of Go's Garbage Collector https://blog.golang.org/ismmkeynote
复盘 GC 算法的发展历程及现状
https://www.infoq.cn/article/development-history-and-current-situation-of-gc-algorithm
Go 1.5 concurrent garbage collector pacing https://docs.google.com/document/d/1wmjrocXIWTr1JxU-3EQBI6BK6KgtiFArkG47XK73xIQ/edit#
Proposal: Eliminate STW stack re-scanning 1.8消除栈重新扫描, 进入sub ms
https://github.com/golang/proposal/blob/master/design/17503-eliminate-rescan.md
twitter golang服务的不同版本golang gc时间对比 https://twitter.com/brianhatfield/status/634166123605331968
Golang源码探索(三) GC的实现原理 https://www.cnblogs.com/zkweb/p/7880099.html
....

C++避坑指南

作者：boolean，腾讯 PCG 后台开发工程师

> 如果，将编程语言比作武功秘籍，C++无异于《九阴真经》。《九阴真经》威力强大、博大精深，经中所载内功、轻功、拳、掌、腿、刀法、剑法、杖法、鞭法、指爪、点穴密技、疗伤法门、闭气神功、移魂大法等等，无所不包，C++亦如是。

C++跟《九阴真经》一样，如果使用不当，很容易落得跟周芷若、欧阳锋、梅超风等一样走火入魔。这篇文章总结了在学习C++过程中容易走火入魔的一些知识点。为了避免篇幅浪费，太常见的误区（如指针和数组、重载、覆盖、隐藏等）在本文没有列出，文中的知识点也没有前后依赖关系，各个知识点基本是互相独立，并没有做什么铺垫，开门见山直接进入正文。

全文链接：https://mp.weixin.qq.com/s/_MmlDCg_9Yx-JA77IwYK4w

现代化 C++ 开发工具 CLion 从入门到精通

作者： allsochen ， 腾讯 PCG 后台开发工程师

这就是 CLion

几年前写过一篇 eclipse C++ 的文章、现在还提 eclipse 的话人民群众会仇恨我，我的朋友和家人也会嘲笑我，唾弃我。因为 JetBrains 家的系列产品 CLion 体验耍 eclipse 好几条街。引用知乎网友对 CLion 和 vim/emacs 的形象比喻，如图所示：



CLion

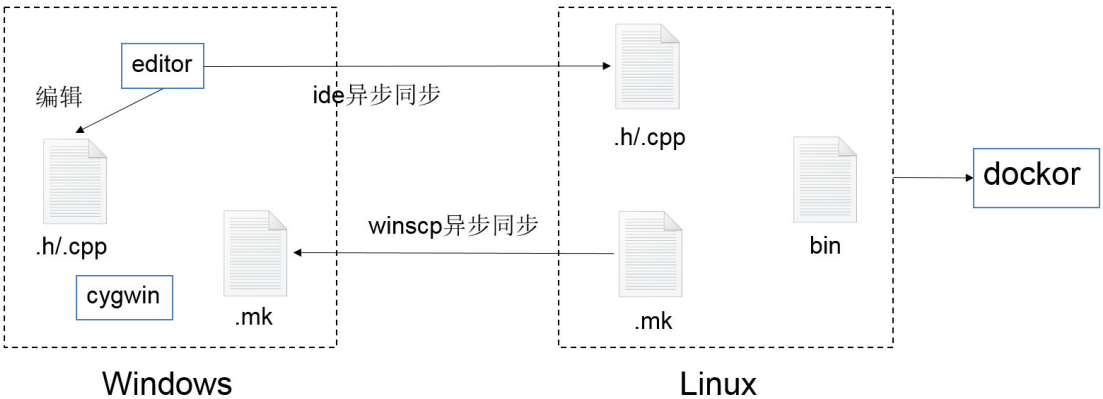


Vim/emacs

从图中可以看出 CLion 的定位是一个完整的现代化 C++工具，包含很多实用的功能，而 vim 则是轻巧锋利，如果你在 CLion 上用 Vim，那就是拿着刀的机器人。这能量你懂的。

起飞前的前奏

根据上次给大家讲解腾讯特有的 windows 开发模式，如下图所示：



Windows 上的环境需要准备炼丹炉：clion 安装、cygwin、jce release 的文件、taf 源码。以上所有涉及到的软件都在 30 的开发机上：/data/home/allsochen/software/clion 上。通过 samba 可以直接看到对应的文件。大家可以直接去取，单不用展示出你的剪刀手。不要直接在 samba 上点击安装，samba 的性能令人堪忧。

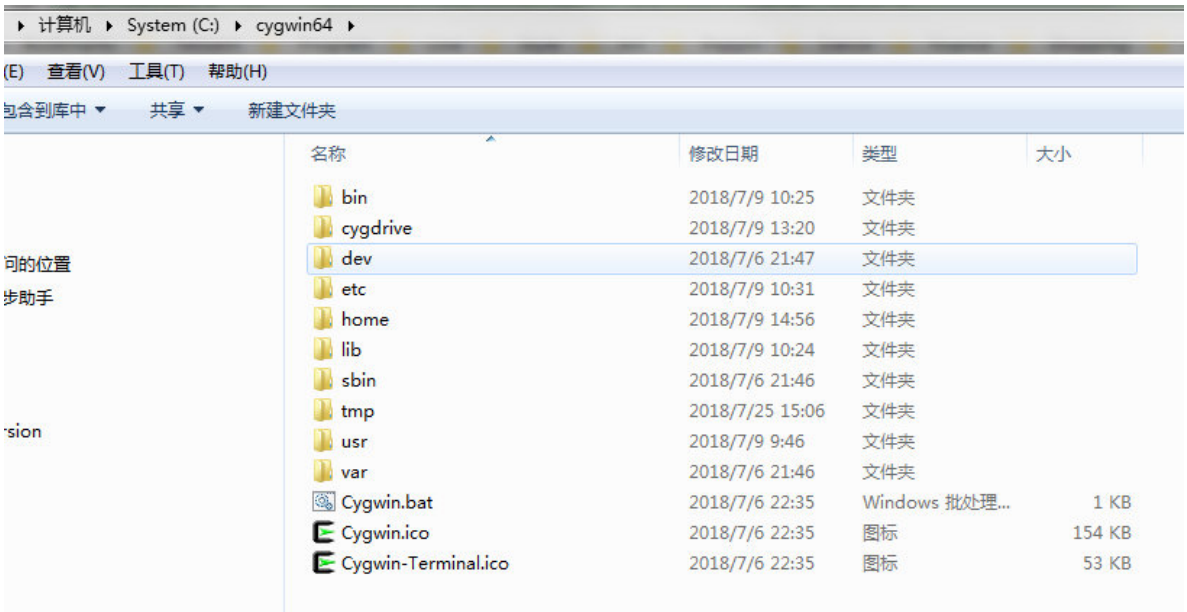
Clion：

1. 安装：直接点击下一步，下一步安装，安装完成
2. 授权：第一次打开提示授权、支持大家购买正版、有 edu 的 email 可以申请免费试用。其他办法你们懂的。
3. 启动参数：由于 clion 是 java 开发的，相对比较耗内存，建议设置最大内存为 4086M，具体的设置在 C:\Program Files\JetBrains\CLion 2018.1.5\bin 目录下的 clion64.exe.vmoptions 文件，把第三行-Xmx2000m 改为-Xmx4086m，同时打开的项目比较多的同学建议再设置大一些，毕竟我们的开发机有 16G，任性。

```
-Xss2m
-Xms256m
-Xmx5086m
-XX:NewSize=128m
-XX:MaxNewSize=128m
-XX:ReservedCodeCacheSize=240m
#-XX:+UseConcMarkSweepGC
-XX:SoftRefLRUPolicyMSPerMB=50
-ea
-Dsun.io.useCanonCaches=false
-Djava.net.preferIPv4Stack=true
-XX:+HeapDumpOnOutOfMemoryError
-XX:-OmitStackTraceInFastThrow
```

Cygwin：

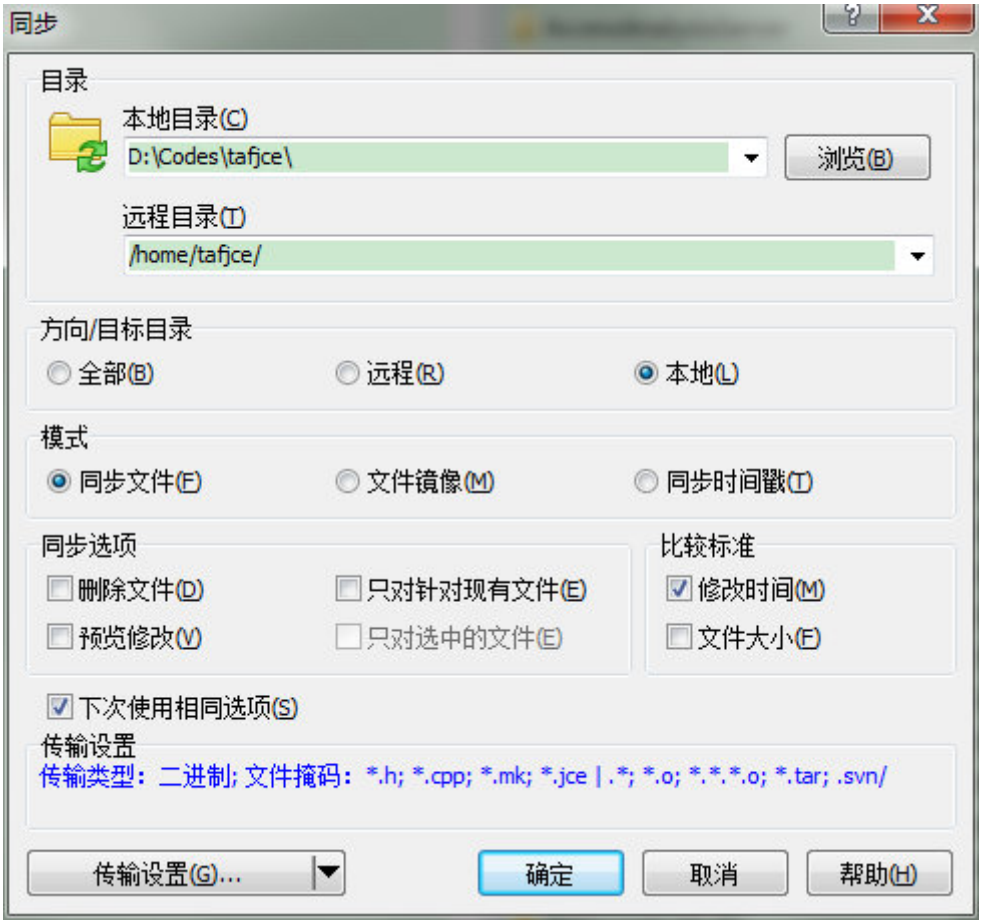
1. 安装：拷贝 cygwin64.zip 到 c 盘解压，解压完的目录如下：



1. Path 设置：为了在 cmd 命令下可以用到 linux 上常用的 grep，find 等命令，在环境变量 path 中加入 C:\cygwin64\bin

Jce 依赖文件：

1. 安装 winscp，通过 winscp 软件把 30 开发机上的/home/tafjce 目录同步到 D:/Codes/tafjce 下
2. 设置同步配置，**请认真看清楚这里的选项，否则误删除开发机上的 release 文件，可能你今年年终奖就 gg 了。**是把远程的目录同步到本地目录，方向/目标目录：本地



1. 设置传输设置：只同步*.h/*.cpp/*.mk/*.jce 文件、排除.svn 目录

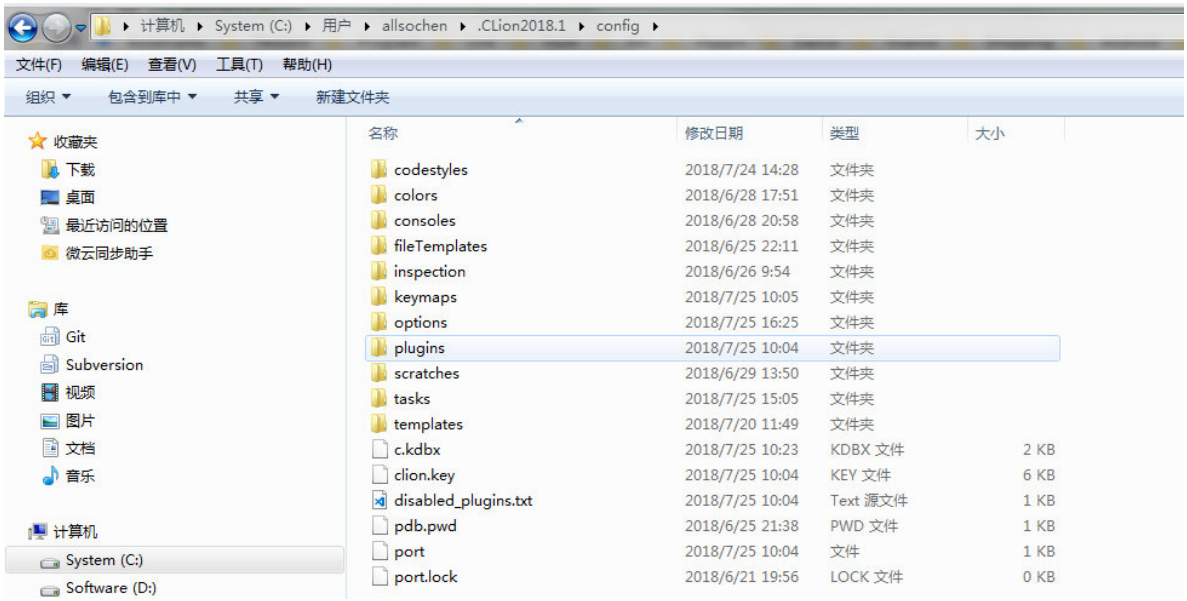


1. 阶段性同步：第一次全量同步后，后续只需要在开发需要的过程中空闲的时间偶尔同步就 ok 了。

字体：推荐 kotlin 之父，JetBrains 公司 CEO 推荐的字体 Lucida Sans Typewriter

CLion 配置：

1. 复制 config.zip 文件到 C:\Users\yourname\.CLion2018.1 下
2. 删除原来的 config 目录
3. 解压 config.zip，解压的目录如下



这个配置的作用是集成了：浏览器 C++开发手册的编程规约、jce editor、m2cmake 等一系列开源的插件和我实现的插件。

Vim 配置：

1. 如果你想把机器人和武士刀合体，尝试下合体后的新力量。
2. 拷贝.ideavimrc 文件到 C:\Users\yourname 的根目录

Taf 支持：

1）拷贝 taf.zip 文件到 D:\Codes\C++目录下解压

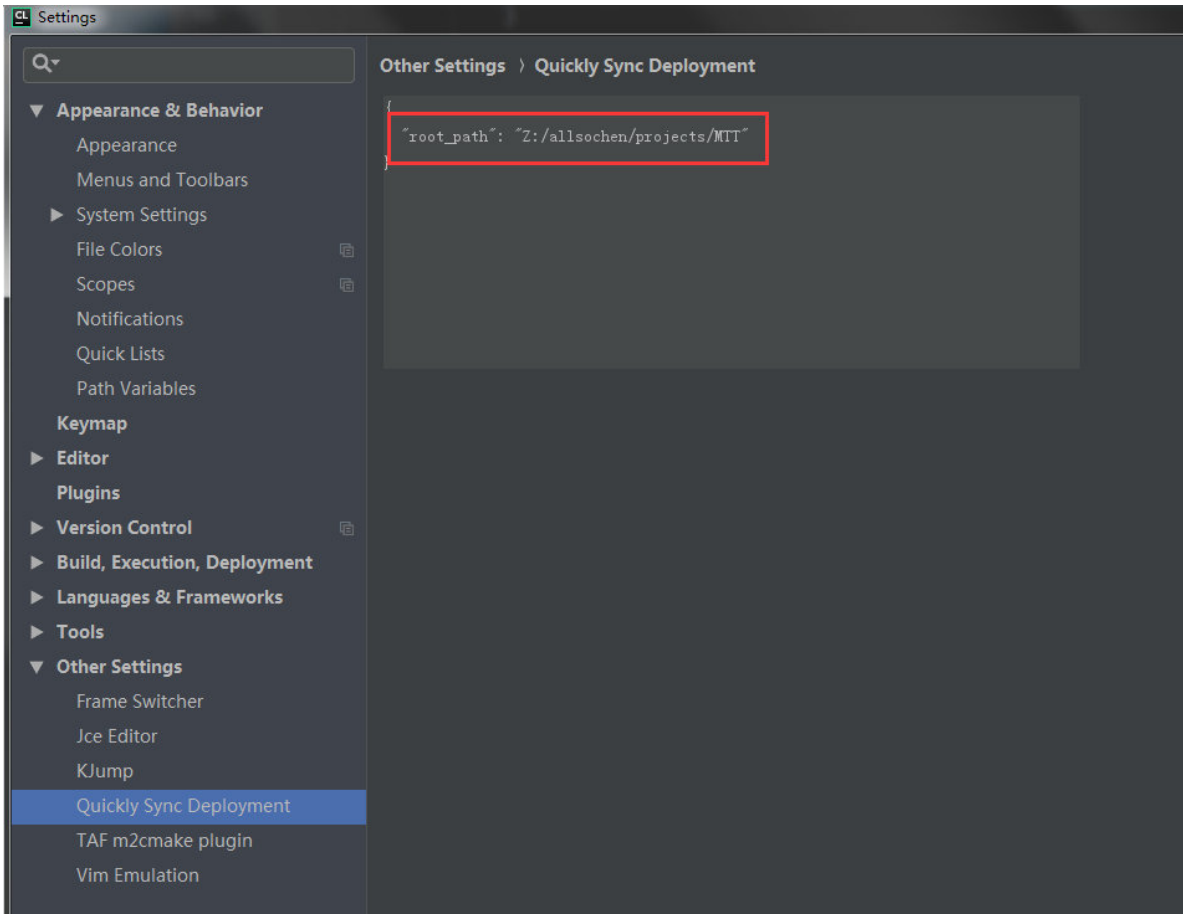
起飞

打开 clion，这时候会提示我的一些无效的工程，直接删除就可以了。然后可以通过 svn、git 或者本地文件引入一个 C++服务，这里切忌不要引入一个具有多个服务的目录，一个项目一个服务。打开一个服务项目后：

自动部署：

把本地的服务目录文件自动同步到目标开发编译的 samba 目录

1. 配置目标同步目录：Files->Settings->Other Settings->Quickly Sync Deployment



> 配置需要同步到自己的开发编译目录，请把 allsochen 改成你的名称

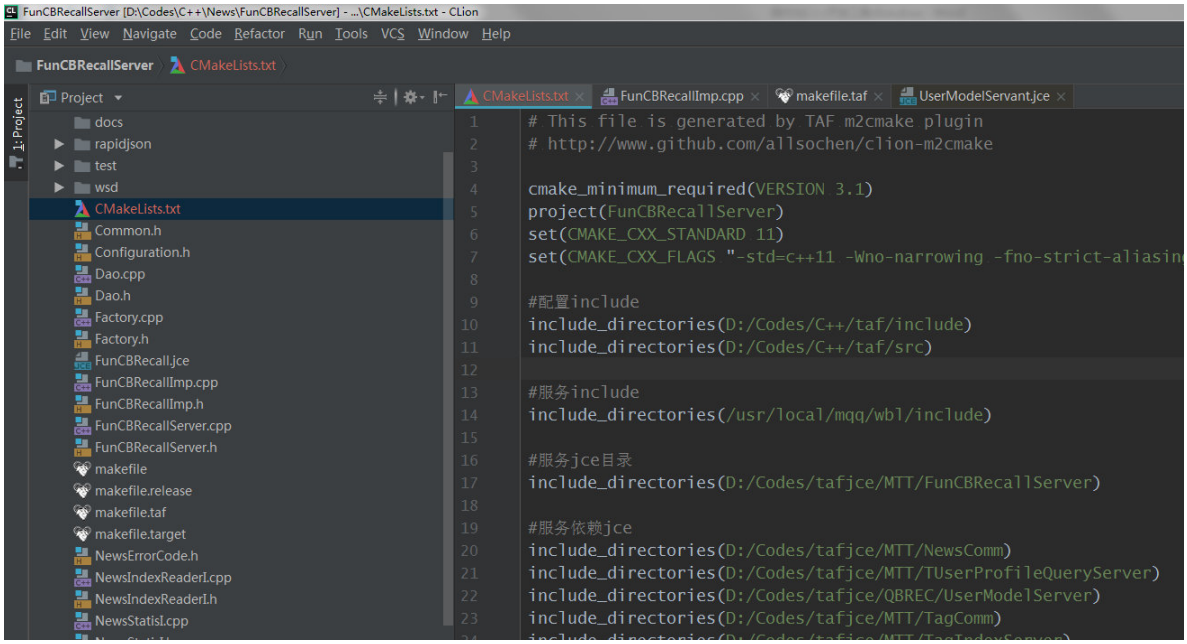
> 2）通过菜单 Tools->Quickly Sync Deployment 进行快速同步部署

3）确认部署：Tools->Deployment->Configuration->OK

> 一定要执行这一步，第二部的插件有 bug，第一次配置会导致不生效，还没有空改插件。

Cmake：

1. 通过 m2cmake 插件把 taf 的 makefile 相关文件生成 CMakeList.txt 文件。选中项目跟目录，右键菜单《Transfer
   TAF makefile to cmake...》



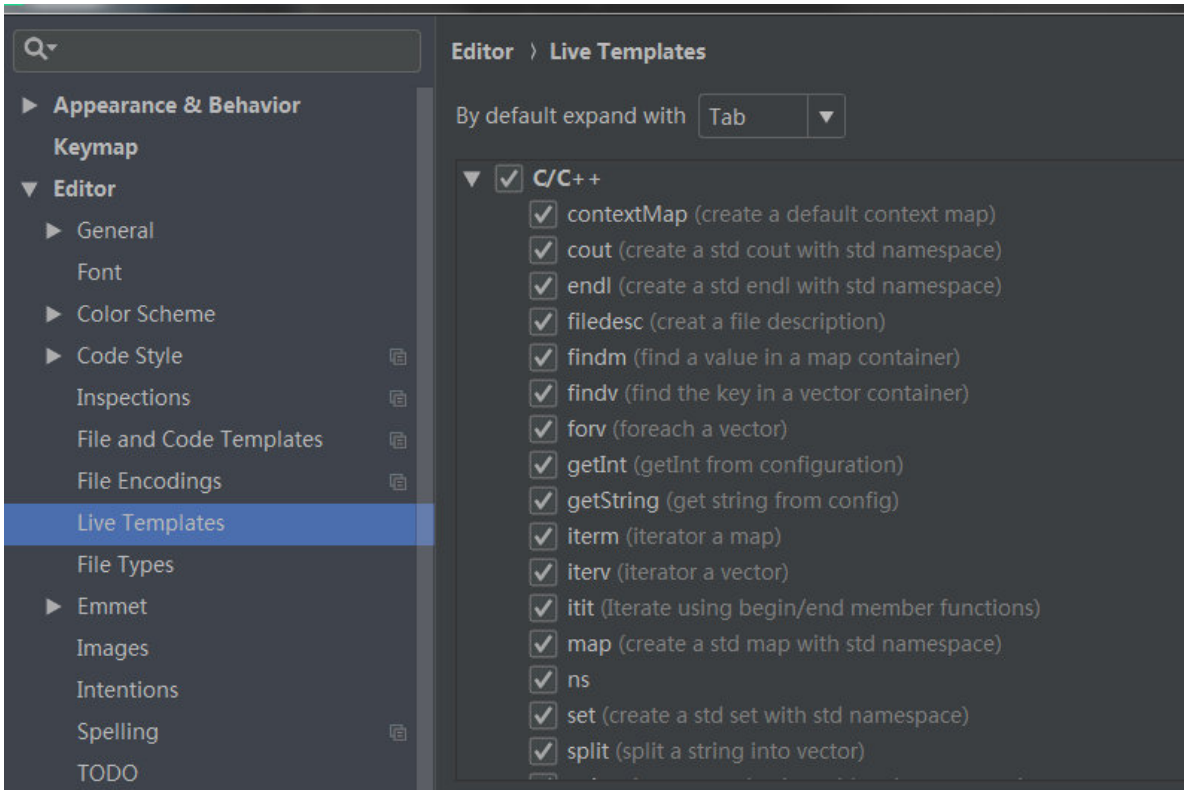
1. 构建项目：选中项目跟目录，右键菜单点击《Reload Cmake Project》，也可以通过快捷键 Alt+K,Alt+M 进行构建

> 构建完成后，尝试下 taf 框架的智能提示和 jce 依赖的智能提示，确认下是否生效。

神秘的力量

1. Live Tempalte

> Live Tempalte 是模板代码的抽象，可以快速生成带有变量的代码模板。比如在打开的 cpp 编辑器中使用 forv，然后按下 tab 键，就会自动生成一个模板 for 循环一个 vector 的模板。从 File->Settings->Editor->Live
> Tempaltes 中可以看到已经配置的模板



1. 常用快捷键：Help->Keymap reference

> JetBrains 的 CEO 说他们的开发工具是不需要用到鼠标的
>
> 这里列举几个常用的：
>
> Shift+Ctrl+N：打开文件
>
> Ctrl+E：打开最近的文件
>
> Ctrl+Y：删除一行
>
> Ctrl+F12：列举出文件的函数，可以输入字母匹配搜索
>
> Ctrl+Q：变量、函数等的说明文档
>
> Ctrl+W：扩展选择
>
> Ctrl+Shift+N：最后编辑点
>
> w：屏幕内光标定位 ,s：再按需要搜索的字母：屏幕搜索字母
>
> Shift+F6：重构命名
>
> Alt+Shift+方向上、下：代码移动
>
> Shift+Enter：以新窗口打开
>
> 更多的需要自己去搜索，只要你用到鼠标的地方，都有一个快捷键适合你。

1. Frame Switch 插件

> 在不同的项目中切换窗口，支持搜索匹配，快捷键 Alt+F2

1. M2cmake 插件

> 自动把 taf 的 makefile 生成 CMakeList.txt 文件
>
> https://github.com/Allsochen/clion-m2cmake

1. QuicklySyncDeployment

> 自动把项目文件部署拷贝到开发机
>
> https://github.com/Allsochen/clion-m2cmake
>
> 目前还有一点小 bug，欢迎有志之士一起维护

1. Jce Editor

> Jce 文件格式化、跳转、语法提示等，格式化 Alt+Ctrl+L
>
> 目前还有一点小 bug，欢迎有志之士一起维护

1. StringStyle

> 提供各种字符串之间的转换，快捷键 Alt+M
>
> https://github.com/Allsochen/StringStyle

代码之美，正则之道

作者： louiszhai，腾讯 IEG 前端开发工程师

> “如果罗列计算机软件领域的伟大发明，我相信绝对不会超过二十项，在这个名单当中，当然应该包括分组交换网络，Web，Lisp，哈希算法，UNIX，编译技术，关系模型，面向对象，XML这些大名鼎鼎的家伙，而正则表达式也绝对不应该被漏掉。”-- Jeffrey Friedl《精通正则表达式》序言。

从 1956 年至今，正则表达式活跃了半个多世纪，其热度依然不减，可见技术半衰期之长，因此，学习正则，不但重要，且受益漫长。

本文涉及 js、php、java、python、bash 等语言，共计 1.2w 字，适合前后端同学系统学习正则并熟练掌握正则！

全文链接：https://mp.weixin.qq.com/s/jH6aimCs5LAL5GBe9bu8rg

编写可靠 bash 脚本的一些技巧

作者： astarsun，腾讯搜索工程师

写过很多 bash 脚本的人都知道，bash 的坑不是一般的多。 其实 bash 本身并不是一个很严谨的语言，但是很多时候也不得不用。以下总结了一些编写可靠的 bash 脚本的小 tips。

0. set -x -e -u -o pipefail

在写脚本时，在一开始（Shebang 之后）加上下面这一句，或者它的缩略版，能避免很多问题，更重要的是能让很多隐藏的问题暴露出来：

```
set -xeuo pipefail
```

下面说明每个参数的作用，以及一些例外的处理方式：

-x： 在执行每一个命令之前把经过变量展开之后的命令打印出来。

这个对于 debug 脚本、输出 Log 时非常有用。 正式运行的脚本也可以不加。

-e： 遇到一个命令失败（返回码非零）时，立即退出。

bash 跟其它的脚本语言最大的不同点之一，应该就是遇到异常时继续运行下一条命令。 这在很多时候会遇到意想不到的问题。加上 -e ，会让 bash 在遇到一个命令失败时，立即退出。

如果有时确实需要忽略个别命令的返回码，可以用 || true 。如：

```
some_cmd || true          # 即使some_cmd失败了，仍然会继续运行
some_cmd || RET=$?        # 或者可以这样来收集some_cmd的返回码，供后面的逻辑判断使用
```

但是在管道串起多条命令的情况下，只有最后一条命令失败时才会退出。如果想让管道中任意一条命令失败就退出，就要用后面提到的-o pipefail 了。

加-e 有时候可能会不太方便，动不动就退出。但还是应该坚持所谓的**fail-fast 原则**，也就是有异常时停止正常运行，而不是继续尝试运行可能存在缺陷的过程。如果有命令可以明确忽略异常，那可以用上面提到的 || true 等方式明确地忽略之。

-u： 试图使用未定义的变量，就立即退出。

如果在 bash 里使用一个未定义的变量，默认是会展开成一个空串。有时这种行为会导致问题，比如：

```
rm -rf $MYDIR/data
```

如果 MYDIR 变量因为某种原因没有赋值，这条命令就会变成 rm -rf /data 。 这就比较搞笑了。。 使用 -u 可以避免这种情况。

但有时候在已经设置了-u 后，某些地方还是希望能把未定义变量展开为空串，可以这样写：

```
${SOME_VAR:-}
#  bash变量展开语法，可以参考：
https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-
Expansion.html
```

-o pipefail： 只要管道中的一个子命令失败，整个管道命令就失败。

pipefail 与-e 结合使用的话，就可以做到管道中的一个子命令失败，就退出脚本。

1. 防止重叠运行

在一些场景中，我们通常不希望一个脚本有多个实例在同时运行。比如用 crontab 周期性运行脚本时，有时不希望上一个轮次还没运行完，下一个轮次就开始运行了。 这时可以用 flock 命令来解决。 flock 通过文件锁的方式来保证独占运行，并且还有一个好处是进程退出时，文件锁也会自动释放，不需要额外处理。

用法 1： 假设你的入口脚本是 myscript.sh，可以新建一个脚本，通过 flock 来运行它：

```
# flock --wait 超时时间   -e 锁文件   -c "要执行的命令"
# 例如：
flock  --wait 5  -e "lock_myscript"  -c "bash myscript.sh"
```

用法 2： 也可以在原有脚本里使用 flock。 可以把文件打开为一个文件描述符，然后使用 flock 对它上锁（flock 可以接受文件描述符参数）。

```
exec 123<>lock_myscript   # 把lock_myscript打开为文件描述符123
flock  --wait 5  123 || { echo 'cannot get lock, exit'; exit 1; }
```

2. 意外退出时杀掉所有子进程

我们的脚本通常会启动好多子脚本和子进程，当父脚本意外退出时，子进程其实并不会退出，而是继续运行着。 如果脚本是周期性运行的，有可能发生一些意想不到的问题。

在 stackoverflow 上找到的一个方法，原理就是利用 trap 命令在脚本退出时 kill 掉它整个进程组。
把下面的代码加在脚本开头区，实测管用：

```
trap "trap - SIGTERM && kill -- -$$" SIGINT SIGTERM EXIT
```

不过如果父进程是用 SIGKILL (kill -9) 杀掉的，就不行了。因为 SIGKILL 时，进程是没有机会运行任何代码的。

3. timeout 限制运行时间

有时候需要对命令设置一个超时时间。这时可以使用 timeout 命令，用法很简单：

```
timeout 600s  some_command arg1 arg2
```

命令在超时时间内运行结束时，返回码为 0，否则会返回一个非零返回码。

timeout 在超时时默认会发送 TERM 信号，也可以用 -s 参数让它发送其它信号。

4. 连续管道时，考虑使用 tee 将中间结果落盘，以便查问题

有时候我们会用到把好多条命令用管道串在一起的情况。如 cmd1 | cmd2 | cmd3 | ...这样会让问题变得难以排查，因为中间数据我们都看不到。

如果改成这样的格式：

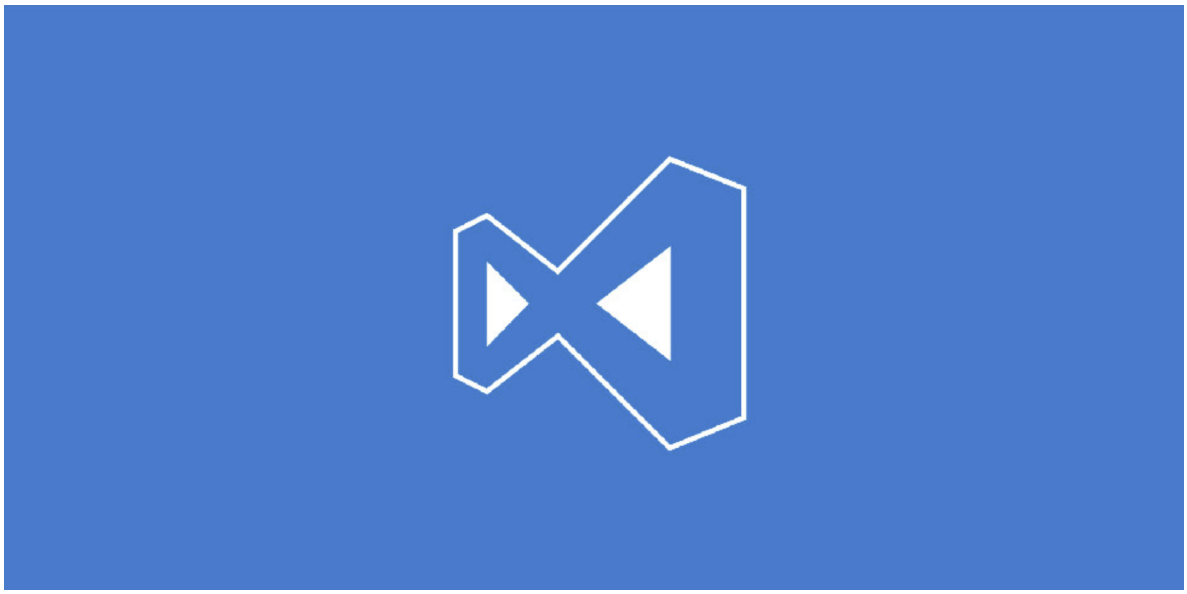
```
cmd1 > out1.dat
cat out1 | cmd2 > out2.dat
cat out2 | cmd3 > out3.dat
```

性能又不太好，因为这样 cmd1, cmd2, cmd3 是串行运行的，这时可以用 tee 命令：

```
cmd1 | tee out1.dat | cmd2 | tee out2.dat | cmd3 > out3.dat
```

微软 VSCode IDE 源码分析

作者： zanewang，腾讯 CSIG web 前端开发工程师



目录

- (1)简介
- (2)技术架构
- (3)启动主进程
- (4)实例化服务
- (5)事件分发
- (6)进程通信

- (7)主要窗口
- (8)开发调试

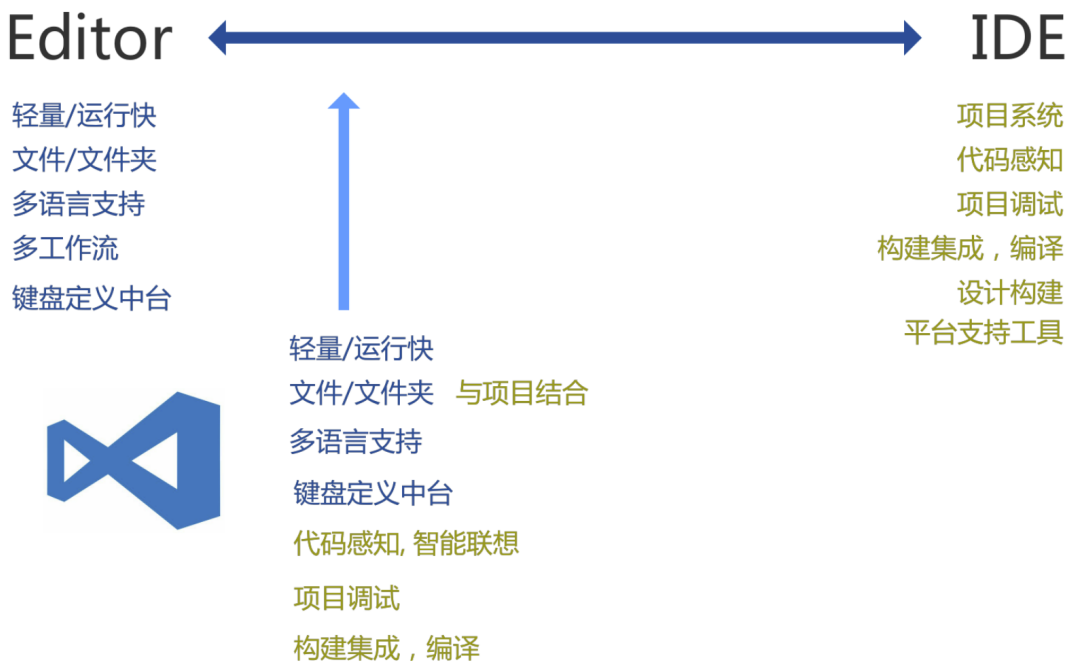
1.简介

Visual Studio Code(简称 VSCode) 是开源免费的 IDE 编辑器，原本是微软内部使用的云编辑器 (Monaco)。

git 仓库地址： https://github.com/microsoft/vscode

通过 Eletron 集成了桌面应用，可以跨平台使用，开发语言主要采用微软自家的 TypeScript。整个项目结构比较清晰，方便阅读代码理解。成为了最流行跨平台的桌面 IDE 应用

微软希望 VSCode 在保持核心轻量级的基础上，增加项目支持，智能感知，编译调试。



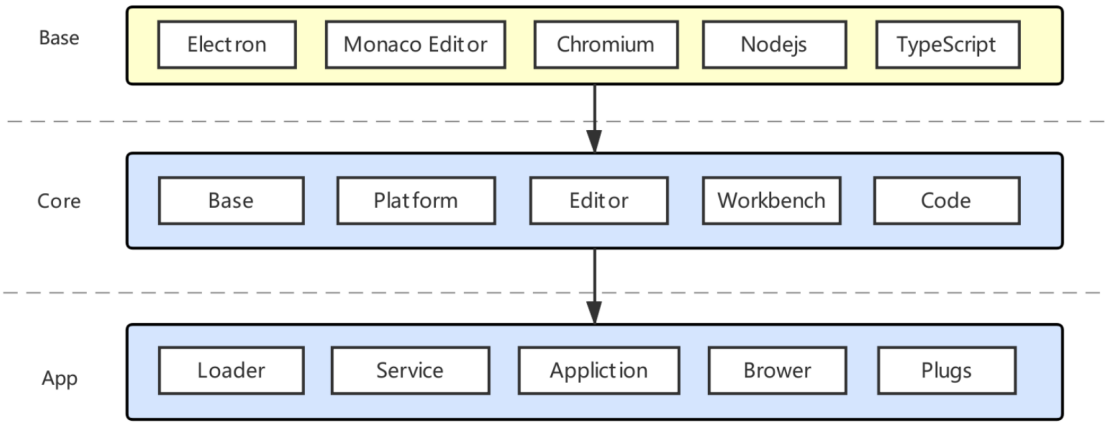
编译安装

下载最新版本,目前我用的是 1.37.1 版本官方的 wiki 中有编译安装的说明 How to Contribute

Linux, Window, MacOS 三个系统编译时有些差别，参考官方文档，在编译安装依赖时如果遇到 connect timeout, 需要进行科学上网。

> 需要注意的一点 运行环境依赖版本 Nodejs x64 version >= 10.16.0, < 11.0.0, python 2.7(3.0 不能正常执行)

2.技术架构



Electron

> Electron 是一个使用 JavaScript, HTML 和 CSS 等 Web 技术创建原生程序的框架，它负责比较难搞的部分，你只需把精力放在你的应用的核心上即可 (Electron = Node.js + Chromium + Native API)

Monaco Editor

> Monaco Editor是微软开源项目, 为 VS Code 提供支持的代码编辑器，运行在浏览器环境中。编辑器提供代码提示，智能建议等功能。供开发人员远程更方便的编写代码，可独立运行。

TypeScript

> TypeScript是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，而且本质上向这个语言添加了可选的静态类型和基于类的面向对象编程

目录结构

```
├── build          # gulp编译构建脚本
├── extensions     # 内置插件
├── product.json   # App meta信息
├── resources      # 平台相关静态资源
├── scripts        # 工具脚本，开发/测试
├── src            # 源码目录
└── typings        # 函数语法补全定义
└── vs
    ├── base         # 通用工具/协议和UI库
    |   ├── browser # 基础UI组件，DOM操作
    |   ├── common  # diff描述，markdown解析器，worker协议，各种工具函数
    |   ├── node    # Node工具函数
    |   ├── parts   # IPC协议（Electron、Node），quickopen、tree组件
    |   ├── test    # base单测用例
    |   └── worker  # Worker factory和main Worker（运行IDE Core：Monaco）
    ├── code         # VSCode主运行窗口
    ├── editor         # IDE代码编辑器
    |   ├── browser      # 代码编辑器核心
    |   ├── common       # 代码编辑器核心
    |   ├── contrib      # vscode 与独立 IDE共享的代码
    |   └── standalone  # 独立 IDE 独有的代码
    ├── platform       # 支持注入服务和平台相关基础服务（文件、剪切板、窗体、状态栏）
    ├── workbench      # 工作区UI布局，功能主界面
    |   ├── api              #
```

```
    |   ├── browser          #
    |   ├── common           #
    |   ├── contrib          #
    |   ├── electron-browser #
    |   ├── services         #
    |   └── test             #
    ├── css.build.js  # 用于插件构建的CSS loader
    ├── css.js        # CSS loader
    ├── editor        # 对接IDE Core（读取编辑/交互状态），提供命令、上下文菜单、hover、
snippet等支持
    ├── loader.js     # AMD loader（用于异步加载AMD模块）
    ├── nls.build.js  # 用于插件构建的NLS loader
    └── nls.js        # NLS（National Language Support）多语言loader
```

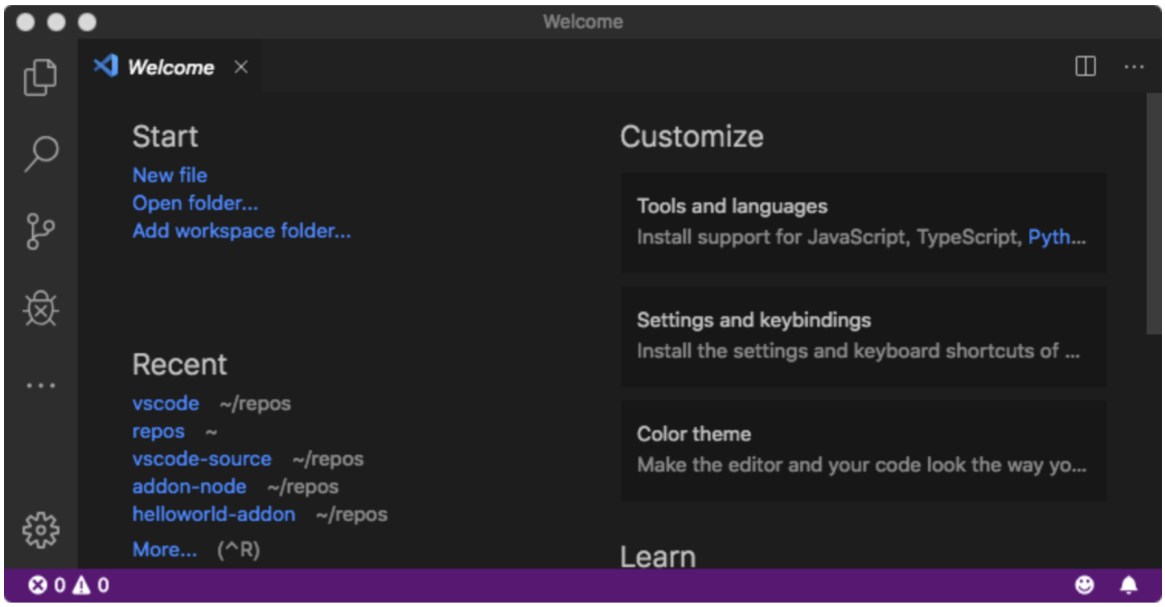
核心层

- base: 提供通用服务和构建用户界面
- platform: 注入服务和基础服务代码
- editor: 微软 Monaco 编辑器，也可独立运行使用
- wrokbench: 配合 Monaco 并且给 viewlets 提供框架：如：浏览器状态栏，菜单栏利用 electron 实现桌面程序

核心环境

整个项目完全使用 typescript 实现，electron 中运行主进程和渲染进程，使用的 api 有所不同，所以在 core 中每个目录组织也是按照使用的 api 来安排，运行的环境分为几类：

- common: 只使用 javascritp api 的代码，能在任何环境下运行
- browser: 浏览器 api, 如操作 dom; 可以调用 common
- node: 需要使用 node 的 api,比如文件 io 操作
- electron-brower: 渲染进程 api, 可以调用 common, brower, node, 依赖electron renderer-process API
- electron-main: 主进程 api, 可以调用: common, node 依赖于electron main-process AP



3.启动主进程

Electron 通过 package.json 中的 main 字段来定义应用入口。

main.js 是 vscode 的入口。

- src/main.js
  _ vs/code/electron-main/main.ts
  _ vs/code/electron-main/app.ts
  _ vs/code/electron-main/windows.ts
  _ vs/workbench/electron-browser/desktop.main.ts * vs/workbench/browser/workbench.ts

```
app.once('ready', function () {
    //启动追踪，后面会讲到，跟性能检测优化相关。
    if (args['trace']) {
        // @ts-ignore
        const contentTracing = require('electron').contentTracing;

        const traceOptions = {
            categoryFilter: args['trace-category-filter'] || '*',
            traceOptions: args['trace-options'] || 'record-until-full,enable-
sampling'
        };

        contentTracing.startRecording(traceOptions, () => onReady());
    } else {
        onReady();
    }
});
function onReady() {
    perf.mark('main:appReady');

    Promise.all([nodeCachedDataDir.ensureExists(),
userDefinedLocale]).then(([cachedDataDir, locale]) => {
        //1. 这里尝试获取本地配置信息，如果有的话会传递到startup
        if (locale && !nlsConfiguration) {
            nlsConfiguration = lp.getNLSConfiguration(product.commit,
userDataPath, metaDataFile, locale);
        }

        if (!nlsConfiguration) {
            nlsConfiguration = Promise.resolve(undefined);
        }


        nlsConfiguration.then(nlsConfig => {

            //4. 首先会检查用户语言环境配置，如果没有设置默认使用英语
            const startup = nlsConfig => {
                nlsConfig._languagePackSupport = true;
                process.env['VSCODE_NLS_CONFIG'] = JSON.stringify(nlsConfig);
                process.env['VSCODE_NODE_CACHED_DATA_DIR'] = cachedDataDir ||
'';

                perf.mark('willLoadMainBundle');
                //使用微软的loader组件加载electron-main/main文件
                require('./bootstrap-amd').load('vs/code/electron-main/main', ()
=> {
                    perf.mark('didLoadMainBundle');
                });
            };

            // 2. 接收到有效的配置传入是其生效，调用startup
```

```
            if (nlsConfig) {
                startup(nlsConfig);
            }

            // 3. 这里尝试使用本地的应用程序
            // 应用程序设置区域在ready事件后才有效
            else {
                let appLocale = app.getLocale();
                if (!appLocale) {
                    startup({ locale: 'en', availableLanguages: {} });
                } else {

                    // 配置兼容大小写敏感，所以统一转换成小写
                    appLocale = appLocale.toLowerCase();
                    // 这里就会调用config服务，把本地配置加载进来再调用startup
                    lp.getNLSConfiguration(product.commit, userDataPath,
metaDataFile, appLocale).then(nlsConfig => {
                        if (!nlsConfig) {
                            nlsConfig = { locale: appLocale, availableLanguages:
{} };
                        }

                        startup(nlsConfig);
                    });
                }
            }
        });
    }, console.error);
}
```

vs/code/electron-main/main.ts

electron-main/main 是程序真正启动的入口,进入 main process 初始化流程.

这里主要做了两件事情:

1. 初始化 Service
2. 启动主实例

直接看 startup 方法的实现,基础服务初始化完成后会加载 CodeApplication, mainIpcServer, instanceEnvironment，调用 startup 方法启动 APP

```
private async startup(args: ParsedArgs): Promise<void> {

        //spdlog 日志服务
        const bufferLogService = new BufferLogService();

        // 1. 调用 createServices
        const [instantiationService, instanceEnvironment] =
this.createServices(args, bufferLogService);
        try {

            // 1.1 初始化Service服务
            await instantiationService.invokeFunction(async accessor => {
                // 基础服务，包括一些用户数据，缓存目录
                const environmentService = accessor.get(IEnvironmentService);
                // 配置服务
```

```
                const configurationService =
accessor.get(IConfigurationService);
                // 持久化数据
                const stateService = accessor.get(IStateService);

                try {
                    await this.initServices(environmentService,
configurationService as ConfigurationService, stateService as StateService);
                } catch (error) {

                    // 抛出错误对话框
                    this.handleStartupDataDirError(environmentService, error);

                    throw error;
                }
            });

            // 1.2 启动实例
            await instantiationService.invokeFunction(async accessor => {
                const environmentService = accessor.get(IEnvironmentService);
                const logService = accessor.get(ILogService);
                const lifecycleService = accessor.get(ILifecycleService);
                const configurationService =
accessor.get(IConfigurationService);

                const mainIpcServer = await this.doStartup(logService,
environmentService, lifecycleService, instantiationService, true);

                bufferLogService.logger = new SpdLogService('main',
environmentService.logsPath, bufferLogService.getLevel());
                once(lifecycleService.onWillShutdown)(() =>
(configurationService as ConfigurationService).dispose());

                return instantiationService.createInstance(CodeApplication,
mainIpcServer, instanceEnvironment).startup();
            });
        } catch (error) {
            instantiationService.invokeFunction(this.quit, error);
        }
    }
```

Service

这里通过 createService 创建一些基础的 Service

运行环境服务 EnvironmentService

src/vs/platform/environment/node/environmentService.ts

通过这个服务获取当前启动目录，日志目录，操作系统信息，配置文件目录，用户目录等。

日志服务 MultiplexLogService

src/vs/platform/log/common/log.ts

默认使用控制台日志 ConsoleLogMainService 其中包含性能追踪和释放信息，日志输出级别

配置服务 ConfigurationService

src/vs/platform/configuration/node/configurationService.ts

从运行环境服务获取内容

生命周期服务 LifecycleService

src/vs/platform/lifecycle/common/lifecycleService.ts

监听事件，electron app 模块 比如：ready， window-all-closed，before-quit

可以参考官方electron app 文档

状态服务 StateService

src/vs/platform/state/node/stateService.ts

通过 FileStorage 读写 storage.json 存储，里记录一些与程序运行状态有关的键值对

请求服务 RequestService

src/vs/platform/request/browser/requestService.ts

这里使用的是原生 ajax 请求，实现了 request 方法

主题服务 ThemeMainService

src/vs/platform/theme/electron-main/themeMainService.ts

这里只设置背景颜色，通过 getBackgroundColor 方法 IStateService 存储

签名服务 SignService

src/vs/platform/sign/node/signService.ts

```
private createServices(args: ParsedArgs, bufferLogService: BufferLogService): [IInstantiationService, typeof process.env] {

    //服务注册容器
    const services = new ServiceCollection();

    const environmentService = new EnvironmentService(args, process.execPath);
    const instanceEnvironment = this.patchEnvironment(environmentService); // Patch `process.env` with the instance's environment
    services.set(IEnvironmentService, environmentService);

    const logService = new MultiplexLogService([new ConsoleLogMainService(getLogLevel(environmentService)), bufferLogService]);
    process.once('exit', () => logService.dispose());
    //日志服务
    services.set(ILogService, logService);
    //配置服务
    services.set(IConfigurationService, new ConfigurationService(environmentService.settingsResource));
    //生命周期
    services.set(ILifecycleService, new SyncDescriptor(LifecycleService));
    //状态存储
    services.set(IStateService, new SyncDescriptor(StateService));
    //网络请求
    services.set(IRequestService, new SyncDescriptor(RequestService));
    //主题设定
    services.set(IThemeMainService, new SyncDescriptor(ThemeMainService));
    //签名服务
    services.set(ISignService, new SyncDescriptor(SignService));
```

```
    return [new InstantiationService(services, true), instanceEnvironment];
}
```

4.实例化服务

SyncDescriptor 负责注册这些服务，当用到该服务时进程实例化使用

src/vs/platform/instantiation/common/descriptors.ts

```
export class SyncDescriptor<T> {
    readonly ctor: any;
    readonly staticArguments: any[];
    readonly supportsDelayedInstantiation: boolean;
    constructor(ctor: new (...args: any[]) => T, staticArguments: any[] = [],
supportsDelayedInstantiation: boolean = false) {
        this.ctor = ctor;
        this.staticArguments = staticArguments;
        this.supportsDelayedInstantiation = supportsDelayedInstantiation;
    }
}
```

main.ts 中 startup 方法调用 invokeFunction.get 实例化服务

```
await instantiationService.invokeFunction(async accessor => {
    const environmentService = accessor.get(IEnvironmentService);
    const configurationService = accessor.get(IConfigurationService);
    const stateService = accessor.get(IStateService);
    try {
        await this.initServices(environmentService, configurationService as
ConfigurationService, stateService as StateService);
    } catch (error) {

        // Show a dialog for errors that can be resolved by the user
        this.handleStartupDataDirError(environmentService, error);

        throw error;
    }
});
```

get 方法调用_getOrCreateServiceInstance，这里第一次创建会存入缓存中下次实例化对象时会优先从缓存中获取对象。

src/vs/platform/instantiation/common/instantiationService.ts

```
invokeFunction<R, TS extends any[] = []>(fn: (accessor: ServicesAccessor,
...args: TS) => R, ...args: TS): R {
    let _trace = Trace.traceInvocation(fn);
    let _done = false;
    try {
        const accessor: ServicesAccessor = {
            get: <T>(id: ServiceIdentifier<T>, isOptional?: typeof optional) =>
{

                if (_done) {
                    throw illegalState('service accessor is only valid during
the invocation of its target method');
```

```
                }

                const result = this._getOrCreateServiceInstance(id, _trace);
                if (!result && isOptional !== optional) {
                    throw new Error(`[invokeFunction] unknown service '${id}'`);
                }
                return result;
            }
        };
        return fn.apply(undefined, [accessor, ...args]);
    } finally {
        _done = true;
        _trace.stop();
    }
}
private _getOrCreateServiceInstance<T>(id: ServiceIdentifier<T>, _trace: Trace):
T {
    let thing = this._getServiceInstanceOrDescriptor(id);
    if (thing instanceof SyncDescriptor) {
        return this._createAndCacheServiceInstance(id, thing, _trace.branch(id,
true));
    } else {
        _trace.branch(id, false);
        return thing;
    }
}
```

vs/code/electron-main/app.ts

这里首先触发 CodeApplication.startup()方法， 在第一个窗口打开 3 秒后成为共享进程，

```
async startup(): Promise<void> {
    ...

    // 1. 第一个窗口创建共享进程
    const sharedProcess =
this.instantiationService.createInstance(SharedProcess, machineId,
this.userEnv);
    const sharedProcessClient = sharedProcess.whenReady().then(() =>
connect(this.environmentService.sharedIPCHandle, 'main'));
    this.lifecycleService.when(LifecycleMainPhase.AfterWindowOpen).then(() => {
        this._register(new RunOnceScheduler(async () => {
            const userEnv = await getShellEnvironment(this.logService,
this.environmentService);

            sharedProcess.spawn(userEnv);
        }, 3000)).schedule();
    });
    // 2. 创建app实例
    const appInstantiationService = await this.createServices(machineId,
trueMachineId, sharedProcess, sharedProcessClient);


    // 3. 打开一个窗口 调用

    const windows = appInstantiationService.invokeFunction(accessor =>
this.openFirstWindow(accessor, electronIpcServer, sharedProcessClient));
```

```
    // 4. 窗口打开后执行生命周期和授权操作
    this.afterWindowOpen();
    ...

    //vscode结束了性能问题的追踪
    if (this.environmentService.args.trace) {
        this.stopTracingEventually(windows);
    }
}
```

openFirstWindow 主要实现 CodeApplication.openFirstWindow 首次开启窗口时，创建 Electron 的 IPC，使主进程和渲染进程间通信。window 会被注册到 sharedProcessClient，主进程和共享进程通信根据 environmentService 提供的参数(path,uri)调用 windowsMainService.open 方法打开窗口

```
private openFirstWindow(accessor: ServicesAccessor, electronIpcServer:
ElectronIPCServer, sharedProcessClient: Promise<Client<string>>): ICodeWindow[]
{

        ...
        // 1. 注入Electron IPC Service, windows窗口管理，菜单栏等服务

        // 2. 根据environmentService进行参数配置
        const macOpenFiles: string[] = (<any>global).macOpenFiles;
        const context = !!process.env['VSCODE_CLI'] ? OpenContext.CLI :
OpenContext.DESKTOP;
        const hasCliArgs = hasArgs(args._);
        const hasFolderURIs = hasArgs(args['folder-uri']);
        const hasFileURIs = hasArgs(args['file-uri']);
        const noRecentEntry = args['skip-add-to-recently-opened'] === true;
        const waitMarkerFileURI = args.wait && args.waitMarkerFilePath ?
URI.file(args.waitMarkerFilePath) : undefined;

        ...

        // 打开主窗口，默认从执行命令行中读取参数
        return windowsMainService.open({
            context,
            cli: args,
            forceNewWindow: args['new-window'] || (!hasCliArgs && args['unity-
launch']),
            diffMode: args.diff,
            noRecentEntry,
            waitMarkerFileURI,
            gotoLineMode: args.goto,
            initialStartup: true
        });
    }
```

vs/code/electron-main/windows.ts

接下来到了 electron 的 windows 窗口，open 方法在 doOpen 中执行窗口配置初始化，最终调用 openInBrowserWindow -> 执行 doOpenInBrowserWindow 是其打开 window,主要步骤如下：

```
private openInBrowserWindow(options: IOpenBrowserWindowOptions): ICodeWindow {
```

```
    ...
    // New window
    if (!window) {
        //1.判断是否全屏创建窗口
         ...
        // 2. 创建实例窗口
        window = this.instantiationService.createInstance(CodeWindow, {
            state,
            extensionDevelopmentPath: configuration.extensionDevelopmentPath,
            isExtensionTestHost: !!configuration.extensionTestsPath
        });

        // 3.添加到当前窗口控制器
        WindowsManager.WINDOWS.push(window);

        // 4.窗口监听器
        window.win.webContents.removeAllListeners('devtools-reload-page'); //
remove built in listener so we can handle this on our own
        window.win.webContents.on('devtools-reload-page', () =>
this.reload(window!));
        window.win.webContents.on('crashed', () => this.onWindowError(window!,
WindowError.CRASHED));
        window.win.on('unresponsive', () => this.onWindowError(window!,
WindowError.UNRESPONSIVE));
        window.win.on('closed', () => this.onWindowClosed(window!));

        // 5.注册窗口生命周期
        (this.lifecycleService as LifecycleService).registerWindow(window);
    }

    ...

    return window;
}
```

doOpenInBrowserWindow 会调用 window.load 方法 在 window.ts 中实现

```
load(config: IWindowConfiguration, isReload?: boolean, disableExtensions?:
boolean): void {

    ...

    // Load URL
    perf.mark('main:loadWindow');
    this._win.loadURL(this.getUrl(configuration));

    ...
}

private getUrl(windowConfiguration: IWindowConfiguration): string {

    ...
    //加载欢迎屏幕的html
    let configUrl = this.doGetUrl(config);
    ...
    return configUrl;
}
```

```
//默认加载 vs/code/electron-browser/workbench/workbench.html
private doGetUrl(config: object): string {
    return `${require.toUrl('vs/code/electron-
browser/workbench/workbench.html')}?
config=${encodeURIComponent(JSON.stringify(config))}`;
}
```

main process 的使命完成, 主界面进行构建布局。

在 workbench.html 中加载了 workbench.js，这里调用 return require('vs/workbench/electron-browser/desktop.main').main(configuration) 实现对主界面的展示。

vs/workbench/electron-browser/desktop.main.ts

创建工作区，调用 workbench.startup()方法，构建主界面展示布局

```
...
async open(): Promise<void> {
    const services = await this.initServices();
    await domContentLoaded();
    mark('willStartWorkbench');

    // 1.创建工作区
    const workbench = new Workbench(document.body, services.serviceCollection,
services.logService);

    // 2.监听窗口变化
    this._register(addDisposableListener(window, EventType.RESIZE, e =>
this.onWindowResize(e, true, workbench)));

    // 3.工作台生命周期
    this._register(workbench.onShutdown(() => this.dispose()));
    this._register(workbench.onWillShutdown(event =>
event.join(services.storageService.close())));

    // 3.启动工作区
    const instantiationService = workbench.startup();

    ...
}
...
```

vs/workbench/browser/workbench.ts

工作区继承自 layout 类，主要作用是构建工作区，创建界面布局。

```
export class Workbench extends Layout {
    ...
    startup(): IInstantiationService {
        try {
            ...

            // Services
            const instantiationService =
this.initServices(this.serviceCollection);
```

```
            instantiationService.invokeFunction(async accessor => {
                const lifecycleService = accessor.get(ILifecycleService);
                const storageService = accessor.get(IStorageService);
                const configurationService =
accessor.get(IConfigurationService);

                // Layout
                this.initLayout(accessor);

                // Registries
                this.startRegistries(accessor);

                // Context Keys

this._register(instantiationService.createInstance(WorkbenchContextKeysHandler))
;

                // 注册监听事件
                this.registerListeners(lifecycleService, storageService,
configurationService);

                // 渲染工作区
                this.renderWorkbench(instantiationService,
accessor.get(INotificationService) as NotificationService, storageService,
configurationService);

                // 创建工作区布局
                this.createWorkbenchLayout(instantiationService);

                // 布局构建
                this.layout();

                // Restore
                try {
                    await this.restoreWorkbench(accessor.get(IEditorService),
accessor.get(IEditorGroupsService), accessor.get(IViewletService),
accessor.get(IPanelService), accessor.get(ILogService), lifecycleService);
                } catch (error) {
                    onUnexpectedError(error);
                }
            });

            return instantiationService;
        } catch (error) {
            onUnexpectedError(error);

            throw error; // rethrow because this is a critical issue we cannot
handle properly here
        }
    }
    ...
}
```

5.事件分发

event

src/vs/base/common/event.ts

程序中常见使用 once 方法进行事件绑定, 给定一个事件，返回一个只触发一次的事件，放在匿名函数返回

```
export function once<T>(event: Event<T>): Event<T> {
    return (listener, thisArgs = null, disposables?) => {
        // 设置次变量，防止事件重复触发造成事件污染
        let didFire = false;
        let result: IDisposable;
        result = event(e => {
            if (didFire) {
                return;
            } else if (result) {
                result.dispose();
            } else {
                didFire = true;
            }

            return listener.call(thisArgs, e);
        }, null, disposables);

        if (didFire) {
            result.dispose();
        }

        return result;
    };
}
```

循环派发了所有注册的事件， 事件会存储到一个事件队列，通过 fire 方法触发事件

private _deliveryQueue?: LinkedList<[Listener, T]>;//事件存储队列

```
fire(event: T): void {
    if (this._listeners) {
        // 将所有事件传入 delivery queue
        // 内部/嵌套方式通过emit发出.
        // this调用事件驱动

        if (!this._deliveryQueue) {
            this._deliveryQueue = new LinkedList();
        }

        for (let iter = this._listeners.iterator(), e = iter.next(); !e.done; e
= iter.next()) {
            this._deliveryQueue.push([e.value, event]);
        }

        while (this._deliveryQueue.size > 0) {
            const [listener, event] = this._deliveryQueue.shift()!;
            try {
                if (typeof listener === 'function') {
                    listener.call(undefined, event);
                } else {
                    listener[0].call(listener[1], event);
                }
```

```
            } catch (e) {
                onUnexpectedError(e);
            }
        }
    }
}
```

6.进程通信

主进程

src/vs/code/electron-main/main.ts

main.ts 在启动应用后就创建了一个主进程 main process，它可以通过 electron 中的一些模块直接与原生 GUI 交互。

```
server = await serve(environmentService.mainIPCHandle);
once(lifecycleService.onWillShutdown)(() => server.dispose());
```

渲染进程

仅启动主进程并不能给你的应用创建应用窗口。窗口是通过 main 文件里的主进程调用叫 BrowserWindow 的模块创建的。

主进程与渲染进程之间的通信

在 electron 中，主进程与渲染进程有很多通信的方法。比如 ipcRenderer 和 ipcMain，还可以在渲染进程使用 remote 模块。

ipcMain & ipcRenderer

- 主进程：ipcMain
- 渲染进程：ipcRenderer

ipcMain 模块和 ipcRenderer 是类 EventEmitter 的实例。

在主进程中使用 ipcMain 接收渲染线程发送过来的异步或同步消息，发送过来的消息将触发事件。

在渲染进程中使用 ipcRenderer 向主进程发送同步或异步消息，也可以接收到主进程的消息。

- 发送消息，事件名为 channel .
- 回应同步消息, 你可以设置 event.returnValue .
- 回应异步消息, 你可以使用 event.sender.send(...)

创建 IPC 服务

src/vs/base/parts/ipc/node/ipc.net.ts

这里返回一个 promise 对象，成功则 createServer

```
export function serve(hook: any): Promise<Server> {
    return new Promise<Server>((c, e) => {
        const server = createServer();

        server.on('error', e);
        server.listen(hook, () => {
            server.removeListener('error', e);
            c(new Server(server));
        });
    });
}
```

创建信道

src/vs/code/electron-main/app.ts

- mainIpcServer * launchChannel
- electronIpcServer
  _ updateChannel
  _ issueChannel
  _ workspacesChannel
  _ windowsChannel
  _ menubarChannel
  _ urlChannel
  _ storageChannel
  _ logLevelChannel

```
private openFirstWindow(accessor: ServicesAccessor, electronIpcServer:
ElectronIPCServer, sharedProcessClient: Promise<Client<string>>): ICodeWindow[]
{

        // Register more Main IPC services
        const launchService = accessor.get(ILaunchService);
        const launchChannel = new LaunchChannel(launchService);
        this.mainIpcServer.registerChannel('launch', launchChannel);

        // Register more Electron IPC services
        const updateService = accessor.get(IUpdateService);
        const updateChannel = new UpdateChannel(updateService);
        electronIpcServer.registerChannel('update', updateChannel);

        const issueService = accessor.get(IIssueService);
        const issueChannel = new IssueChannel(issueService);
        electronIpcServer.registerChannel('issue', issueChannel);

        const workspacesService = accessor.get(IWorkspacesMainService);
        const workspacesChannel = new WorkspacesChannel(workspacesService);
        electronIpcServer.registerChannel('workspaces', workspacesChannel);

        const windowsService = accessor.get(IWindowsService);
        const windowsChannel = new WindowsChannel(windowsService);
        electronIpcServer.registerChannel('windows', windowsChannel);
        sharedProcessClient.then(client => client.registerChannel('windows',
windowsChannel));

        const menubarService = accessor.get(IMenubarService);
```

```
        const menubarChannel = new MenubarChannel(menubarService);
        electronIpcServer.registerChannel('menubar', menubarChannel);

        const urlService = accessor.get(IURLService);
        const urlChannel = new URLServiceChannel(urlService);
        electronIpcServer.registerChannel('url', urlChannel);

        const storageMainService = accessor.get(IStorageMainService);
        const storageChannel = this._register(new
GlobalStorageDatabaseChannel(this.logService, storageMainService));
        electronIpcServer.registerChannel('storage', storageChannel);

        // Log level management
        const logLevelChannel = new
LogLevelSetterChannel(accessor.get(ILogService));
        electronIpcServer.registerChannel('loglevel', logLevelChannel);
        sharedProcessClient.then(client => client.registerChannel('loglevel',
logLevelChannel));

        ...

        // default: read paths from cli
        return windowsMainService.open({
            context,
            cli: args,
            forceNewWindow: args['new-window'] || (!hasCliArgs && args['unity-
launch']),
            diffMode: args.diff,
            noRecentEntry,
            waitMarkerFileURI,
            gotoLineMode: args.goto,
            initialStartup: true
        });
    }
```

每一个信道，内部实现两个方法 listen 和 call

例如：src/vs/platform/localizations/node/localizationsIpc.ts

构造函数绑定事件

```
export class LocalizationsChannel implements IServerChannel {

    onDidLanguagesChange: Event<void>;

    constructor(private service: ILocalizationsService) {
        this.onDidLanguagesChange = Event.buffer(service.onDidLanguagesChange,
true);
    }

    listen(_: unknown, event: string): Event<any> {
        switch (event) {
            case 'onDidLanguagesChange': return this.onDidLanguagesChange;
        }

        throw new Error(`Event not found: ${event}`);
    }
```

```
    call(_: unknown, command: string, arg?: any): Promise<any> {
        switch (command) {
            case 'getLanguageIds': return this.service.getLanguageIds(arg);
        }

        throw new Error(`Call not found: ${command}`);
    }
}
```

7.主要窗口

workbench.ts 中 startup 里面 Workbench 负责创建主界面
src/vs/workbench/browser/workbench.ts

```
startup(): IInstantiationService {
    try {

        ...

        instantiationService.invokeFunction(async accessor => {

            // 渲染主工作界面
            this.renderWorkbench(instantiationService,
accessor.get(INotificationService) as NotificationService, storageService,
configurationService);

            // 界面布局
            this.createWorkbenchLayout(instantiationService);

            // Layout
            this.layout();

            // Restore
            try {
                await this.restoreWorkbench(accessor.get(IEditorService),
accessor.get(IEditorGroupsService), accessor.get(IViewletService),
accessor.get(IPanelService), accessor.get(ILogService), lifecycleService);
            } catch (error) {
                onUnexpectedError(error);
            }
        });

        return instantiationService;
    } catch (error) {
        onUnexpectedError(error);

        throw error; // rethrow because this is a critical issue we cannot
handle properly here
    }
}
```

渲染主工作台，渲染完之后加入到 container 中，container 加入到 parent, parent 就是 body 了。

this.parent.appendChild(this.container);

```
private renderWorkbench(instantiationService: IInstantiationService,
notificationService: NotificationService, storageService: IStorageService,
configurationService: IConfigurationService): void {

        ...

        //TITLEBAR_PART 顶部操作栏
        //ACTIVITYBAR_PART 最左侧菜单选项卡
        //SIDEBAR_PART 左侧边栏，显示文件，结果展示等
        //EDITOR_PART 右侧窗口，代码编写,欢迎界面等
        //STATUSBAR_PART 底部状态栏
        [
            { id: Parts.TITLEBAR_PART, role: 'contentinfo', classes:
['titlebar'] },
            { id: Parts.ACTIVITYBAR_PART, role: 'navigation', classes:
['activitybar', this.state.sideBar.position === Position.LEFT ? 'left' :
'right'] },
            { id: Parts.SIDEBAR_PART, role: 'complementary', classes:
['sidebar', this.state.sideBar.position === Position.LEFT ? 'left' : 'right'] },
            { id: Parts.EDITOR_PART, role: 'main', classes: ['editor'], options:
{ restorePreviousState: this.state.editor.restoreEditors } },
            { id: Parts.PANEL_PART, role: 'complementary', classes: ['panel',
this.state.panel.position === Position.BOTTOM ? 'bottom' : 'right'] },
            { id: Parts.STATUSBAR_PART, role: 'contentinfo', classes:
['statusbar'] }
        ].forEach(({ id, role, classes, options }) => {
            const partContainer = this.createPart(id, role, classes);

            if
(!configurationService.getValue('workbench.useExperimentalGridLayout')) {
                // TODO@Ben cleanup once moved to grid
                // Insert all workbench parts at the beginning. Issue #52531
                // This is primarily for the title bar to allow overriding -
webkit-app-region
                this.container.insertBefore(partContainer,
this.container.lastChild);
            }

            this.getPart(id).create(partContainer, options);
        });

        // 将工作台添加至container dom渲染
        this.parent.appendChild(this.container);
    }
```

workbench 最后调用 this.layout()方法，将窗口占据整个界面，渲染完成

```
layout(options?: ILayoutOptions): void {
        if (!this.disposed) {
            this._dimension = getClientArea(this.parent);

            if (this.workbenchGrid instanceof Grid) {
                position(this.container, 0, 0, 0, 0, 'relative');
                size(this.container, this._dimension.width,
this._dimension.height);

                // Layout the grid widget
```

```
                this.workbenchGrid.layout(this._dimension.width,
this._dimension.height);
            } else {
                this.workbenchGrid.layout(options);
            }

            // Emit as event
            this._onLayout.fire(this._dimension);
        }
    }
```

8.开发调试

```
app.once('ready', function () {
    //启动追踪
    if (args['trace']) {
        // @ts-ignore
        const contentTracing = require('electron').contentTracing;

        const traceOptions = {
            categoryFilter: args['trace-category-filter'] || '*',
            traceOptions: args['trace-options'] || 'record-until-full,enable-
sampling'
        };

        contentTracing.startRecording(traceOptions, () => onReady());
    } else {
        onReady();
    }
});
```

启动追踪

这里如果传入 trace 参数，在 onReady 启动之前会调用 chromium 的收集跟踪数据，
提供的底层的追踪工具允许我们深度了解 V8 的解析以及其他时间消耗情况，

一旦收到可以开始记录的请求，记录将会立马启动并且在子进程是异步记录听的. 当所有的子进程都收到 startRecording 请求的时候，callback 将会被调用.

categoryFilter 是一个过滤器，它用来控制那些分类组应该被用来查找.过滤器应当有一个可选的 - 前缀来排除匹配的分类组.不允许同一个列表既是包含又是排斥.

contentTracing.startRecording(options, callback)

- options Object
  _ categoryFilter String
  _ traceOptions String
- callback Function

[关于 trace 的详细介绍]（https://www.w3cschool.cn/electronmanual/electronmanual-content-tracing.html）

结束追踪

contentTracing.stopRecording(resultFilePath, callback)

- resultFilePath String

- callback Function
  在成功启动窗口后，程序结束性能追踪，停止对所有子进程的记录.

子进程通常缓存查找数据，并且仅仅将数据截取和发送给主进程.这有利于在通过 IPC 发送查找数据之前减小查找时的运行开销，这样做很有价值.因此，发送查找数据，我们应当异步通知所有子进程来截取任何待查找的数据.

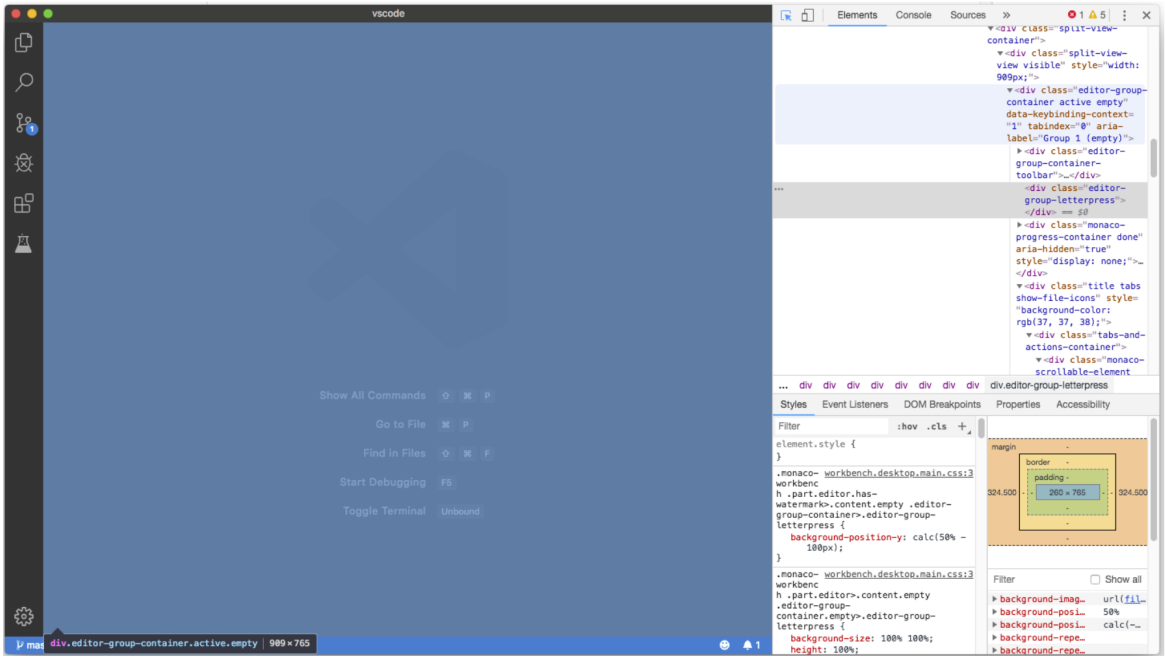
一旦所有子进程接收到了 stopRecording 请求，将调用 callback ，并且返回一个包含查找数据的文件.

如果 resultFilePath 不为空，那么将把查找数据写入其中，否则写入一个临时文件.实际文件路径如果不为空，则将调用 callback .

debug

调试界面在菜单栏找到 Help->Toggle Developers Tools

调出 Chrome 开发者调试工具进行调试



参考

https://electronjs.org/docs

https://github.com/microsoft/vscode/wiki/How-to-Contribute

https://github.com/Microsoft/vscode/wiki/Code-Organization

http://xzper.com/2016/04/17/vscode%E6%BA%90%E7%A0%81%E5%89%96%E6%9E%90/

http://www.ayqy.net/blog/vs-code%E6%BA%90%E7%A0%81%E7%AE%80%E6%9E%90/

Python 自动翻译成 C++

作者：byronhe，腾讯 WXG 开发工程师

一、问题背景

随着深度学习的广泛应用，在搜索引擎/推荐系统/机器视觉等业务系统中，越来越多的深度学习模型部署到线上服务。

机器学习模型在离线训练时，一般要将输入的数据做特征工程预处理，再输入模型在 TensorFlow PyTorch 等框架上做训练。

1.常见的特征工程逻辑

常见的特征工程逻辑有：

1. 分箱/分桶 离散化
2. log/exp 对数/幂等 math numpy 常见数学运算
3. 特征缩放/归一化/截断
4. 交叉特征生成
5. 分词匹配程度计算
6. 字符串分隔匹配判断 tong
7. 缺省值填充等
8. 数据平滑
9. onehot 编码，hash 编码等

这些特征工程代码，当然一般使用深度学习最主要的语言 **python** 实现。

二、业务痛点

离线训练完成，模型上线部署后，同样要**用 C++ 重新实现** 这些 python 的特征工程逻辑代码。

我们发现，**“用 C++ 重新实现”** 这个步骤，给实际业务带来了大量的问题：

1. 繁琐，费时费力，极容易出现 python 和 C++ 代码**不一致**
2. **不一致**会直接影响模型在线上的效果，导致大盘业务指标不如预期，产生各种 bad case
3. **不一致**难以发现，无法测试，无法监控，经常要靠用户投诉反馈，甚至大盘数据异常才能发现

1. 业界方案

针对这些问题，我调研了这些业界方案：

《推荐系统中模型训练及使用流程的标准化》
https://www.infoq.cn/article/2E6LCqb1GeqFRAjkkjX3

《自主研发、不断总结经验，美团搜索推荐机器学习平台》
https://cloud.tencent.com/developer/article/1357309

《京东电商推荐系统实践》
https://www.infoq.cn/article/1OkKmb_gEYNR3YqC9RcW

> “模型线上线下一致性问题对于模型效果非常重要，我们使用特征日志来实时记录特征，保证特征的一致性。这样离线处理的时候会把实时的用户反馈，和特征日志做一个结合生成训练样本，然后更新到模型训练平台上，平台更新之后在推送到线上，这样整个排序形成了一个闭环。”

总结起来，有几种思路：

1. 在线特征存储起来给离线用
2. 在线 C++ 代码编译成 so 导出给离线用
3. 根据一份配置生成离线和在线代码
4. 提取公共代码，加强代码复用，等软件工程手段，减少不一致

2. 自动翻译方案

(1) .已有方案的缺点

但这些思路都有各种缺点：

1. 所有在线请求的所有特征，这个存储量数据量很大
2. 算法改代码需要等待后台开发，降低了算法同学的工作效率
3. 特征处理代码的复杂度转移到配置文件中，不一定能充分表达，而且配置格式增加学习成本
4. 就这边真实离线特征处理代码来看，大部分代码都无法抽取出公共代码做复用。

(2). 翻译器

回到问题出发点考虑，显而易见，这个问题归根结底就是需要一个“ python 到 c++ 的翻译器 ”。

那其实“翻译器 Transpiler ”，和编译器解释器类似，也是个古老的热门话题了，比如 WebAssembly, CoffeeScript ， Babel ,Google Closure Compiler， f2c。

于是一番搜索，发现 python 到 C++ 的翻译器也不少，其中 Pythran 是新兴比较热门的开源项目。

于是一番尝试后，借助 pythran，我们实现了：

1. 一条命令 **全自动把 Python 翻译成等价 C++**
2. 严格等价保证改写，彻底消除不一致
3. **完全去掉重新实现** 这块工作量，后台开发成本降到 0 ，彻底解放生产力
4. 算法同学继续使用纯 python，开发效率无影响，**无学习成本**
5. 并能推广到其他需要 **python 改写成后台 C++ 代码** 的业务场景，解放生产力

三、pythran 的使用流程

(1). 安装

一条命令安装：

```
pip3 install pythran
```

(2). 写 Python 代码

下面这个 python demo，是 pythran 官方 demo

```
import math
import numpy as np

def zero(n, m):
    return [[0]*n for col in range(m)]

#pythran export matrix_multiply(float list list, float list list)
def matrix_multiply(m0, m1):
    new_matrix = zero(len(m0),len(m1[0]))
    for i in range(len(m0)):
        for j in range(len(m1[0])):
            for k in range(len(m1)):
                new_matrix[i][j] += m0[i][k]*m1[k][j]
    return new_matrix

#pythran export arc_distance(float[], float[], float[], float[])
def arc_distance(theta_1, phi_1, theta_2, phi_2):
    """
    Calculates the pairwise arc distance
    between all points in vector a and b.
    """
```

```
    temp = (np.sin((theta_2-theta_1)/2)**2
           + np.cos(theta_1)*np.cos(theta_2) * np.sin((phi_2-phi_1)/2)**2)
    distance_matrix = 2 * np.arctan2(np.sqrt(temp), np.sqrt(1-temp))
    return distance_matrix


#pythran export dprod(int list, int list)
def dprod(l0,l1):
    """wow, generator expression, zip and sum."""
    return sum(x * y for x, y in zip(l0, l1))


#pythran export get_age(int )
def get_age(age):
    if age <= 20:
        age_x = '0_20'
    elif age <= 25:
        age_x = '21_25'
    elif age <= 30:
        age_x = '26_30'
    elif age <= 35:
        age_x = '31_35'
    elif age <= 40:
        age_x = '36_40'
    elif age <= 45:
        age_x = '41_45'
    elif age <= 50:
        age_x = '46_50'
    else:
        age_x = '50+'
    return age_x
```

(3). Python 转成 C++

一条命令完成翻译

```
pythran -e demo.py -o  demo.hpp
```

(4). 写 C++ 代码调用

pythran/pythonic/ 目录下是 python 标准库的 C++ 等价实现，翻译出来的 C++ 代码需要 include 这些头文件。

写个 C++ 代码调用

```
#include "demo.hpp"
#include "pythonic/numpy/random/rand.hpp"
#include <iostream>

using std::cout;
using std::endl;

int main() {
  pythonic::types::list<pythonic::types::list<double>> m0 = {{2.0, 3.0},
```

```
                                                        {4.0, 5.0}},
                                                m1 = {{1.0, 2.0},
                                                      {3.0, 4.0}};
  cout << m0 << "*" << m1 << "\n=\n"
       << __pythran_demo::matrix_multiply()(m0, m1) << endl
       << endl;

  auto theta_1 = pythonic::numpy::random::rand(3),
       phi_1 = pythonic::numpy::random::rand(3),
       theta_2 = pythonic::numpy::random::rand(3),
       phi_2 = pythonic::numpy::random::rand(3);
  cout << "arc_distance " << theta_1 << "," << phi_1 << "," << theta_2 << ","
       << phi_2 << "\n=\n"
       << __pythran_demo::arc_distance()(theta_1, phi_1, theta_2, phi_2) << endl
       << endl;

  pythonic::types::list<int> l0 = {2, 3}, l1 = {4, 5};
  cout << "dprod " << l0 << "," << l1 << "\n=\n"
       << __pythran_demo::dprod()(l0, l1) << endl
       << endl;

  cout << "get_age 30 = " << __pythran_demo::get_age()(30) << endl << endl;

  return 0;
}
```

(5). 编译运行

```
g++ -g -std=c++11 main.cpp -fopenmp -march=native -DUSE_XSIMD -I
/usr/local/lib/python3.6/site-packages/pythran/ -o pythran_demo

./pythran_demo
```

四、pythran 的功能与特性

(1). 介绍

按官方定义，Pythran 是一个 AOT (Ahead-Of-Time - 预先编译) 编译器。 给科学计算的 python 加注解后，pythran 可以把 python 代码变成接口相同的原生 python 模块，大幅度提升性能。

并且 pythran 也可以利用 OpenMP 多核和 SIMD 指令集。

支持 python 3 和 Python 2.7 。

pythran 的 manual 挺详细：https://pythran.readthedocs.io/en/latest/MANUAL.html

(2). 功能

pythran 并不支持完整的 python， 只支持 python 语言特性的一个子集:

- polymorphic functions 多态函数(翻译成 C++ 的泛型模板函数)
- lambda
- list comprehension 列表推导式
- map, reduce 等函数
- dictionary, set, list 等数据结构
- exceptions 异常

- file handling 文件处理
- 部分 numpy

不支持的功能：

- classes 类
- polymorphic variables 可变类型变量

(3). 支持的数据类型和函数

pythran export 可以导出函数和全局变量。支持导出的数据类型，BNF 定义是：

```
    argument_type = basic_type
                  | (argument_type+)    # this is a tuple
                  | argument_type list    # this is a list
                  | argument_type set    # this is a set
                  | argument_type []+    # this is a ndarray, C-style
                  | argument_type [::]+    # this is a strided ndarray
                  | argument_type [:,...,:]+ # this is a ndarray, Cython style
                  | argument_type [:,...,3]+ # this is a ndarray, some dimension
fixed
                  | argument_type:argument_type dict    # this is a dictionary

    basic_type = bool | byte | int | float | str | None | slice
               | uint8 | uint16 | uint32 | uint64 | uintp
               | int8 | int16 | int32 | int64 | intp
               | float32 | float64 | float128
               | complex64 | complex128 | complex256
```

可以看到基础类型相当全面，支持各种 整数，浮点数，字符串，复数。

复合类型支持 tuple, list, set, dict, numpy.ndarray 等，对应 C++ 代码的类型实现在 pythran/pythonic/include/types/ 下面，可以看到比如 dict 实际就是封装了一下 std::unordered_map

https://pythran.readthedocs.io/en/latest/SUPPORT.html

可以看到支持的 python 基础库，其中常用于机器学习的 numpy 支持算比较完善。

五、pythran 的基本原理

和常见的编译器/解释器类似， pythran 的架构是分成 3 层：

1. python 代码解析成抽象语法树 AST 。用 python 标准库自带的的 ast 模块实现
2. 代码优化。在 AST 上做优化，有多种 transformation pass，比如 deadcode_elimination 死代码消除，loop_full_unrolling 循环展开 等。还有 Function/Module/Node 级别的 Analysis，用来遍历 AST 供 transformation 利用。
3. 后端，实现代码生成。目前有 2 个后端，Cxx / Python， Cxx 后端可以把 AST 转成 C++ 代码（Python 后端用来调试）。

目前看起来 ， pythran 还欠缺的：

1. 字符串处理能力欠缺，缺少 str.encode()/str.decode() 对 utf8 的支持
2. 缺少正则表达式 regex 支持

看文档要自己加也不麻烦，看业务需要可以加。

六、相关文章

- 《京东电商推荐系统实践》
  https://www.infoq.cn/article/1OkKmb_gEYNR3YqC9RcW
- 《自主研发、不断总结经验，美团搜索推荐机器学习平台》
  https://cloud.tencent.com/developer/article/1357309
- 《推荐系统中模型训练及使用流程的标准化》
  https://www.infoq.cn/article/2E6LCqb1GeqFRAjkkjX3
- numba
  http://numba.pydata.org

系统架构

如何设计一个高可用、高并发秒杀系统

作者：vincentsu，腾讯 PCG 后台开发工程师

> 如今的互联网已经在海量服务领域有了很成熟的理论，因此自己也很庆幸，能够从 0 到 1 完整践行海量服务。微视春节项目中的集卡瓜分活动，是一个典型的秒杀场景，自己参与其中，分享一些心得和总结。

如今的互联网已经在海量服务领域有了很成熟的理论，因此自己也很庆幸，能够从 0 到 1 完整践行海量服务。微视春节项目中的集卡瓜分活动，是一个典型的秒杀场景，自己参与其中，分享一些心得和总结。

秒杀系统的难点

- 友好的用户体验
  - 用户不能接受破窗的体验，例如：系统超时、系统错误的提示，或者直接 404 页面
- 瞬时高并发流量的挑战
  - 木桶短板理论，整个系统的瓶颈往往都在 DB，如何设计出高并发、高可用系统?

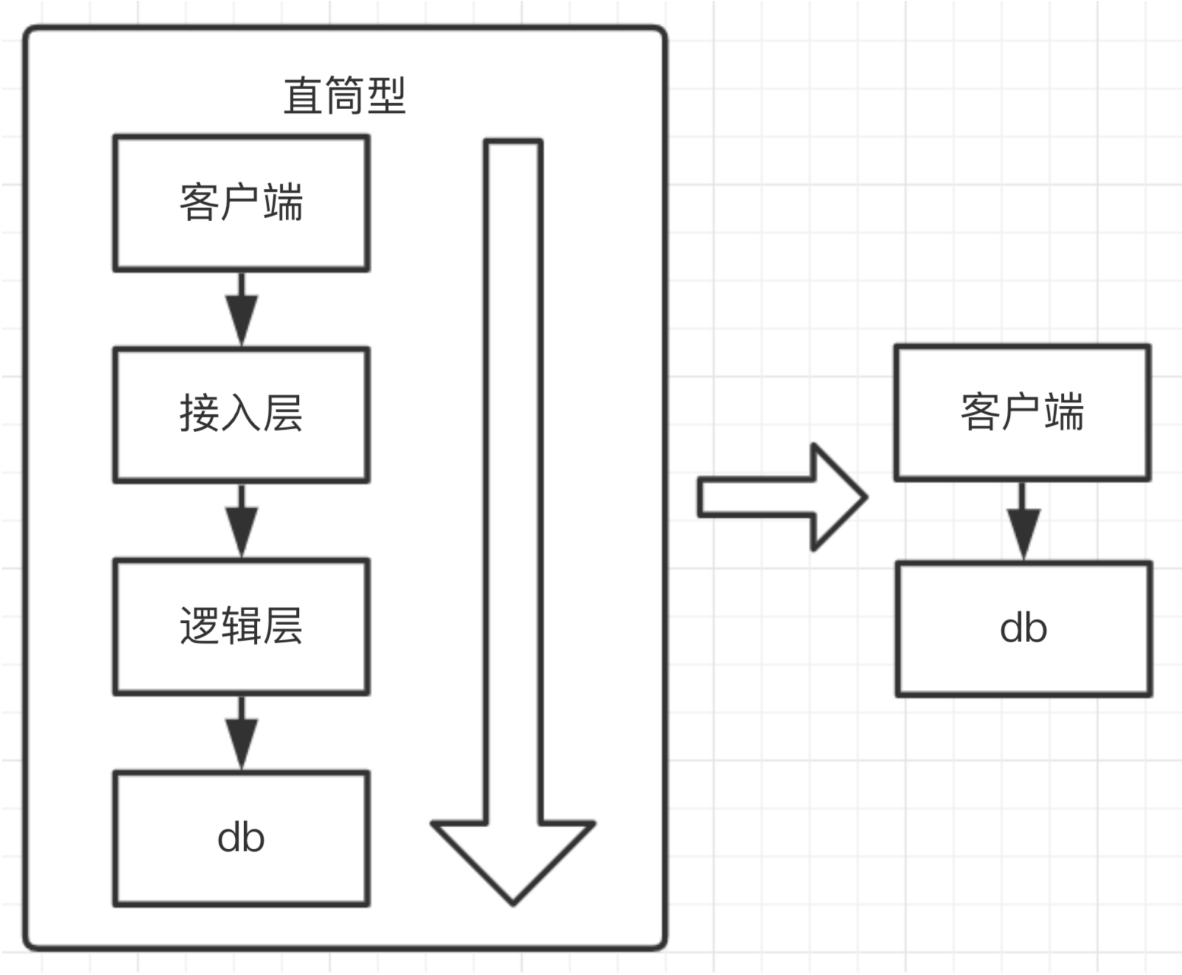
如何设计



上图是一个典型的互联网业务，用户完成一个写操作，一般会通过接入层和逻辑层，这里的服务都是无状态，可以通过平行拓展去解决高并发的问题；到了 db 层，必须要落到介质中，可以是磁盘/ssd/内存，如果出现 key 的冲突，会有一些并发控制技术，例如 cas/加锁/串行排队等。

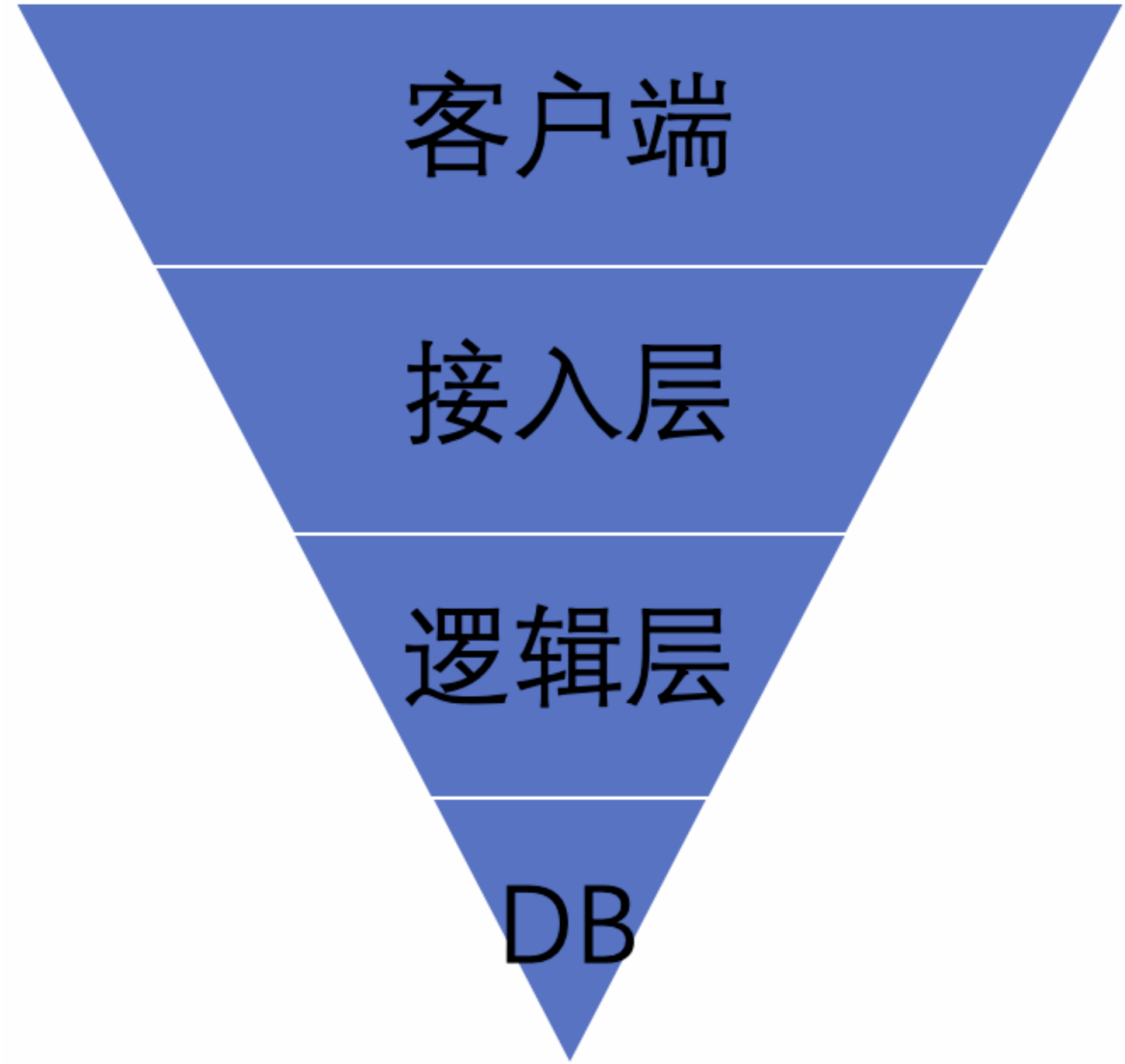
直筒型

直筒型业务，指的是用户请求 1:1 的洞穿到 db 层，如下图所示。在比较简单的业务中，才会采用这个模型。随着业务规模复杂度上来，一定会有 db 和逻辑层分离、逻辑层和接入层分离。



漏斗型

漏斗型业务，指的是，用户的请求，从客户端到 db 层，层层递减，递减的程度视业务而定。例如当 10w 人去抢 1 个物品时，db 层的请求在个位数量级，这就是比较理想的模型。如下图所示

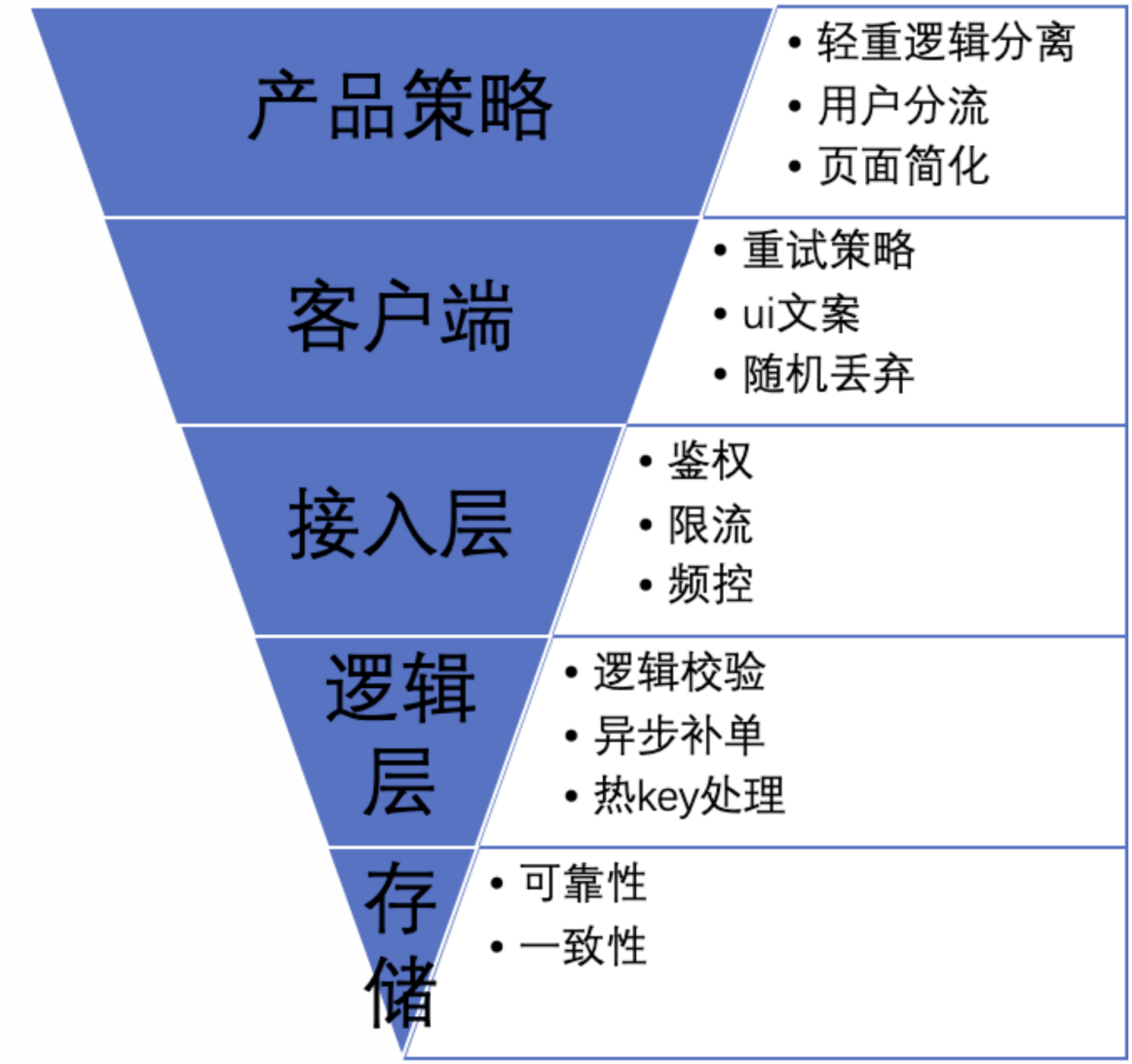


这个模型，是高并发的基础，翻译一下就是下面这些：

- 及早发现，及早拒绝
- Fast Fail
- 前端保护后端

如何实现漏斗型系统

漏斗型系统需要从产品策略/客户端/接入层/逻辑层/DB 层全方位立体的设计。



产品策略

- 轻重逻辑分离，以秒杀为例，将抢到和到账分开；
  - 抢到，是比较轻的操作，库存扣成功后，就可以成功了
  - 到账，是比较重的操作，需要涉及到到事务操作
- 用户分流，以整点秒杀活动为例，在 1 分钟内，陆续对用户放开入口，将所有用户请求打散在 60s 内，请求就可以降一个数量级
- 页面简化，在秒杀开始的时候，需要简化页面展示，该时刻只保留和秒杀相关的功能。例如，秒杀开始的时候，页面可以不展示推荐的商品。

客户端

- 重试策略非常关键，如果用户秒杀失败了，频繁重试，会加剧后端的雪崩。如何重试呢？根据后端返回码的约定，有两种方法：
  - 不允许重试错误，此时 ui 和文案都需要有一个提示。同时不允许重试
  - 可重试错误，需要策略重试，例如二进制退避法。同时文案和 ui 需要提示。
- ui 和文案，秒杀开始前后，用户的所有异常都需要有精心设计的 ui 和文案提示。例如：【当前活动太火爆，请稍后再重试】【你的货物堵在路上，请稍后查看】等
- 前端随机丢弃请求可以作为降级方案，当用户流量远远大于系统容量时，人工下发随机丢弃标记，用户本地客户端开始随机丢弃请求。

接入层

- 所有请求需要鉴权，校验合法身份

    - 如果是长链接的服务，鉴权粒度可以在 session 级别；如果是短链接业务，需要应对这种高并发流量，例如 cache 等
- 根据后端系统容量，需要一个全局的限流功能，通常有两种做法：
    - 设置好 N 后，动态获取机器部署情况 M，然后下发单机限流值 N/M。要求请求均匀访问，部署机器统一。
    - 维护全局 key，以时间戳建 key。有热 key 问题，可以通过增加更细粒度的 key 或者定时更新 key 的方法。
- 对于单用户/单 ip 需要频控，主要是防黑产和恶意用户。如果秒杀是有条件的，例如需要完成 xxx 任务，解锁资格，对于获得资格的步骤，可以进行安全扫描，识别出黑产和恶意用户。

逻辑层

- 逻辑层首先应该进入校验逻辑，例如参数的合法性，是否有资格，如果失败的用户，快速返回，避免请求洞穿到 db。
- 异步补单，对于已经扣除秒杀资格的用户，如果发货失败后，通常的两种做法是：
    - 事务回滚，回滚本次行为，提示用户重试。这个代价特别大，而且用户重试和前面的重试策略结合的话，用户体验也不大流畅。
    - 异步重做，记录本次用户的 log，提示用户【稍后查看，正在发货中】，后台在峰值过后，启动异步补单。需要服务支持幂等
- 对于发货的库存，需要处理热 key。通常的做法是，维护多个 key，每个用户固定去某个查询库存。对于大量人抢红包的场景，可以提前分配。

存储层

对于业务模型而言，对于 db 的要求需要保证几个原则：

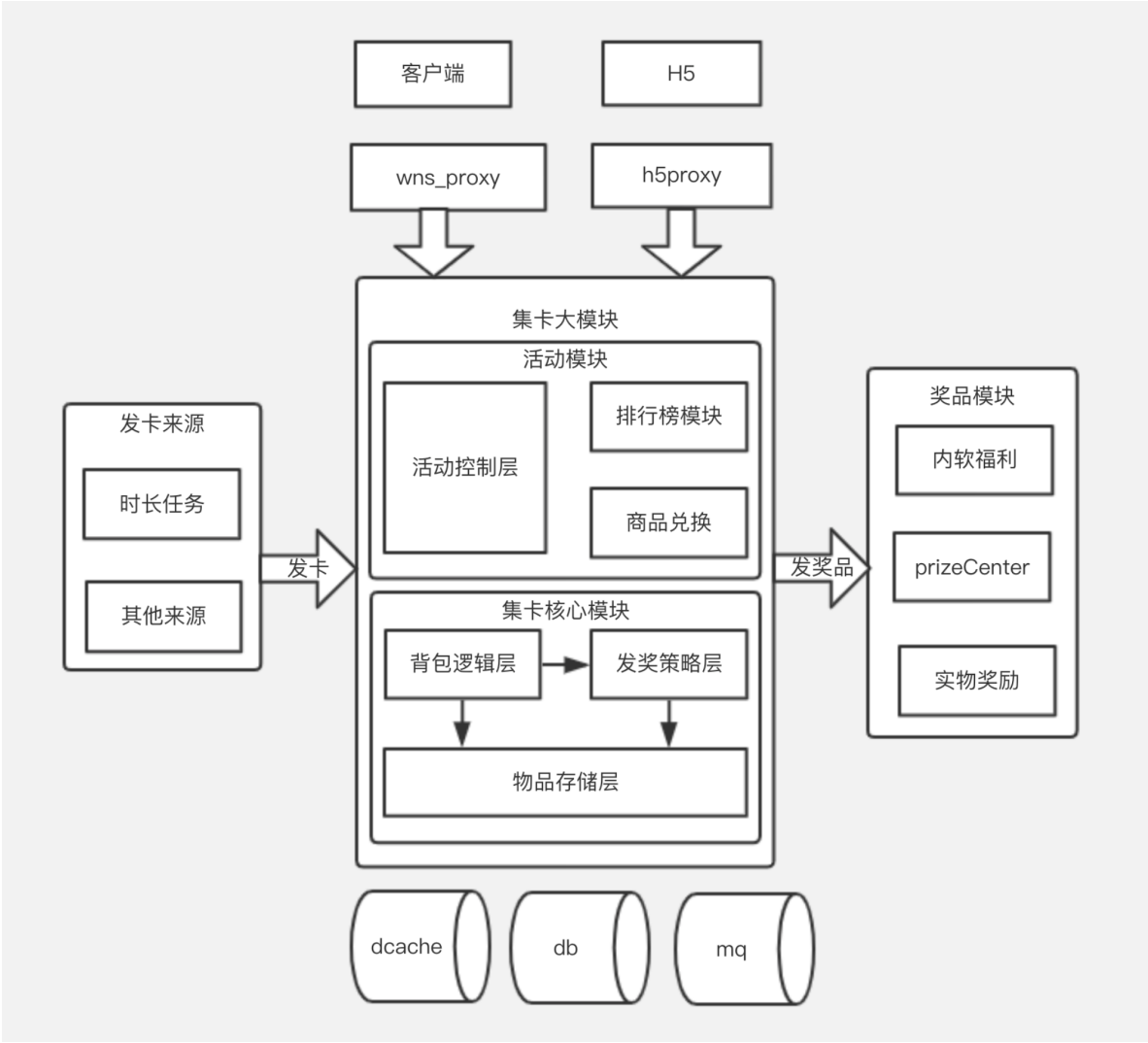
- 可靠性
    - 主备：主备能互相切换，一般要求在同城跨机房
    - 异地容灾：当一地异常，数据能恢复，异地能选主
    - 数据需要持久化到磁盘，或者更冷的设备
- 一致性
    - 对于秒杀而言，需要严格的一致性，一般要求主备严格的一致。

实践——微视集卡瓜分系统

微视集卡瓜分项目属于微视春节项目之一。用户的体验流程如下：



架构图



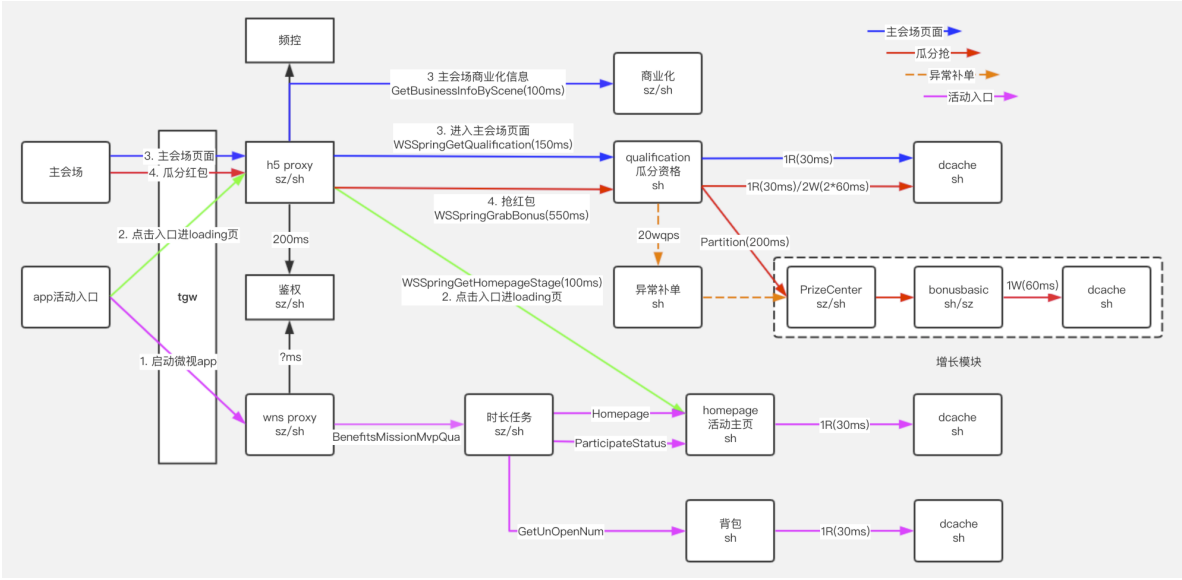
- 客户端主要是微视主 app 和 h5 页面，主 app 是入口，h5 页面是集卡活动页面和瓜分页面。
- 逻辑部分为分：发卡来源、集卡模块、奖品模块，发卡来源主要是任务模块；集卡模块主要由活动模块和集卡模块组成。瓜分部分主要在活动控制层。
- 奖品模块主要是发钱和其他奖品。

瓜分降级预案

为了做好瓜分时刻的高并发，对整个系统需要保证两个重要的事情：

- 全链路梳理，包括调用链的合理性和时延设置
- 降级服务预案分析，提升系统的鲁棒性

如下图所示，是针对瓜分全链路调用分析如下图，需要特别说明的几点：

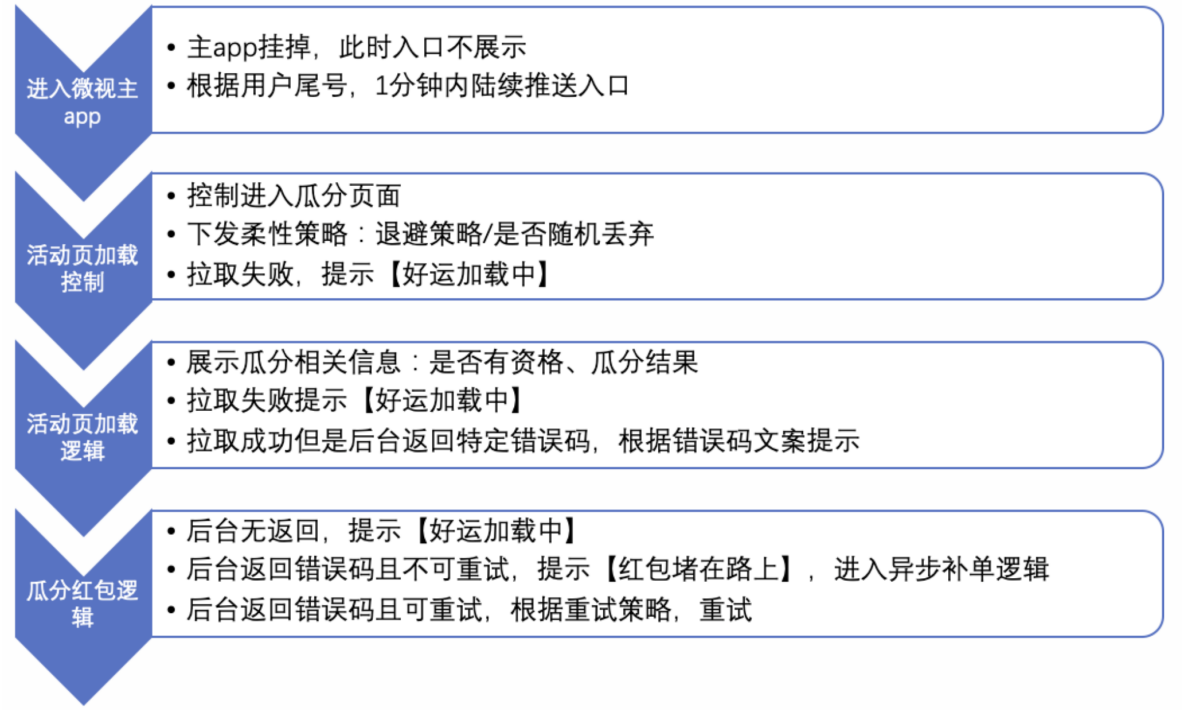


- 时延很重要，需要全链路分析。不但可以提高吞吐量，而且可以快速暴露系统的瓶颈。
- 峰值时刻，补单逻辑需要关闭，避免加剧雪崩。

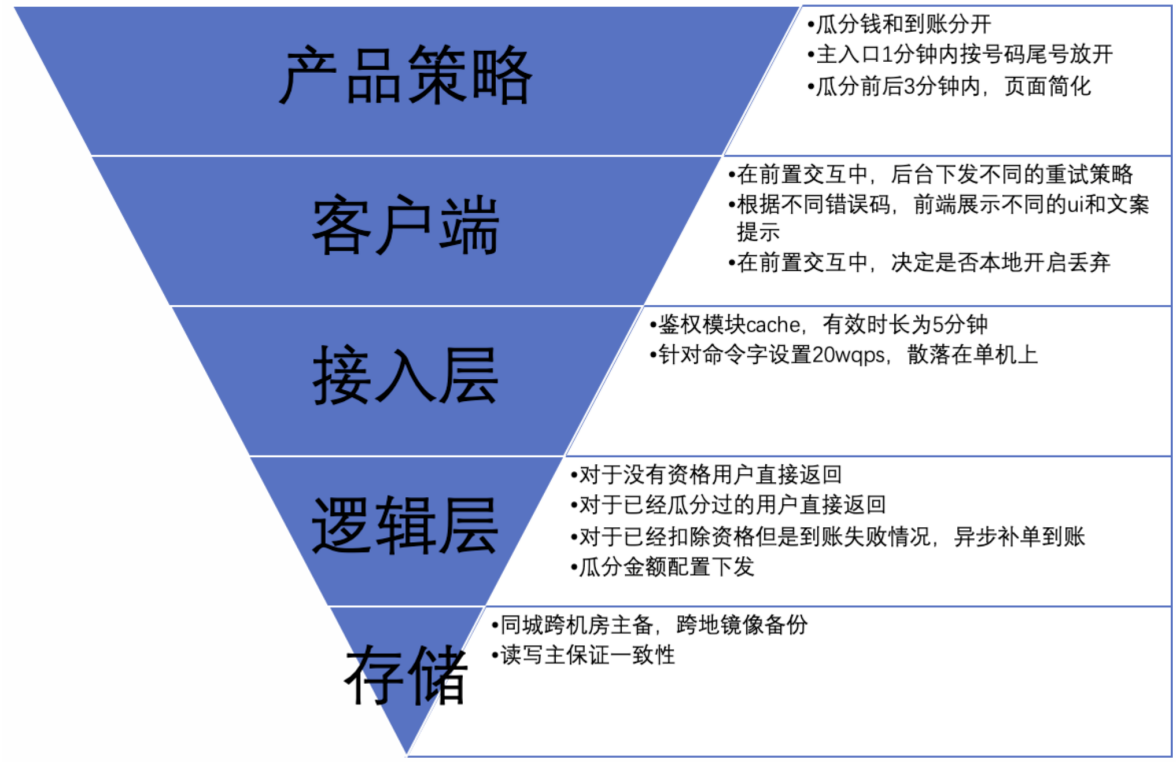
我们的降级预案大概如下：

- 一级预案，瓜分时刻前后 5 分钟自动进入：
  - 入口处 1 分钟内陆续放开入口倒计时，未登录用户不弹入口
  - 主会场排队，以进入主会场 100wqps 为例，超过了进入排队，由接入层频控控制
  - 拉取资格接口排队，拉取资格接口 100wqps，超过了进入排队，由接入层频控控制
  - 抢红包排队，抢红包 100wqps，超过了进入排队，由接入层频控控制
  - 红包到账排队，如果资格扣除成功，现金发放失败，进入排队，24 小时内到账。异步补单
  - 入口处调用后端非关键 rpc:ParticipateStatus，手动关闭
  - 异步补单逻辑关闭。
- 二级预案，后端随机丢请求，接入层频控失效或者下游服务过载，手动开启进入
- 三级预案，前端随机丢请求，后端服务过载或者宕机进入。手动开启

综上，整个瓜分时刻体验如下所示：



回顾下漏斗模型，总结下整个实践：



腾讯课堂：两周流量激增百倍的后台扩容和性能优化实战

作者：andyawang，腾讯 CSIG 后台开发工程师

> 疫情期间，学校网课需求激增，腾讯课堂 2 天上线极速版，2周内支持同时在线人数超百倍增长，对整个后台挑战非常大。整整2个月下来，同合作团队一起，白天7点开始盯监控和开发版本，凌晨12点例行压测和发布扩容，踩过很多坑也取得很多收获，这里拎几个关键点记录下。

腾讯课堂停课不停学

项目背景

大年初一，吃着火锅唱着歌，突然收到重庆十一中的求助信：受疫情影响，年后学校无法开学，高三老师学生都很担心影响到高考，问腾讯课堂能否提供线上平台给高三复课，拉开了整个停课不停学专项的序幕。

极速版的诞生

由于课堂是面向线上培训机构的，这次想把十一中这样的传统线下校园，搬到腾讯课堂内上课才发现困难重重：

1. 入驻：学校各类资质和机构完全不一样，审核周期长
2. 发课：机构发课有很多规范约束，而学校用课程表排课，一个个发课成本高
3. 直播：学校老师转线上上课，普遍说直播工具有上手成本

耗了很多人力才把十一中的入驻发课和老师培训搞完，同时其他学校也陆续找过来了，才发现根本没人力能对接这么多学校。就在这时，小马哥发话了：“把入驻发课全砍掉，快速做个腾讯课堂极速版，老师下载完就能自助上课了”

初3收到军令状，公司2个通宵生死急速开发完，初6凌晨团队体验，解决完体验问题后白天急速上线外发。随着各省市教育厅陆续宣布使用急速版复课，课堂pcu/dau开始起飞，短短2-3周，各项指标百倍增长，AppStore免费类排行榜进Top10，教育类稳居Top1。





疫情期间开发缩影

白天7点开始盯监控和开发版本，凌晨12点例行压测和发布扩容。才发现企业微信一周小结是按凌晨5点为界：

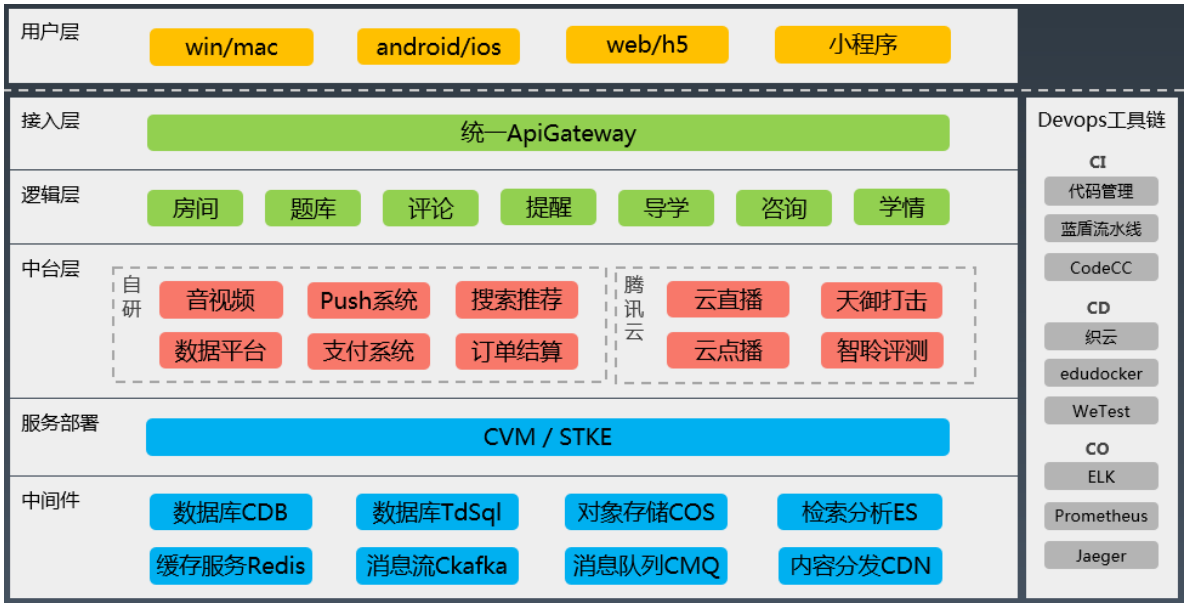






架构挑战

下面是课堂后台架构图，按之前的架构设计和模块部署，突然要在2周内支持量100倍增涨，同时还要开发校园版需求，时间赶且任务重。这里分五个阶段把架构挑战和解决策略介绍下。

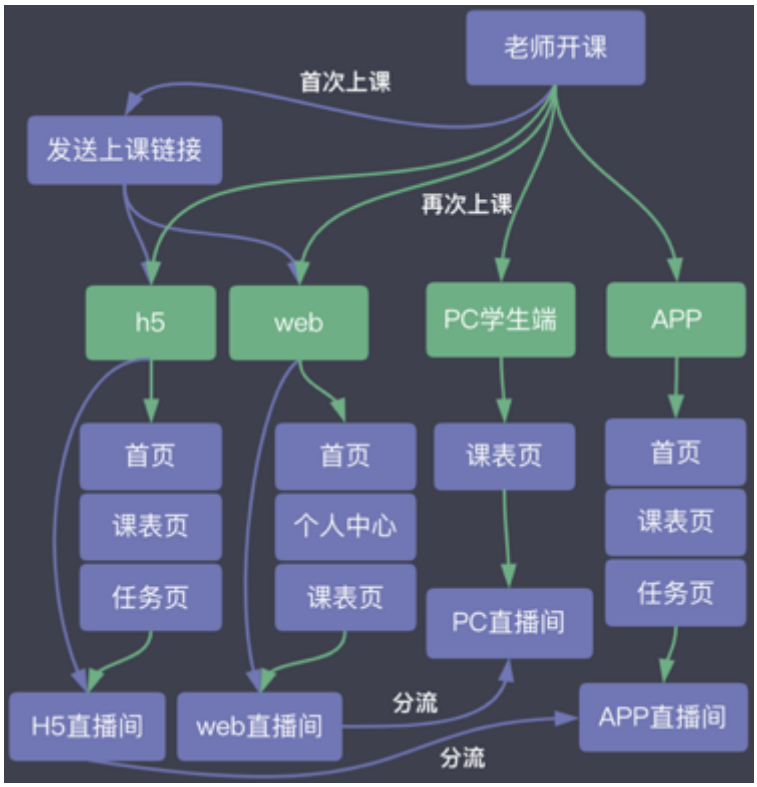


阶段1：先抗住后优化

面对2周100倍量级增长，重构肯定来不及了，且过大改动仓促上线反而会增加不稳定因素。所以初期思路就是“先抗住后优化”：梳理极速版用户路径，评估路径上各模块容量，快速扩容后，每天凌晨例行全链路压测，持续验证迭代。

模块梳理和接口裁剪

和产品讨论完用户路径和访问量级后，各页面qps也基本有个数了，就开始梳理每个页面调用接口列表，明确每个接口要支撑的qps：



由于课堂微服务很多，为了争取时间，需聚焦核心路径减少梳理复杂度，对于非核心体验和风险较大的这2类接口，抛出来和产品讨论做页面接口裁剪。

系统容量评估

模块和接口梳理清楚后，就开始分负责人对系统做容量评估。

要特别关注木桶效应，很多新同学只评估了逻辑Svr这一层，其实从用户端->接入层->逻辑层->存储层全链路都得涉及，不能让任一环节成短板。

扩容扩容扩容！！

各模块扩容数算清楚后，剩下的就是申请机器部署了。扩容本该是个很简单的活，但因为历史债务等问题，部分模块费了很多周折，比如：

1. 容器化和上k8s不彻底
2. 部分c++模块依赖ShmAgent，扩容流程极其繁琐
3. 扩容svr导致DB链接数爆炸

   ...

针对扩容暴露的问题，从接入->逻辑->存储，沉淀了很多设计经验，后面得彻底改造掉，特别是k8s这块应尽快全面铺开。这里终极目标是：针对量级暴涨的情况，不用花人力做评估和扩容，整个系统就有自动伸缩的能力。

全链路压测

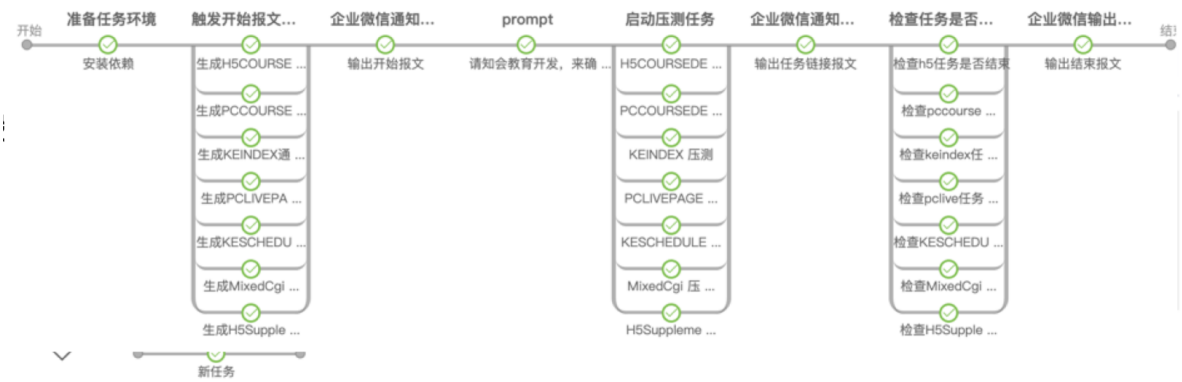
在初步扩容完后，为了防止梳理不全、评估不准等情况，例行全链路压测验证非常重要，可以挖掘性能瓶颈和系统隐患。

在测试同学给力支持下，每天例行执行下面流程，持续扩容优化迭代：

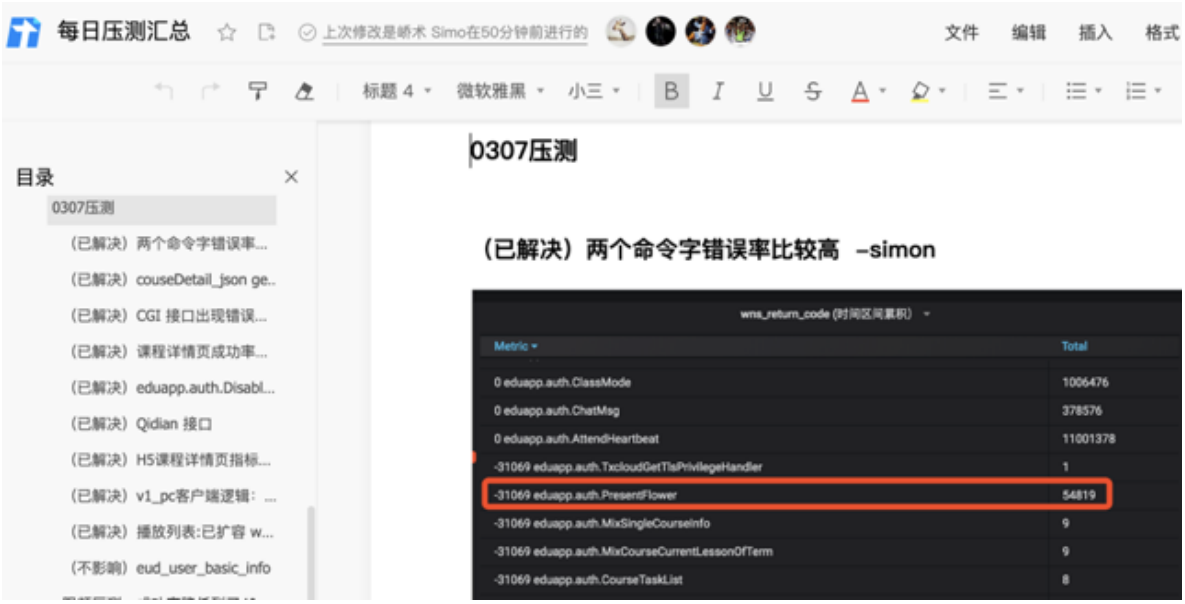
1. 校准压测模型：非常重要，压测用例设计会直接关系到压测效果
2. 确定压测目标：把每个模块/接口的压测qps确定下来
3. 执行压测任务：凌晨12点启动整站压测流水线，执行星海用例，输出压测结论

4. 回归压测结果：压测不达标接口记录doc，尽快暴露隐患，责任人分析原因给解决方案

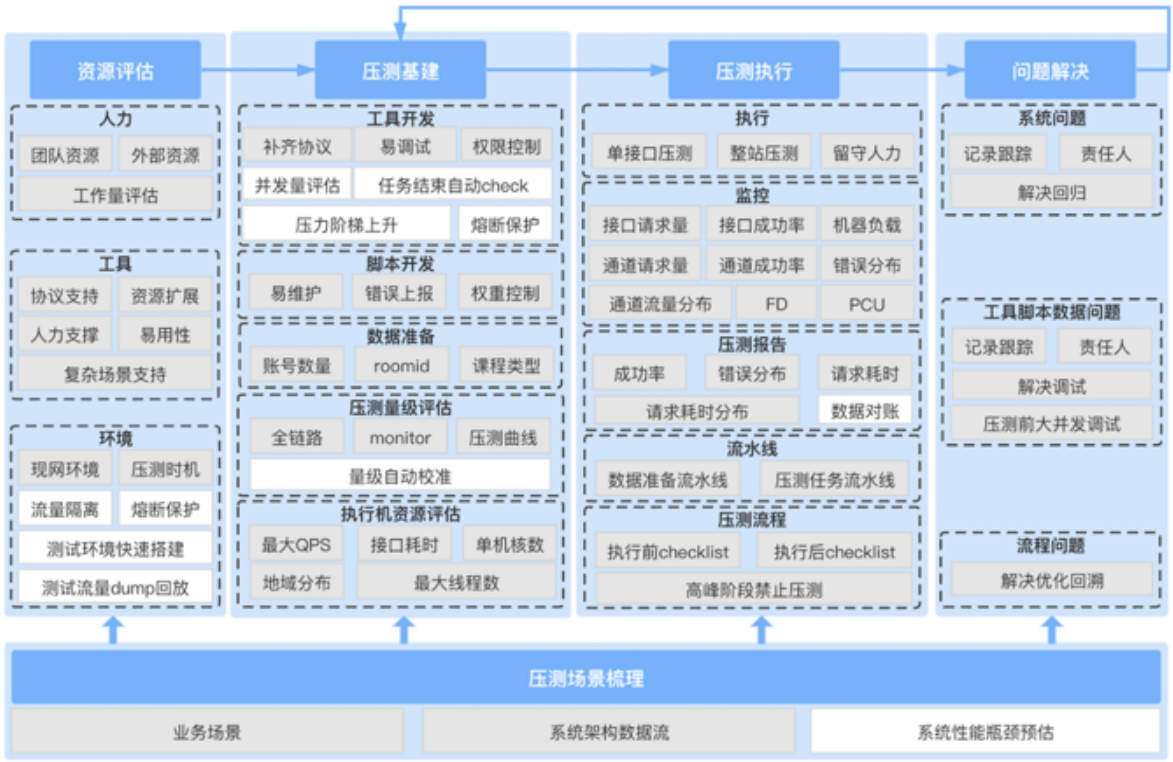
压测QCI流水线：



每日不达标记录：



全链路压测方案：



阶段2：瓶颈最先出在DB数据层

根据先抗住后优化的思路，可扩容的都容易解决，架构瓶颈会最先出现在伸缩性差、不容易扩容的环节上，经常会是数据层，课堂这次也中招了。

核心DB，一挂全挂

由于之前量较小，课堂大部分模块使用同个DB实例（ip+port），上量前这个核心DB的cpu在20%、qps在1k左右，评估下来风险很大：

1. 扩展性差：主机没法扩展，从机不建议超5组，且有主备延迟风险
2. 耦合度高：任一svr链接数或sql没控制好，就算是边缘Svr都可能搞垮DB一挂全挂
3. 梳理复杂：涉及svr数100+，时间太赶来不及逐个梳理

也是历史设计的坑，后面数据层设计要多考虑吞吐量和可扩展性。但回不去了，硬骨头得啃下来，立了专项找DBA同学一起分析优化，主要有下面几块：

	功能
压测分析	模拟600w后SQL读写量级压测，明确读写瓶颈
扩容RO组	从1主1从（业务混部），扩到1主5从（独立部署），规划清晰切更多读SQL到从机
业务分离	较独立的 评论/推荐/搜索/数据 模块，风险小快速切库独立部署
慢SQL优化	优化Top的40+条慢SQL，通过索引/切备/服务Cache/避免扫全表等降低主DB负载
链接数优化	非核心且占Top链接数模块，引入ProxySql中转层做链接池
参数调整	主DB超30s读SQLKill，防异常SQL雪崩
完善告警	CPU负载、链接数、慢查询数、主备不同步延迟等

业务横向拆分

根据压测发现非常明显的28原则，比如top1的写sql占总量82%，是搜索推荐模块定时刷权重用的，这类模块相对独立，和其他模块表关联join等操作少，方便业务拆分。对于这类模块，像搜索推荐、数据分析、评论系统等，快速切独立DB解耦，规避互相影响。

方便业务拆分的切走后，剩下能快速上的就是读写分离扩ro组了。快速扩了4个ro组，把较独立模块sql切过去，规避互相影响，也分摊了主机压力。因为复制模式是半同步的，也需关注主备同步延时，做好监控，特别是一些对延迟敏感的模块。

慢查询优化

横向能拆的都快速搞了，主DB风险还是很高，除了升级DB机器配置，剩下就只能逐个做慢sql优化了。采用mysqldumpslow对慢查询日志做归并排序，就可很清楚平均耗时/扫描行数/返回记录数top的慢sql，基本优化也是围绕着索引来，比如：

1. 查询没有走索引
2. 访问的数据量太大走全表
3. OR的情况无法使用索引
4. 在索引上面直接进行函数计算
5. 没有使用联合索引
6. 唯一索引没有命中数据

优化效果：主db峰值cpu负载从20%下降到5%左右。

链接数优化

链接数上也出过一些很惊险的case：鉴权svr凌晨扩100台机器，没考虑到对DB链接数影响，svr起来后DB链接数瞬间增长2k+差点爆掉。

除了对top的svr减少链接数外，引入DB代理也是个较快的解决方案，由于之前上云对ProxySql和NginxTcpProxy都有实践过，所以这次刚好也使用上。

具体可参考之前的文章《谈下mysql中间件（问题域、业内组件）》：

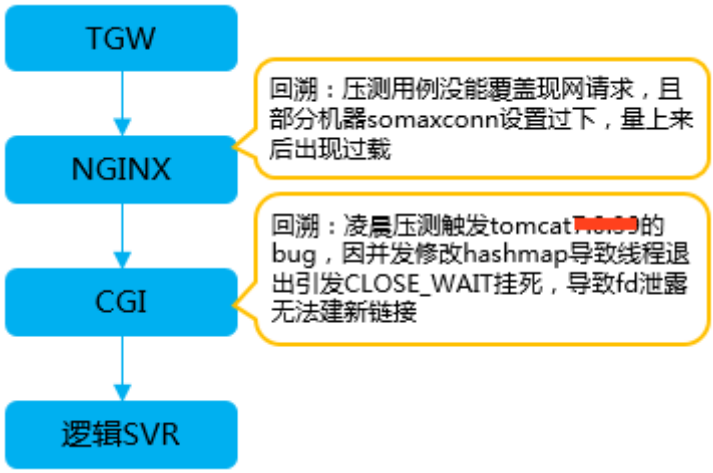
https://cloud.tencent.com/developer/article/1349133

优化效果：主db峰值链接数从4.6k下降到3.8k。

阶段3：核心模块逐个专项击破

接入：血的教训

课堂支持学生在pc/web/app/ipad/h5/小程序等多端进行学习，接入模块属于基础组件改动不大，但这次web接入模块却出了2个问题，确实不应该：



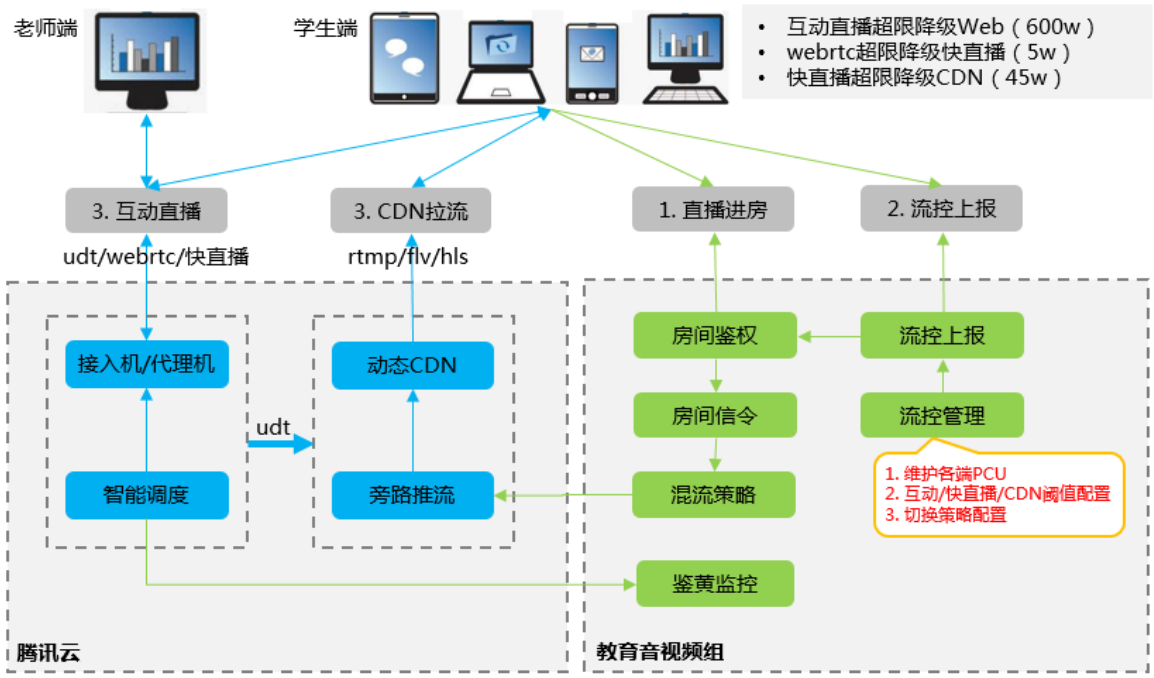
针对这2个问题也做了专项总结：

1. 校准压测用例，更模拟现网流量
2. nginx动静分离，上报等接口也独立出去
3. nginx规则精简，接入层尽量打薄，逻辑后移
4. 统一机器net.core.somaxconn等参数配置，重点监控告警
5. 压测完要清理战场，关注fd等指标是否恢复
6. 升级tomcat规避挂死bug
7. 升级tomcat遵循RFC3986规范，规避特殊字符影响

直播：站在云的肩膀上

课堂最核心的模块就是音视频，直播的进房成功率/首帧延迟/卡顿率/音画同步时延/分辨率等指标直接影响用户核心体验。由于直播模块之前已全部切云，这次站在云的肩膀上，业务不仅直接使用了云的多种直播模式，云音视频团队在整个疫情期间也提供非常给力的质量保障。

下面是具体的直播架构，业务通过流控Svr来控制各端走哪种直播模式：



消息：走推走拉?

随着极速版普及，各学校对单房间同时在线人数要求也越来越高，从3w->6w->30w->150w。

对于大房间消息系统而言，核心要解决消息广播风暴的问题（30w人房间，有100人在同1秒发言，会产生30w*100=3kw/s消息），随着扩散因子的变大，之前纯推的方案已不能满足需求，参考了业内直播间的一些IM方案：

	具体策略
走推	• 实时性好 • 消息量受广播系数影响大 • 通过消息限频合并，分割小房间等策略优化大房间问题 • 关注消息可靠性
走拉	• 定期轮训 • 实现简单易扩展，垒机器可解决 • 通过间隔周期退火等策略减少无效轮询调用 • 关注消息顺序性
推拉结合	• 小房间走推 • 大房间走拉 • 做好大小房间判定的阈值处理

结合对比方案，也针对课堂产品特性，在原来纯推架构上新迭代一版推拉结合的架构。

对于IM系统的设计，推荐一下这个文章：《新手入门一篇就够：从零开发移动端IM》

http://www.52im.net/thread-464-1-1.html

阶段4：做好核心路径的防过载和柔性降级

在非常有限的时间里，逻辑层根本来不及重构，容量评估也不一定精准，过载保护和柔性降级就显得尤其重要。

因为没时间全盘优化，在量突然暴涨的情况下，某个模块过载或者爆bug的概率会变大，所以在用户登录->查课表->上课的核心路径上，必须增加足够容错能力来提高可用性。

雪崩来得猝不及防

疫情初期课堂就遇到一个雪崩的case：直播间拉取成员列表接口有失败毛刺，因为web没做异常保护，失败直接把循环拉取间隔时间置0，导致接口调用量越滚越大，B侧拉取涨了10倍后雪崩超时。由于没预埋开关等控制策略，得回滚web版本才解决。

这个case暴露了过载保护的缺失，一方面web没对异常返回做合理处理保护后端，一方面svr没有识别雪崩请求做限频，除此之外，也缺少一些配置开关可快速控制web循环间隔时间，导致雪崩来得猝不及防。

过载保护策略

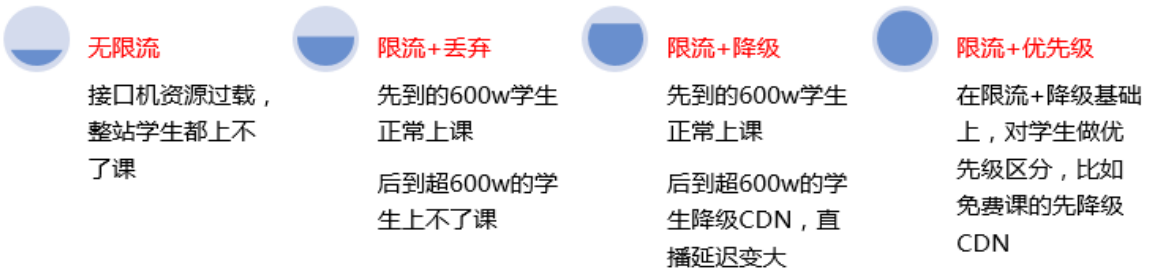
针对这类badcase，对核心路径的服务做了很多过载保护和柔性降级策略，这里把一些典型方案记录下：

	具体策略
自动扩容	基于TKE的HPA机制自动伸缩
流量控制	限制最大吞吐量或链接数，对超过阈值做降级处理
过载熔断	把过载的请求丢弃，保护后端
轻重分离	对易过载模块分离部署，规避级联影响到整站
降级开关	用户端预埋功能控制开关，过载等异常情况可手动关闭

限流和熔断

高并发场景为了规避过载的级联传递，防止全链路崩溃，制定合理的限流和熔断策略是2个常见的解决方案。

以这次疫情互动直播限流场景为例，互动直播默认只部署最多支撑600w同时在线的接口机资源，如果哪天突发超过了600w学生：



限流算法选择上，最常见就是漏桶和令牌桶。不是说令牌桶就是最好的，只有最合适的，有时简单的计数器反而更简单。golang拓展库 golang.org/x/time/rate 就提供了令牌桶限流器，3个核心api：

```
1、func (*Limiter) Allow: 没有取到token返回false，消耗1个token返回true
2、func (*Limiter) Wait: 阻塞等待，直到取到1个token
3、func (*Limiter) Reserve: 返回token信息，返回需要等待多久才有新的token
```

除了算法外，怎么把限流集成到框架、分布式限流实现、限流后请求优先级选择等问题，可以做得更深入，但很遗憾这次没时间搞，后面继续实践。

熔断是另一个重要防过载策略，其中熔断器Hystrix最为著名，github.com/afex/hystrix-go 就提供了其golang版本实现，使用也简单。其实L5就包含了熔断能力，包括熔断请求数阈值、错误率阈值和自动恢复探测策略。

Apollo配置中心

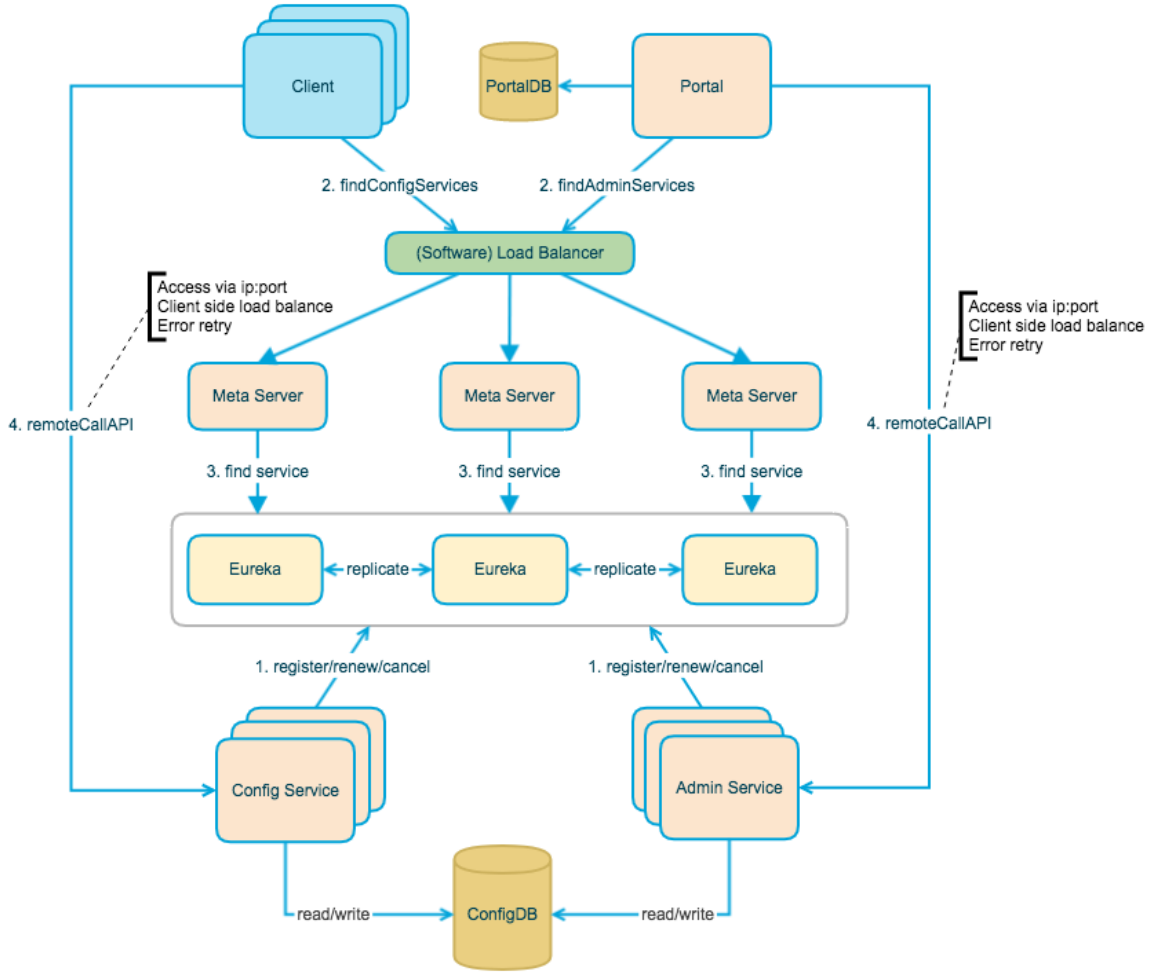
好的组件都是用脚投票，这次疫情期间，很多策略开关和阈值控制都是用Apollo配置中心来做，实现配置热更新，在高可用上实践也不错。很多时候多预埋这些配置就是用来保命的，当监控发现趋势不对时，可快速调整规避事故发生，简单列些例子：

1. 后端限流阈值大小，后端要过载时可调小
2. Cache缓存时间，数据层负载高时可调大
3. 非核心路径后端调用开关，必须时关闭调用补上降级默认值
4. 前端定时调用的间隔时间，后端要过载时可调大

   ……

当然，如果可做到系统自动触发调整配置就更进一步了，当时有想过但时间太赶没实践，有兴趣同学可思考实践下。

Apollo是携程开源的分布式配置中心，能够集中化管理应用不同环境配置，实现配置热更新，具备规范的权限、流程治理等特性，适用于微服务配置管理场景。补个架构图推荐下：



阶段5：服务性能优化实战

在抗住前2周最猛的流量增长后，下来很长一段时间都是在优化服务的性能和稳定性、处理用户反馈和打磨产品体验上。这里沉淀3个服务性能优化上印象较深刻的点。

分析利器 pprof+torch

在性能分析上，对比c++，golang提供了更多好用的工具，基本每次性能分析都是先用pprof+torch跑一把。通过框架中嵌入net/http/pprof并监听http遥测端口，管理后台就可随时得到svr协程/cpu/内存等相关指标，比如优化前的成员列表svr火焰图case。

结合代码，便可快速有些优化思路，比如：

1. 分离B/C调用部署
2. 优化pb序列化，如做些cache
3. 简化定时器使用场景
4. 调整大对象使用优化gc消耗

最终根据这些优化思路改版，让超200ms的比例从0.2%降到0.002%以下：



缓存设计和踩坑

回过头看，大部分服务性能瓶颈还是在数据层或Rpc调用上，很多时候数据一致性要求没那么高，加缓存是最简单的首选方案。

关于缓存的设计，无论是本地缓存、分布式缓存、多级缓存，还是Cache Aside、Read/Write Through、Write Behind Caching等缓存模式，就看哪种更适合业务场景，这里也不累赘，核心说下这次实践中踩的2个坑：

1、缓存击穿

- 案例：高频访问的缓存热key突然失效，导致对这个key的读瞬间压到DB上飙高负载
- 方案：使用异步更新或者访问DB加互斥锁

2、缓存穿透

- 案例：访问DB中被删除的key，这些key在缓存中也没有，导致每次读直接透到DB
- 方案：把这些key也缓存起来，但要关注恶意扫描的影响

为啥qps压不上去?

疫情期间，有一个现象很奇怪但又经常出现：压测时cpu很低，pprof+torch看不出什么异常，数据层返回也很快，但吞吐量就是上不去。一开始思路较少，后面也慢慢知道套路了，这里列几个真实的case供参考：

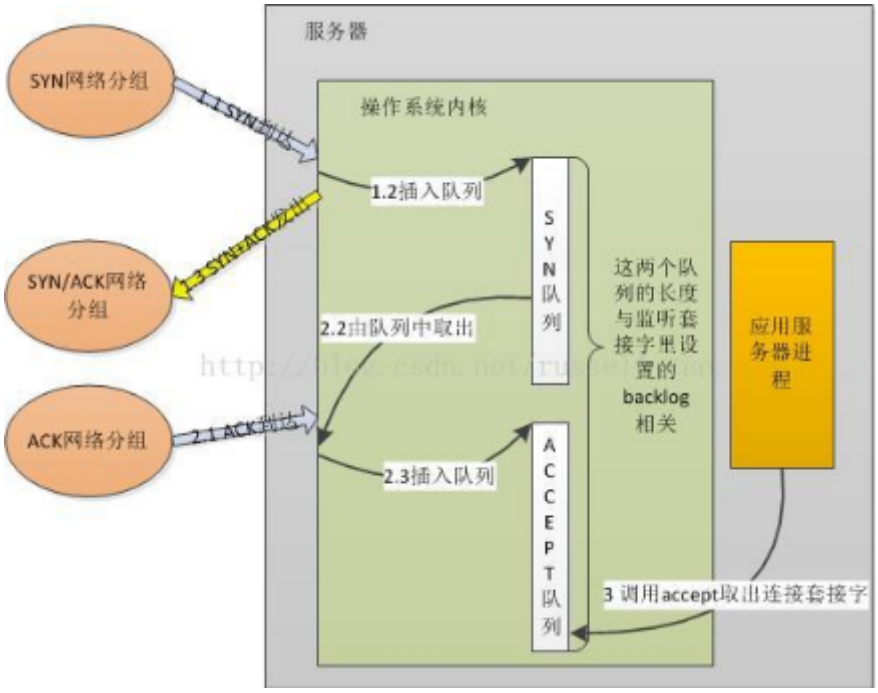
1. 锁竞争：如死锁、锁粒度太大等，关注锁时间上报
2. 打日志：日志量过大等导致磁盘IO彪高，在高并发场景尤其要注意精简日志量
3. 进程重启：如panic或oom导致进程被kill，重启过程请求超时，要补齐进程重启监控
4. 队列丢包：如请求缓存队列设置过小等，要关注队列溢出监控

   ……

比如最后这点，就遇过这样的case：一次凌晨压测，其他机器都正常，就2个新机器死活一直超时，业务指标也看不出区别，折腾了好一阵，才发现monitor上 监听队列溢出(ListenOverflows) 这个值有毛刺异常。

继续深挖下去，证明请求在tcp队列就溢出了，tcp的accept队列长度=min(backlog,SOMAXCONN)，查看新机器内核配置 net.core.somaxconn=128，确实比其他机器小，神坑。

所以后续也增加了服务器tcp的半连接和全连接队列相关监控：



从技术演变的角度看互联网后台架构

作者： likezhang

这是去年在部门内部做的一个面向后台开发新同学的课程，因为其他BG一些同学要求分享，所以发一下。其实内容都是些常见开源组件的high level描述，比如flask, express框架，中间件的演化，micro service的概念，一些对nosql/column based db的概念介绍，docker的一些简单概念等等。从单个概念来说，这只是一些科普。

但是为什么当时要开这门课呢？重点是我发现很多新入职的后台开发同学并不太清楚自己做的东西在现代互联网整体架构中处于一个什么样的角色，而在IEG内部则因为游戏开发和互联网开发的一些历史性差异，有些概念并不清晰。

拿中间件来说，很多web application不用啥中间件一样可以跑很好，那么是不是都要上redis？到底解决什么问题？中间件又存在什么问题？中台和中间件又是个什么关系？如果开个mq就是中间件，微服务又是要做啥？

如果能从这十多年来互联网应用的整个tech stack变化去看待backend architecture的一些改变，应该是一件有趣也有意思的事情。这是当时写这个ppt开课的初衷。

全文链接：https://mp.weixin.qq.com/s/ybQd4zXyfQlUdbvbQl0h9A

Linux系统——架构浅析

作者： jackycao，腾讯 IEG 客户端开发工程师

> 掐指一算自己从研究生开始投入到Linux的海洋也有几年的时间，即便如此依然对其各种功能模块一知半解。无数次看了Linux内核的技术文章后一头雾水，为了更系统地更有方法的学Linux，特此记录。 掐指一算自己从研究生开始投入到Linux的海洋也有几年的时间，即便如此依然对其各种功能模块一知半解。无数次看了Linux内核的技术文章后一头雾水，为了更系统地更有方法的学Linux，特此记录。

1991年，还在芬兰赫尔辛基大学上学的Linus Torvalds在自己的Intel 386计算机上开发了属于他自己的第一个程序，并利用Internet发布了他开发的源代码，将其命名为Linux，从而创建了Linux操作系统，并在同年公开了Linux的代码，从而开启了一个伟大的时代。在之后的将近30年的时间里，越来越多的工程师投入到Linux，帮助不断完善Linux的功能。现在的Linux系统架构凭借优秀的分层和模块化的设计，融合了大量的设备和不同的物理架构。

写这篇文章，也是对Linux系统的一个非常简单的介绍，主要讲解Linux的**进程调度、内存管理、设备驱动、文件系统、网络模块**。

全文链接：https://mp.weixin.qq.com/s/Kk9nuRMu3Ce4DCZY381Hlw

深入理解计算机系统：进程

作者：levonli，腾讯 CSIG 后台开发工程师

> 这是篇读书笔记，每次重读CSAPP都有新的认知，尤其是在进入了后台通道之后才感受到每天和进程打交道的感觉是如此深刻。

全文链接：https://mp.weixin.qq.com/s/z6K8C56FnNVKu6XAQefViQ

zookeeper 源码和实践浅析

作者： runnerzhang，腾讯 CSIG 后台开发工程师

ZooKeeper 是个针对大型分布式系统的高可用、高性能且具有一致性的开源协调服务，被广泛的使用。对于开发人员，ZooKeeper 是一个学习和实践分布式组件的不错的选择。本文对 ZooKeeper 的源码进行简析，也会介绍 ZooKeeper 实践经验，希望能帮助到初学 ZooKeeper 的同事。文章部分内容参考了一些网络文章，已标注在末尾参考文献中。

导言

初衷

在业务中使用了 ZooKeeper 作为消息系统，在开发和运维过程中，也遇到一些问题，萌发了阅读源码窥视实现细节的想法。同时我们运维的 ZooKeeper 集群规模和数据规模非常大，也想把运维的经验分享出来供参考去规避风险点和性能调优。

目标读者

本文是介绍 ZooKeeper 基础知识和源码分析的**入门级**材料，适合用于**初步**进入分布式系统的开发人员，以及使用 ZooKeeper 进行生产经营的应用程序运维人员。

本文内容介绍

第 1 章节：主要介绍 ZooKeeper 使命、地位、基础的概念和基本组成模块。

第 2 章节：主要介绍 ZooKeeper 内部运行原理，此部分主要从书籍《ZooKeeper 分布式过程协同技术详解》摘录，对于有 ZooKeeper 基础的可以略过。写第二章节的主要目的，不先陷入解析源码的繁琐的实现上，而是从系统和底层看 ZooKeeper 如何运行，通过从高层次介绍其所使用的协议，以及 ZooKeeper 所采用的在提高性能的同时还具备容错能力的机制，。

第 3 章节：简析 ZooKeeper 的源码实现，主要目的去介绍 ZooKeeper 集群的工作流程，给出看源码的简要指引，能更快上手去深入阅读源码

第 4 章节：主要介绍业务用 zookeeper 做消息系统的实践，在实践中的优化点和踩坑的地方，由于业务场景和规模的差别，关注点和优化点也差别很大，也欢迎在评论区更新使用 ZooKeeper 共性问题。

ZooKeeper 简介

在大数据和云计算盛行的今天，应用服务由很多个独立的程序组成，这些独立的程序则运行在形形色色，千变万化的一组计算机上，而如何让一个应用中的多个独立的程序协同工作是一件非常困难的事情。而 ZooKeeper 就是一个分布式的，开放源码的分布式应用程序协调服务。它使得应用开发人员可以更多的关注应用本身的逻辑，而不是协同工作上。从系统设计看，ZooKeeper 从文件系统 API 得到启发，提供一组简单的 API，使得开发人员可以实现通用的协作任务，例如选举主节点，管理组内成员的关系，管理元数据等，同时 ZooKeeper 的服务组件运行在一组专用的服务器之上，也保证了高容错性和可扩展性。

本章节主要从 ZooKeeper 的使命、ZooKeeper 基础、工业级案例，来简要介绍 ZooKeeper。

ZooKeeper 的使命

Apache ZooKeeper 的官方介绍，ZooKeeper is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration maintenance, and groups and naming. It is designed to be easy to program，and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C.

ZooKeeper: A Distributed Coordination Service for Distributed Applications

ZooKeeper is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration maintenance, and groups and naming. It is designed to be easy to program to, and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C.

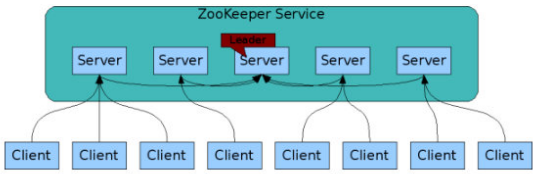
Coordination services are notoriously hard to get right. They are especially prone to errors such as race conditions and deadlock. The motivation behind ZooKeeper is to relieve distributed applications the responsibility of implementing coordination services from scratch.

Design Goals

ZooKeeper is simple. ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical namespace which is organized similarly to a standard file system. The name space consists of data registers - called znodes, in ZooKeeper parlance - and these are similar to files and directories. Unlike a typical file system, which is designed for storage, ZooKeeper data is kept in-memory, which means ZooKeeper can achieve high throughput and low latency numbers.

The ZooKeeper implementation puts a premium on high performance, highly available, strictly ordered access. The performance aspects of ZooKeeper means it can be used in large, distributed systems. The reliability aspects keep it from being a single point of failure. The strict ordering means that sophisticated synchronization primitives can be implemented at the client.

ZooKeeper is replicated. Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a sets of hosts called an ensemble.



The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.

Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.

ZooKeeper is ordered. ZooKeeper stamps each update with a number that reflects the order of all ZooKeeper transactions. Subsequent operations can use the order to implement higher-level abstractions, such as synchronization primitives.

ZooKeeper is fast. It is especially fast in "read-dominant" workloads. ZooKeeper applications run on thousands of machines, and it performs best where reads are more common than writes, at ratios of around 10:1.

从上面的官方介绍看，ZooKeeper 主要的系统功能是在分布式系统中协作多个任务。例如，典型的主-从工作模式中，我们需要主节点和从节点进行协作，在从节点处于空闲状态时会通知主节点可以接受工作，于是主节点就会分配任务给从节点，同时我们只想有一个主节点，而很多进程可能都想成为主节点，这些操作都是要在多个任务中进行协作。另外，协同并不总是采取像主节点选举或者加锁等同步原语的形式，**配置元数据**也是一个进程通知其他进程需要做什么的一种常用数据，例如，在一个主-从系统中，从节点需要知道任务已经分配给他们，即便在主节点发生崩溃的情况下，这些信息也需要有效。

在 ZooKeeper 之前，一些系统也可以采用分布式锁管理器或者分布式数据库来实现协作，例如，用数据库，redis 实现分布式锁。

那么 ZooKeeper 改变了什么呢？ZooKeeper 的设计更专注于任务协作，它不提供任何锁的接口或者通用的存储数据的接口，也没有强加任何特殊的同步原语，而是提供一个更加敏捷健壮的分布协作方案，例如在主-从模型中，ZooKeeper 没有为应用实现主节点选举，或者进程存活与否的跟踪功能，但是，ZooKeeper 提供了实现这些任务的工具，对于实现什么样的协同任务，有开发人员自己决定。

分布式系统中关键在于进程通信，其有两种选择：直接通过网络进行信息交换，或者读写某些共享存储。对于 ZooKeeper 实现协作和同步原语本质上是使用**共享存储模型**，即开发的应用是连接到 ZooKeeper 服务器端的客户端，他们连接到 ZooKeeper 服务器端进行相关的操作，以来影响服务器端存储的共享数据，最终应用间实现协作。

ZooKeeper 不适合的场景

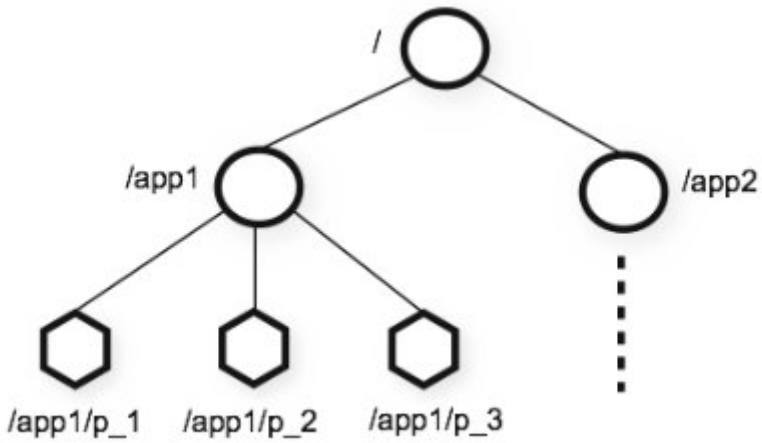
整个 ZooKeeper 的服务器集群管理着应用协作的关键数据，ZooKeeper 不适合用作海量的数据存储，对于需要海量的应用数据的情况，可以使用数据库和分布式文件系统，所以在设计应用时，最佳实践是把应用数据和协同数据独立分开。

ZooKeeper 基础简介

本小节简要介绍 ZooKeeper 数据结构、监控与通知、服务架构，详细的介绍请查看 ZooKeeper 技术文档和书籍。

1.2.1 ZooKeeper 数据结构

ZooKeeper 采用类似于文件系统的层级树状结构进行管理 Znode，并且暴露操作 API 接口。



Znode 的节点类型：在新建 znode 节点，需要指定该节点的类型，不同的类型决定了 znode 节点的行为方式，znode 的类型分为**持久节点、时节点、有序节点，**组合 4 中类型**，持久的，临时的，持久有序的，临时有序的。**对于持久节点，只能主动调用 delete 来删除，而临时的 znode，在当创建该节点的客户端崩溃或者关闭了与 ZooKeeper 的连接时，这个节点就会被删除。一般持久类型的 znode 为应用保存数据，即使 znode 的创建者不再属于应用系统时，数据会保存下来而不丢失。临时 Znode 仅当创

建者的会话有效时这些信息必须有效保存，会话超时或者主动关闭时，临时 znode 会自动消失。**有序 Znode 节点**是被分配唯一一个单调递增的整数。

API 接口

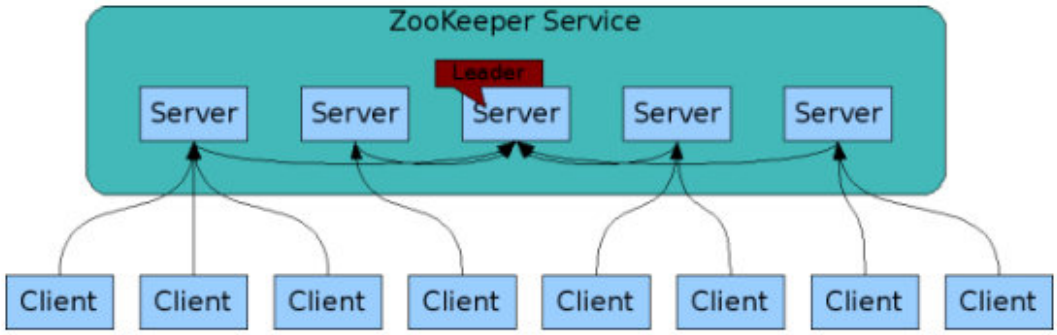
> create /path data 创建一个名为/path 的 znode 节点，并包含数据 data。
>
> delete /path 删除名为/path 的 znode。
>
> exists /path 检查是否存在名为/path 的节点。
>
> setData /path data
>
> getData /path
>
> getChildren /path

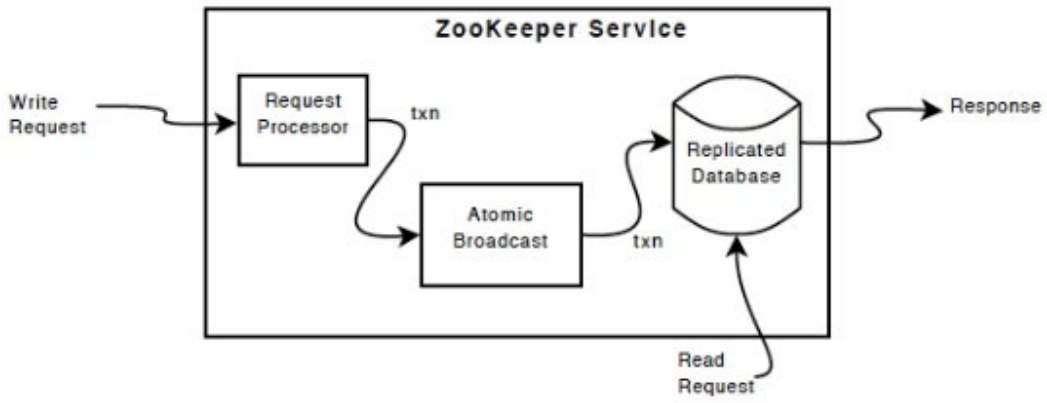
需要注意的是，ZooKeeper 并不允许局部写入或读取 znode 节点的数据。当设置一个 znode 节点的数据或读取时，znode 节点的内容会被整个替换或者全部读取进来，特别是 getChildren，如果是数据量比较大，会获取大量的数据。

1.2.2 ZooKeeper 监视与通知

ZooKeeper 客户端获得服务器的数据或者变化，不是通过轮询的模式，而是基于通知的机制，客户端向 ZooKeeper 服务器端注册需要接收通知的 znode，通过对 znode 设置监视点来接收通知，需要强调的是监视点是一个**单次触发**的操作。

1.2.3 ZooKeeper 架构





ZooKeeper 服务器端运行于两种模式下：独立模式和仲裁模式。独立服务器只有一个单独的服务器，ZooKeeper 状态无法复制。而在仲裁模式下，具有一组 ZooKeeper 服务器，称为 ZooKeeper 集合，它们之间可以进行状态的复制，并同时服务客户端的请求。不过服务器集合并不会让客户端等待每个服务器完成数据保存后再继续，而是在满足仲裁数目的服务器保存或者同步了状态就会返回给客户端。在解决这一分布式数据一致性，ZooKeeper 采用 ZAB(ZooKeeper Atomic Broadcast)的一致性协议，关于 ZAB 协议后面会详细的介绍。

ZooKeeper 客户端在服务器集群中执行任何请求前必须先与服务器建立会话（**session**），客户端提交给 ZooKeeper 的所有操作均关联在一个会话上。客户端初始化连接到集合中某个服务器或一个独立的服务器，客户端提供**TCP 协议**与服务器进行连接并通信，但当会话无法与当前连接的服务器继续通信时，会话就可能转移到另外一个服务器，ZooKeeper 客户端**透明地**转移一个会话到不同的服务器。需要指明的，会话提供了顺序保障，同一个会话中的请求会以 FIFO（先进先出）顺序执行。

ZooKeeper 应用案例

Apache HBase

HBase 是一个通常与 Hadoop 一起使用的数据存储仓库。在 HBase 中，ZooKeeper 用于选举一个集群内的主节点，以便跟踪可用的服务器，并保持集群的元数据。

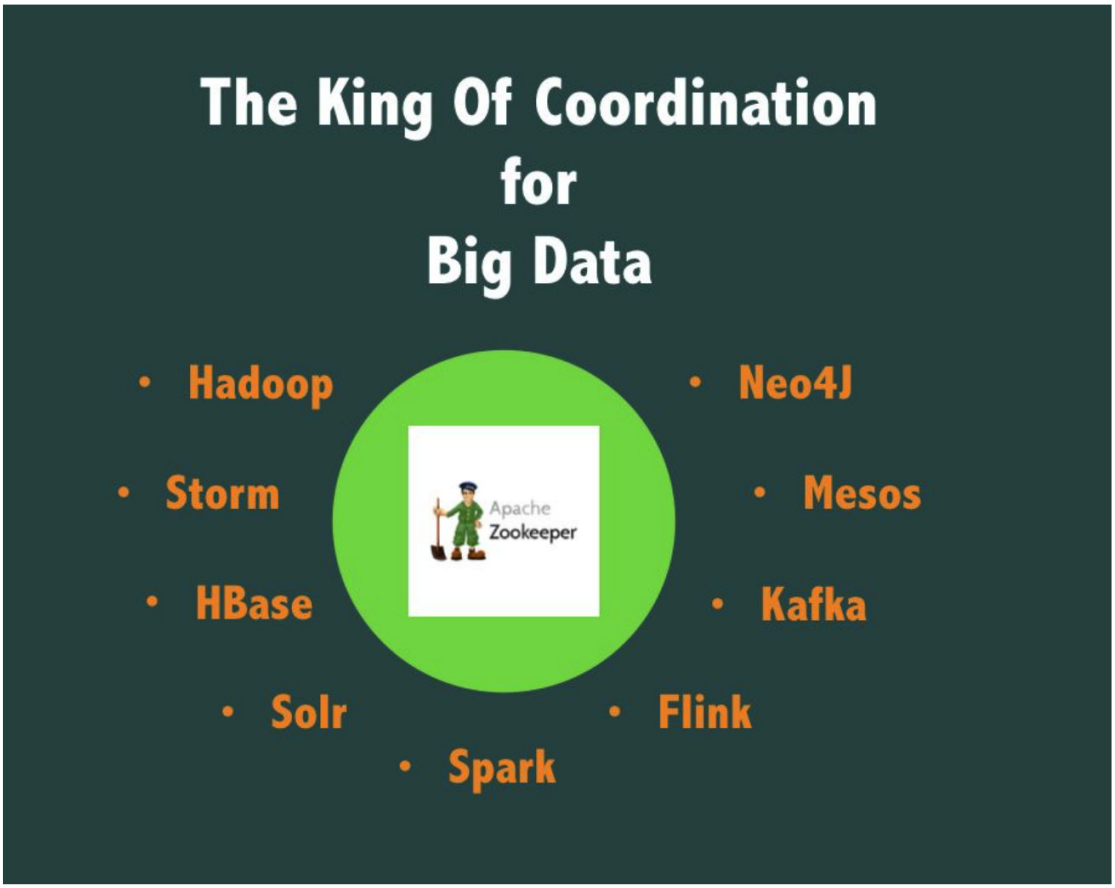
Apache Kafka

Kafka 是一个基于发布-订阅模型的消息系统。其中 ZooKeeper 用于检测崩溃，实现主题的发现，并保持主题的生产和消费状态。

Apache Solr

Solr 是一个企业级的搜索平台，它使用 ZooKeeper 来存储集群的元数据，并协作更新这些元数据。

ZooKeeper 应该是 “The King Of Coordination for Big Data”!



ZooKeeper 内部实现原理

本章节是介绍 ZooKeeper 内部是如何运行的，通过从高层次介绍其所使用的协议，以及 ZooKeeper 所采用的在提供高性能的同时还具有容错能力的机制。总体来说 ZooKeeper 运行于一个集群环境中，选举出某个服务器作为群首（Leader），其他服务器追随群首（Follower）。群首作为中心处理所有对 ZooKeeper 系统变更的请求，它就像一个定序器，建立了所有对 ZooKeeper 状态的更新的顺序，追随者接收群首所发出更新操作请求，并对这些请求进行处理，以此来保障状态更新操作不会发生碰撞。

2.1 请求、事务、标识符

ZooKeeper 服务器会在本地处理只读请求（exists、getData、getChildren），例如一个服务器接收客户端的 getData 请求，服务器读取该状态信息，并把这些信息返回给客户端。

那些会改变 ZooKeeper 状态的客户端请求（create，delete 和 setData）将会转发到群首，群首执行对应的请求，并形成状态的更新，称为事务（transaction），其中事务要以**原子**方式执行。同时，一个事务还要具有**幂等性**，事务的幂等性在我们进行恢复处理时更加简单，后面我们可以看到如何利用幂等性进行数据恢复或者灾备。

在群首产生了一个事务，就会为该事务分配一个标识符，称为会话 id（zxid），通过 Zxid 对事务进行标识，就可以按照群首所指定的顺序在各个服务器中按序执行。服务器之间在进行新的群首选举时也会交换 zxid 信息，这样就可以知道哪个无故障服务器接收了更多的事务，并可以同步他们之间的状态信息。

Zxid 为一个 long 型（64 位）整数，分为两部分：时间戳（epoch）部分和计数器（counter）部分。每一部分为 32 位，在我们讨论 zab 协议时，我们就会发现时间戳（epoch）和计数器（counter）的具体作用，我们通过 zab 协议来广播各个服务器的状态变更信息。

2.2 群首选举

群首为集群中的服务器选择出来的一个服务器，并会一直被集群所认可。设置群首的目的是为了对客户端所发起的 ZooKeeper 状态更新请求进行**排序**，包括 create，setData 和 delete 操作。群首将每一个请求转换为一个事务，将这些事务发送给追随者，确保集群按照群首确定的顺序接受并处理这些事务。

每个服务器启动后进入 LOOKING 状态，开始选举一个新的群首或者查找已经存在的群首。如果群首已经存在，其他服务器就会通知这个新启动的服务器，告知哪个服务器是群首，于此同时，新服务器会与群首建立连接，以确保自己的状态与群首一致。如果群首中的所有的服务器均处于 LOOKING 状态，这些服务器之间就会进行通信来选举一个群首，通过信息交换对群首选举达成共识的选择。在本次选举过程中胜出的服务器将进入 LEADING 状态，而集群中其他服务器将会进入 FOLLOWING 状态。

具体看，一个服务器进入 LOOKING 状态，就会发送向集群中每个服务器发送一个通知信息，该消息中包括该服务器的投票（vote）信息，投票中包含服务器标识符（sid）和最近执行事务的 zxid 信息。

当一个服务器收到一个投票信息，该服务器将会根据以下规则修改自己的投票信息：

1. 将接收的 voteId 和 voteZxid 作为一个标识符，并获取接收方当前的投票中的 zxid，用 myZxid 和 mySid 表示接收方服务器自己的值。
2. 如果（voteZxid > myZxid）或者（voteZxid == myZxid 且 voteId >mySid）,保留当前的投票信息。
3. 否则，修改自己的投票信息，将 voteZxid 赋值给 myZxid，将 voteId 赋值给 mySid。

从上面的投票过程可以看出，只有最新的服务器将赢得选举，因为其拥有最近一次的 zxid。如果多个服务器拥有的最新的 zxid 值，其中的 sid 值最大的将会赢得选举。

当一个服务器连接到仲裁数量的服务器发来的投票都一样时，就表示群首选举成功，如果被选举的群首为某个服务器自己，该服务器将会开始行使群首角色，否则就会成为一个追随者并尝试连接被选举的群首服务器。一旦连接成功，追随者和群首之间将会进行状态同步，在同步完成后，追随者才可以进行新的请求。

2.3 Zab：状态更新的广播协议

在接收到一个写请求操作后，追随者会将请求转发给群首，群首将会探索性的执行该请求，并将执行结果以事务的方式对状态更新进行广播。如何确认一个事务是否已经提交，ZooKeeper 由此引入了 zab 协议，即原子广播协议（**ZooKeeper Atomic Broadeast protocol**）。该协议提交一个事务非常简单，类型于一个两阶段提交。

1. 群首向所有追随者发送一个 PROPOSAL 消息 p。

2. 当一个追随者接收到消息 p 后，会响应群首一个 ACK 消息，通知群首其已接受该提案（proposal）。
3. 当收到仲裁数量的服务器发送的确认消息后（该仲裁数包括群首自己），群首就会发送消息通知追随者进行提交（COMMIT）操作。

Zab 保障了以下几个重要的属性

如果群首按顺序广播了事务 T 和事务 T，那么每个服务器在提交 T 事务前保证事务 T 已经完成提交。

如果某个服务器按照事务 T 和事务 T 的顺序提交了事务，所有其他服务器也必然会在提交事务 T 前提交事务 T。

第一个属性保证事务在服务器之间传送顺序的一致，而第二个竖向保证服务器不会跳过任何事务。

2.4 观察者

观察者与追随者有一些共同的特点，他们提交来自群首的提议，不同于追随者的是，观察者不参与选举过程，他们仅仅学习经由 INFORM 消息提交的提议。

引入观察者的一个主要原因是提高读请求的可扩展性。通过加入多个观察者，我们可以在不牺牲写操作的吞吐率的前提下服务更多的读操作。但是引入观察者也不是完全没有开销，每一个新加入的观察者将对应于每一个已提交事务点引入的一条额外消息。

采用观察者的另外一个原因是进行跨多个数据中心部署。由于数据中心之间的网络链接延时，将服务器分散于多个数据中心将明显地降低系统的速度。引入观察者后，更新请求能够先以高吞吐量和低延迟的方式在一个数据中心内执行，接下来再传播到异地的其他数据中心得到执行。

2.5 服务器的构成

群首，追随者，观察者根本上都是服务器。在实现服务器主要抽象概念是请求处理器。请求处理器是对处理流水线上不同阶段的抽象，每个服务器实现一个请求处理器的序列。

2.5.1 独立服务器

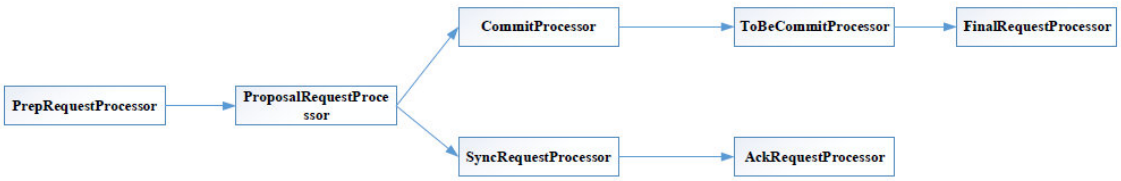
PrepRequestProcessor 接受客户端的请求并执行这个请求，处理结果则是生成一个事务。不过只有改变 ZooKeeper 状态的操作才会产生事务，对于读操作并不会产生任何事务。

SyncRequestProcessor 负责将事务持久化到磁盘上。实际上就是将事务数据按照顺序追加到事务日志中，并形成快照数据。

最后一个处理器为 FinalRequestProcessor，如果 Request 对象包含事务数据，该处理器就会接受对 ZooKeeper 数据树的修改，否则，该处理器会从数据树中读取数据并返回客户端。

2.5.2 群首服务器

在切换到仲裁模式时，服务器的流水线则有一些变化。



第一个处理器同样是 PrepRequestProcessor，而之后的处理器则为 ProposalRequestProcessor，该处理器会准备一个提议，并将该提议发送给跟随者，并且会把所有请求转发给 CommitRequestProcessor，对于写操作请求，还会把请求转发给 SyncRequestProcessor 处理器。

SyncRequestProcessor 和独立服务器的功能一样，是持久化事务到磁盘上，执行完后会触发 AckRequestProcessor 处理器，它仅仅生成确认消息并返回给自己。

CommitRequestProcessor 会将收到足够多的确认消息的提议进行提交。

2.5.3 追随者和观察者服务器

Follower 服务器是先从 FollowerRequestProcessors 处理器开始，该处理器接收并处理客户端请求，FollowerRequestProcessors 处理器之后转发请求给 CommitRequestProcessor，同时也会转发写请求到群首服务器。CommitRequestProcessor 会直接转发读取请求到 FinalRequestProcessor 处理器，而且对于写请求，在转发前会等待提交事务。而群首接收到一个新的写请求时会生成一个提议，之后转发到追随者服务器，在收到一个提议，追随服务器会发送这个提议到 SyncRequestProcessor，SendRequestProcessor 会向群首发送确认消息。

当群首服务器接收到足够多确认消息来提交这个提议是，群首就会发送提交事务消息给追随者，当收到提交的事务消息时，追随者就通过 CommitRequestProcessor 处理器进行处理。为了保证执行的顺序，CommitRequestProcessor 处理器会在收到一个写请求处理器时暂停后续的请求处理。

对于观察者服务器不需要确认提议消息，因此观察者服务器并不需要发送确认消息给群首服务器，一般情况下，也不用持久化事务到磁盘。对于观察者服务器是否持久化事务到磁盘，以便加速观察者服务器的恢复速度，可以根据具体情况决定。

2.6 本地存储

SyncRequestProcessor 处理器是用于处理提议写入的日志和快照。

日志和磁盘的使用

服务器通过事务日志来持久化事务。在接受一个提议时，一个服务器就会将提议的事务持久化到事务日志中，该事务日志保存在服务器本地磁盘中，而事务将会按照顺序追加其后。**写事务日志是写请求操作的关键路径**，因此 ZooKeeper 必须有效处理写日志问题。在持久化事务到磁盘时，还有一个重要说明：现代操作系统通常会缓存脏页（Dirty Page），并将他们异步写入磁盘介质。然而，我们需要在继续之前，要确保事务已经被持久化。因此我们需要冲刷（Flush）事务到磁盘介质。

冲刷在这里就是指我们告诉操作系已经把脏页写入到磁盘，并在操作完成后返回。同时为了提高 ZooKeeper 系统的运行速度，也会使用组提交和补白的。其中组提交是指一次磁盘写入时追加多个事务，可以减少磁盘寻址的开销。补白是指在文件中预分配磁盘存储块。

快照

快照是 ZooKeeper 数据树的拷贝副本，每一个服务器会经常以序列化整个数据树的方式来提取快照，并将这个提取的快照保存到文件。服务器在进行快照时不需要进行协作，也不需要暂停处理请求。因此服务器在进行快照时还会继续处理请求，所以当快照完成时，数据树可能又发生了变化，称为快照是模糊的，因为它们不能反映出在任意给定的时间点数据树的准确的状态。

2.7 服务器与会话

会话（session）是 ZooKeeper 的一个重要的抽象。保证请求有序，临时 znode 节点，监控点都与会话密切相关。因此会话的跟踪机制对 ZooKeeper 来说也是非常重要的。

在独立模式下，单个服务器会跟踪所有的会话，而在仲裁模式下则由群首服务器来跟踪和维护。而追随者服务器仅仅是简单地把客户端连接的会话信息转发到群首服务器。

为了保证会话的存活，服务器需要接收会话的心跳信息。心跳的形式可以是一个新的请求或者显式的 ping 信息。两种情况下，服务器通过更新会话的过期时间来触发会话活跃，在仲裁模式下，群首服务器发送一个 PING 信息给它的追随者们，追随者们返回自从最新一次 PING 消息之后的一个 session 列表。群首服务器每半个 tick 就会发送一个 ping 信息给追随者们。

2.8 服务器与监视点

监视点是由读取操作所设置的**一次性触发器**，每个监视点有一个特定操作来触发，即通过监视点，客户端可以对指定的 znode 节点注册一个通知请求，在发生时就会收到一个单次的通知。监视点只会存在内存，而不会持久化到硬盘，当客户端与服务端的连接断开时，它的所有的监视点会从内存中清除。因为客户端也会维护一份监视点的数据，在重连之后，监视点数据会再次同步到服务端。

2.9 客户端

在客户端库中有 2 个主要的类：ZooKeeper 和 ClientCnxn，写客户端应用程序时通过实例化 ZooKeeper 类来建立一个会话。一旦建立起一个会话，ZooKeeper 就会使用一个会话标识符来关联这个会话。这个会话标识符实际上是有服务端所生产的。

ClientCnxn 类管理连接到 server 的 socket 连接。该类维护一个可连接的 ZooKeeper 的服务列表，并当连接断掉的时候无缝地切换到其他服务器，当重连到一个其他的服务器时会使用同一个会话，客户端也会重置所有的监视点到刚连接的服务器上。

2.10 序列化

对于网络传输和磁盘保存的序列化消息和事务，ZooKeeper 使用了 Hadoop 中的 Jute 来做序列化。

ZooKeeper 源码浅析

以 3.5.5 版本作为分析。主要从服务端，客户端，以及服务端和客户端结合的部分分析源码。在分析源码时，主要从数据结构，类结构，线程模型，流程等方面看。（注：本章节参考了网上 ZooKeeper 的分析文章，借用了不少文字描述。）

3.1 服务端

ZooKeeper 服务的启动方式分为三种，即单机模式、伪分布式模式、分布式模式。本章节主要研究分布式模式的启动模型，其主要要经过 Leader 选举，集群数据同步，启动服务器。

分布式模式下的启动过程包括如下阶段，

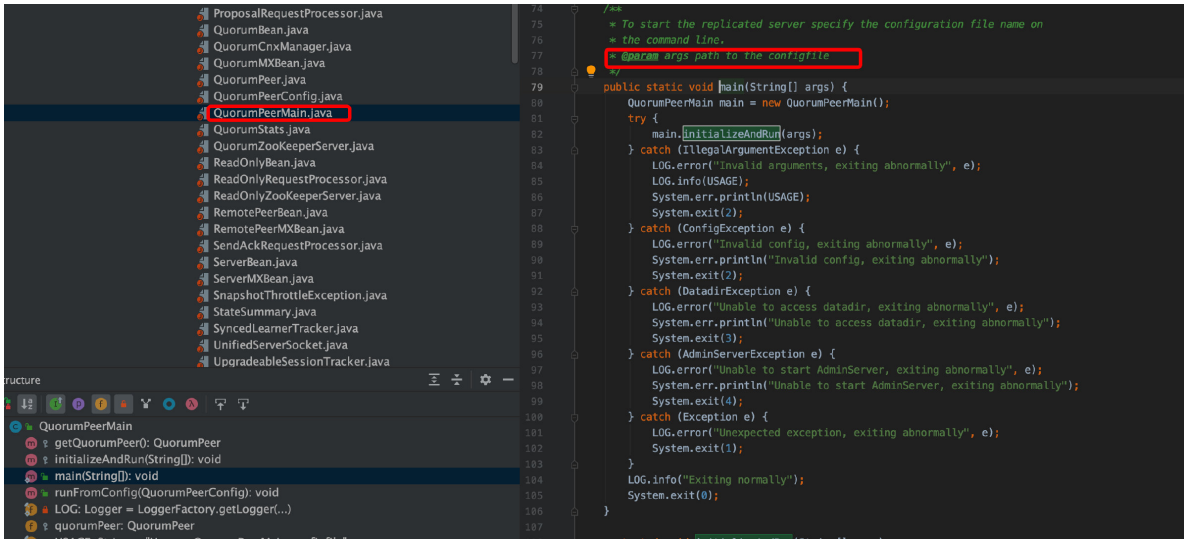
- 解析 config 文件；
- 数据恢复；
- 监听 client 连接(但还不能处理请求)；
- bind 选举端口监听 server 连接；
- 选举；
- 初始化 ZooKeeperServer；
- 数据同步；
- 同步结束，启动 client 请求处理能力。

3.1.1 服务端启动流程（分布式模式）

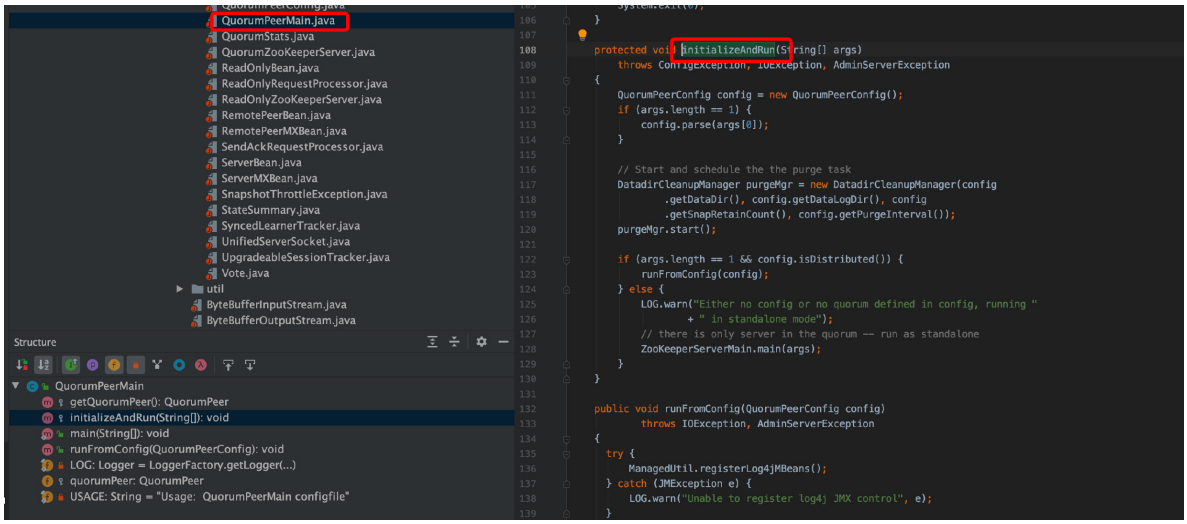
注：本章节主要是参考网上 blog 文章，对部分内容作了调整与处理。

具体细节如下：

1. ZooKeeper 启动类是 QuorumPeerMain，是将配置文件通过参数方式传入。



2. DatadirCleanupManager 线程，由于 ZooKeeper 的任何一个变更操作都产生事务，事务日志需要持久化到硬盘，同时当写操作达到一定量或者一定时间间隔后，会对内存中的数据进行一次快照并写入到硬盘上的 snapshop 中，快照为了缩短启动时加载数据的时间从而加快整个系统启动。而随着运行时间的增长生成的 transaction log 和 snapshot 将越来越多，所以要定期清理，DatadirCleanupManager 就是启动一个 TimeTask 定时任务用于清理 DataDir 中的 snapshot 及对应的 transaction
log。



3. 根据配置中的 servers 数量判断是集群环境还是单机环境，如果单机环境以 standalone 模式运行直接调用 ZooKeeperServerMain.main()方法，否则进入集群模式中。
4. 创建 ServerCnxnFactory 实例，
ServerCnxnFactory 从名字就可以看出其是一个工厂类，负责管理 ServerCnxn，ServerCnxn 这个类代表了一个客户端与一个 server 的连接，每个客户端连接过来都会被封装成一个 ServerCnxn 实例用来维护了服务器与客户端之间的 Socket 通道。
5. QuorumPeer.start()是 ZooKeeper 中非常重要的一个方法入口，

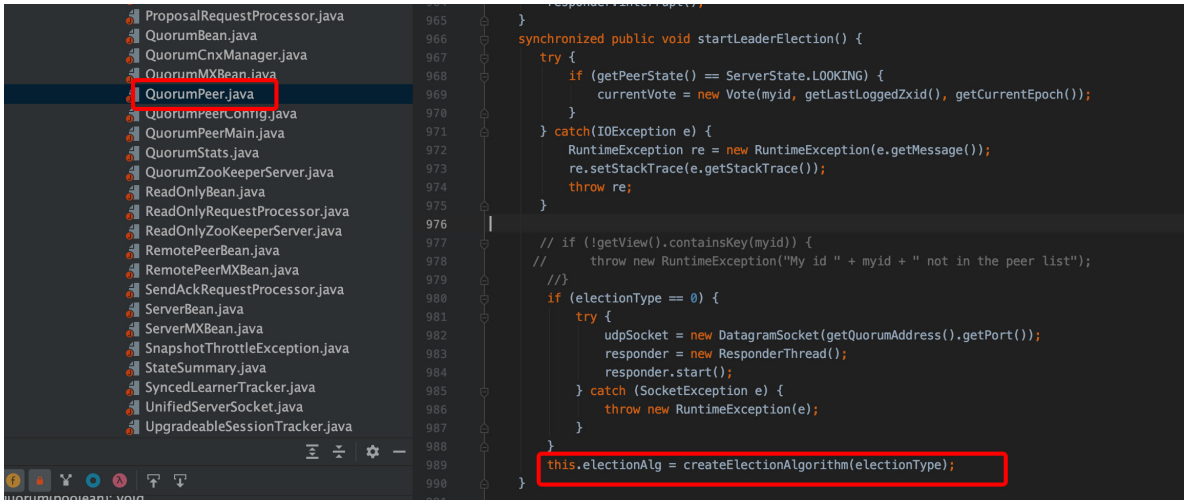


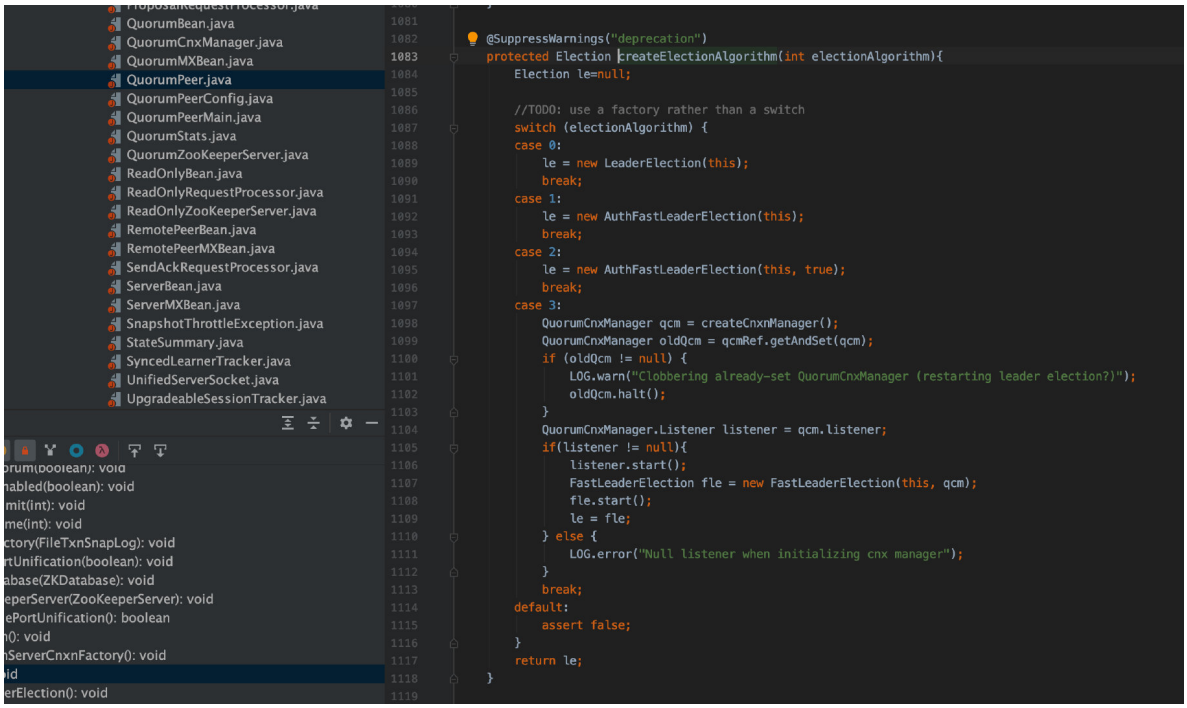
start 方法实现的业务主要包含四个方面：

1.loadDataBase：涉及到的核心类是 ZKDatabase，并借助于 FileTxnSnapLog 工具类将 snap 和 transaction
log 反序列化到内存中，最终构建出内存数据结构 DataTree。

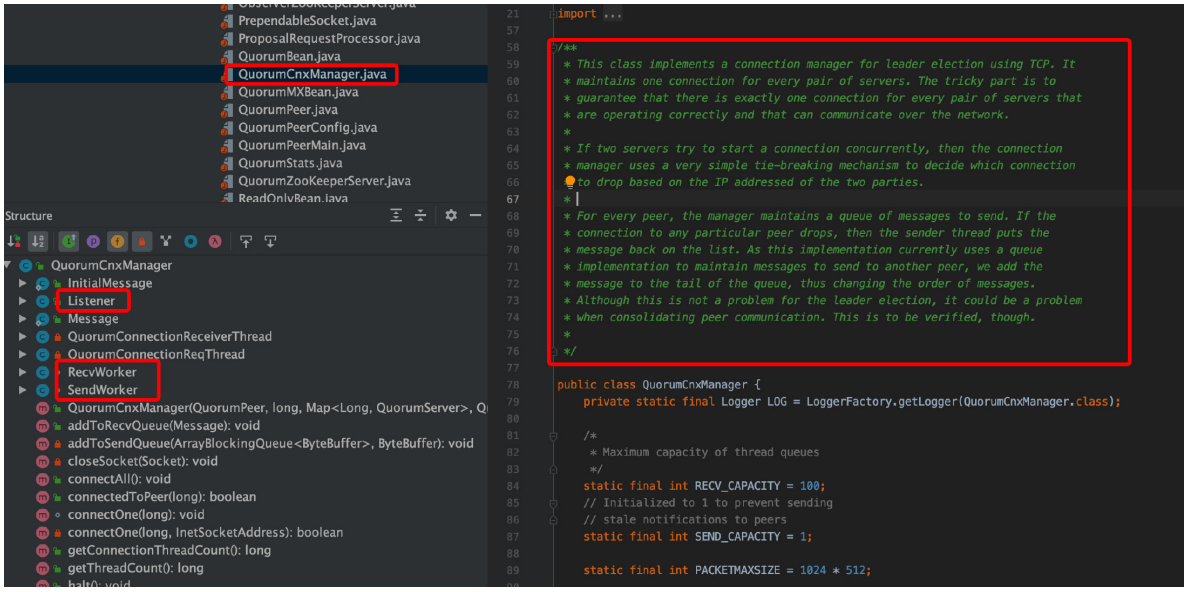
2.cnxnFactory.start：之前介绍过 ServerCnxnFactory 作用，ServerCnxnFactory 本身也可以作为一个线程。

3.startLeaderElection()：这个主要是初始化一些 Leader 选举工作。

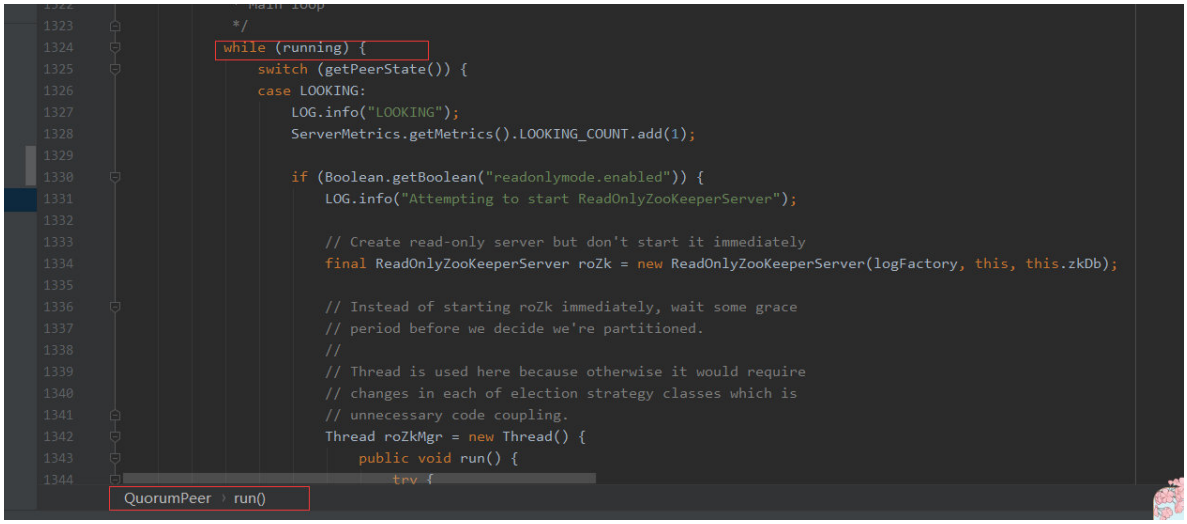




Leader 选举涉及到节点间的网络 IO，QuorumCnxManager 就是负责集群中各节点的网络 IO，QuorumCnxManager 包含一个内部类 Listener，Listener 是一个线程，这里启动 Listener 线程，主要启动选举监听端口并处理连接进来的 Socket；FastLeaderElection 就是封装了具体选举算法的实现。



4.super.start()：QuorumPeer 本身也是一个线程，其继承了 Thread 类，这里就是启动 QuorumPeer 线程，就是执行 QuorumPeer.run 方法。



QuorumPeer 线程进入到一个无限循环模式，不停的通过 getPeerState 方法获取当前节点状态，然后执行相应的分支逻辑。大致流程可以简单描述如下：

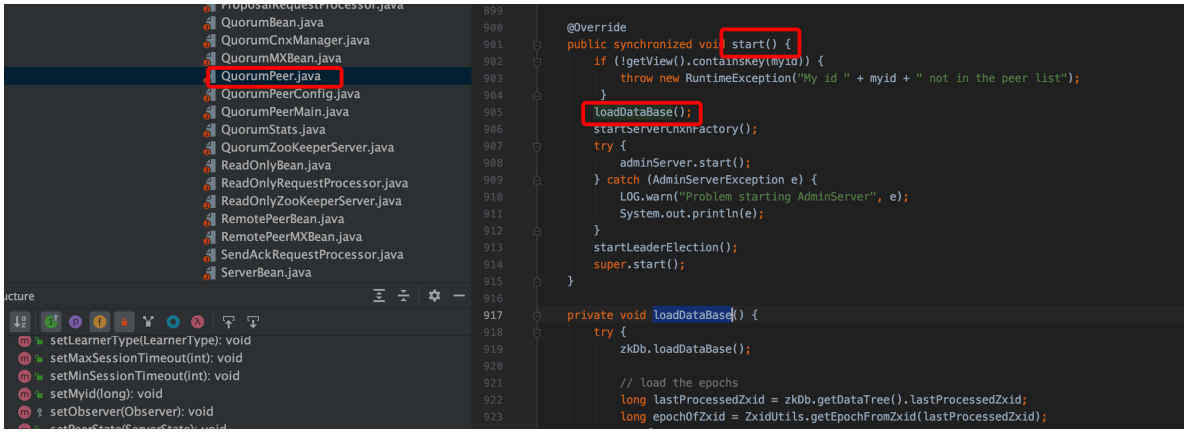
a.首先系统刚启动时 serverState 默认是 LOOKING，表示需要进行 Leader 选举，这时进入 Leader 选举状态中，会调用 FastLeaderElection.lookForLeader 方法，lookForLeader 方法内部也包含了一个循环逻辑，直到选举出 Leader 才会跳出 lookForLeader 方法，如果选举出的 Leader 就是本节点，则将 serverState=LEADING 赋值，否则设置成 FOLLOWING 或 OBSERVING。

b.然后 QuorumPeer.run 进行下一轮次循环，通过 getPeerState 获取当前 serverState 状态，如果是 LEADING，则表示当前节点当选为 LEADER，则进入 Leader 角色分支流程，执行作为一个 Leader 该干的任务；如果是 FOLLOWING 或 OBSERVING，则进入 Follower 或 Observer 角色，并执行其相应的任务。注意：进入分支路程会一直阻塞在其分支中，直到角色转变才会重新进行下一轮次循环，比如 Follower 监控到无法与 Leader 保持通信了，会将 serverState 赋值为 LOOKING，跳出分支并进行下一轮次循环，这时就会进入 LOOKING 分支中重新进行 Leader 选举。

3.1.2 服务器各阶段

3.1.2.1 数据恢复

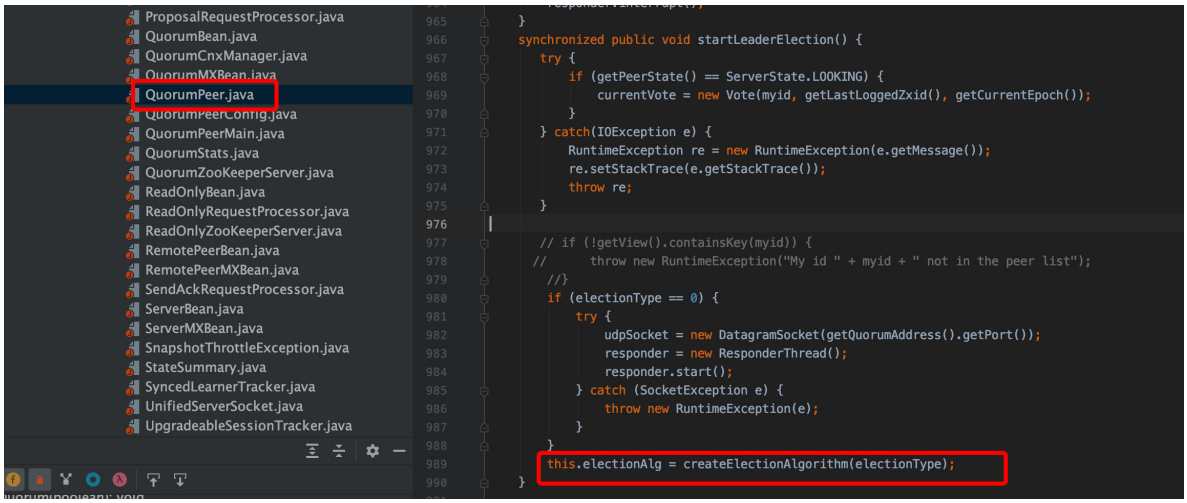
在服务器启动阶段需要进行数据恢复阶段。

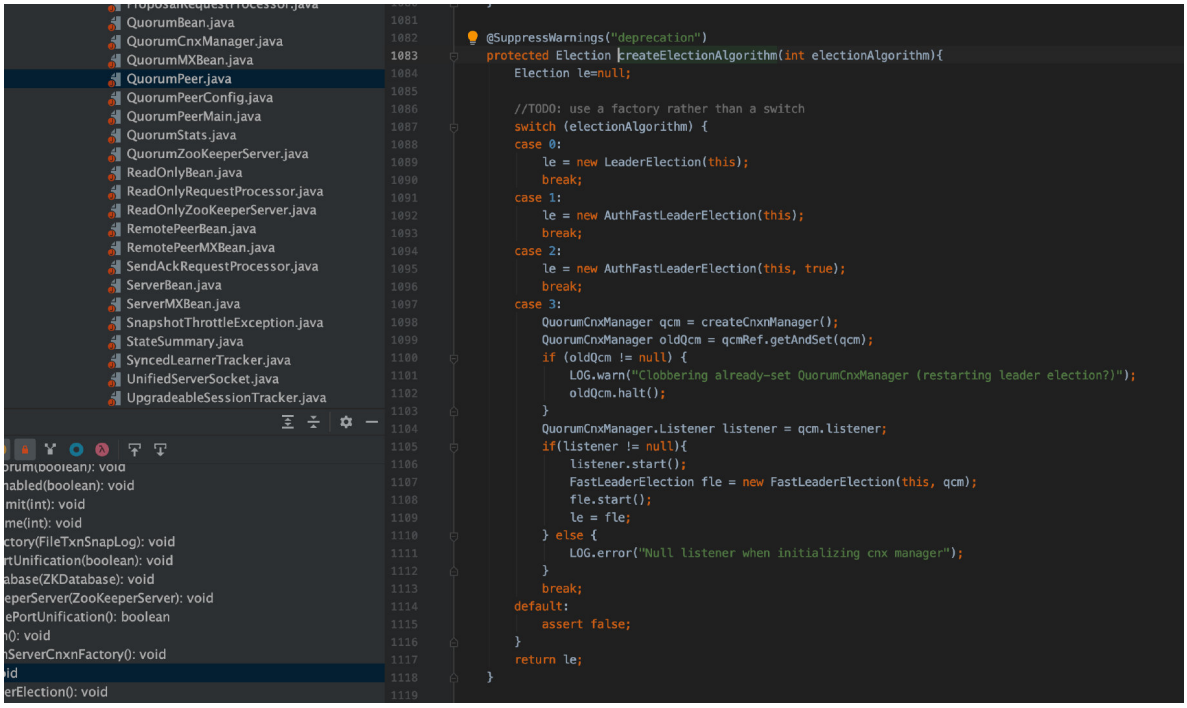


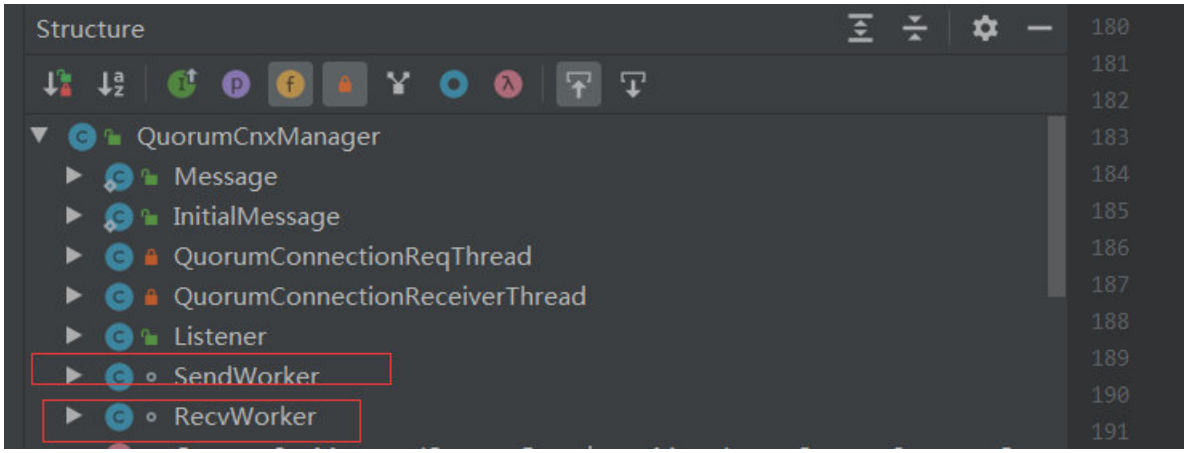
3.1.2.2 Leader 选举

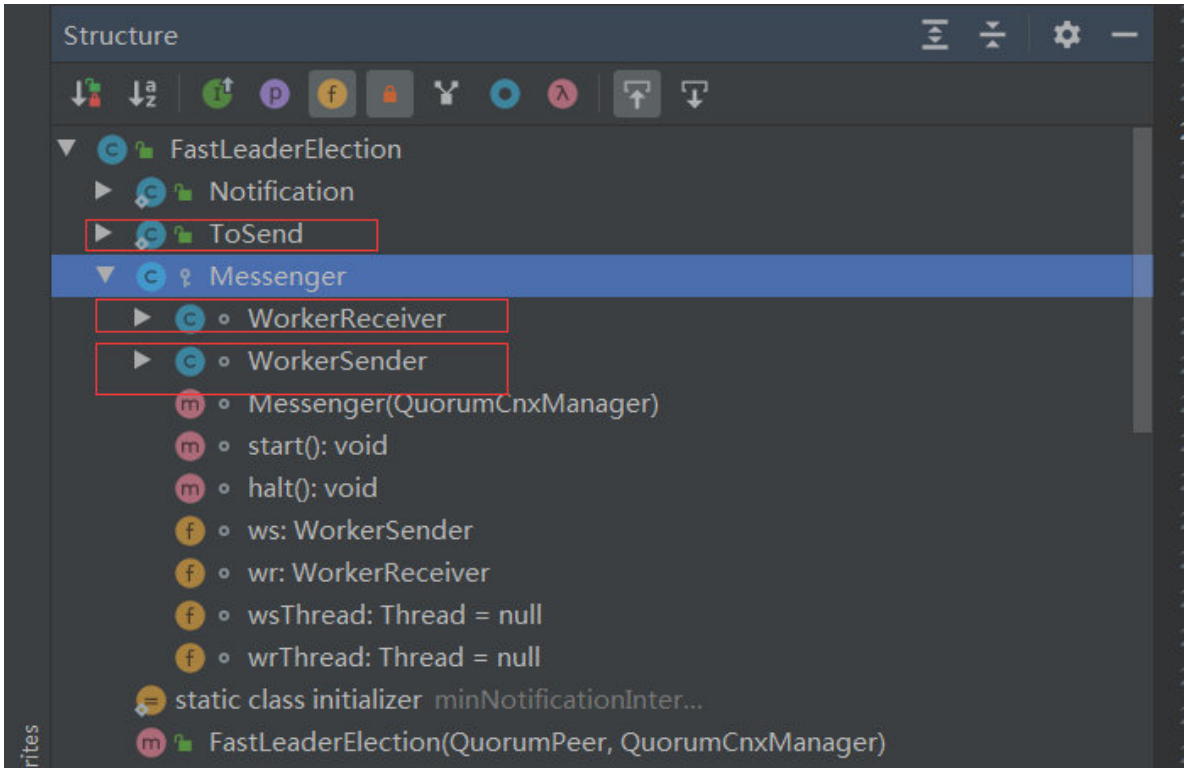
Leader 选举初始化 QuorumPeer.startLeaderElection()，Leader 选举涉及到两个核心类：QuorumCnxManager 和 FastLeaderElection。

Leader 选举期间集群中各节点之间互相进行投票，就会涉及到网络 IO 通信，QuorumCnxManager 就是用来管理维护选举期间网络 IO 通信的工具类。选举算法逻辑被封装在 FastLeaderElection 类。

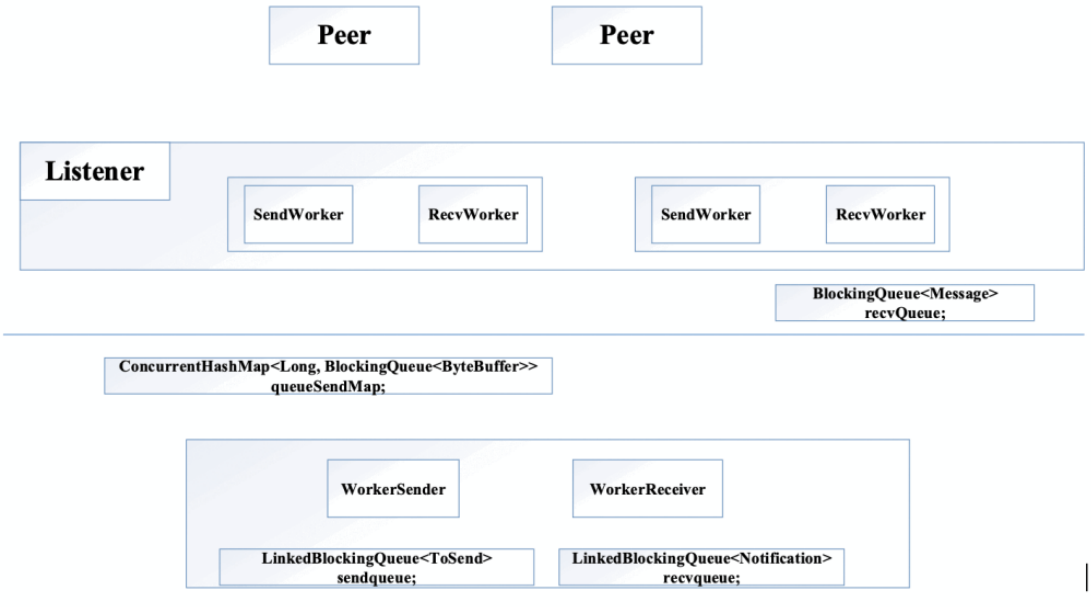








在 createElectionAlgorithm()算法中，创建一个 QuorumCnxManager 实例，启动 QuorumCnxManager.Listener 线程，构建选举算法 FastLeaderElection，然后相互交互投票信息，进入 Leader 选举过程。



QuorumCnxManager 有一个内部类 Listener，初始化一个 ServerSocket，然后在一个 while 循环中调用 accept 接收客户端(注意：这里的客户端指的是集群中其它服务器)连接。当有客户端连接进来后，会将该客户端 Socket 封装成 RecvWorker 和 SendWorker，它们都是线程，分别负责和该 Socket 所代表的客户端进行读写。其中，RecvWorker 和 SendWorker 是成对出现的，每对负责维护和集群中的一台服务器进行网络 IO 通信。

FastLeaderElection 负责 Leader 选举核心规则算法实现，包含了两个内部类 WorkerSender 和 WorkerReceiver 线程。

FastLeaderElection 中进行选举时广播投票信息时，将**投票信息写入到对端服务器**大致流程如下：

a、将数据封装成 ToSend 格式放入到 sendqueue；

b、WorkerSender 线程会一直轮询提取 sendqueue 中的数据，当提取到 ToSend 数据后，会获取到集群中所有参与 Leader 选举节点(除 Observer 节点外的节点)的 sid，如果 sid 即为本机节点，则转成 Notification 直接放入到 recvqueue 中，因为本机不再需要走网络 IO；否则放入到 queueSendMap 中，key 是要发送给哪个服务器节点的 sid，ByteBuffer 即为 ToSend 的内容，queueSendMap 维护的着当前节点要发送的网络数据信息，由于发送到同一个 sid 服务器可能存在多条数据，所以 queueSendMap 的 value 是一个 queue 类型；

c、QuorumCnxManager 中的 SendWorkder 线程不停轮询 queueSendMap 中是否存在自己要发送的数据，每个 SendWorkder 线程都会绑定一个 sid 用于标记该 SendWorkder 线程和哪个对端服务器进行通信，因此，queueSendMap.get(sid)即可获取该线程要发送数据的 queue，然后通过 queue.poll()即可提取该线程要发送的数据内容；

d、然后通过调用 SendWorkder 内部维护的 socket 输出流即可将数据写入到对端服务器。

FastLeaderElection 中进行选举时广播投票信息时，从**对端服务器读取投票信息**的大致流程如下：

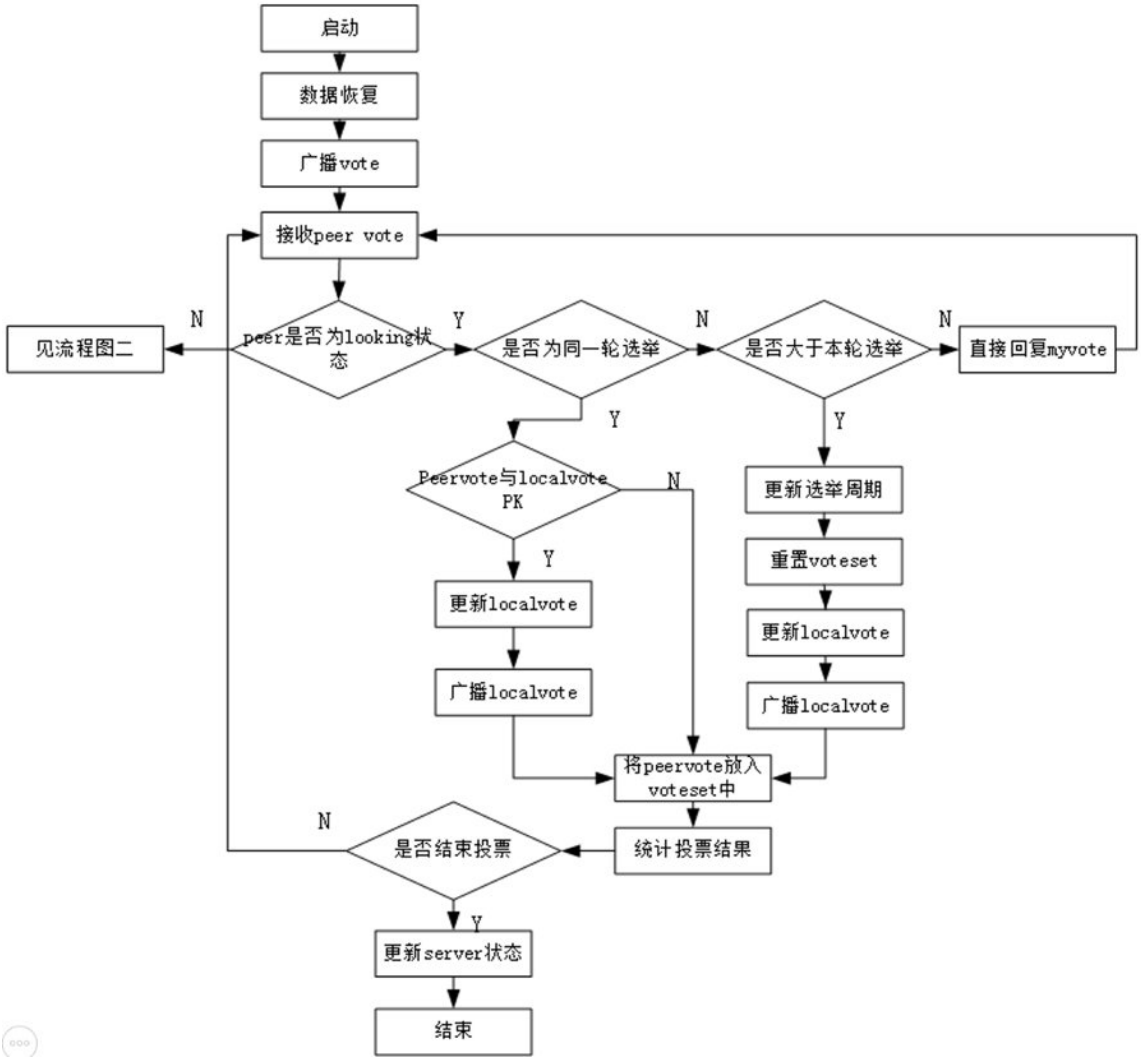
a、QuorumCnxManager 中的 RecvWorker 线程会一直从 Socket 的输入流中读取数据，当读取到对端发送过来的数据时，转成 Message 格式并放入到 recvQueue 中；

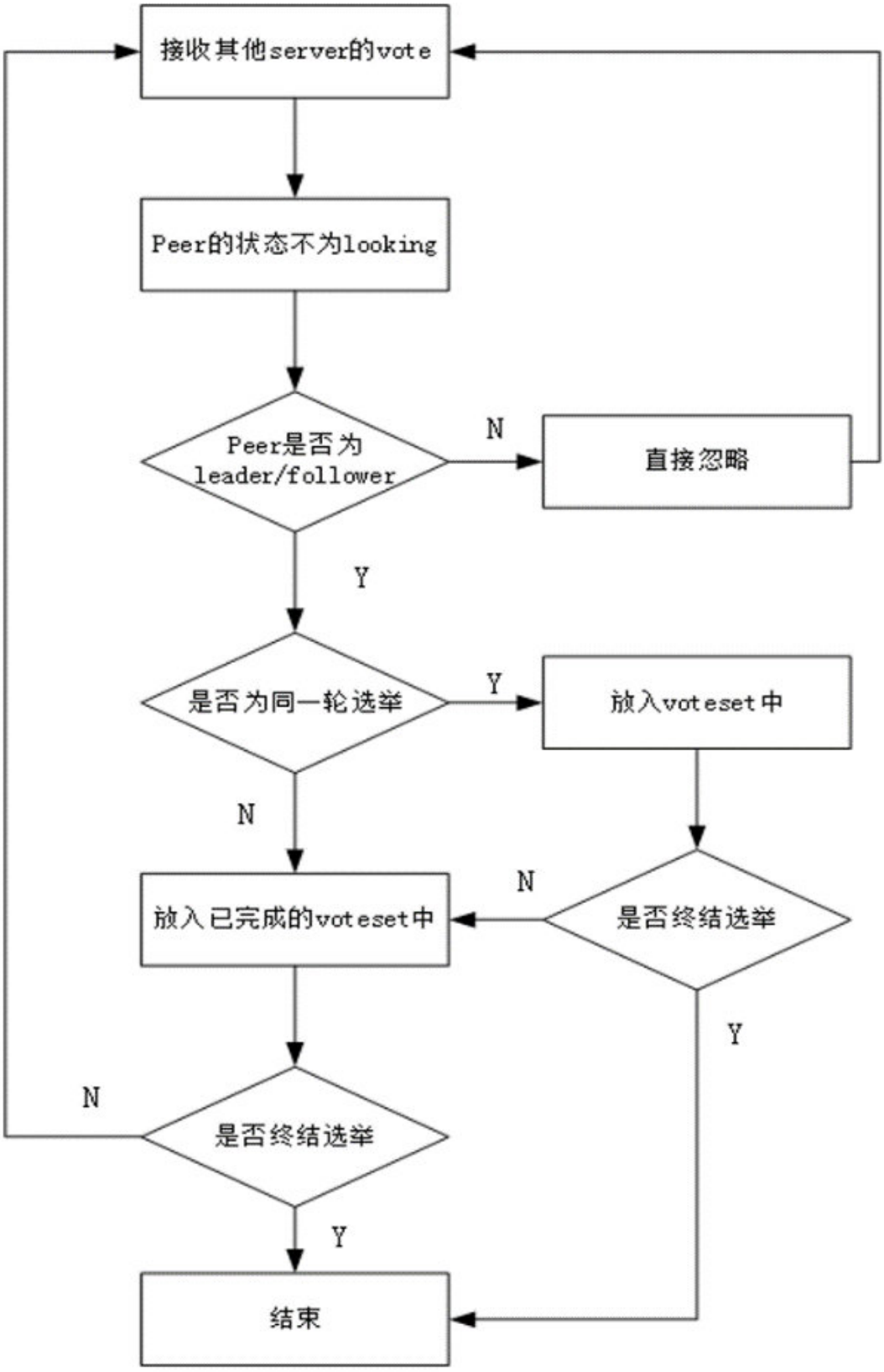
b、FastLeaderElection.WorkerReceiver 线程会轮询方式从 recvQueue 提取数据并转成 Notification 格式放入到 recvqueue 中；

c、FastLeaderElection 从 recvqueu 提取所有的投票信息进行比较 最终选出一个 Leader。

Leader 选举算法实现

上面已经介绍了 Leader 选举期间网络 IO 的大致流程，下面介绍下具体选举算法如何实现。


启动
数据恢复
广播vote
接收peer vote
peer是否为looking状态
N
Y
见流程图二
是否为同一轮选举
N
Y
是否大于本轮选举
N
Y
直接回复myvote
Peervote与localvote PK
N
Y
更新选举周期
重置voteset
更新localvote
广播localvote
更新localvote
广播localvote
将peervote放入voteset中
统计投票结果
是否结束投票
N
Y
更新server状态
结束



QuorumPeer 线程中会有一个 Loop 循环，获取 serverState 状态后进入不同分支，当分支退出后继续下次循环，FastLeaderElection 选举策略调用就是发生在检测到 serverState 状态为 LOOKING 时进入到 LOOKING 分支中调用的。

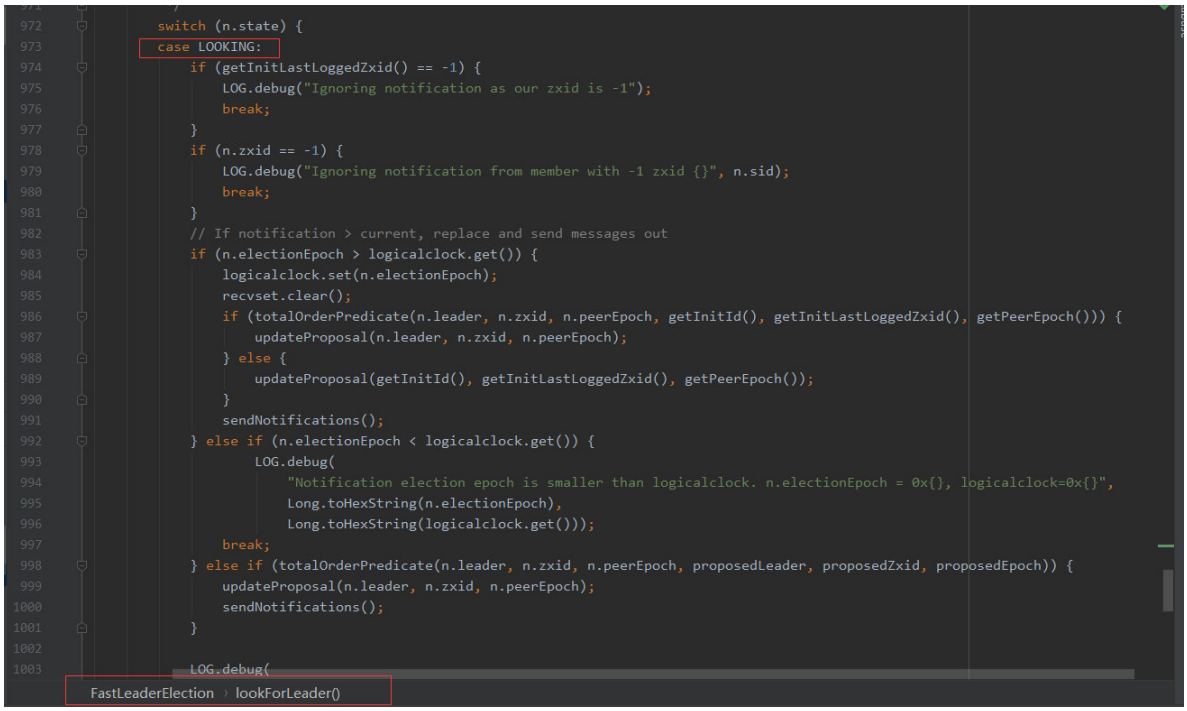
进入到 LOOKING 分支执行的代码逻辑：

setCurrentVote(makeLEStrategy().lookForLeader());

从上面代码可以看出，Leader 选举策略入口方法为：FastLeaderElection.lookForLeader()方法。当 QuorumPeer.serverState 变成 LOOKING 时，该方法会被调用，表示执行新一轮 Leader 选举。下面来看下 lookForLeader 方法的大致实现逻辑：

1.更新自己期望投票信息，即自己期望选哪个服务器作为 Leader(用 sid 代替期望服务器节点)以及该服务器 zxid、epoch 等信息，第一次投票默认都是投自己当选 Leader，然后调用 sendNotifications 方法广播该投票到集群中所有可以参与投票服务器，广播涉及到网络 IO 流程前面已讲解，这里就不再细说；

其中，updateProposal()方法有三个参数：a.期望投票给哪个服务器(sid)、b.该服务器的 zxid、c.该服务器的 epoch，在后面会看到这三个参数是选举 Leader 时的核心指标，后面再介绍。



首先对之前提到的选举轮次 electionEpoch 进行判断，这里分为三种情况：

a.只有对方发过来的投票的 electionEpoch 和当前节点相等表示是同一轮投票，即投票有效，然后调用 totalOrderPredicate()对投票进行 PK，返回 true 代表对端胜出，则表示第一次投票是错误的(第一次都是投给自己)，更新自己投票期望对端为 Leader，然后调用 sendNotifications()将自己最新的投票广播出去。返回 false 则代表自己胜出，第一次投票没有问题，就不用管

b.如果对端发过来的 electionEpoch 大于自己，则表明重置自己的 electionEpoch，然后清空之前获取到的所有投票 recvset，因为之前获取的投票轮次落后于当前则代表之前的投票已经无效了，然后调用 totalOrderPredicate()将当前期望的投票和对端投票进行 PK，用胜出者更新当前期望投票，然后调用 sendNotifications()将自己期望头破广播出去。注意：这里不管哪一方胜出，都需要广播出去，而不是步骤 a 中己方胜出不需要广播，这是因为由于 electionEpoch 落后导致之前发出的所有投票都是无效的，所以这里需要重新发送

c.如果对端发过来的 electionEpoch 小于自己，则表示对方投票无效，直接忽略不进行处理

totalOrderPredicate()实现了对投票进行 PK 规则：

```
707         */
708        protected boolean totalOrderPredicate(long newId, long newZxid, long newEpoch, long curId, long curZxid, long curEpoch) {
709            LOG.debug(
710                "id: {}, proposed id: {}, zxid: 0x{}, proposed zxid: 0x{}",
711                newId,
712                curId,
713                Long.toHexString(newZxid),
714                Long.toHexString(curZxid));
715
716            if (self.getQuorumVerifier().getWeight(newId) == 0) {
717                return false;
718            }
719
720            /*
721             * We return true if one of the following three cases hold:
722             * 1- New epoch is higher
723             * 2- New epoch is the same as current epoch, but new zxid is higher
724             * 3- New epoch is the same as current epoch, new zxid is the same
725             *  as current zxid, but server id is higher.
726             */
727
728            return ((newEpoch > curEpoch)
729                    || ((newEpoch == curEpoch)
730                        && ((newZxid > curZxid)
731                            || ((newZxid == curZxid)
732                                && (newId > curId)))));
733        }
734
735        /**
736         * Given a set of votes, return the SyncedLearnerTracker which is used to
```

FastLeaderElection › totalOrderPredicate()

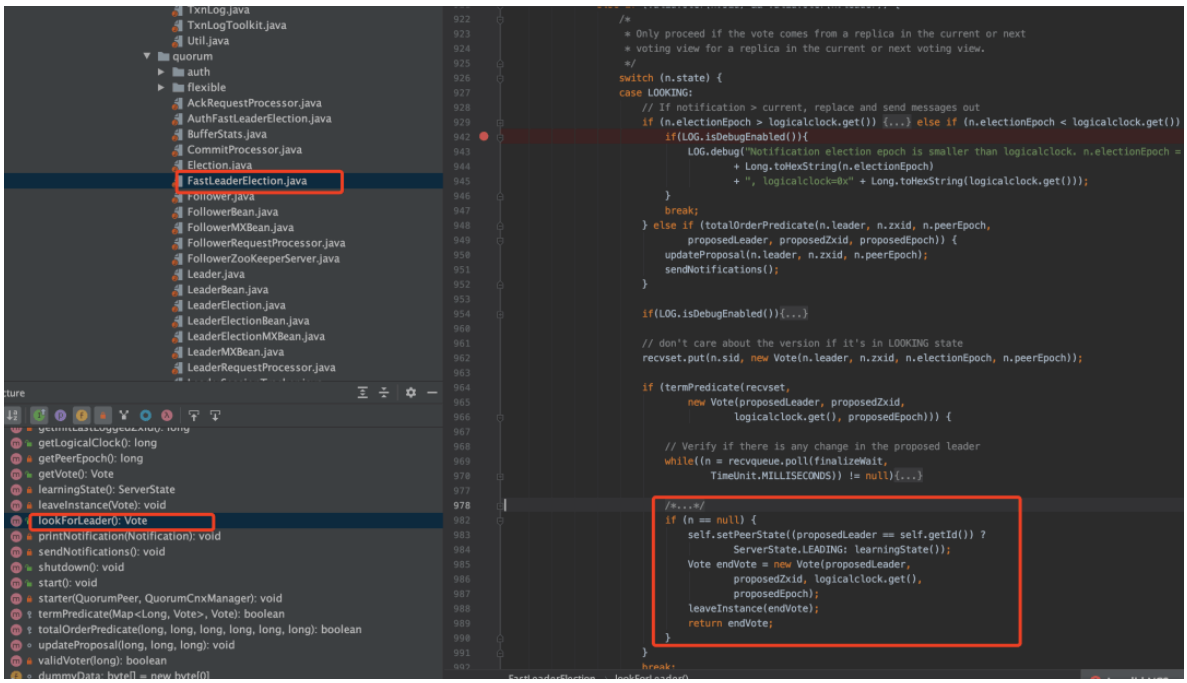
下面简单说下这个 PK 逻辑原理(胜出一方代表更有希望成为 Leader):

1、首先比较 epoch，哪个 epoch 哪个胜出，前面介绍过 epoch 代表了 Leader 的轮次，是一个递增的，epoch 越大就意味着数据越新，Leader 数据越新则可以减少后续数据同步的效率，当然应该优先选为 Leader；

2、然后才是比较 zxid，由于 zxid=epoch+counter，第一步已经把 epoch 比较过了，其实这步骤只是相当于比较 counter 大小，counter 越大则代表数据越新，优先选为 Leader。注：其实第 1 和第 2 可以合并到一起，直接比较 zxid 即可，因为 zxid=epoch+counter，第 1 比较显的有些多余

3、如果前两个指标都没法比较出来，只能通过 sid 来确定，zxid 相等说明两个服务器的数据是一致的，所以选哪个当 Leader 其实没有区别，这里就选择 sid 大的当 Leader .

下面来看下 LOOKING 分支的最后一部分逻辑:



3.1.2.3 集群数据同步

Leader 选举的流程，ZooKeeper 集群在 Leader 选举完成后，集群中的各个节点就确定了自己的角色信息：Leader、Follower 或 Observer。

```
                    break;
                case LEADING:
                    LOG.info("LEADING");
                    try {
                        setLeader(makeLeader(logFactory));
                        leader.lead();
                        setLeader(null);
                    } catch (Exception e) {
                        LOG.warn("Unexpected exception", e);
                    } finally {
                        if (leader != null) {
                            leader.shutdown("Forcing shutdown");
                            setLeader(null);
                        }
                        updateServerState();
                    }
                    break;
                }
            }
        } finally {
            LOG.warn("QuorumPeer main thread exited");
            MBeanRegistry instance = MBeanRegistry.getInstance();
            instance.unregister(jmxQuorumBean);
```

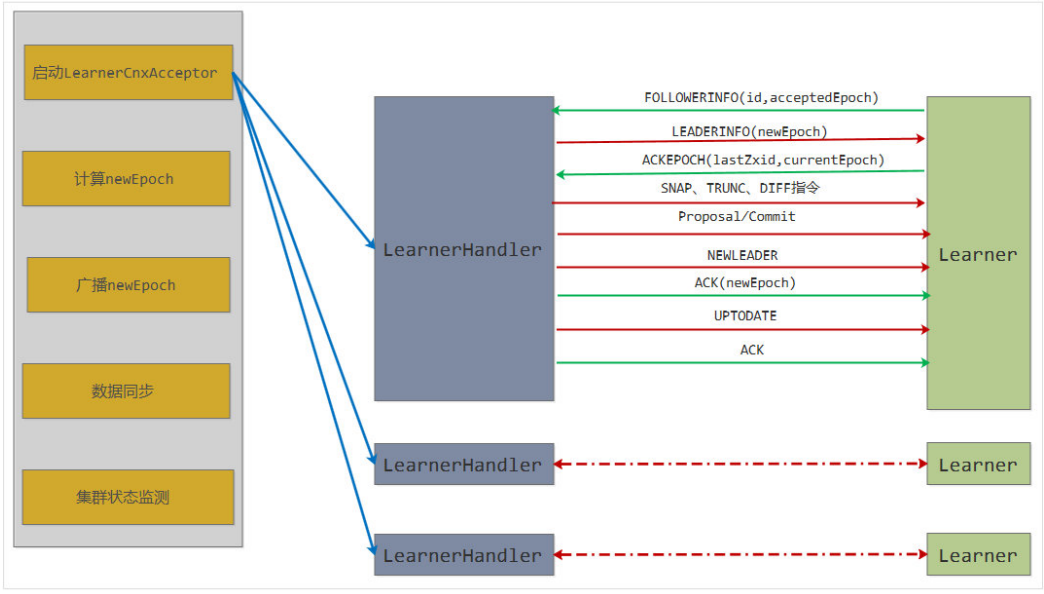
QuorumPeer › run()

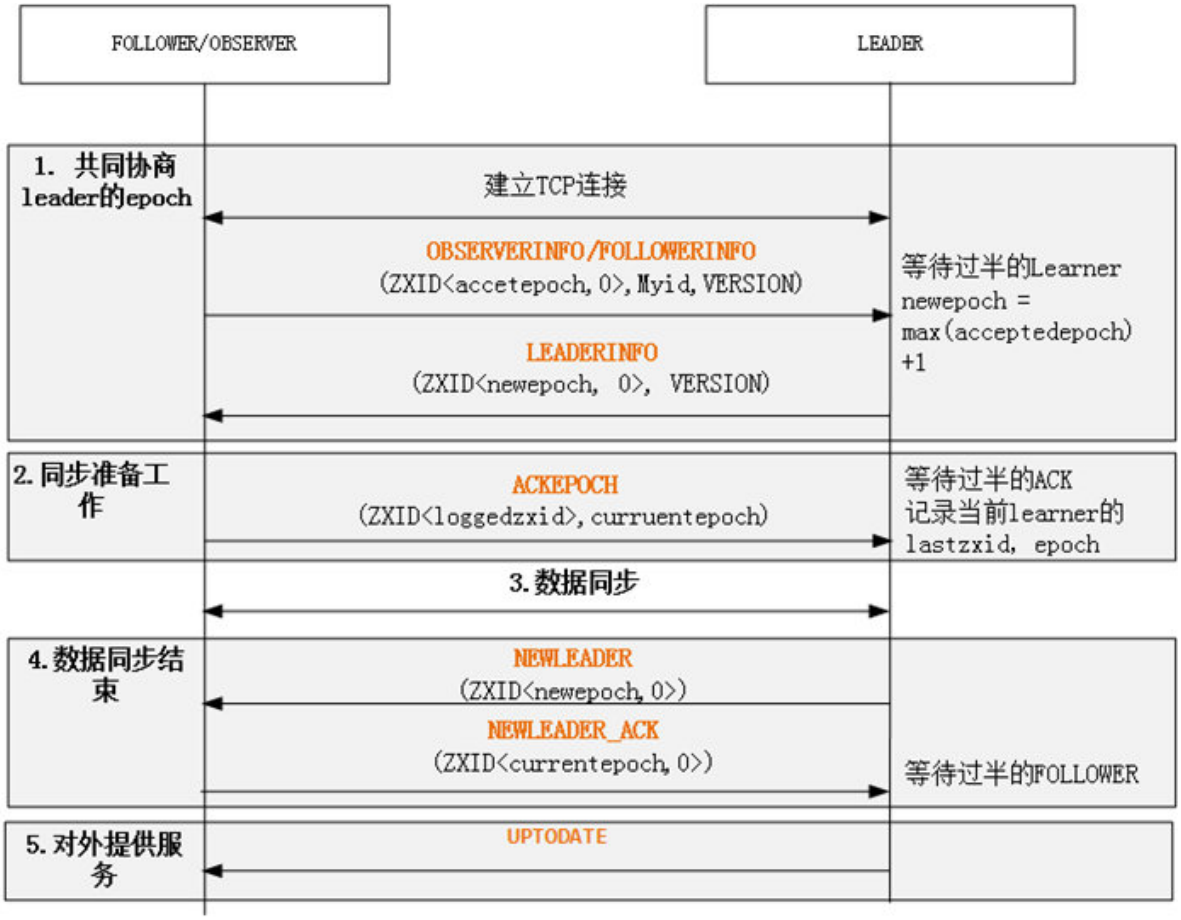
如上述代码所述，节点确定了自己的角色后，就会进入自己的角色分支：对于 Leader 而言创建 Leader 实例并调用其 lead()函数，对于 Follower 而言创建 Follower 实例并调用其 followLeader()函数，对于 Observer 而言创建 Observer 实例并调用其 observeLeader()函数。在这三个函数中，服务器会进行相关的初始化并完成最终的启动。

对于 Follower 和 Observer 而言，主要的初始化工作是要建立与 Leader 的连接并同步 epoch 信息，最后完成与 Leader 的数据同步。而 Leader 会启动 LearnerCnxAcceptor 线程，该线程会接受来自 Follower 和 Observer（统称为 Learner）的连接请求并为每个连接创建一个 LearnerHandler 线程，该线程会负责包括数据同步在内的与 learner 的一切通信。

Learn(Follower 或 Observer)节点会主动向 Leader 发起连接，ZooKeeper 就会进入集群同步阶段，集群同步主要完成集群中各节点状态信息和数据信息的一致。选出新的 Leader 后的流程大致分为：计算 epoch、统一 epoch、同步数据、广播模式等四个阶段。其中其前三个阶段：计算 epoch、统一 epoch、同步数据就是这一节主要介绍的集群同步阶段的主要内容，这三个阶段主要完成新 Leader 与集群中的节点完成同步工作，处于这个阶段的 zk 集群还没有真正做好对外提供服务的能力，可以看着是新 leader 上任后进行的内部沟通、前期准备工作等，只有等这三个阶段全部完成，新 leader 才会真正的成为 leader，这时 zk 集群会恢复正常可运行状态并对外提供服务。

被选举为 Leader 角色的节点，会创建一个 Leader 实例，然后执行 Leader.lead()进入到 Leader 角色的任务分支中，其流程大致如下所示：





Leader 分支大致可以分为 5 个阶段：启动 LearnerCnxAcceptor 线程、计算 newEpoch、广播 newEpoch、数据同步和集群状态监测。

Leader.lead()方法控制着 Leader 角色节点的主体流程，其实现较为简单，大致模式都是通过阻塞方法阻塞当前线程，直到该阶段完成 Leader 线程才会被唤醒继续执行下一个阶段；而每个阶段实现的具体细节及大量的网络 IO 操作等都在 LearnerHandler 中实现。比如计算 newEpoch，Leader 中只会判断 newEpoch 计算完成没，没有计算完成就会进入阻塞状态挂起当前 Leader 线程，直到集群中一半以上的节点同步了 epoch 信息后 newEpoch 正式产生才会唤醒 Leader 线程继续向下执行；而计算 newEpoch 会涉及到 Leader 去收集集群中大部分 Learner 服务器的 epoch 信息，会涉及到大量的网络 IO 通信等内容，这些细节部分都在 LearnerHandler 中实现。

涉及到网络 IO 就会存在 Server 和 Client，这里的 Server 就是 Leader，Client 就是 Learner(Follower 和 Observer 统称 Learner)，对于 Server 端，主要关注 Leader 和 LearnerHandler 这两个类，而对于 Client 端，根据角色分类主要关注 Follower 或 Observer 这两个类。
ZooKeeper 中主要存在三个端口：

1、客户端请求端口：对应于配置中的 clientPort，默认是 2181，就是客户端连接 ZK 对其进行增删改操作的端口；

2、集群选举端口：之前分析过的集群中 Leader 选举涉及到网络 IO 使用的端口，对应于配置中“server.0=10.80.8.3:2888:2999”这里的 2999 就是集群选举端口；

3、集群同步端口：Leader 选举出后就会涉及到 Leader 和 Learner 之间的数据同步问题，集群同步端口的作用就是做这个使用的，对应于配置中”server.0=10.80.8.3:2888:2999“这里的 2888；

启动 LearnerCnxAcceptor 线程

Leader 首先会启动一个 LearnerCnxAcceptor 线程，这个线程做的工作就非常简单了，就是不停的循环 accept 接收 Learner 端的网络请求(这里的监听端口就是上面说的同步监听端口，而不是选举端口)，Leader 选举结束后被分配为 Follower 或 Observer 角色的节点会主动向 Leader 发起连接，Leader 端接收到一个网络连接就会封装成一个 LearnerHandler 线程。

Leader 类可以看成一个总管，和每个 Learner 服务器的交互任务都会被分派给 LearnerHandler 这个助手完成，当 Leader 检测到一个任务被一半以上的 LearnerHandler 处理完成，即认为该阶段结束，进入下一个阶段。

计算 epoch

epoch 在 ZooKeeper 中是一个很重要的概念，前面也介绍过了：epoch 就相当于 Leader 的身份编号，就如同身份证编号一样，每次选举产生一个新 Leader 时，都会为该 Leader 重新计算出一个新 epoch。epoch 被设计成一个递增值，比如上一个 Leader 的 epoch 是 1，假如重新选举新的 Leader 就会被分配 epoch=1。

epoch 作用：可以防止旧 Leader 活过来后继续广播之前旧提议造成状态不一致问题，只有当前 Leader 的提议才会被 Follower 处理。ZooKeeper 集群所有的事务请求操作都要提交由 Leader 服务器完成，Leader 服务器将事务请求转成一个提议(Proposal)并分配一个事务 ID(zxid)后广播给 Learner，zxid 就是由 epoch 和 counter(递增)组成，当存在旧 leader 向 follower 发送命令的时候，follower 发现 zxid 所在的 epoch 比当前的小，则直接拒绝，防止出现不一致性。

统一 epoch

newEpoch 计算完成后，该值只有 Leader 知道，现在需要将 newEpoch 广播到集群中所有的服务器节点上，让他们都更新下新 Leader 的 epoch 信息，这样他们在处理请求时会根据 epoch 判断该请求是不是当前新 Leader 发出的，可以防止旧 Leader 活过来后继续广播之前旧提议造成状态不一致问题，只有当前 Leader 的提议才会被 Follower 处理。

总结：广播 newEpoch 流程也比较简单，就是将之前计算出来的 newEpoch 封装到 LEADERINFO 数据包中，然后广播到集群中的所有节点，同时会收到 ACKEPOCH 回复数据包，当集群中一半以上的节点进行了回复则可以认为 newEpoch 广播完成，则进入下一阶段。同样，为避免线程一直阻塞，休眠线程依然会被添加超时时间，超时后仍未完成则抛出 InterruptedException 异常重新进入 Leader 选举状态。

数据同步

之前分析过 Leader 的选举策略：lastZxid 越大越会被优先选为 Leader。lastZxid 是节点上最大的事务 ID，由于 zxid 是递增的，lastZxid 越大，则表示该节点处理的数据越新，即数据越完整。所以，被选为 Leader 的节点数据完整性越高，为了数据一致性，这时就需要其它节点和 Leader 进行数据同步保持数据一致性。

数据同步四种情况：

DIFF，learner 比 leader 少一些数据；

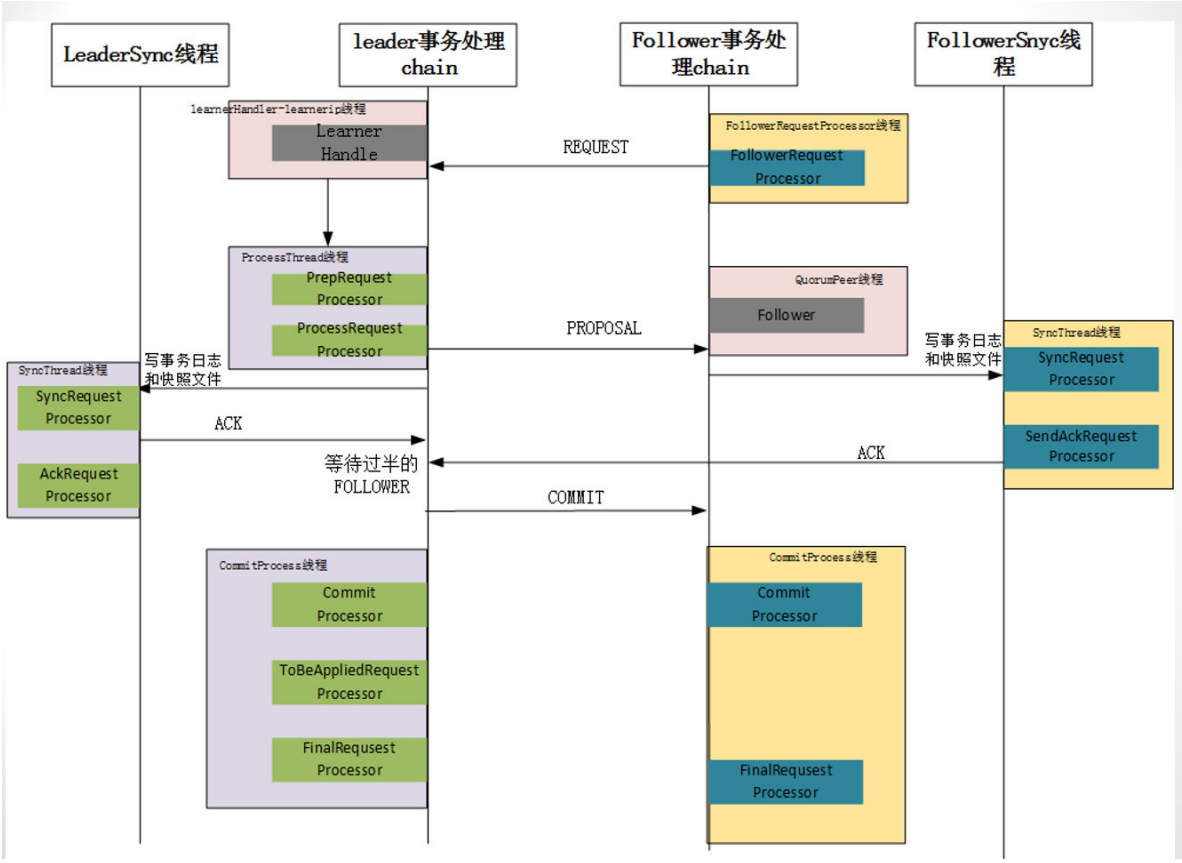
TRUNC，learner 数据比 leader 多；

DIFF+TRUNC，learner 对 leader 多数据又少数据；

SNAP，learner 比 leader 少很多数据。

3.1.2.4 服务角色

群首，追随者，观察者根本上都是服务器，在实现服务器主要抽象概念是请求处理器。请求处理器是对处理流水线上不同阶段的抽象，每个服务器在初始化时实现一个请求处理器的序列。对于请求处理器，ZooKeeper 代码里有一个叫 RequestProcessor 的接口，这个接口的主要方法是**processRequest**，它接受一个 Request 参数，在一个请求处理器的流水线中，对于相邻处理器的请求的处理是通过队列实现解耦合。当一个处理器有一条请求需要下一个处理器进行处理时，它将这条请求加入队列中。然后，它将处于等待状态直到下一个处理器处理完此消息。本节主要看看各个服务器的请求处理器序列初始化和对队列的使用与处理，处理器的细节可以参考源码。

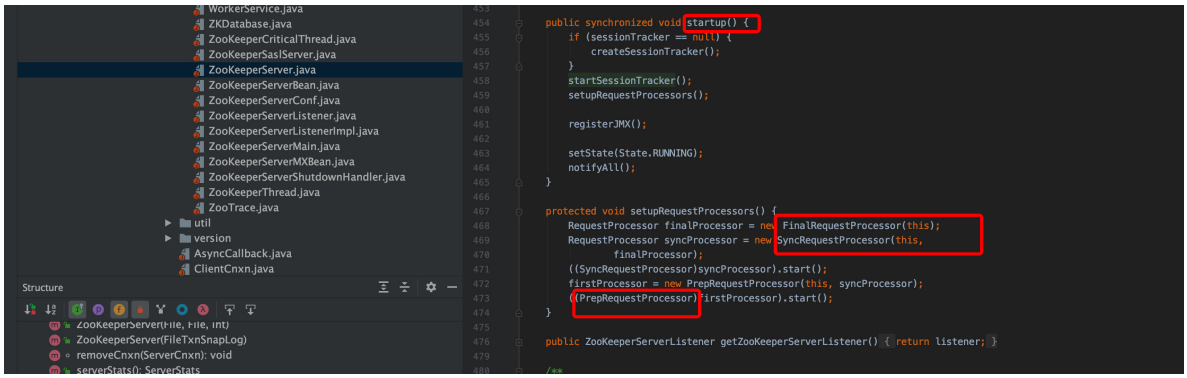


1. **独立服务器**

独立服务器请求链



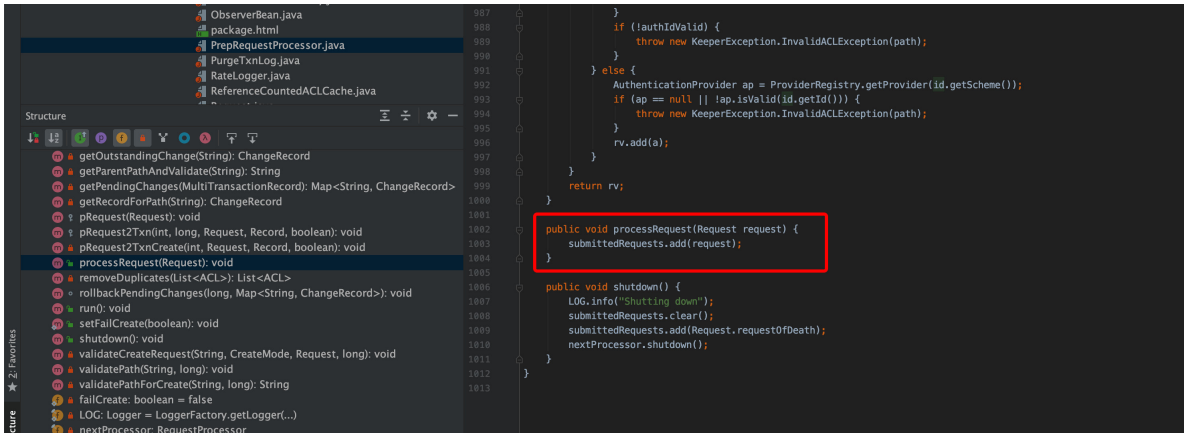
独立服务器是从 ZooKeeperServerMain.java 开始，

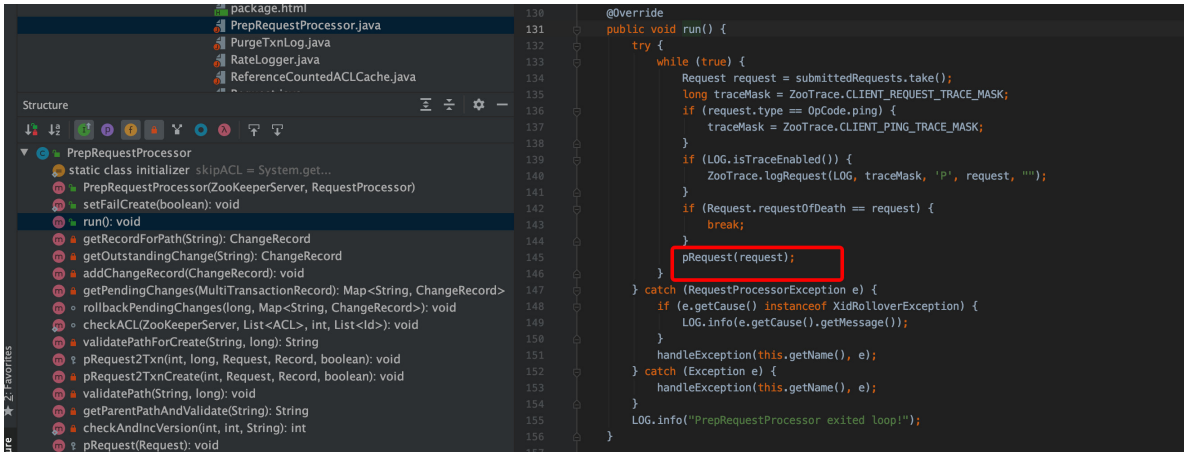


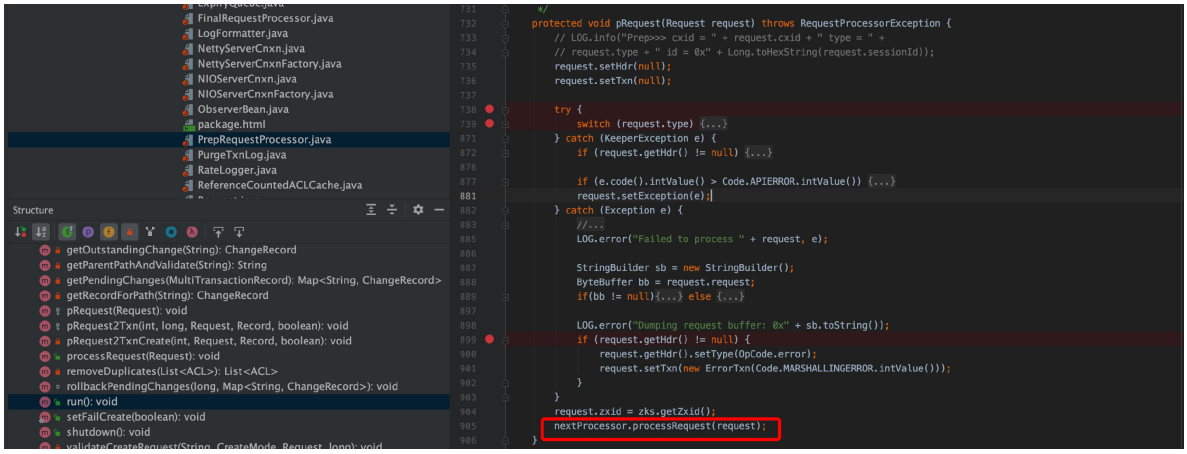
在 PrepRequestProcessor 中，消费请求队列 submittedRequests，数据结构如下

LinkedBlockingQueue<Request> submittedRequests = new LinkedBlockingQueue<Request>();

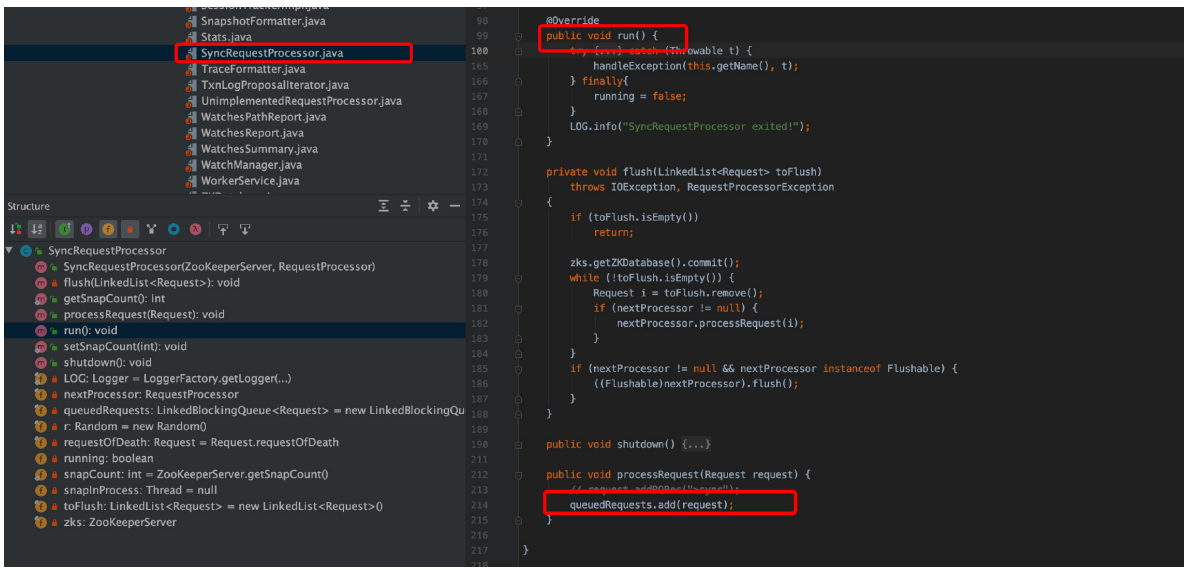
PrepRequestProcessor 接受客户端的请求并执行这个请求，处理结果则是生成一个事务。不过只有改变 ZooKeeper 状态的操作才会产生事务，对于读操作并不会产生任何事务。

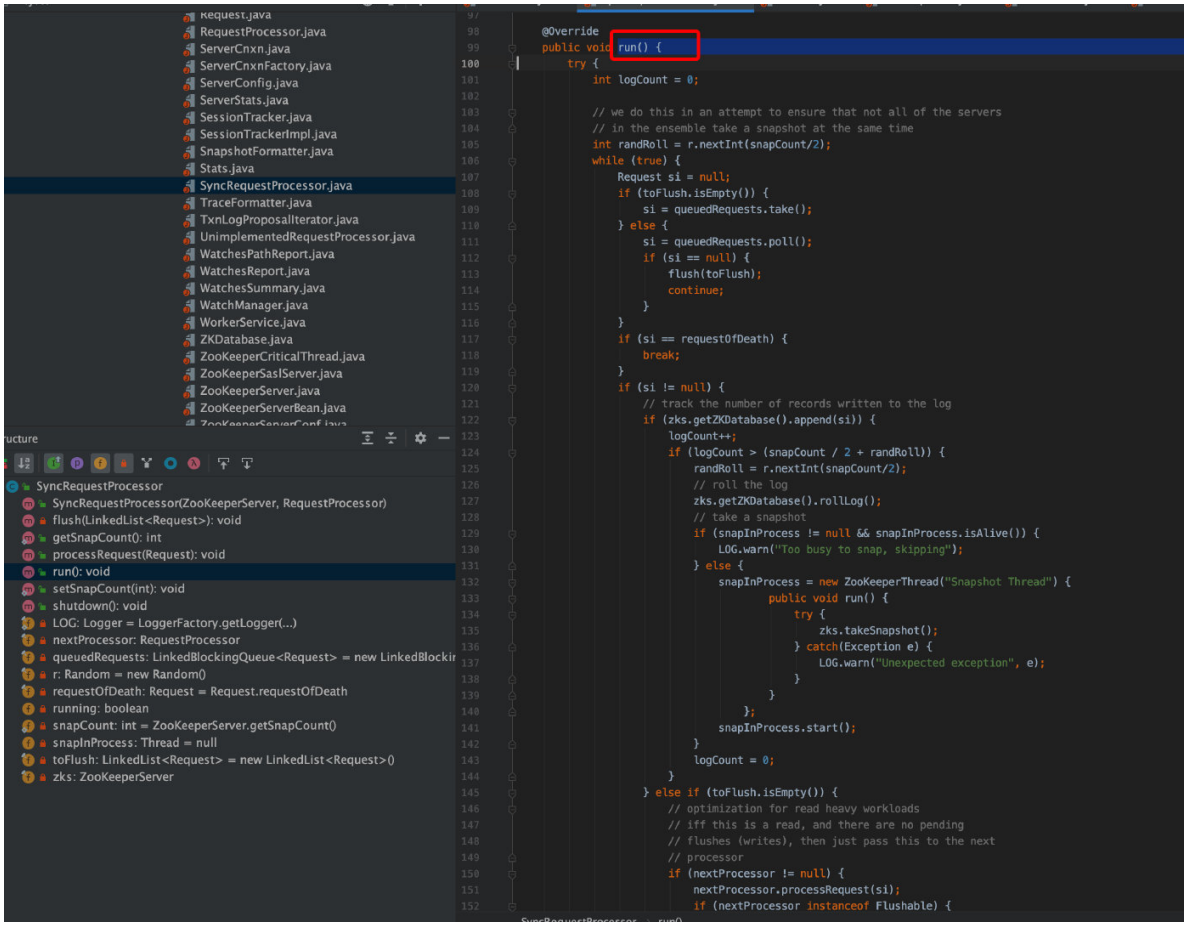




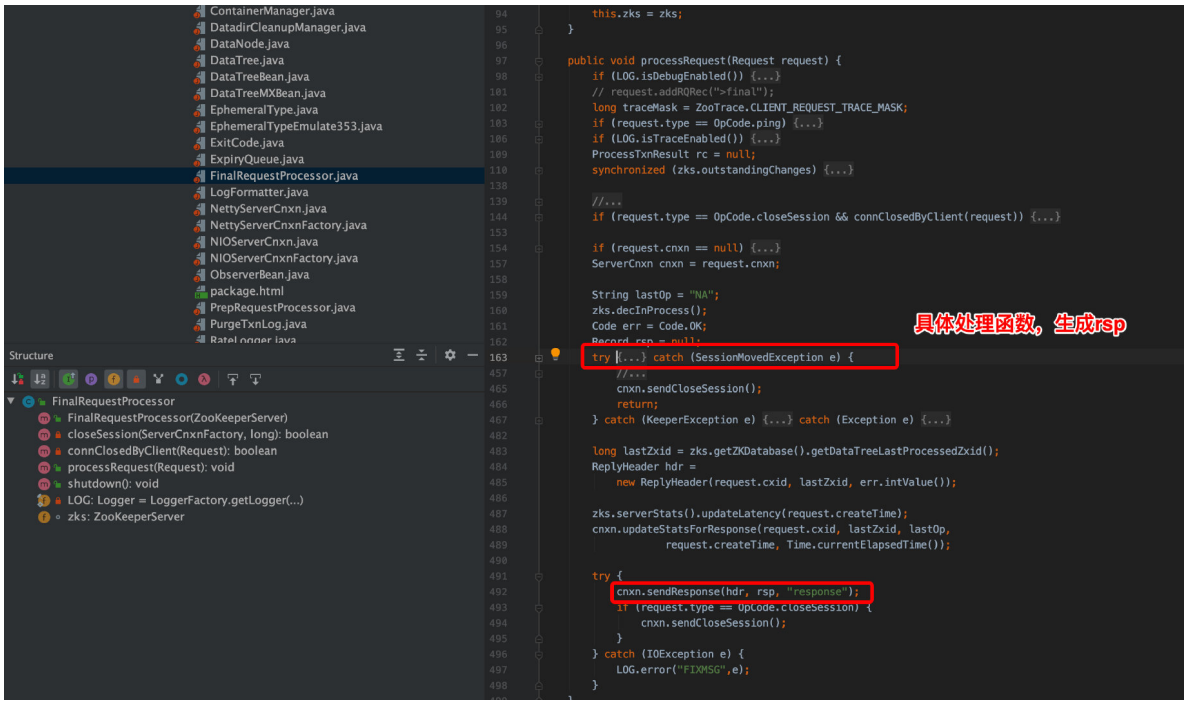


SyncRequestProcessor.java，SyncRequestProcessor 负责将事务持久化到磁盘上。实际上就是将事务数据按照顺序追加到事务日志中，并形成快照数据。



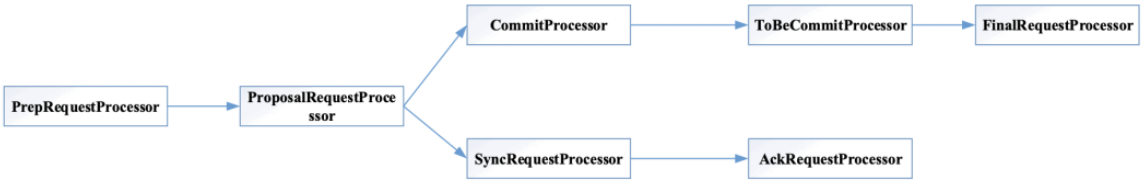


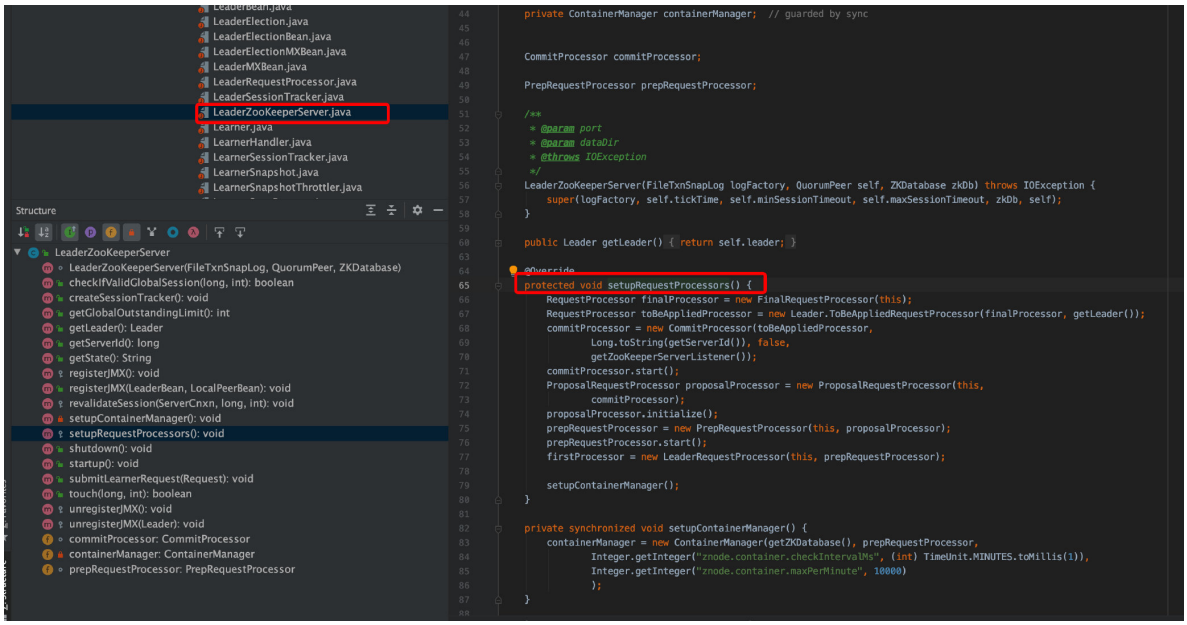
FinalRequestProcessor.java,FinalRequestProcessor，如果 Request 对象包含事务数据，该处理器就会接受对 ZooKeeper 数据树的修改，否则，该处理器会从数据树中读取数据并返回客户端。

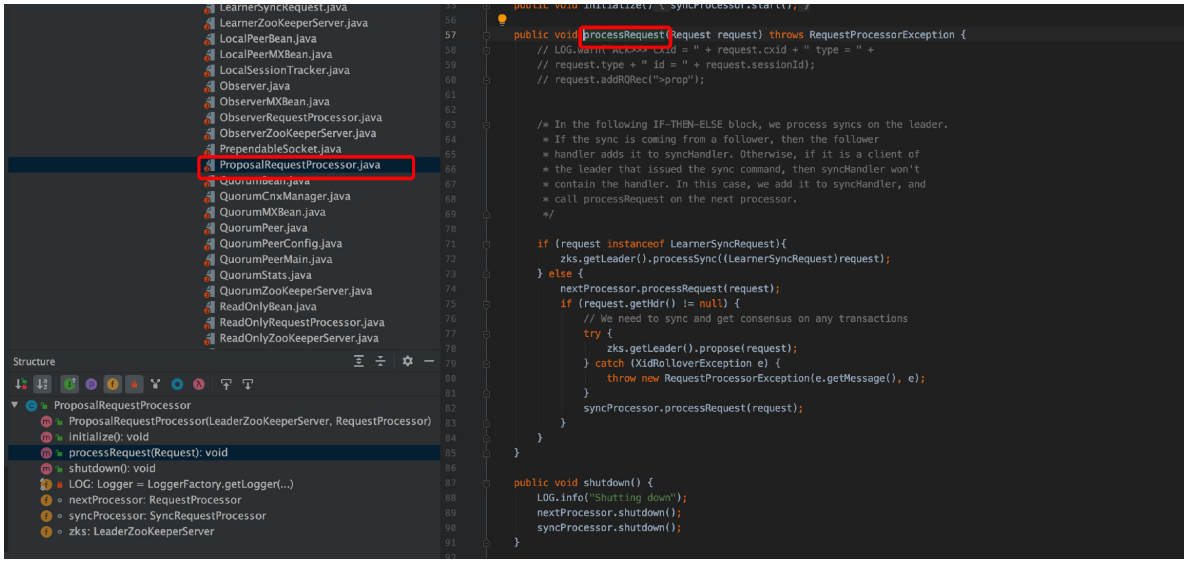


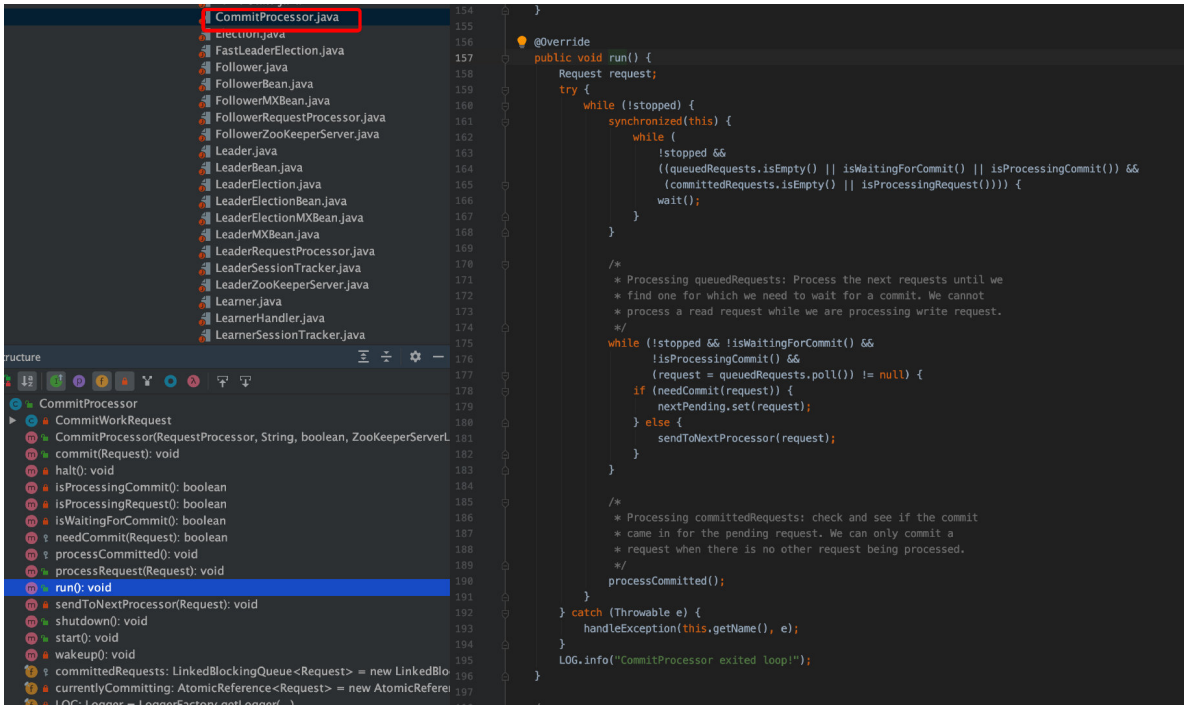
群首服务器（Leader）

请求链

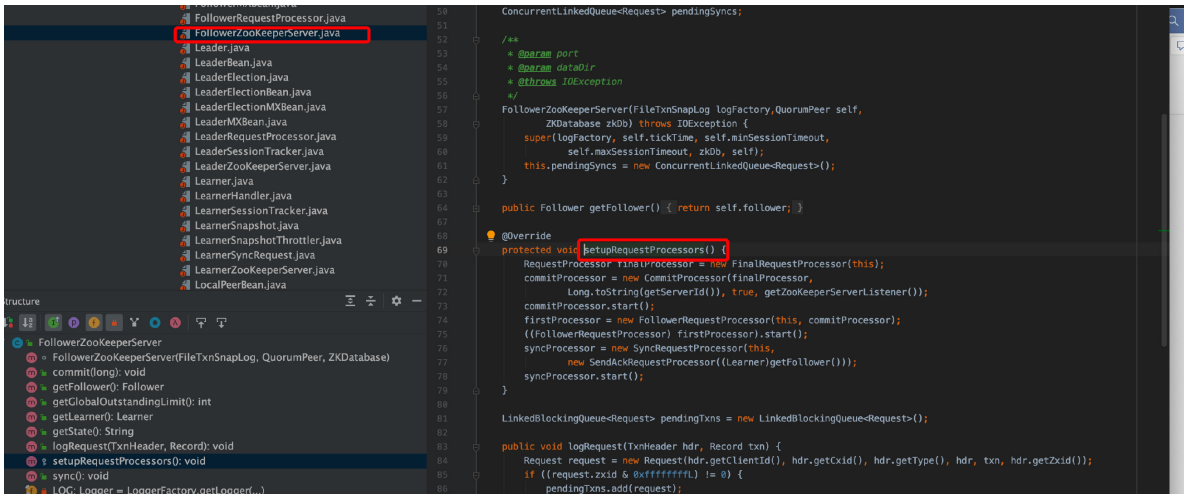

CommitProcessor
ToBeCommitProcessor
FinalRequestProcessor
PrepRequestProcessor
ProposalRequestProcessor
SyncRequestProcessor
AckRequestProcessor


LeaderElection.java
LeaderElectionBean.java
LeaderElectionMXBean.java
LeaderMXBean.java
LeaderRequestProcessor.java
LeaderSessionTracker.java
LeaderZooKeeperServer.java
Learner.java
LearnerHandler.java
LearnerSessionTracker.java
LearnerSnapshot.java
LearnerSnapshotThrottler.java
Structure
LeaderZooKeeperServer
LeaderZooKeeperServer(FileTxnSnapLog, QuorumPeer, ZKDatabase)
checkIfValidGlobalSession(long, int): boolean
createSessionTracker(): void
getGlobalOutstandingLimit(): int
getLeader(): Leader
getServerId(): long
getState(): String
registerJMX(): void
registerJMX(LeaderBean, LocalPeerBean): void
revalidateSession(ServerCnxn, long, int): void
setupContainerManager(): void
setupRequestProcessors(): void
shutdown(): void
startup(): void
submitLearnerRequest(Request): void
touch(long, int): boolean
unregisterJMX(): void
unregisterJMX(Leader): void
commitProcessor: CommitProcessor
containerManager: ContainerManager
prepRequestProcessor: PrepRequestProcessor
private ContainerManager containerManager; // guarded by sync
CommitProcessor commitProcessor;
PrepRequestProcessor prepRequestProcessor;
/**
* @param port
* @param dataDir
* @throws IOException
*/
LeaderZooKeeperServer(FileTxnSnapLog logFactory, QuorumPeer self, ZKDatabase zkDb) throws IOException {
super(logFactory, self.tickTime, self.minSessionTimeout, self.maxSessionTimeout, zkDb, self);
}
public Leader getLeader() { return self.leader; }
@Override
protected void setupRequestProcessors() {
RequestProcessor finalProcessor = new FinalRequestProcessor(this);
RequestProcessor toBeAppliedProcessor = new Leader.ToBeAppliedRequestProcessor(finalProcessor, getLeader());
commitProcessor = new CommitProcessor(toBeAppliedProcessor,
Long.toString(getServerId()), false,
getZooKeeperServerListener());
commitProcessor.start();
ProposalRequestProcessor proposalProcessor = new ProposalRequestProcessor(this,
commitProcessor);
proposalProcessor.initialize();
prepRequestProcessor = new PrepRequestProcessor(this, proposalProcessor);
prepRequestProcessor.start();
firstProcessor = new LeaderRequestProcessor(this, prepRequestProcessor);
setupContainerManager();
}
private synchronized void setupContainerManager() {
containerManager = new ContainerManager(getZKDatabase(), prepRequestProcessor,
Integer.getInteger("znode.container.checkIntervalMs", (int) TimeUnit.MINUTES.toMillis(1)),
Integer.getInteger("znode.container.maxPerMinute", 10000)
);
}

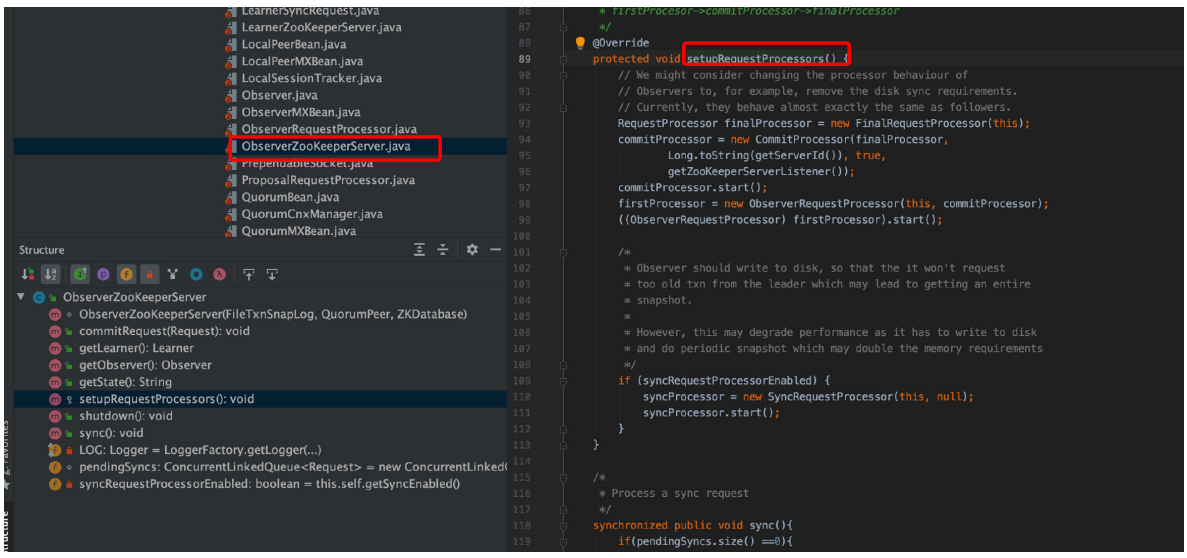

LearnerSyncRequest.java
LearnerZooKeeperServer.java
LocalPeerBean.java
LocalPeerMXBean.java
LocalSessionTracker.java
Observer.java
ObserverMXBean.java
ObserverRequestProcessor.java
ObserverZooKeeperServer.java
PrependableSocket.java
ProposalRequestProcessor.java
QuorumCnxManager.java
QuorumMXBean.java
QuorumPeer.java
QuorumPeerConfig.java
QuorumPeerMain.java
QuorumStats.java
QuorumZooKeeperServer.java
ReadOnlyBean.java
ReadOnlyRequestProcessor.java
ReadOnlyZooKeeperServer.java
Structure
ProposalRequestProcessor
ProposalRequestProcessor(LeaderZooKeeperServer, RequestProcessor)
initialize(): void
processRequest(Request): void
shutdown(): void
LOG: Logger = LoggerFactory.getLogger(...)
nextProcessor: RequestProcessor
syncProcessor: SyncRequestProcessor
zks: LeaderZooKeeperServer
public void processRequest(Request request) throws RequestProcessorException {
// LOG.warn("Ack>>> cxid = " + request.cxid + " type = " +
// request.type + " id = " + request.sessionId);
// request.addRQRec(">prop");
/* In the following IF-THEN-ELSE block, we process syncs on the leader.
* If the sync is coming from a follower, then the follower
* handler adds it to syncHandler. Otherwise, if it is a client of
* the leader that issued the sync command, then syncHandler won't
* contain the handler. In this case, we add it to syncHandler, and
* call processRequest on the next processor.
*/
if (request instanceof LearnerSyncRequest){
zks.getLeader().processSync((LearnerSyncRequest)request);
} else {
nextProcessor.processRequest(request);
if (request.getHdr() != null) {
// We need to sync and get consensus on any transactions
try {
zks.getLeader().propose(request);
} catch (XidRolloverException e) {
throw new RequestProcessorException(e.getMessage(), e);
}
syncProcessor.processRequest(request);
}
}
}
public void shutdown() {
LOG.info("Shutting down");
nextProcessor.shutdown();
syncProcessor.shutdown();
}



Follower



Observer



参考资料：https://www.jianshu.com/p/45f8a966fb47

3.1.3 服务端的线程

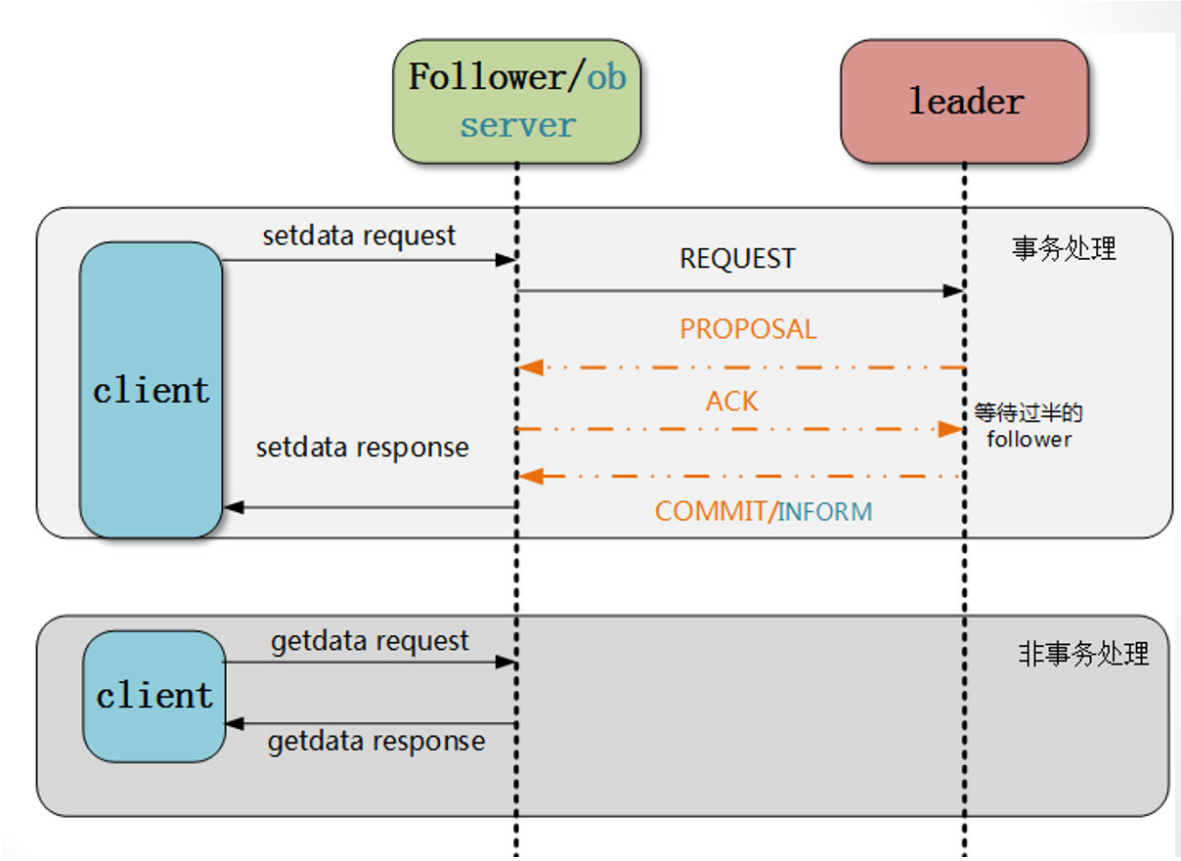
3.2 客户端

从整体看，客户端启动的入口时 ZooKeeperMain，在 ZooKeeperMain 的 run()中，创建出控制台输入对象（jline.console.ConsoleReader），然后它进入 while 循环，等待用户的输入。同时也调用 connectToZK 连接服务器并建立会话（session），在 connect 时创建 ZooKeeper 对象，在 ZooKeeper 的构造函数中会创建客户端使用的 NIO
socket，并启动两个工作线程 sendThread 和 eventThread，两个线程被初始化为守护线程。

sendThread 的 run()是一个无限循环,除非运到了 close 的条件,否则他就会一直循环下去,比如向服务端发送心跳,或者向服务端发送我们在控制台输入的数据以及接受服务端发送过来的响应。

eventThread 线程负责队列事件和处理 watch。

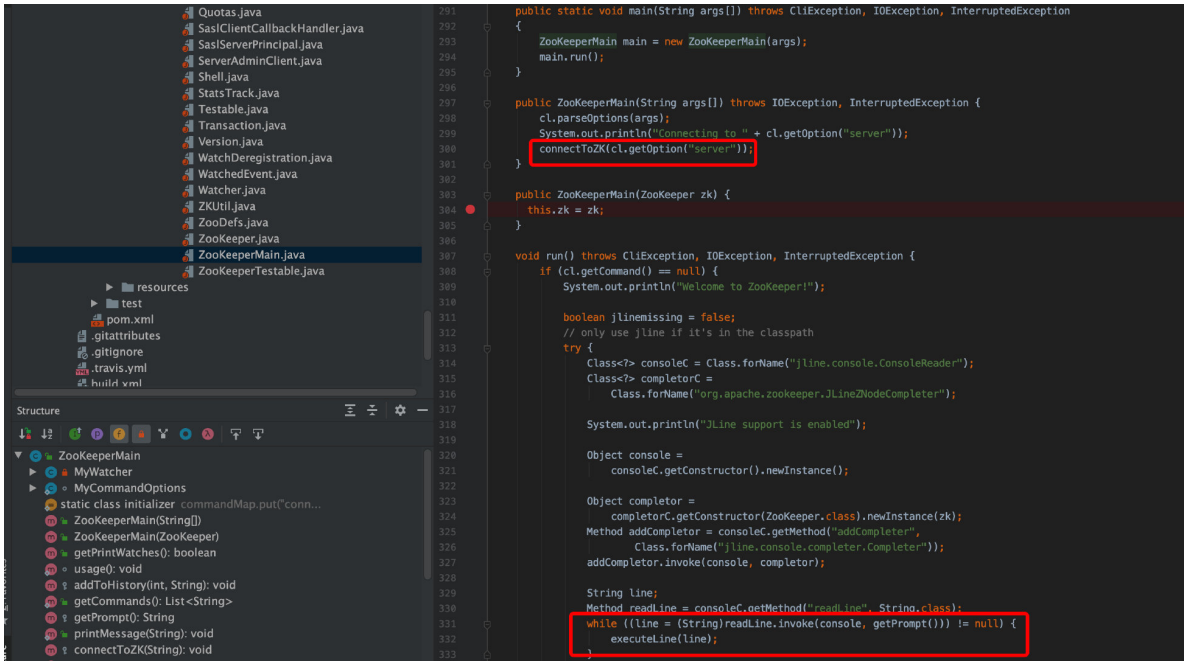
客户端也会创建一个 clientCnxn，由 ClientCnxnSocketNIO.java 负责 IO 数据通信。



客户端的场景说明（事务、非事务请求类型）。

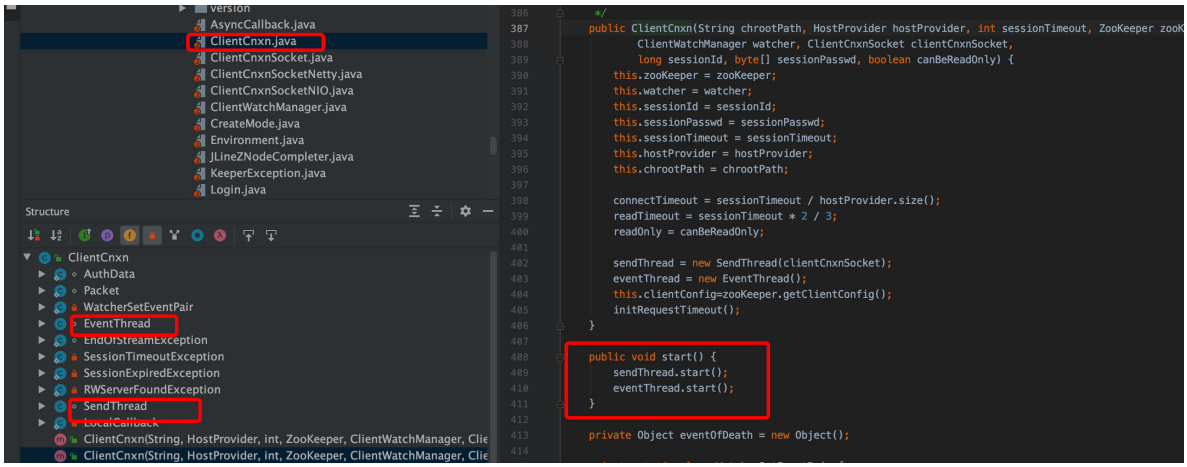
客户端源码解析

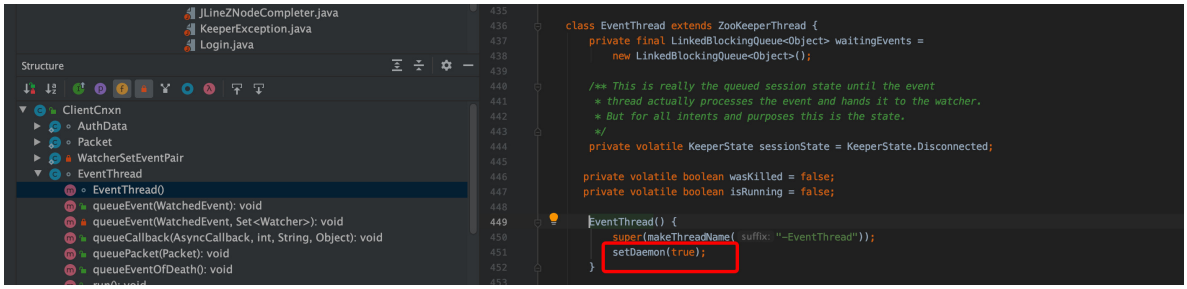
ZooKeeperMain 初始化

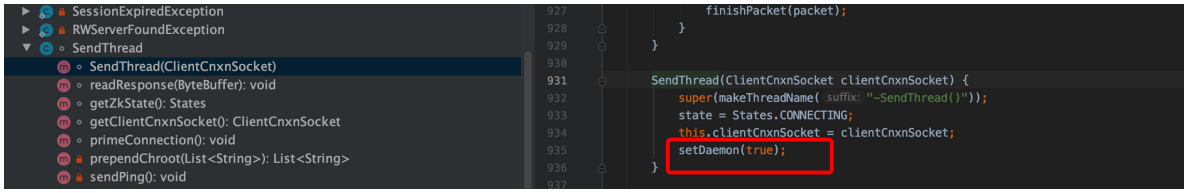


ZooKeeper 的构造函数，cnxn.start()会创建 sendThread 和 eventThread 守护线程。

```
1212        */
1213       public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher,
1214               long sessionId, byte[] sessionPasswd, boolean canBeReadOnly,
1215               HostProvider aHostProvider, ZKClientConfig clientConfig) throws IOException {
1216           LOG.info("Initiating client connection, connectString=" + connectString
1217                   + " sessionTimeout=" + sessionTimeout
1218                   + " watcher=" + watcher
1219                   + " sessionId=" + Long.toHexString(sessionId)
1220                   + " sessionPasswd="
1221                   + (sessionPasswd == null ? "<null>" : "<hidden>"));
1222
1223           if (clientConfig == null) {
1224               clientConfig = new ZKClientConfig();
1225           }
1226           this.clientConfig = clientConfig;
1227           watchManager = defaultWatchManager();
1228           watchManager.defaultWatcher = watcher;
1229
1230           ConnectStringParser connectStringParser = new ConnectStringParser(
1231                   connectString);
1232           hostProvider = aHostProvider;
1233
1234           cnxn = new ClientCnxn(connectStringParser.getChrootPath(),
1235                   hostProvider, sessionTimeout, this, watchManager,
1236                   getClientCnxnSocket(), sessionId, sessionPasswd, canBeReadOnly);
1237           cnxn.seenRwServerBefore = true; // since user has provided sessionId
1238           cnxn.start();
1239       }
```







在 ClientCnxn.java 中，有两个重要的数据结构。

```
/**

* These are the packets that have been sent and are waiting for a response.

*/

private final LinkedList<Packet> pendingQueue = new LinkedList<Packet>();

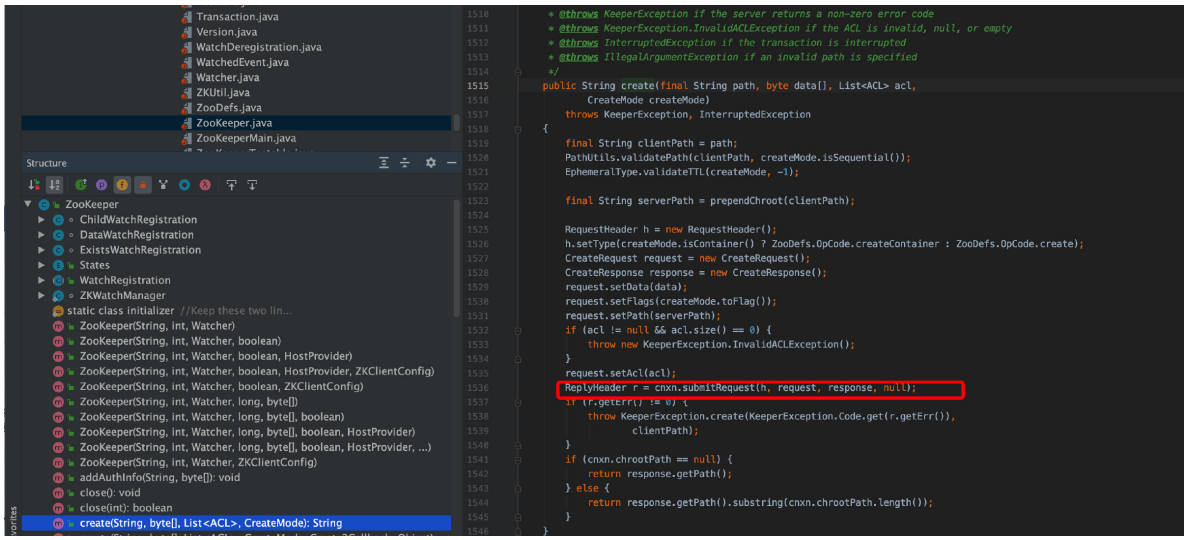
 /**

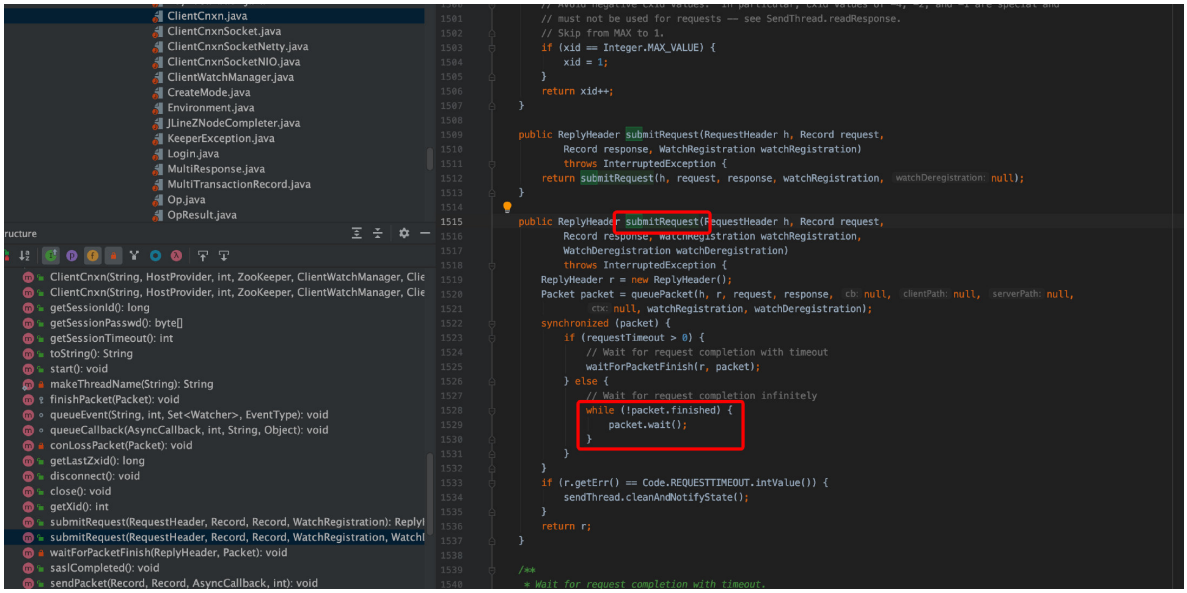
  * These are the packets that need to be sent.

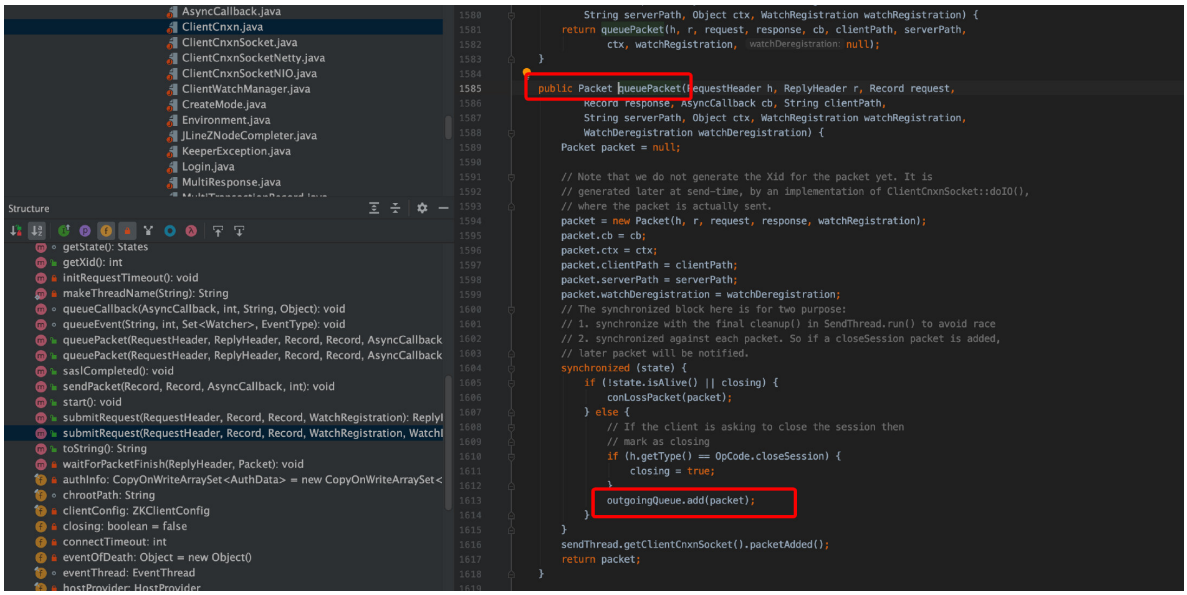
 */
```

private final LinkedBlockingDeque<Packet> outgoingQueue = new LinkedBlockingDeque<Packet>();

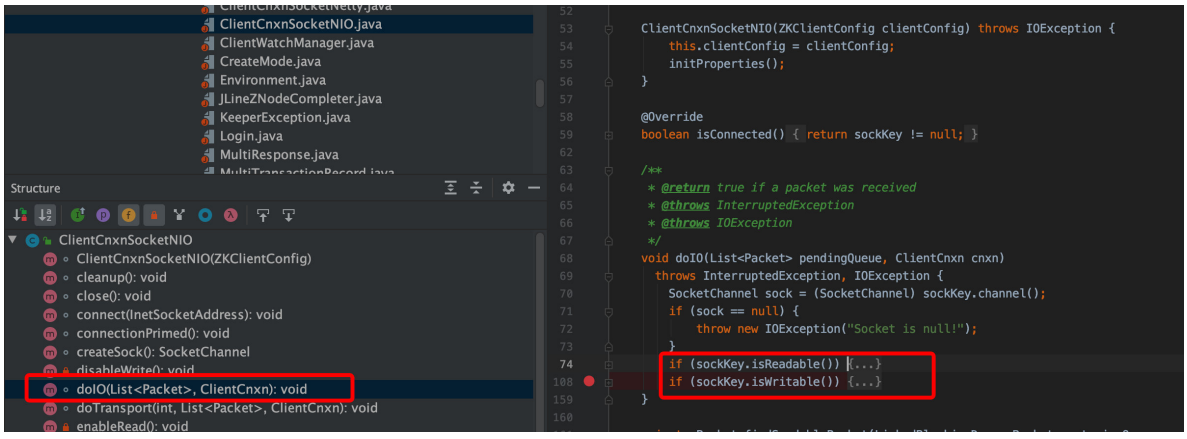
ZooKeeper 类中的对用户的输入参数转换为对 ZK 操作，会调用 cnxn.submitRequest()提交请求,在 ClientCnxn.java 中会把请求封装为 Packet 并写入 outgoingQueue，待 sendThread 线程消费发送给服务端，对于同步接口，调用 cnxn.submitRequest()会阻塞，其中客户端等待是自旋锁。



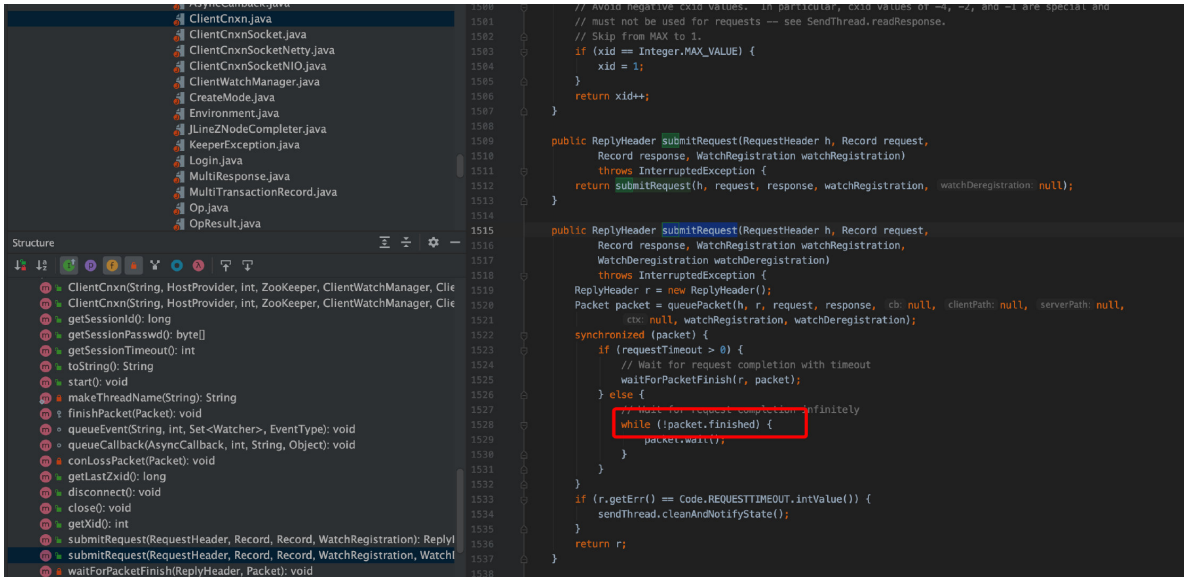


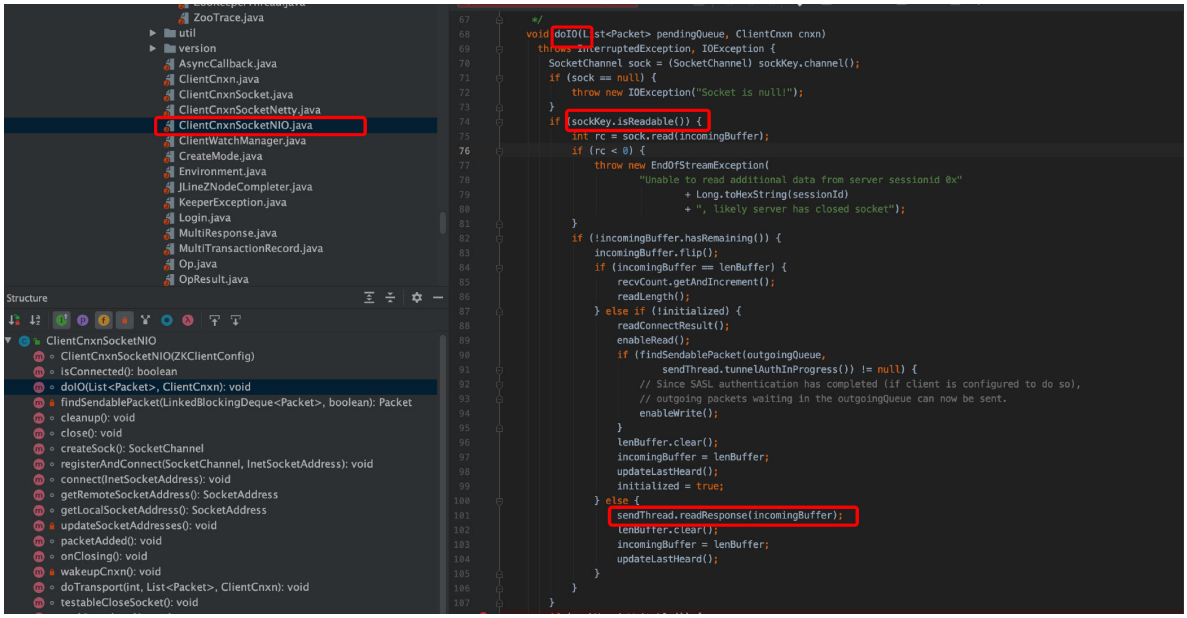


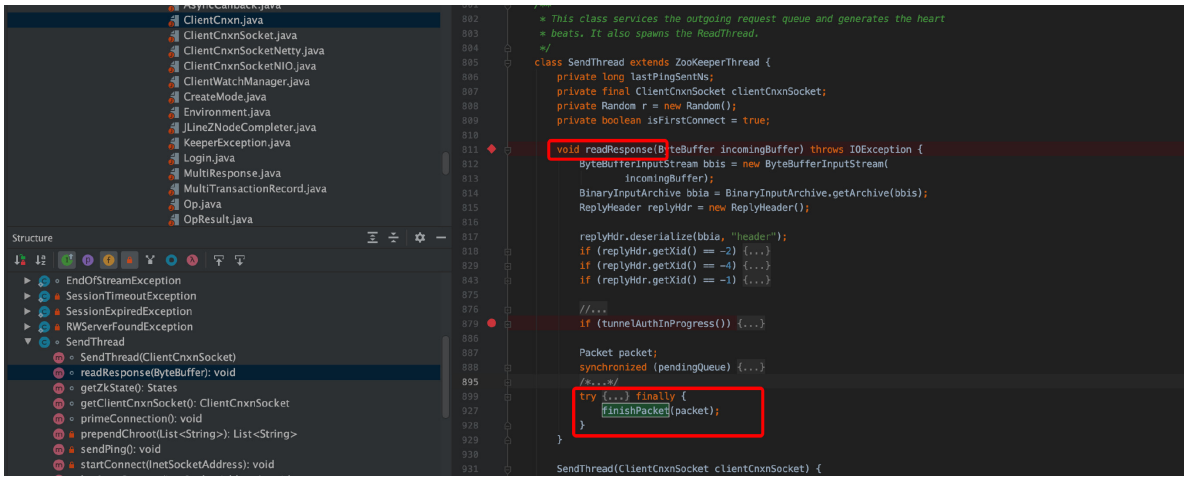
ClientCnxnSocketNIO.java 主要是调用 dcIO(), 其中读就绪,读取服务端发送过来的数据，写就绪, 往客户端发送用户在控制台输入的命令。

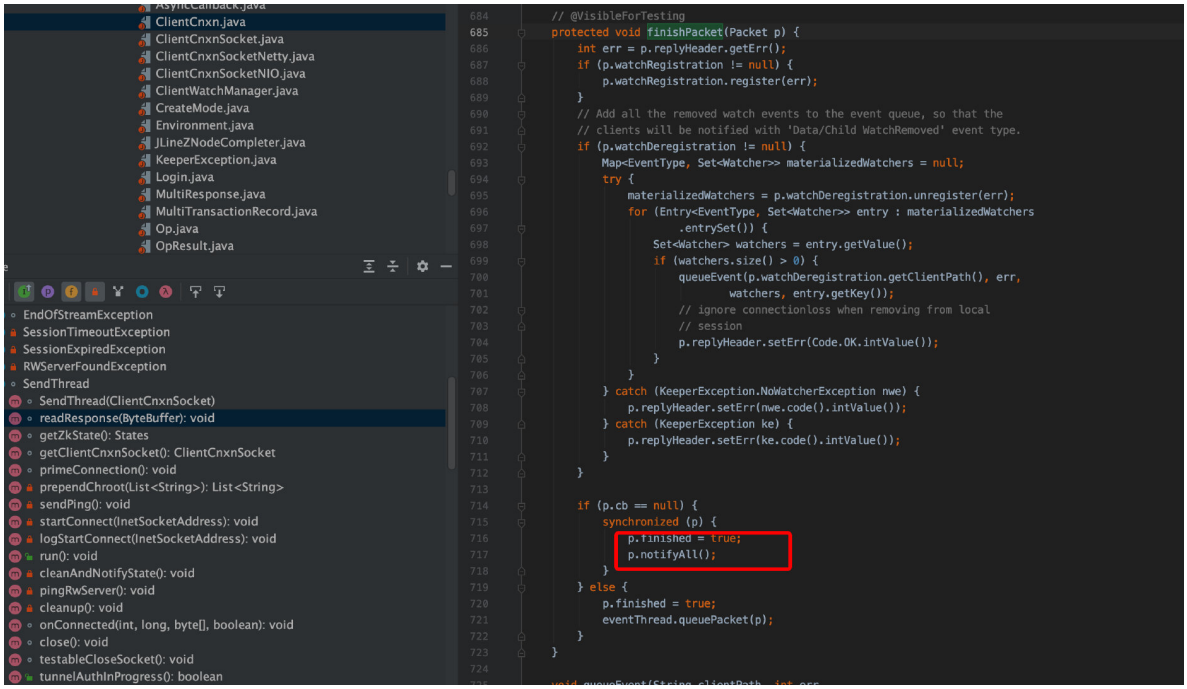


从上面源码看，客户端在 cnxn.submitRequest()，会自旋等待服务端的结果，直到 Packet 的 finished 被设置为 true。ClientCnxnSocketNIO.java 调用 dcIO()，read 逻辑中，会调用 sendThread.readResponse(),
在 sendThread.readResponse()函数中的 finally 中调用 finshPacket()设置 finished 为 true，进而客户端阻塞解除，返回结果。


ClientCnxn.java
ClientCnxnSocket.java
ClientCnxnSocketNetty.java
ClientCnxnSocketNIO.java
ClientWatchManager.java
CreateMode.java
Environment.java
JLineZNodeCompleter.java
KeeperException.java
Login.java
MultiResponse.java
MultiTransactionRecord.java
Op.java
OpResult.java
Structure
ClientCnxn(String, HostProvider, int, ZooKeeper, ClientWatchManager, Clie
ClientCnxn(String, HostProvider, int, ZooKeeper, ClientWatchManager, Clie
getSessionId(): long
getSessionPasswd(): byte[]
getSessionTimeout(): int
toString(): String
start(): void
makeThreadName(String): String
finishPacket(Packet): void
queueEvent(String, int, Set<Watcher>, EventType): void
queueCallback(AsyncCallback, int, String, Object): void
conLossPacket(Packet): void
getLastZxid(): long
disconnect(): void
close(): void
getXid(): int
submitRequest(RequestHeader, Record, Record, WatchRegistration): ReplyH
submitRequest(RequestHeader, Record, Record, WatchRegistration, WatchI
waitForPacketFinish(ReplyHeader, Packet): void
// Skip from MAX to 1.
if (xid == Integer.MAX_VALUE) {
xid = 1;
return xid++;
public ReplyHeader submitRequest(RequestHeader h, Record request,
Record response, WatchRegistration watchRegistration)
throws InterruptedException {
return submitRequest(h, request, response, watchRegistration, watchDeregistration: null);
public ReplyHeader submitRequest(RequestHeader h, Record request,
Record response, WatchRegistration watchRegistration,
WatchDeregistration watchDeregistration)
throws InterruptedException {
ReplyHeader r = new ReplyHeader();
Packet packet = queuePacket(h, r, request, response, cb: null, clientPath: null, serverPath: null,
ctx: null, watchRegistration, watchDeregistration);
synchronized (packet) {
if (requestTimeout > 0) {
// Wait for request completion with timeout
waitForPacketFinish(r, packet);
} else {
// Wait for request completion infinitely
while (!packet.finished) {
packet.wait();
}
}
if (r.getErr() == Code.REQUESTTIMEOUT.intValue()) {
sendThread.cleanAndNotifyState();
}
return r;
}


ZooTrace.java
util
version
AsyncCallback.java
ClientCnxn.java
ClientCnxnSocket.java
ClientCnxnSocketNetty.java
ClientCnxnSocketNIO.java
ClientWatchManager.java
CreateMode.java
Environment.java
JLineZNodeCompleter.java
KeeperException.java
Login.java
MultiResponse.java
MultiTransactionRecord.java
Op.java
OpResult.java
Structure
ClientCnxnSocketNIO
ClientCnxnSocketNIO(ZKClientConfig)
isConnected(): boolean
doIO(List<Packet>, ClientCnxn): void
findSendablePacket(LinkedBlockingDeque<Packet>, boolean): Packet
cleanup(): void
close(): void
createSock(): SocketChannel
registerAndConnect(SocketChannel, InetSocketAddress): void
connect(InetSocketAddress): void
getRemoteSocketAddress(): SocketAddress
getLocalSocketAddress(): SocketAddress
updateSocketAddresses(): void
packetAdded(): void
onClosing(): void
wakeupCnxn(): void
doTransport(int, List<Packet>, ClientCnxn): void
testableCloseSocket(): void
*/
void doIO(List<Packet> pendingQueue, ClientCnxn cnxn)
throws InterruptedException, IOException {
SocketChannel sock = (SocketChannel) sockKey.channel();
if (sock == null) {
throw new IOException("Socket is null!");
}
if (sockKey.isReadable()) {
int rc = sock.read(incomingBuffer);
if (rc < 0) {
throw new EndOfStreamException(
"Unable to read additional data from server sessionid 0x"
+ Long.toHexString(sessionId)
+ ", likely server has closed socket");
}
if (!incomingBuffer.hasRemaining()) {
incomingBuffer.flip();
if (incomingBuffer == lenBuffer) {
recvCount.getAndIncrement();
readLength();
} else if (!initialized) {
readConnectResult();
enableRead();
if (findSendablePacket(outgoingQueue,
sendThread.tunnelAuthInProgress()) != null) {
// Since SASL authentication has completed (if client is configured to do so),
// outgoing packets waiting in the outgoingQueue can now be sent.
enableWrite();
}
lenBuffer.clear();
incomingBuffer = lenBuffer;
updateLastHeard();
initialized = true;
} else {
sendThread.readResponse(incomingBuffer);
lenBuffer.clear();
incomingBuffer = lenBuffer;
updateLastHeard();
}
}
}


ClientCnxn.java
ClientCnxnSocket.java
ClientCnxnSocketNetty.java
ClientCnxnSocketNIO.java
ClientWatchManager.java
CreateMode.java
Environment.java
JLineZNodeCompleter.java
KeeperException.java
Login.java
MultiResponse.java
MultiTransactionRecord.java
Op.java
OpResult.java
Structure
EndOfStreamException
SessionTimeoutException
SessionExpiredException
RWServerFoundException
SendThread
SendThread(ClientCnxnSocket)
readResponse(ByteBuffer): void
getZkState(): States
getClientCnxnSocket(): ClientCnxnSocket
primeConnection(): void
prependChroot(List<String>): List<String>
sendPing(): void
startConnect(InetSocketAddress): void
* This class services the outgoing request queue and generates the heart
* beats. It also spawns the ReadThread.
*/
class SendThread extends ZooKeeperThread {
private long lastPingSentNs;
private final ClientCnxnSocket clientCnxnSocket;
private Random r = new Random();
private boolean isFirstConnect = true;
void readResponse(ByteBuffer incomingBuffer) throws IOException {
ByteBufferInputStream bbis = new ByteBufferInputStream(
incomingBuffer);
BinaryInputArchive bbia = BinaryInputArchive.getArchive(bbis);
ReplyHeader replyHdr = new ReplyHeader();
replyHdr.deserialize(bbia, "header");
if (replyHdr.getXid() == -2) {...}
if (replyHdr.getXid() == -4) {...}
if (replyHdr.getXid() == -1) {...}
//...
if (tunnelAuthInProgress()) {...}
Packet packet;
synchronized (pendingQueue) {...}
/*...*/
try {...} finally {
finishPacket(packet);
}
}
SendThread(ClientCnxnSocket clientCnxnSocket) {



扩展阅读：

https://www.cnblogs.com/ZhuChangwu/p/11587615.html

服务端和客户端结合部分

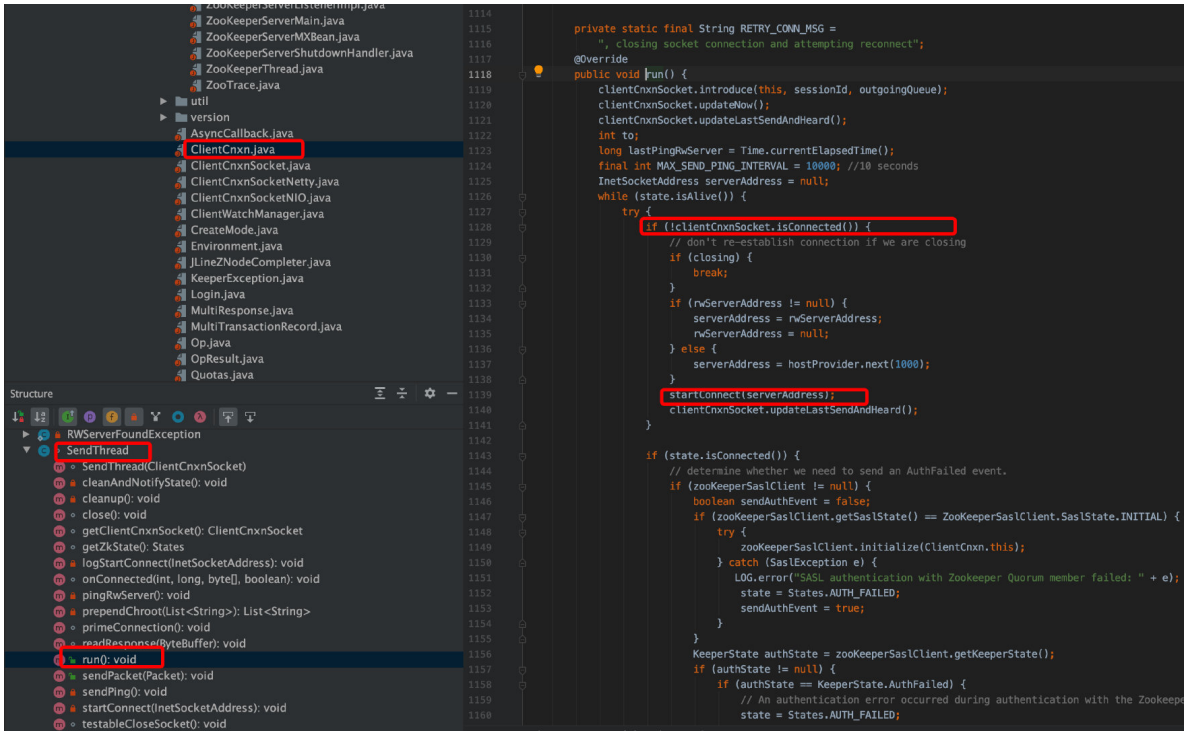
会话（Session）

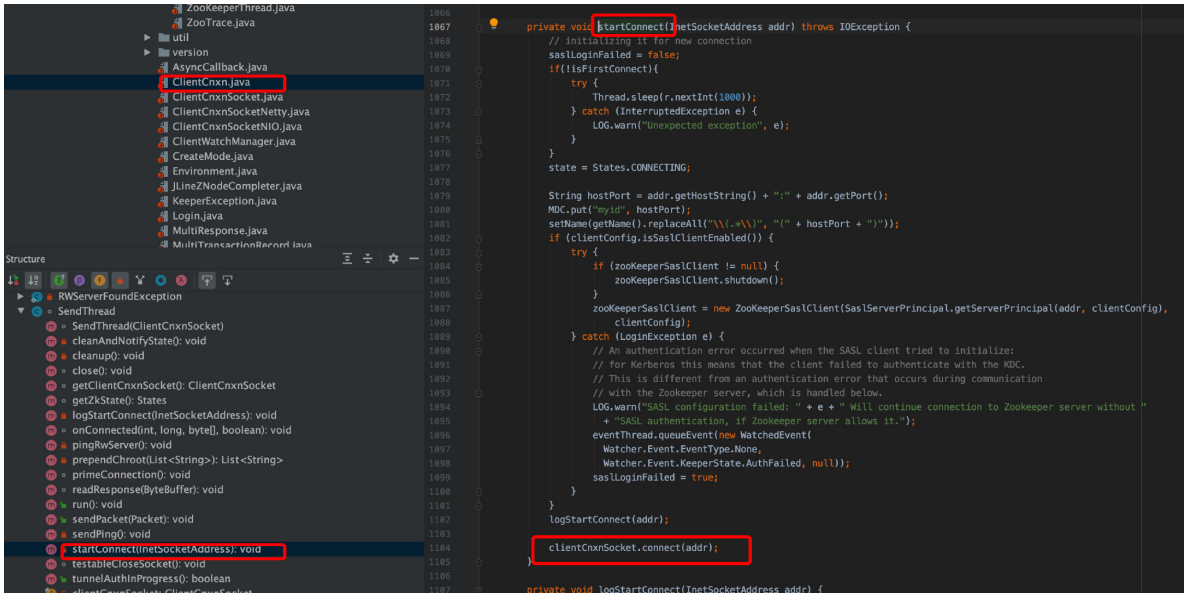
Client 建立会话的流程如下：

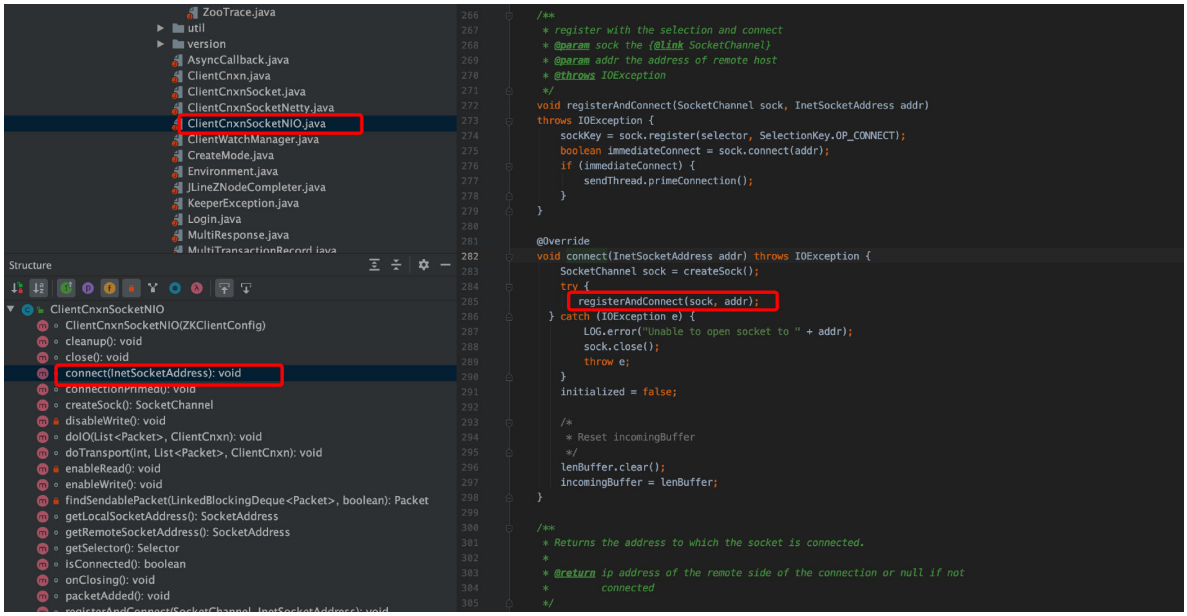
1. 服务端启动，客户端启动；
2. 客户端发起 socket 连接；
3. 服务端 accept socket 连接，socket 连接建立；
4. 客户端发送 ConnectRequest 给 server；
5. server 收到后初始化 ServerCnxn，代表一个和客户端的连接，即 session，server 发送 ConnectResponse 给 client；
6. client 处理 ConnectResponse，session 建立完成。

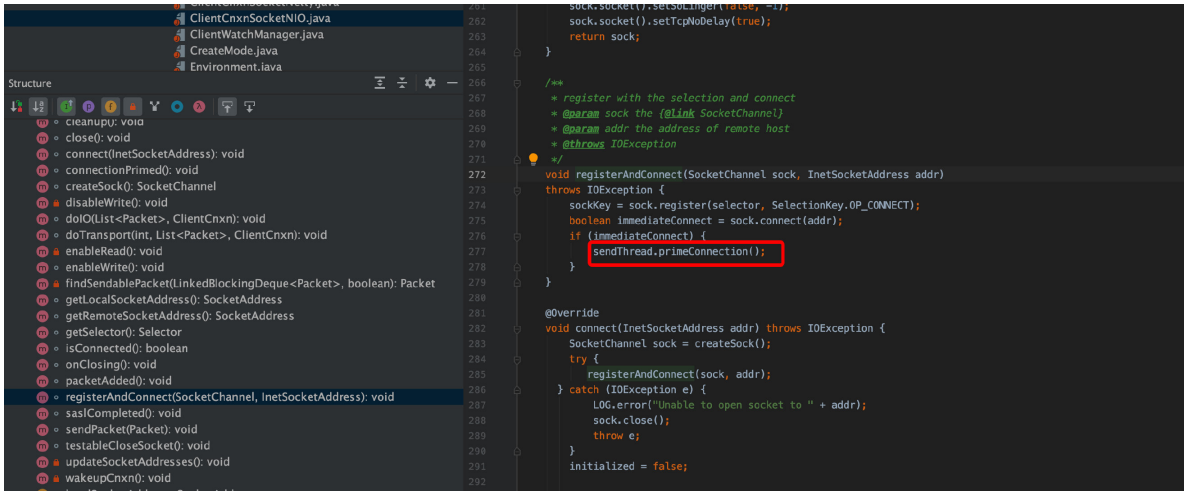
客户端源码分析

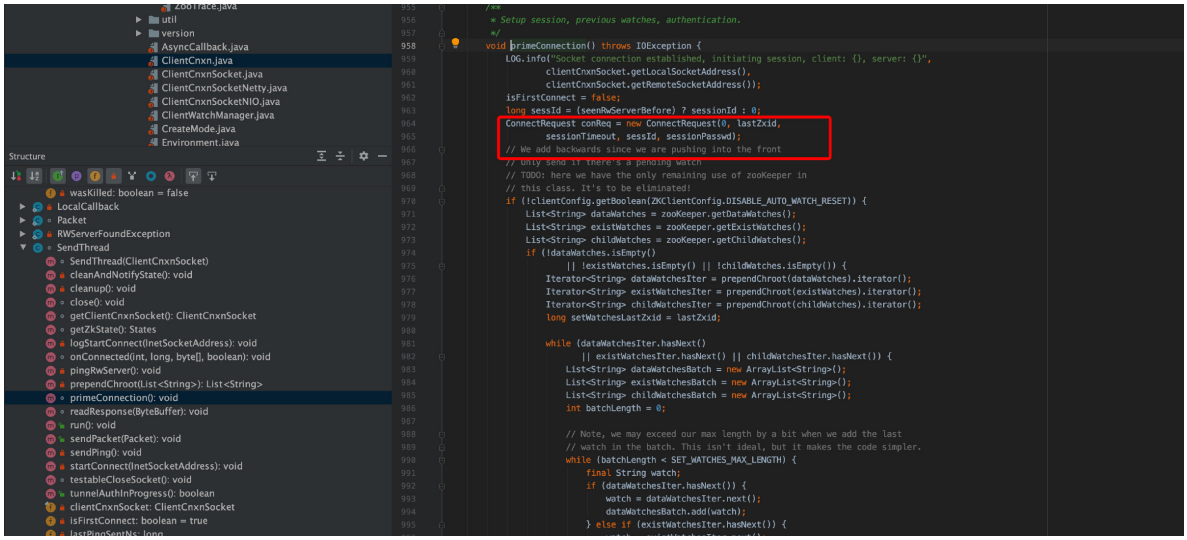
在 clientCnxn.java 中，run 是一个 while 循环，只要 client 没有被关闭会一直循环，每次循环判断当前 client 是否连接到 server，如果没有则发起连接，发起连接调用了 startConnect。


ZooKeeperServerMain.java
ZooKeeperServerMXBean.java
ZooKeeperServerShutdownHandler.java
ZooKeeperThread.java
ZooTrace.java
util
version
AsyncCallback.java
ClientCnxn.java
ClientCnxnSocket.java
ClientCnxnSocketNetty.java
ClientCnxnSocketNIO.java
ClientWatchManager.java
CreateMode.java
Environment.java
JLineZNodeCompleter.java
KeeperException.java
Login.java
MultiResponse.java
MultiTransactionRecord.java
Op.java
OpResult.java
Quotas.java
Structure
RWServerFoundException
SendThread
SendThread(ClientCnxnSocket)
cleanAndNotifyState(): void
cleanup(): void
close(): void
getClientCnxnSocket(): ClientCnxnSocket
getZkState(): States
logStartConnect(InetSocketAddress): void
onConnected(int, long, byte[], boolean): void
pingRwServer(): void
prependChroot(List<String>): List<String>
primeConnection(): void
readResponse(ByteBuffer): void
run(): void
sendPacket(Packet): void
sendPing(): void
startConnect(InetSocketAddress): void
testableCloseSocket(): void
private static final String RETRY_CONN_MSG =
", closing socket connection and attempting reconnect";
@Override
public void run() {
clientCnxnSocket.introduce(this, sessionId, outgoingQueue);
clientCnxnSocket.updateNow();
clientCnxnSocket.updateLastSendAndHeard();
int to;
long lastPingRwServer = Time.currentElapsedTime();
final int MAX_SEND_PING_INTERVAL = 10000; //10 seconds
InetSocketAddress serverAddress = null;
while (state.isAlive()) {
try {
if (!clientCnxnSocket.isConnected()) {
// don't re-establish connection if we are closing
if (closing) {
break;
}
if (rwServerAddress != null) {
serverAddress = rwServerAddress;
rwServerAddress = null;
} else {
serverAddress = hostProvider.next(1000);
}
startConnect(serverAddress);
clientCnxnSocket.updateLastSendAndHeard();
}
if (state.isConnected()) {
// determine whether we need to send an AuthFailed event.
if (zooKeeperSaslClient != null) {
boolean sendAuthEvent = false;
if (zooKeeperSaslClient.getSaslState() == ZooKeeperSaslClient.SaslState.INITIAL) {
try {
zooKeeperSaslClient.initialize(ClientCnxn.this);
} catch (SaslException e) {
LOG.error("SASL authentication with Zookeeper Quorum member failed: " + e);
state = States.AUTH_FAILED;
sendAuthEvent = true;
}
}
KeeperState authState = zooKeeperSaslClient.getKeeperState();
if (authState != null) {
if (authState == KeeperState.AuthFailed) {
// An authentication error occurred during authentication with the Zookeepe
state = States.AUTH_FAILED;


ZooKeeperThread.java
ZooTrace.java
util
version
AsyncCallback.java
ClientCnxn.java
ClientCnxnSocket.java
ClientCnxnSocketNetty.java
ClientCnxnSocketNIO.java
ClientWatchManager.java
CreateMode.java
Environment.java
JLineZNodeCompleter.java
KeeperException.java
Login.java
MultiResponse.java
Structure
RWServerFoundException
SendThread
SendThread(ClientCnxnSocket)
cleanAndNotifyState(): void
cleanup(): void
close(): void
getClientCnxnSocket(): ClientCnxnSocket
getZkState(): States
logStartConnect(InetSocketAddress): void
onConnected(int, long, byte[], boolean): void
pingRwServer(): void
prependChroot(List<String>): List<String>
primeConnection(): void
readResponse(ByteBuffer): void
run(): void
sendPacket(Packet): void
sendPing(): void
startConnect(InetSocketAddress): void
testableCloseSocket(): void
tunnelAuthInProgress(): boolean
private void startConnect(InetSocketAddress addr) throws IOException {
// initializing it for new connection
saslLoginFailed = false;
if(!isFirstConnect){
try {
Thread.sleep(r.nextInt(1000));
} catch (InterruptedException e) {
LOG.warn("Unexpected exception", e);
}
}
state = States.CONNECTING;
String hostPort = addr.getHostString() + ":" + addr.getPort();
MDC.put("myid", hostPort);
setName(getName().replaceAll("\\(.+\\)", "(" + hostPort + ")"));
if (clientConfig.isSaslClientEnabled()) {
try {
if (zooKeeperSaslClient != null) {
zooKeeperSaslClient.shutdown();
}
zooKeeperSaslClient = new ZooKeeperSaslClient(SaslServerPrincipal.getServerPrincipal(addr, clientConfig),
clientConfig);
} catch (LoginException e) {
// An authentication error occurred when the SASL client tried to initialize:
// for Kerberos this means that the client failed to authenticate with the KDC.
// This is different from an authentication error that occurs during communication
// with the Zookeeper server, which is handled below.
LOG.warn("SASL configuration failed: " + e + " Will continue connection to Zookeeper server without "
+ "SASL authentication, if Zookeeper server allows it.");
eventThread.queueEvent(new WatchedEvent(
Watcher.Event.EventType.None,
Watcher.Event.KeeperState.AuthFailed, null));
saslLoginFailed = true;
}
}
logStartConnect(addr);
clientCnxnSocket.connect(addr);
}
private void logStartConnect(InetSocketAddress addr) {

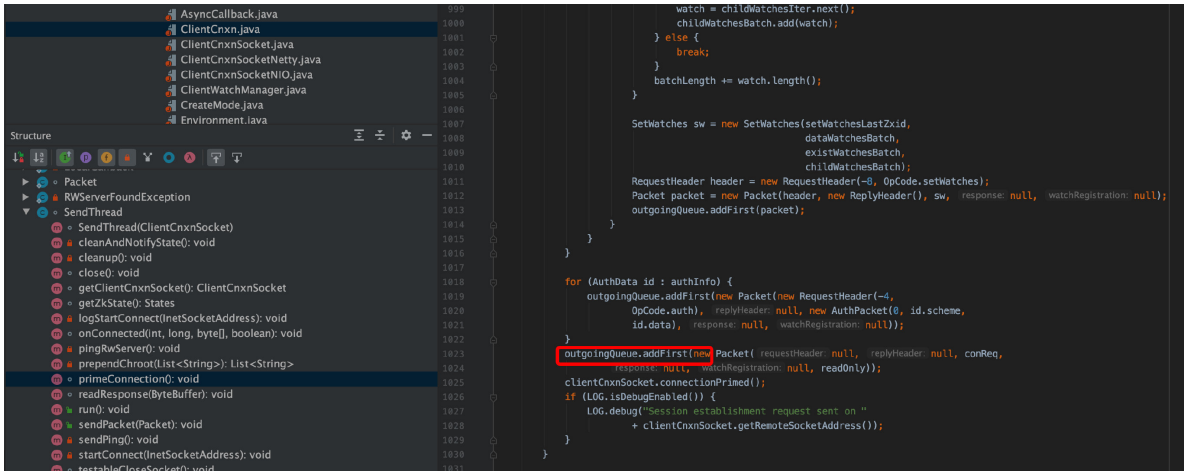

ZooTrace.java
util
version
AsyncCallback.java
ClientCnxn.java
ClientCnxnSocket.java
ClientCnxnSocketNetty.java
ClientCnxnSocketNIO.java
ClientWatchManager.java
CreateMode.java
Environment.java
JLineZNodeCompleter.java
KeeperException.java
Login.java
MultiResponse.java
Structure
ClientCnxnSocketNIO
ClientCnxnSocketNIO(ZKClientConfig)
cleanup(): void
close(): void
connect(InetSocketAddress): void
connectionPrimed(): void
createSock(): SocketChannel
disableWrite(): void
doIO(List<Packet>, ClientCnxn): void
doTransport(int, List<Packet>, ClientCnxn): void
enableRead(): void
enableWrite(): void
findSendablePacket(LinkedBlockingDeque<Packet>, boolean): Packet
getLocalSocketAddress(): SocketAddress
getRemoteSocketAddress(): SocketAddress
getSelector(): Selector
isConnected(): boolean
onClosing(): void
packetAdded(): void
/**
* register with the selection and connect
* @param sock the {@link SocketChannel}
* @param addr the address of remote host
* @throws IOException
*/
void registerAndConnect(SocketChannel sock, InetSocketAddress addr)
throws IOException {
sockKey = sock.register(selector, SelectionKey.OP_CONNECT);
boolean immediateConnect = sock.connect(addr);
if (immediateConnect) {
sendThread.primeConnection();
}
}
@Override
void connect(InetSocketAddress addr) throws IOException {
SocketChannel sock = createSock();
try {
registerAndConnect(sock, addr);
} catch (IOException e) {
LOG.error("Unable to open socket to " + addr);
sock.close();
throw e;
}
initialized = false;
/*
* Reset incomingBuffer
*/
lenBuffer.clear();
incomingBuffer = lenBuffer;
}
/**
* Returns the address to which the socket is connected.
*
* @return ip address of the remote side of the connection or null if not
* connected
*/





> 在 connect 是，传递了如下参数，

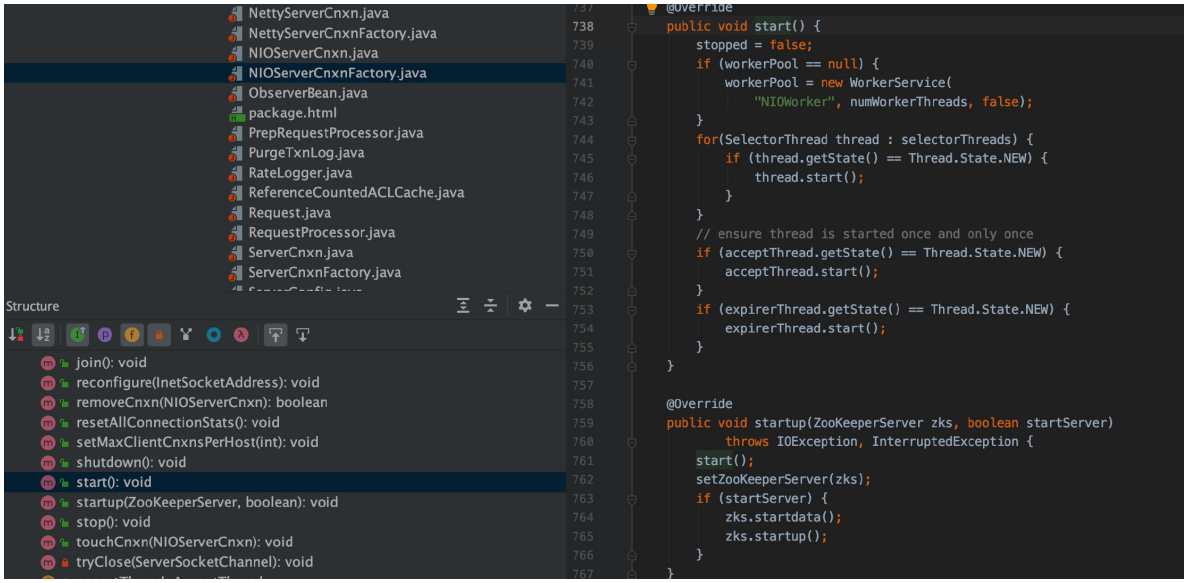
1. lastZxid：上一个事务的 id；
2. sessionTimeout：client 端配置的 sessionTimeout；
3. sessId：sessionId，如果之前建立过连接取的是上一次连接的 sessionId
4. sessionPasswd：session 的密码；

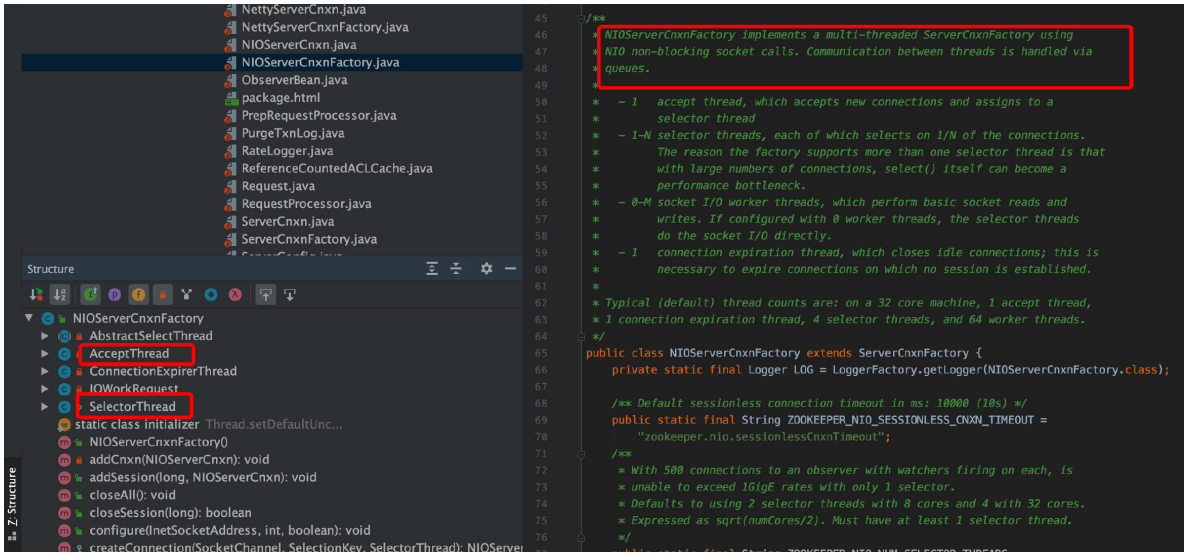


服务端源码分析

server 在启动后，会暴露给客户端连接的地址和端口以提供服务。**我们先看一下** NIOServerCnxnFactory，主要是启动三个线程。

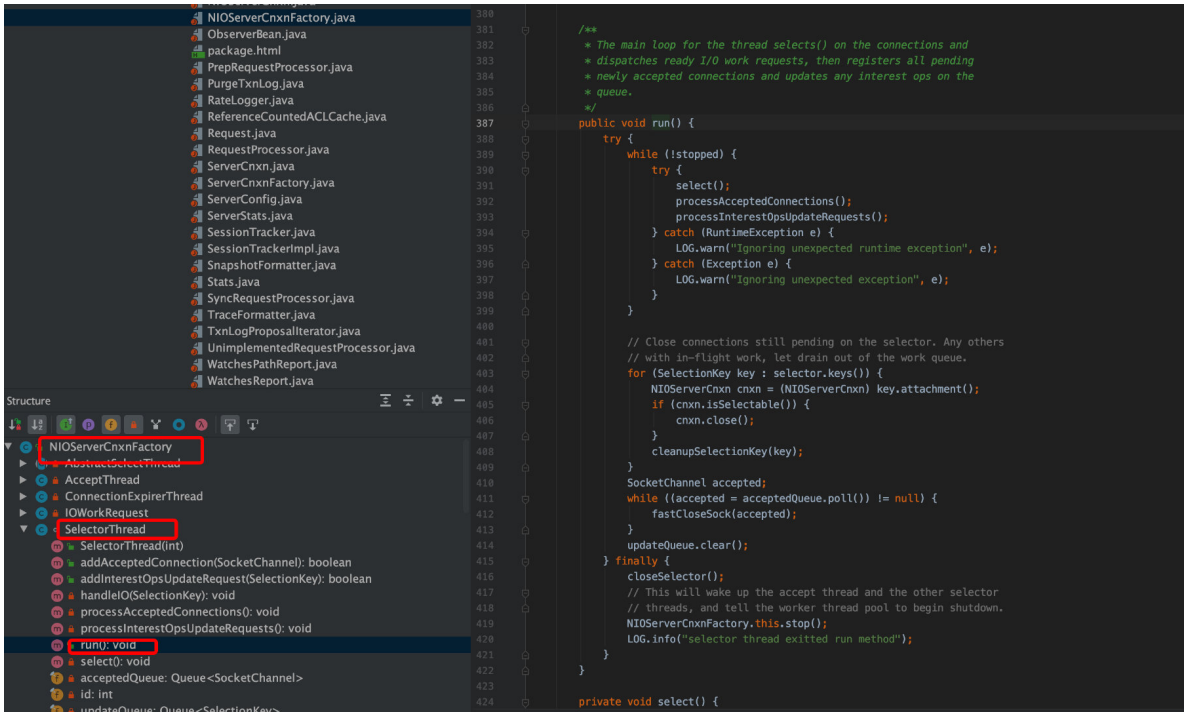
1. AcceptThread：用于接收 client 的连接请求，建立连接后交给 SelectorThread 线程处理
2. SelectorThread：用于处理读写请求
3. ConnectionExpirerThread：检查 session 连接是否过期

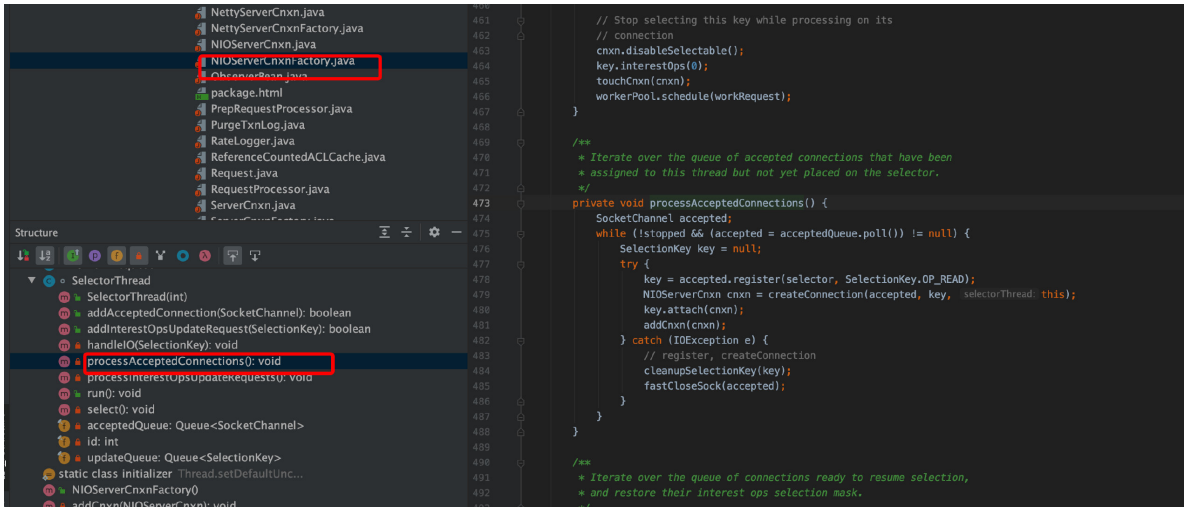




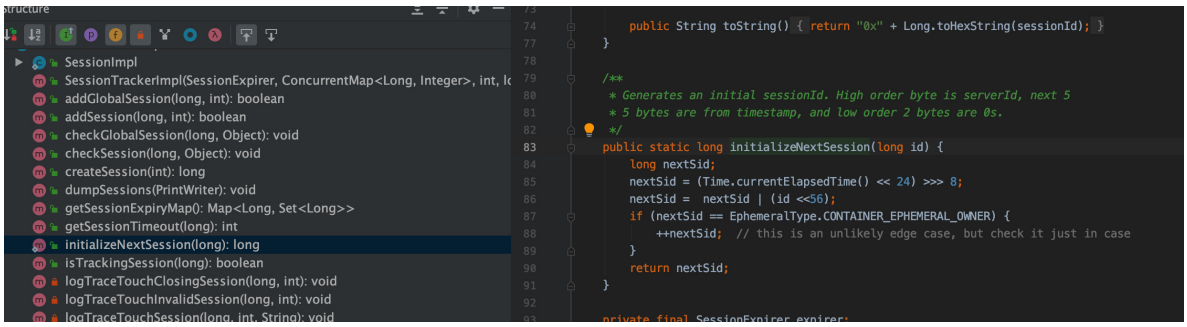
client 发起 socket 连接的时候，server 监听了该端口，接收到 client 的连接请求，然后把建立练级的 SocketChannel 放入队列里面，交给 SelectorThread 处理。

SelectorThread 是一个不断循环的线程，每次循环都会处理刚刚建立的 socket 连接。





session 生成算法



监视（Watch）

本小节主要看看 ZooKeeper 怎么设置监视和监控点的通知。ZooKeeper 可以定义不同类型的通知，如监控 znode 的数据变化，监控 znode 子节点的变化，监控 znode 的创建或者删除。ZooKeeper 的服务端实现了监视点管理器（watch
manager）。

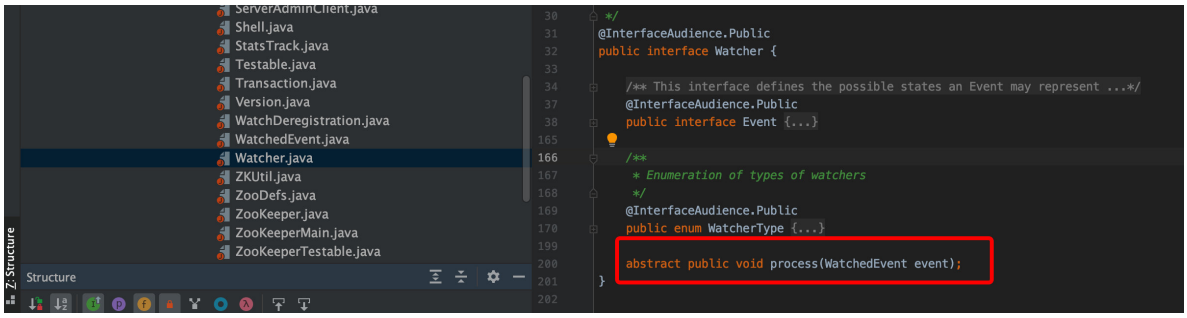
一个 WatchManager 类的实例负责管理当前已经注册的监视点列表，并负责触发他们，监视点只会存在内存且为本地服务端的概念，所有类型的服务器都是使用同样的方式处理监控点。

DataTree 类中持有一个监视点管理器来负责子节点监控和数据的监控。

在服务端触发一个监视点，最终会传播到客户端，负责处理传播的为服务端的 cnxn 对象（ServerCnxn 类），此对象表示客户端和服务端的连接并实现了 Watcher 接口。Watch.process 方法序列化了监视点事件为一定的格式，以便于网络传送。ZooKeeper 客户端接收序列化的监视点事件，并将其反序列化为监控点事件的对象，并传递给应用程序。

以同步版本的 GetData 为例

Watcher 接口的定义，如果设置了监视点，我们要实现 process 函数。



客户端 Watcher 的 process()接口

```
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.Watcher;

public class Master implements Watcher {
    ZooKeeper zk;
    String hostPort;

    Master(String hostPort) {
        this.hostPort = hostPort; ❶
    }

    void startZK() {
        zk = new ZooKeeper(hostPort, 15000, this); ❷
    }

    public void process(WatchedEvent e) {
        System.out.println(e); ❸
    }

    public static void main(String args[])
        throws Exception {
        Master m = new Master(args[0]);

        m.startZK();

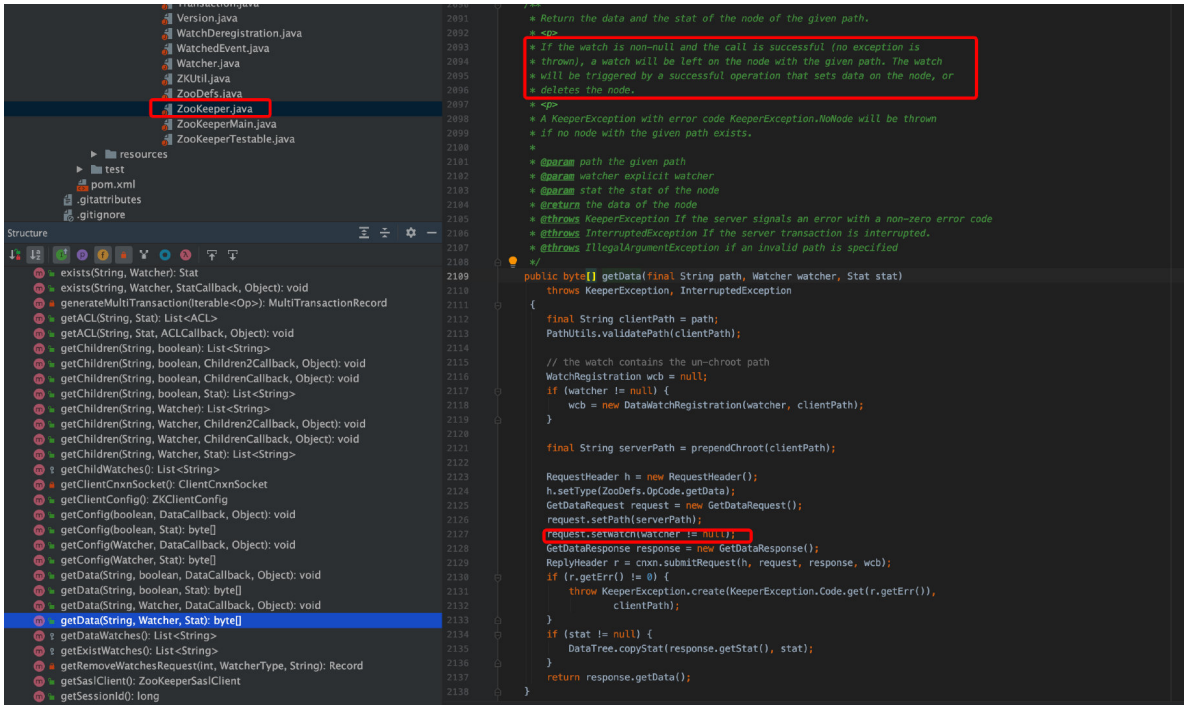
        // wait for a bit
        Thread.sleep(60000); ❹
    }
```

客户端 watcher 实现

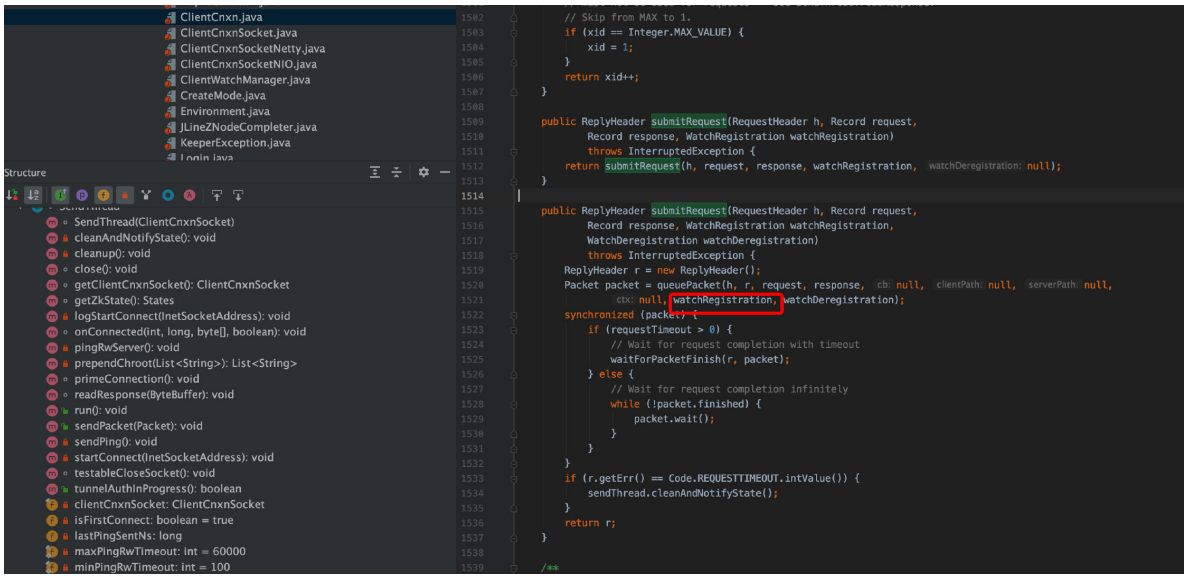
在客户端 GetData 时，如果注册 watch 监控点到服务端，在 watch 的 path 的 value 变化时，服务端会通知客户端该变化。

在客户端的 GetData 方法中（ZooKeeper 类的 GetData）：

- 创建 WatchRegistration wcb= new DataWatchRegistration(watcher, clientPath)，path 和 watch 封装进了一个对象；
- 创建一个 request，设置 type 为 GetData 对应的数值；
- request.setWatch(watcher != null)，setWatch 参数为一个 bool 值。
- 调用 ClientCnxn.submitRequest(...)，
  将请求包装为 Packet，queuePacket()方法的参数中存在创建的 path+watcher 的封装类 WatchRegistration，请求会被 sendThread 消费发送到服务端。



客户端 GetData()

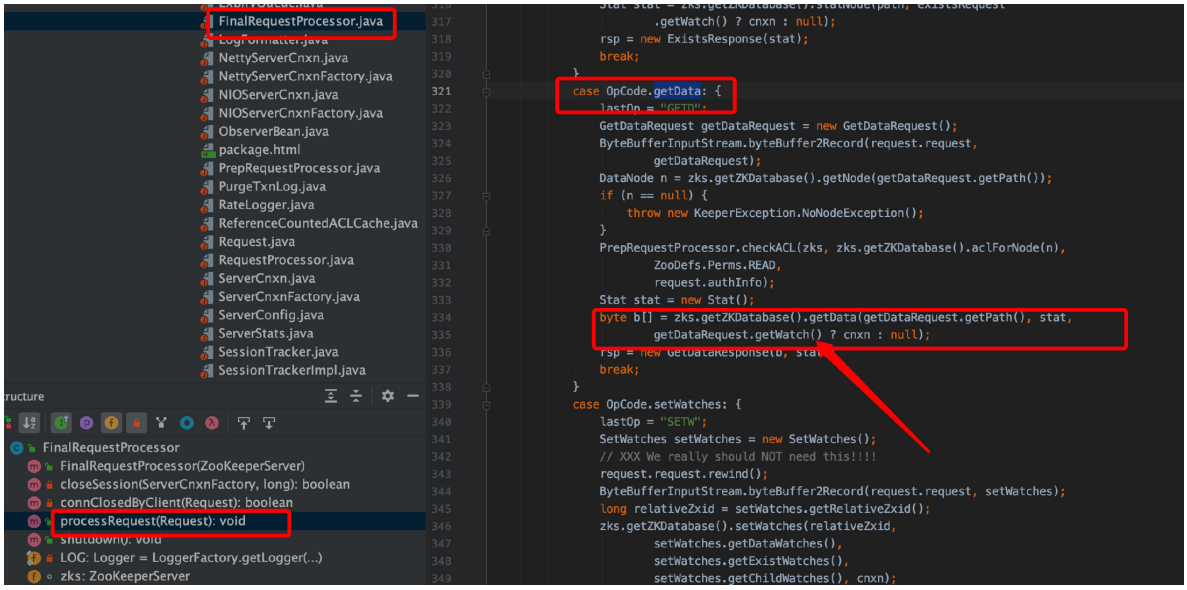


客户端 submitRequest()

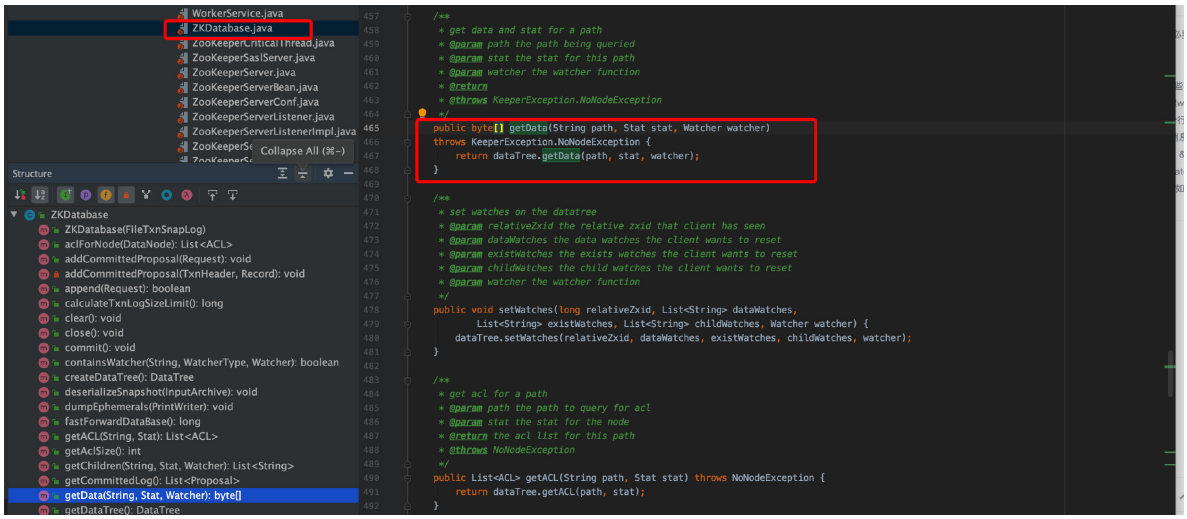
服务端如何处理 GetData 请求呢，由于是读请求，我们可以直接看 FinalRequestProcessor 处理器的 public
void processRequest(Request request){}方法，看它针对 GetData()方式的请求做出了哪些动作。

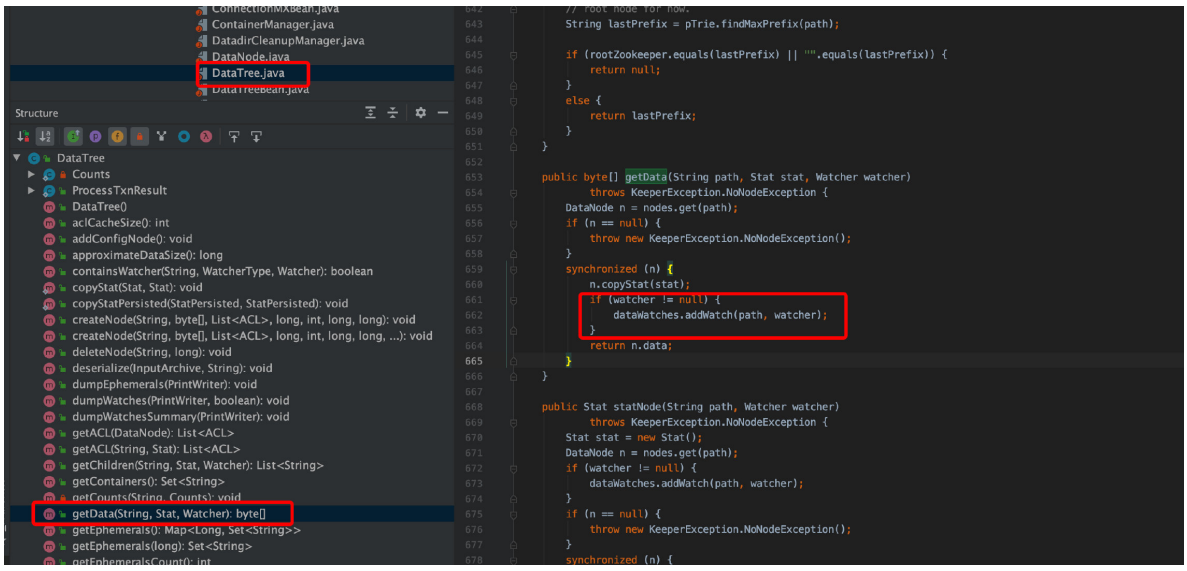
zks.getZKDatabase().getData(getDataRequest.getPath(), stat, getDataRequest.getWatch() ? cnxn : null)

根据 watcher 的“有无”给服务端添加不同的 Watcher。服务端 GetData()函数，在服务端维护了一份 path+watcher 的 map，如果设置 watcher，服务端会保存该 path 的 watcher。

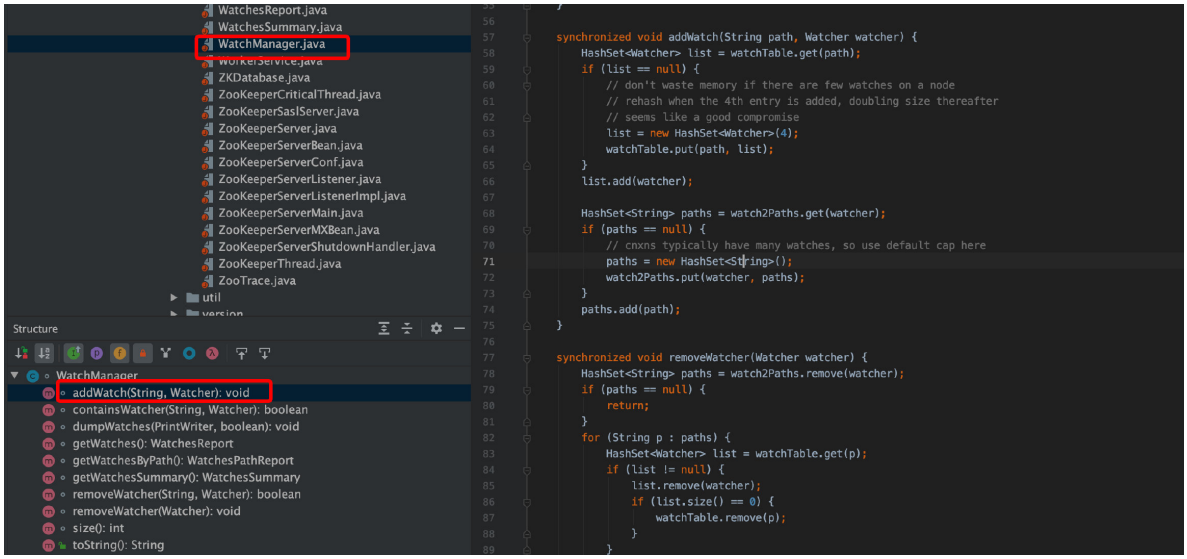


服务端 GetData()





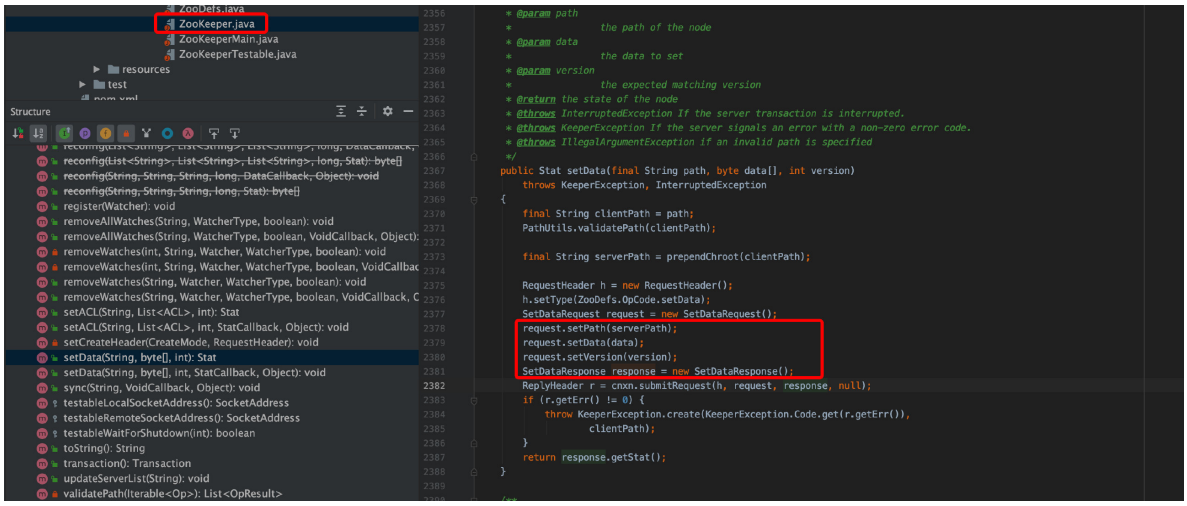
服务端 GetData()



服务端 addWatch ()

为了测试服务端监视通知客户端，我们在客户端本地输入的命令，

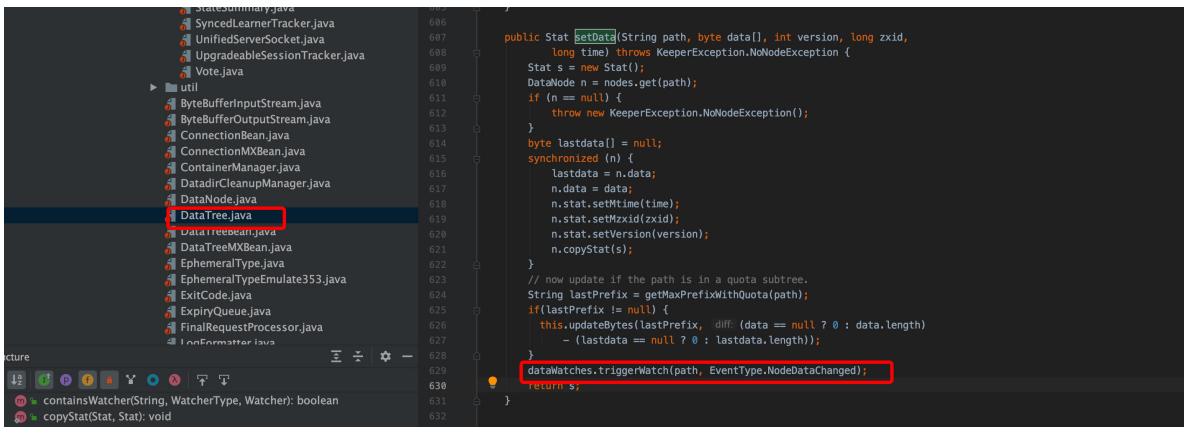
set /path newValue



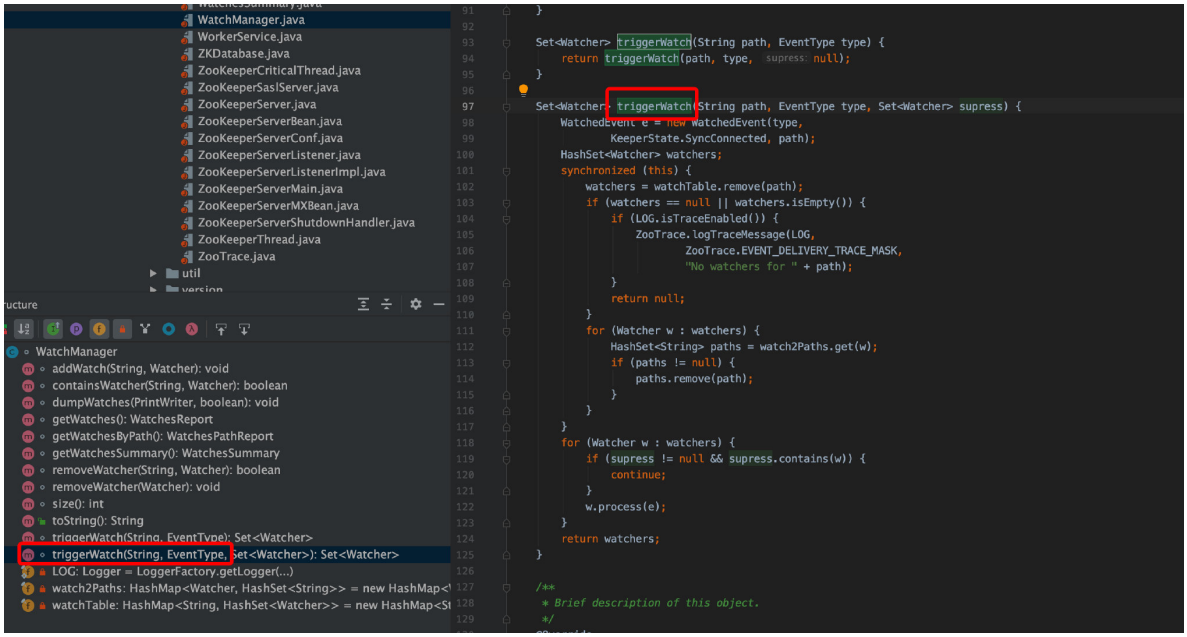
客户端 SetData()

从 SetData 的源码看，本次的 submitRequest 参数中，WatchRegistration==null，可以推断，服务端在 FinalRequestProcessor 中再处理时取出的 watcher==null,也就不会将 path+watcher 保存进 maptable 中，其他的处理过程和上面 GetData 类似。

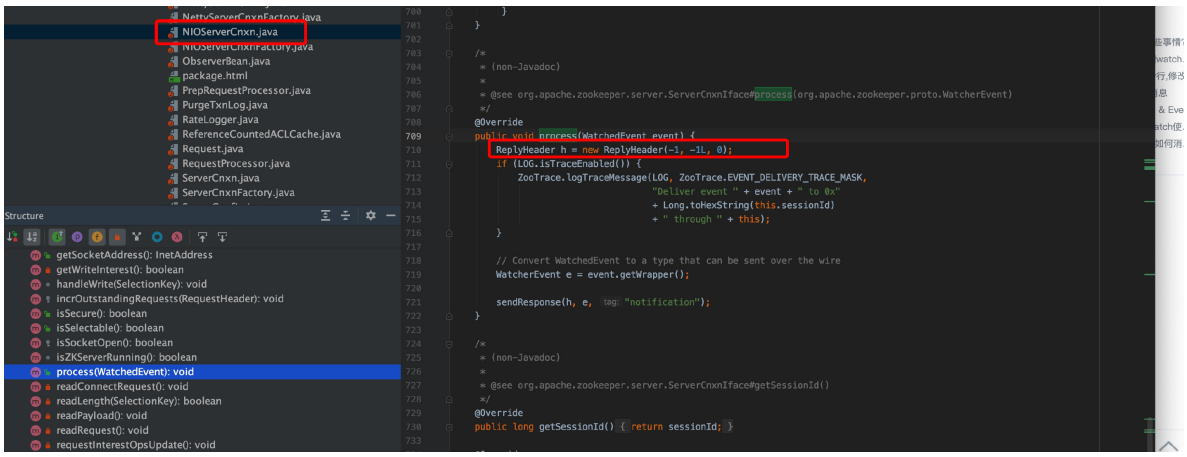
服务端在满足触发监控点时，并通过 cnxn 的 process()方法处理（NIOServerCnxn 类）通知到客户端。在服务端处理的 SetData()函数看，Set 数值后，会触发 watch 回调，即 triggerWatch()。



服务端 SetData()



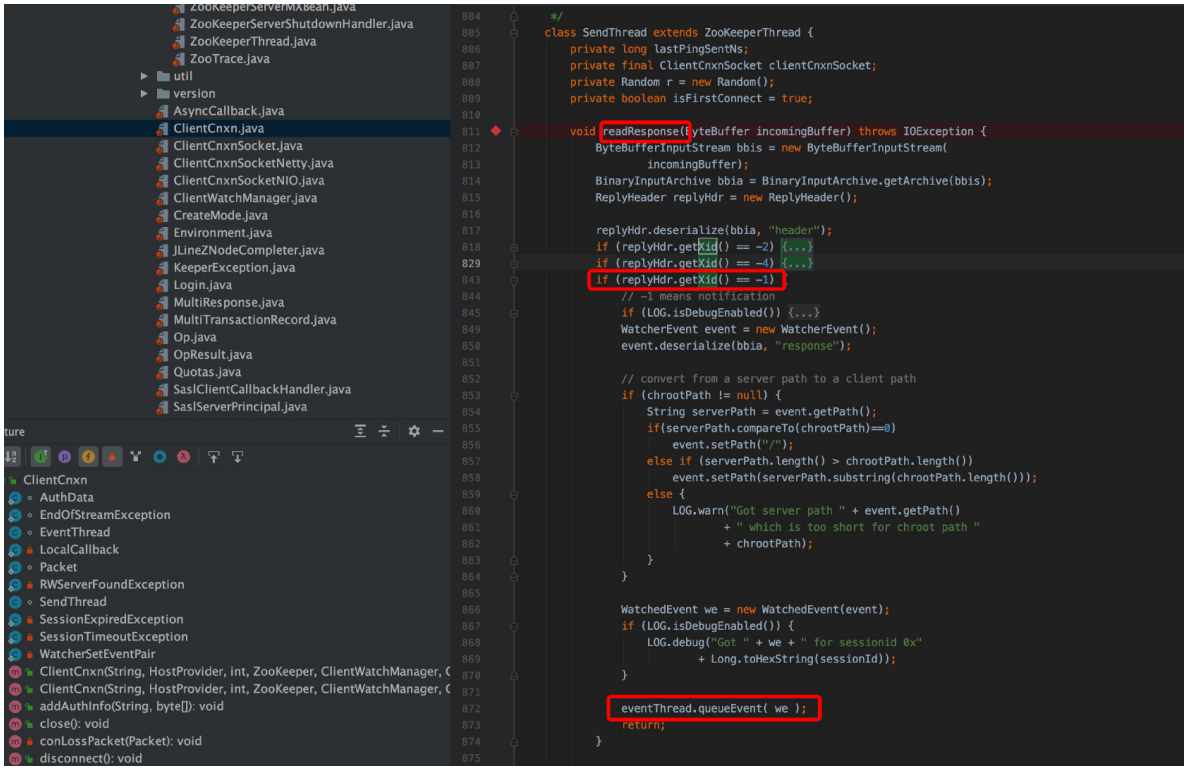
服务端 triggerWatch ()



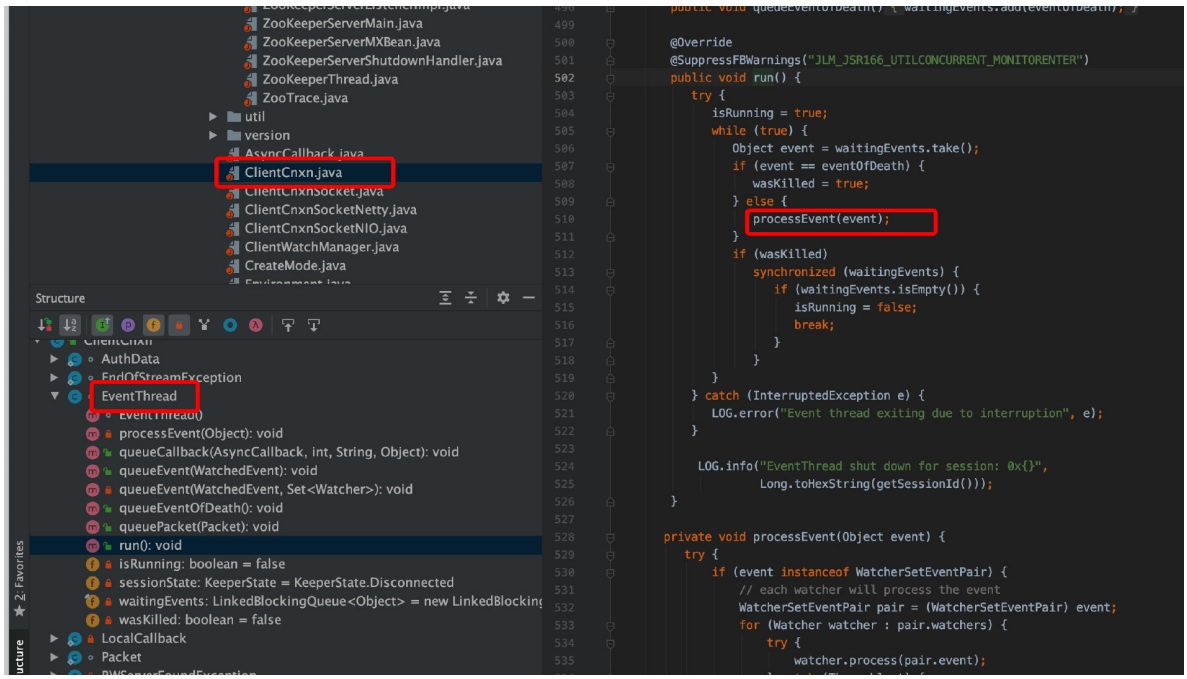
服务端 NIOServerCnxn 的 process()

从上面看服务端在往客户端发送事务型消息, 并且 new ReplyHeader(-1, -1L,0)第一个位置上的参数是-1。

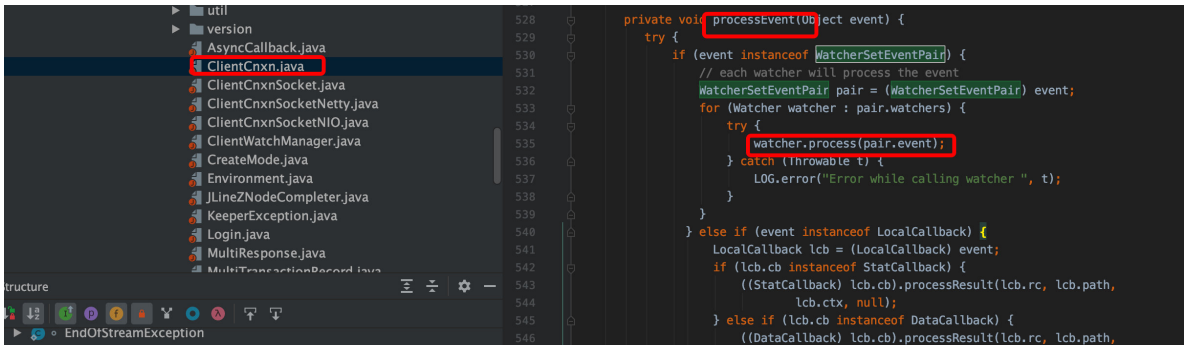
在客户端的 SendThread 读就绪源码部分（readResponse），在 readResponse 函数中会判断 xid==-1 时然后调用 eventThread.queueEvent(we)，把响应交给 EventThread 处理。



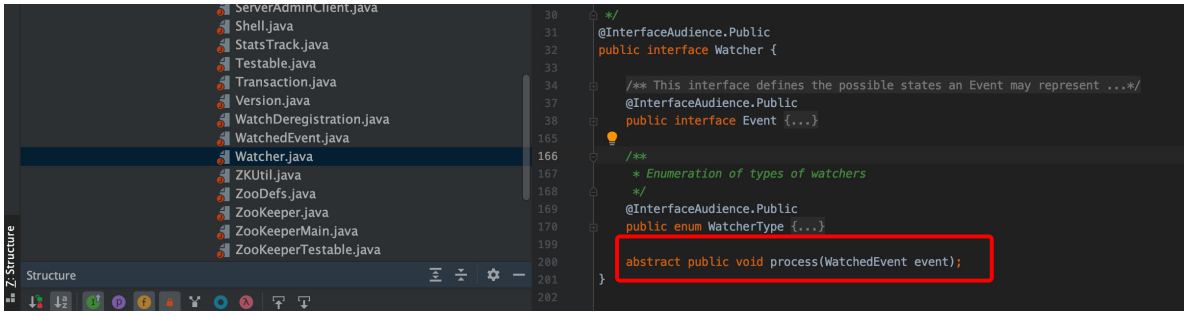
其 eventThread 是一个守护线程，run()函数在 while(true)去消费 waitingEvents，最终调用会 watcher.process(pair.event)，其中 process 是 watcher 的 process 的接口的实现，从而完成 wacher 回调处理。



客户端 eventThread 的 run()



客户端 processEvent()



客户端接口 watch process()

ZooKeeper 实践经验

4.1 业务的控制面架构

控制面架构

在业务处理逻辑中，API 会写 ZooKeeper 和 db 的，agent 作为客户端连接 ZooKeeper 集群，并注册 watch 到感兴趣的节点，在 watch 的 znode 发生变化时，服务端触发通知 agent，agent 感知到数据变化，经过数据转换，再通过适当的接口下发到设备上。

ZooKeeper 部署结构

客户端会根据域名解析访问 Observer，客户端不会直接连接主集群，做到读写分离。

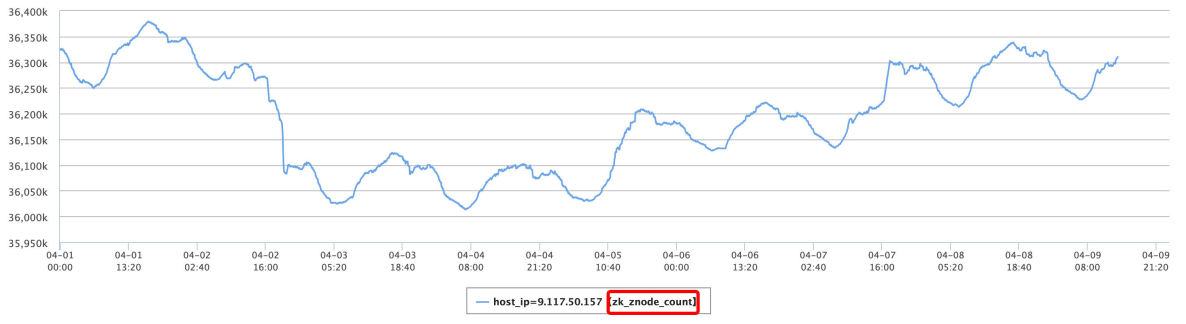
4.2 ZooKeeper 集群的特点

目前我们运维的 ZooKeeper 集群规模大，客户端数目也很大，导致 znode 数目和 watcher 数目也是巨大的，这个运维带来重大的挑战。

ZooKeeper 集群规模，以地域级集群举例（2020 前）

地域	集群规模（设备数目）	备注
上海	168	
广州	95	
北京	41	
其他	4 ~ 12	

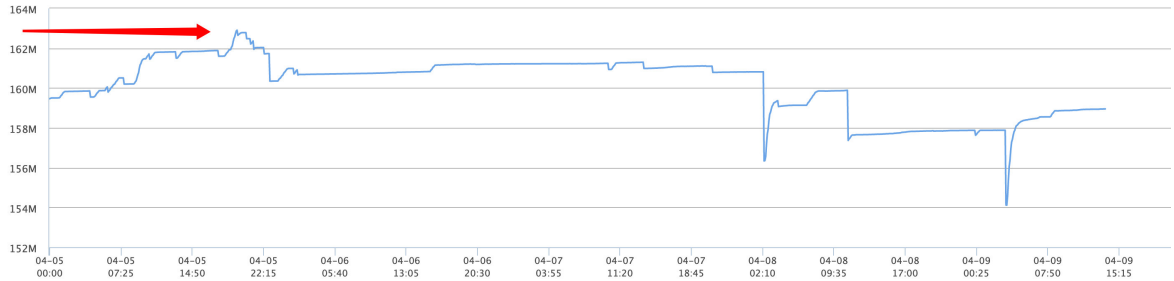
ZooKeeper 集群的 Znode 数目



该设备管理的 znode 节点数目高达 3 千万+，同时我们也可以看出 znode 节点在动态的变化，波谷在晚上，这些变化就是用户进行扩缩容。

ZooKeeper 集群的 Watch 数目

选择单台设备看



该设备管理的 watcher 节点数目高达 1.6 亿 watch 数目，同时我们也可以看出 watch 节点在动态的变化，波谷在晚上，这些变化就是用户进行扩缩容。

实践场景分析和优化措施

4.3.1 灾备集群搭建

痛点：由于各种原因，ZooKeeper 集群可能发起重新选举，并且在选举过程中，集群服务会不可用，更有甚者，长时间选举不出来 Leader，需要重启集群。同时，现网会遇到机房裁撤，需要迁移 ZooKeeper 的服务器，特别是 3.5 版本前，ZooKeeper 没有提供**重配置**（reconfig），在迁移集群时，需要复杂的启用服务器，风险很大。

在现网运营中，出现过半个小时以上，服务不可用的情况，灾备集群的搭建显得十分重要。

ZooKeeper 数据存储的一个优点是，数据的存储方式是一样的，通过事务日志和快照的合并可以得到正确的数据视图，可以拷贝日志文件和快照文件到另外的新集群。

目前我们切换新旧集群还是人工参与，不过可以大幅度降低服务不可用的整体时间。在搭建灾备集群时，也会遇到环境，配置，机型等问题，需要在实践中摸索，并能熟练的切换。

4.3.2 Observer 单核高负载时 Observer 数据落地慢

触发点

ZK 数据有突发写入时，子树数据量大。

故障现象

客户端感知数据变化慢，下发配置不及时，导致用户业务受影响。

故障过程

1. ZooKeeper 数据有突发写入时；
2. 客户端从 Observer 拉取大子树（children 很多的节点的 children 列表）；
3. 触发 Observer 发生单核高负载，高负载 CPU 主要处理 getChildren 时的数据序列化去了；

4．客户端看见从 Observer
getChildren 回来的数据是很旧的数据，而此时 ZooKeeper 数据早就写入主集群了；

5．客户端一次不能看见的数据变化特别慢，导致客户端花了很长时间才感知并在本地处理完这些突发写入。

故障原因分析

写子树时，触发客户端的 Children 事件，由于 ZooKeepeer 实现的机制不能单独通知哪个 Children 节点变化，客户端必须自己去 getChildren 获得全量的 Children 节点（例如 Children 层机有 10w 节点，在新增一个节点，客户端需要下拉 10w+的数据到本地），如果 Children 数量很大，会极大消耗 Observer 的性能，在 Observer 高负载后处理不及时，导致下发配置延时。

优化措施

1.扩容 Observer，并进行监控 Observer 的状态（cpu，内存）；

2.大量子节点树二级分组优化，把 getChildren 拉取数据的规模降低。

3.客户端开启多进程，根据适当的指标分组，然后分配到不同进程去管理节点，可以加速并发，进程的管理节点规模要尽量均衡性。

4.客户端可以延时拉取，例如如果要插入 10 个节点，在获得第一次 watch 通知后可以 hold 一个随机事件，再去拉取数据，这样在 hold time 时，节点现象变化完成，可以一下子拉取到所有变化现象，而不是在每个节点变化时都 get 一次，加大对服务端的压力，不过这个 hold time 的是否开启要根据具体的业务场景决定。

4.3.3 服务器 Full GC 导致会话异常

触发点

ZooKeeper 的服务端机器发生了 gc，gc 时间过长，gc 结束后发生会话超时处理。

故障现象

长时间的 gc 后，会话超时，客户端再请求服务器时，遇到异常，客户端会重启。服务端断开大量的客户端时，会带来**连接冲击**。

4.3.4 机房网络中断，大量连接冲击 Observer

触发点

客户端，Observer，主集群跨区部署，某区机房网络短暂中断。

连接冲击现象

集群有连接冲击发生时，closeSession 事务导致所有 Observer 无法快速处理新建的连接和其他请求，从而客户端主动断连，又出现更多的 closeSession。几乎无法自行恢复。



单台 Observer 临时节点的数量变化



集群中 Fellower 数量变化

故障过程

阶段 1：网络异常，Observer 和主集群的通信中断，Leader 把 Observer 踢出集群（从上图的**Fellower 的数量**变化可以看出），大量客户端开始断连（从上图的**临时节点的数量变化**可以看出）；

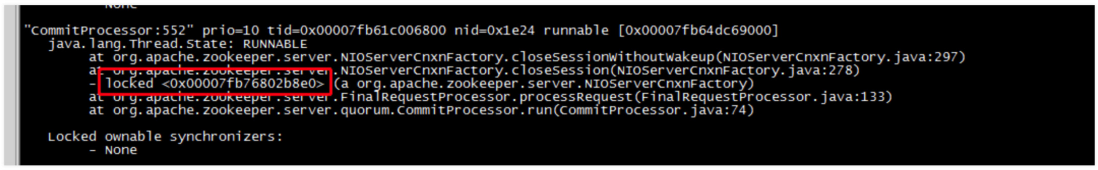
阶段 2: 网络恢复后 Observer 感知到了被踢出，进入自恢复逻辑；

阶段 3: Observer 同步完新事务，并进入 Serving 状态；

阶段 4: 大量客户端开始重连 Observer，Observer 没有限制住连接冲击导致卡死。

故障原因

在阶段 4，观察分析 Observer 的 pps 不是很高，不过处理事务非常慢，线程栈发现有两个线程互相卡慢，使得 closeSession 事务无法在 Observer 上有效执行，也使 NIO 连接接入层线程无法处理连接的数据接收和数据回复和建立新连接。





优化措施

限制或者抑制连接冲击。在故障时，根据 tcp 状态为 established 的连接数量动态限制连接，不过 established 的连接数量其未过阀值，但是观察到 fd 仍是满的，大部分连接处于 tcp 的 close-wait 状态，其中 fd 消耗过多，如果 Observer 落地日志的话，也会造成写 binlog 或 snapshot 失败导致进程异常退出。

4.3.6 initlimit 和 syncLimit 参数配置对集群和会话的影响

initLimit 参数

initLimit 是追随者最初连接到群首时的超时值，单位为 tick 值的倍数。当某个追随者最初与群首建立连接时，它们之间会传输相当多的数据，尤其是追随者落后整体很多时。配置 initLimit 参数值取决于群首与追随者之间的网络传输速度情况，以及传输的数据量大小。如果 ZooKeeper 中保存的数据量特别大时或者网络非常缓慢时，就需要增大 initLimit。

故障场景：在相同数据量的情况下，对于一个正常运行中的 3 节点主集群，如果一台 follower 重启或一台 observer 想要加入集群：initLimit 过小，会使这台机器无法加入主集群。

原因分析

ZooKeeper 的 3.4.4 版本的 observer/follower 启动时会读取一次 snapshot，在选举逻辑知道 leader 信息后，与 leader
quorum 端口(2001、2888）交互前，还会再读取一次 snapshot。

另外，initLimit 影响 leader 对 observer/follower 的 newLeaderAck（ZooKeeper3.4.4 或 3.4.6 版本），成员加入集群前，成员机器上会进行一次 snapshot 刷出，耗时如果过长，会使 leader 对 observer 或 follower 的的 newLeaderAck 读取超时（tickTime*initLimit）。如果此时正处理 leader 刚选举完要给一个 follower 同步数据的时候，还会导致 leader 不能及时收到足够数量的 newLeaderAck 而导致集群组建失败。

在 ZooKeePeer 的 3.5 版本后，初始化加载 snapshot 只会加载一次，不过需要同步的数据量比较大时，initLimit 还是要调大一些。

syncLimit 参数

syncLimit 是追随者与群首进行 sync 操作时的超时值，单位为 tick 值的倍数。

追随者总是会稍微落后于群首，但是因为服务器负载或者网络问题，就会导致追随者落后群首太多，甚至需要放弃该追随者，如果群首与追随者无法进行 sync 操作，而且超过了 syncLimit 的 tick 时间，就会放弃该追随者。

优化措施：

1. 测试追随者与群首的网络情况，进行规划配置，并实时监控集群数据量的变化。
2. 提高服务端的性能，网卡性能。

参考文献

- ZooKeeper-选举实现分析
- 《Apache ZooKeeper 官网》
- 《ZooKeeper github》
- 《zookeeper-分布式过程协同技术详解》【美】里德,【美】Flavio Junqueira 著
- ZooKeeper 源码分析
- 阿里巴巴为什么不用 ZooKeeper 做服务发现?
- ZooKeeper-选举实现分析
- ZooKeeper 源码分析

如何设计一个 A/B test?

作者： jiangeliu，PCG 开发工程师

实验设计

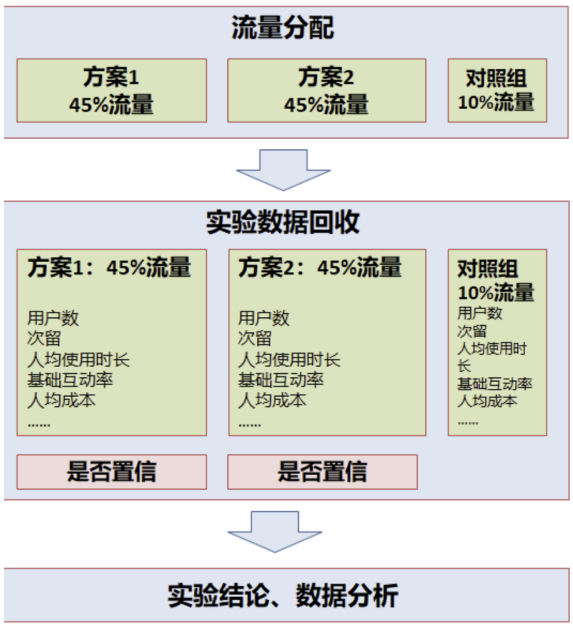
AB Test 实验一般有 2 个目的：

1. 判断哪个更好：例如，有 2 个 UI 设计，究竟是 A 更好一些，还是 B 更好一些，我们需要实验判定
2. 计算收益：例如，最近新上线了一个直播功能，那么直播功能究竟给平台带了来多少额外的 DAU，多少额外的使用时长，多少直播以外的视频观看时长等

我们一般比较熟知的是上述第 1 个目的，对于第 2 个目的，**对于收益的量化，计算 ROI，往往对数据分析师和管理者非常重要。**

对于一般的 ABTest 实验，其实本质上就是把平台的流量均匀分为几个组，每个组添加不同的策略，然后根据这几个组的用户数据指标，例如：留存、人均观看时长、基础互动率等等核心指标，最终选择一个最好的组上线。

实验的几个基本步骤一般如下：



如何分配流量

- 尽快得到实验结论，预期需要多少天
- 对体验影响最小

（此处暂假定实验平台分流是均匀的，分析角度，先不考虑AA Test）

实验效果

- 各项指标怎么样，可信吗？
- 哪个方案效果更好？
- 方案1、方案2的ROI更高？
- 有没有新奇效应的存在？
- 对于不同的群体，存在差异吗？

实验结论

- 定性定量判断
- 分析性结论和建议，人群、边际效应等
- 下一步还需要进行什么实验

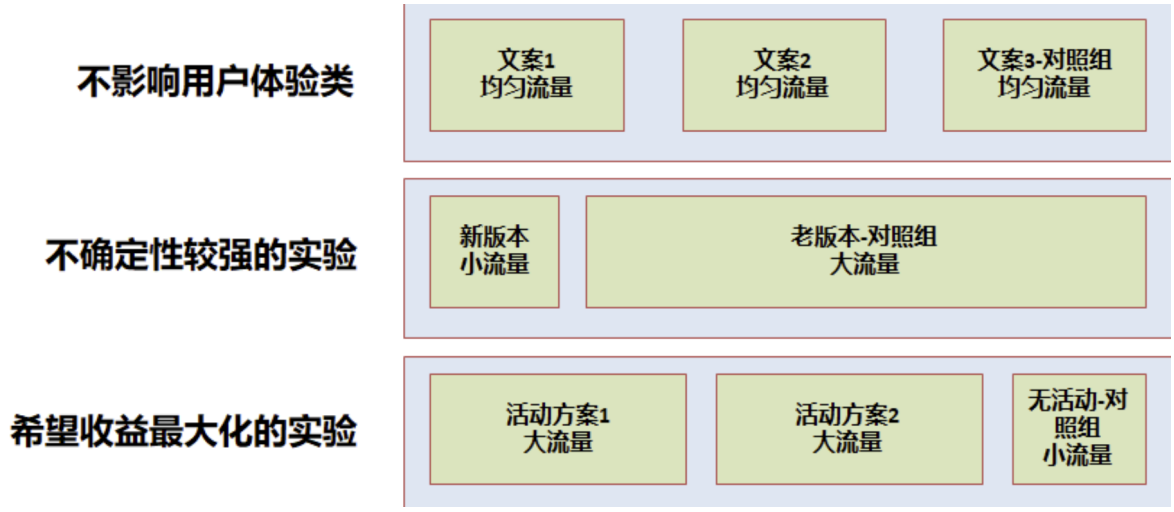
流量分配

实验设计时有两个目标：

- 希望尽快得到实验结论，尽快决策
- 希望收益最大化，用户体验影响最小

因此经常需要在流量分配时有所权衡，一般有以下几个情况：

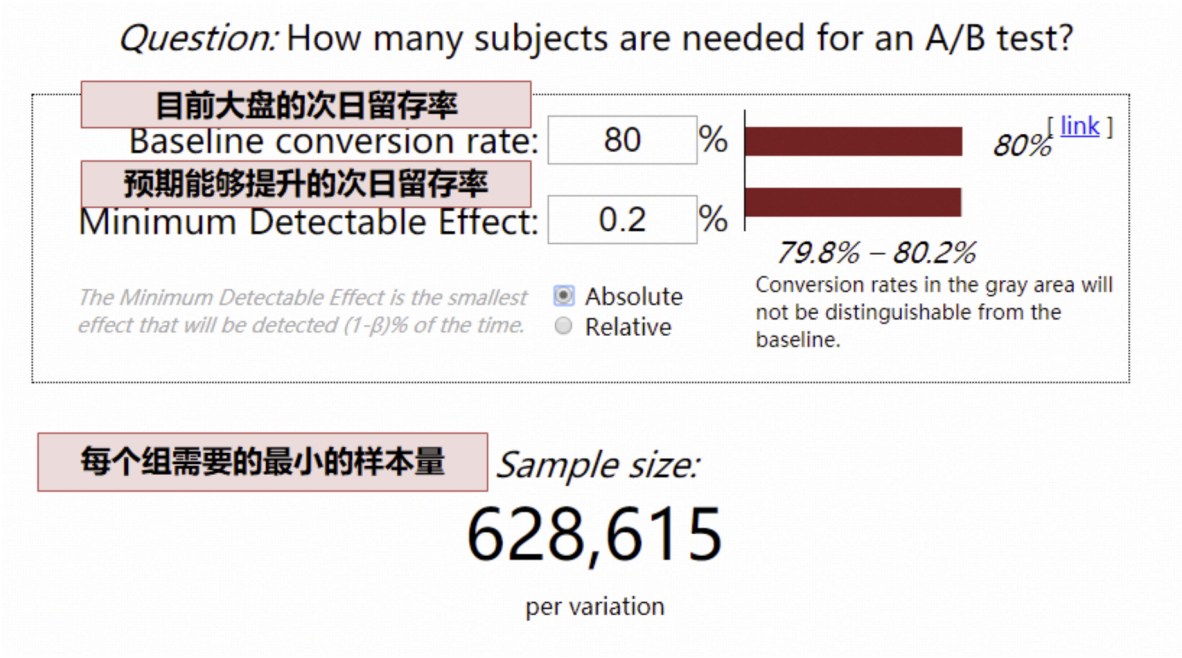
- **不影响用户体验：**如 UI 实验、文案类实验等，一般可以均匀分配流量实验，可以快速得到实验结论
- **不确定性较强的实验：**如产品新功能上线，一般需小流量实验，尽量减小用户体验影响，在允许的时间内得到结论
- **希望收益最大化的实验：**如运营活动等，尽可能将效果最大化，一般需要大流量实验，留出小部分对照组用于评估 ROI



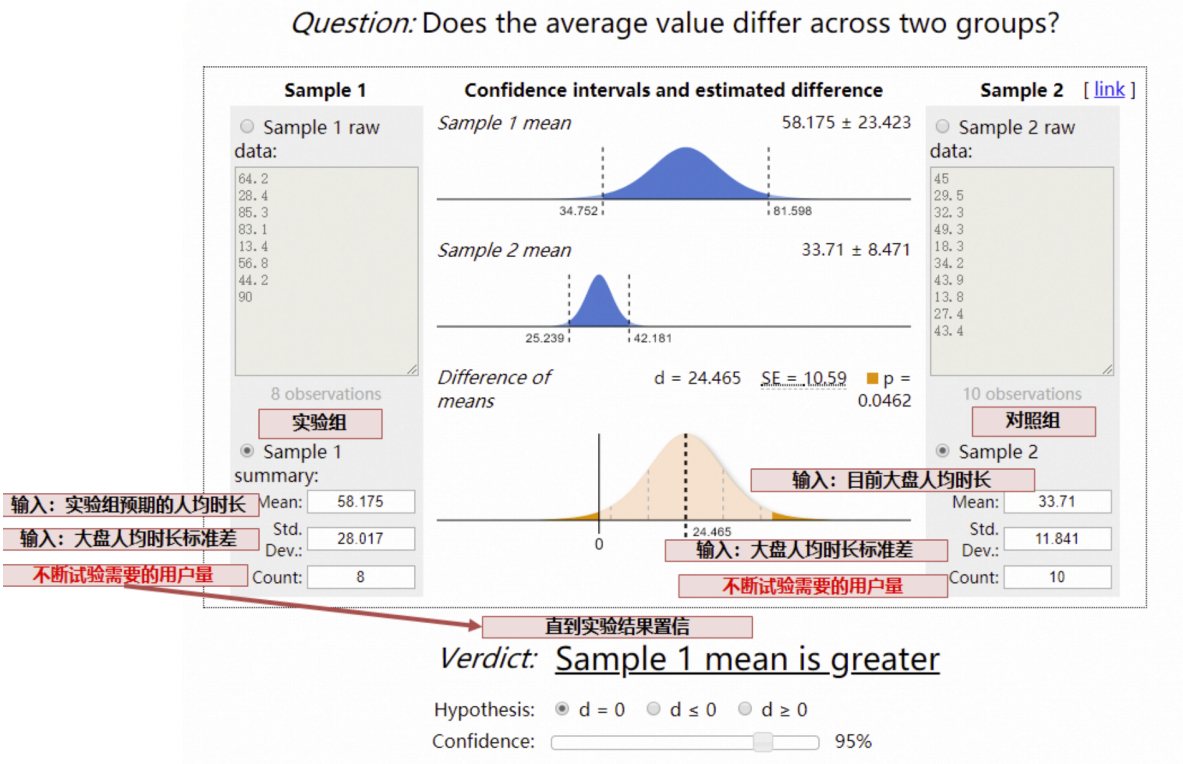
根据实验的预期结果，大盘用户量，确定实验所需最小流量，可以通过一个网站专门计算所需样本量：

- 以次日留存率为例，目前大盘次日留存率 80%，预期实验能够提升 0.2pp
  （这里的留存率可以转换为点击率、渗透率等等，只要是比例值就可以，**如果估不准，为了保证实验能够得到结果，此处可低估，不可高估，也就是 0.2pp 是预期能够提升地最小值**）
- 网站计算，最少样本量就是 63W
  （**这里的最少样本量，指的是最少流量实验组的样本量**）

- 如果我们每天只有 5W 的用户可用于实验（5W 的用户，指最少流量实验组是 5W 用户），63/ 5 = 13 天，我们需要至少 13 天才能够得到实验结论



如果我们预期提升的指标是人均时长、人均 VV 等，可能就比较复杂了，我们需要运用 t 检验反算，需要的样本量：



实验效果

我们以一个稍复杂点的运营活动实验为例，活动有方案 1、方案 2，同时为了量化 ROI，对照组没有运营活动。

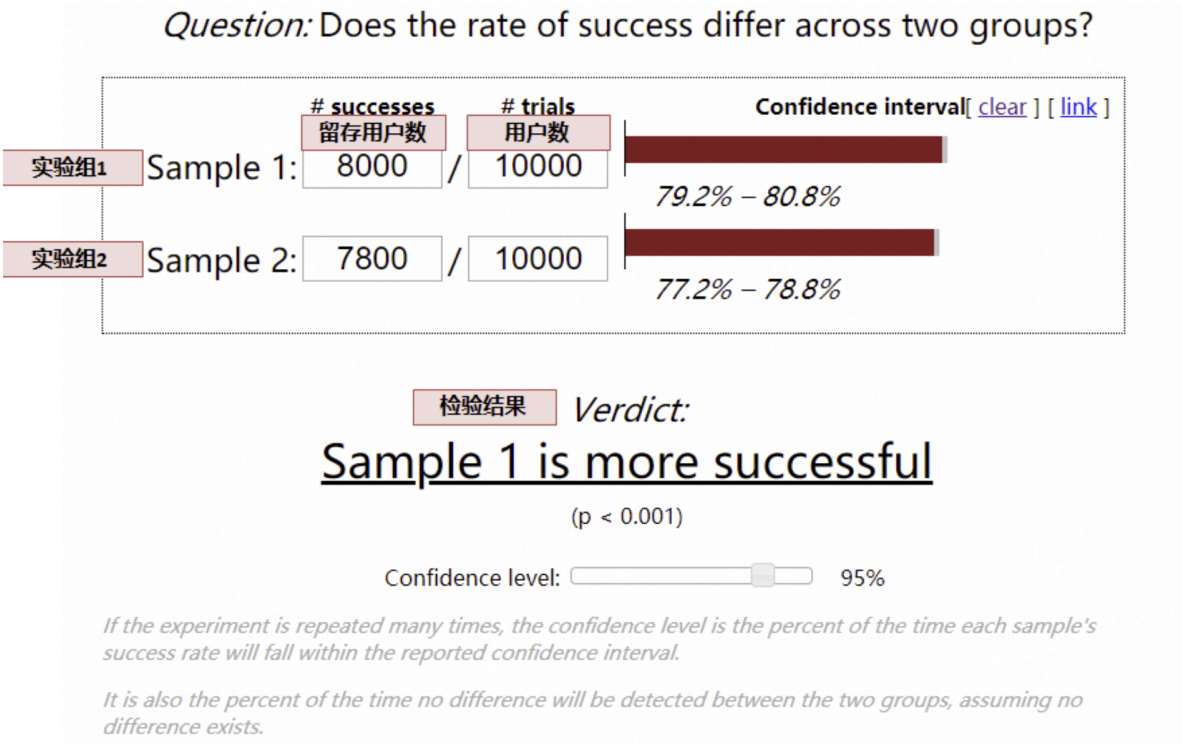


需要回答几个问题

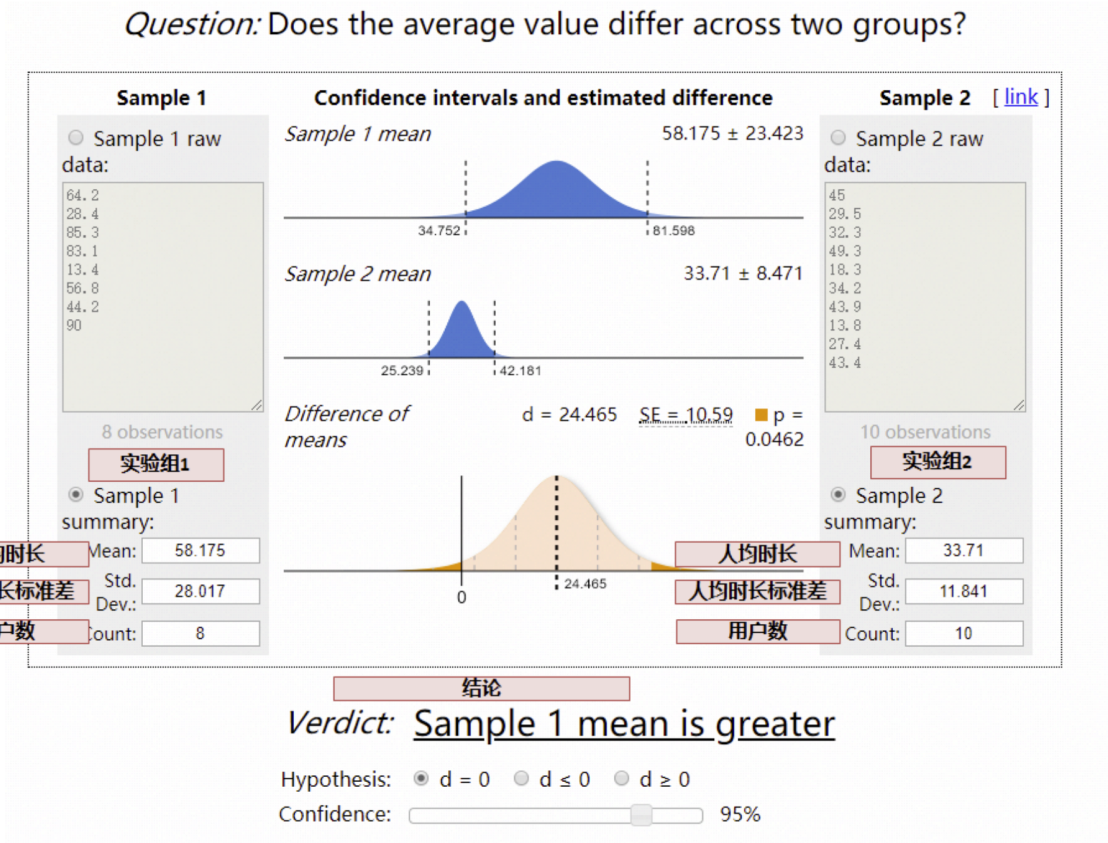
1. 方案 1 和方案 2，哪个效果更好?
2. 哪个 ROI 更高?
3. 长期来看哪个更好?
4. 不同群体有差异吗?

第 1 个问题，方案 1 和方案 2，哪个效果更好?

还是要运用假设检验，对于留存率、渗透率等漏斗类指标，采用卡方检验：



对于人均时长类等均值类指标，采用t 检验：



通过上假设检验，如果结论置信，我们就能够得到方案 1 和方案 2 在哪像指标更好（有显著性差异），对于不置信的结论，尽管方案 1 和方案 2 的指标可能略有差异，但可能是数据正常波动产生。

第 2 个问题，哪个 ROI 更高?

一般有活动相比无活动，留存、人均时长等各项指标均会显著，我们不再重复上述的假设检验过程。

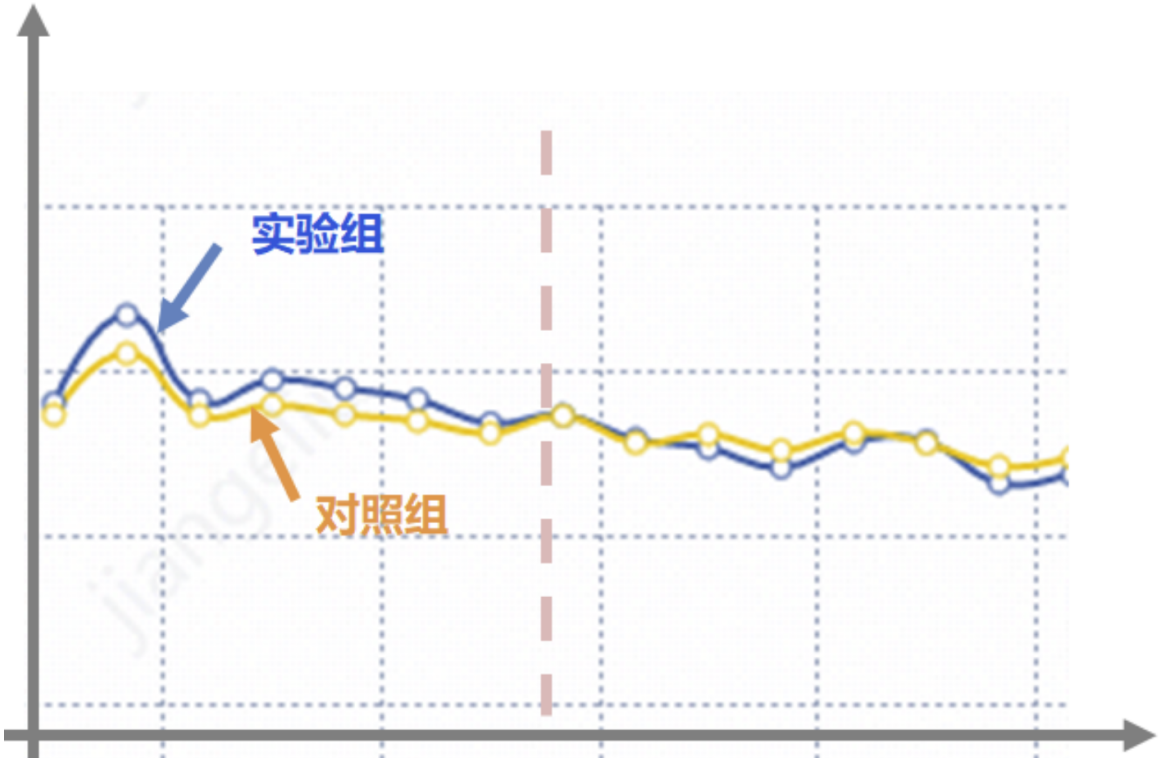
对于 ROI 的计算，成本方面，每个实验组成本可以直接计算，对于收益方面，就要和对照组相比较，假定以总日活跃天（即 DAU 按日累计求和）作为收益指标，需要假设不做运营活动，DAU 会是多少，可以通过对照组计算，即：

- 实验组假设不做活动日活跃天 = 对照组日活跃天 * （实验组流量 / 对照组流量）
- 实验组收益 = 实验组日活跃天 - 实验组假设不做活动日活跃天

这样就可以量化出每个方案的 ROI。

第 3 个问题，长期来看哪个更好?

这里就要考虑新奇效应的问题了，一般在实验上线前期，用户因为新鲜感，效果可能都不错，因此在做评估的时候，需要观测指标到稳定态后，再做评估。



例如有的时候出现，刚刚上线前期，实验组效果更好，但是经过以端时间，用户的新鲜感过去了，实验组的效果可能更差，因此，从长远收益来看，我们应该选择对照组，是实验组的新奇效应欺骗了我们，在做实验分析时，应剔除新奇效应的部分，待平稳后，再做评估

第 4 个问题，不同用户群体有差异吗?

很多情况下，对新用户可能实验组更好，老用户对照组更好；对年轻人实验组更好，中年人对照组更好，

作为数据分析师，分析实验结论时，还要关注用户群体的差异。

实验结束

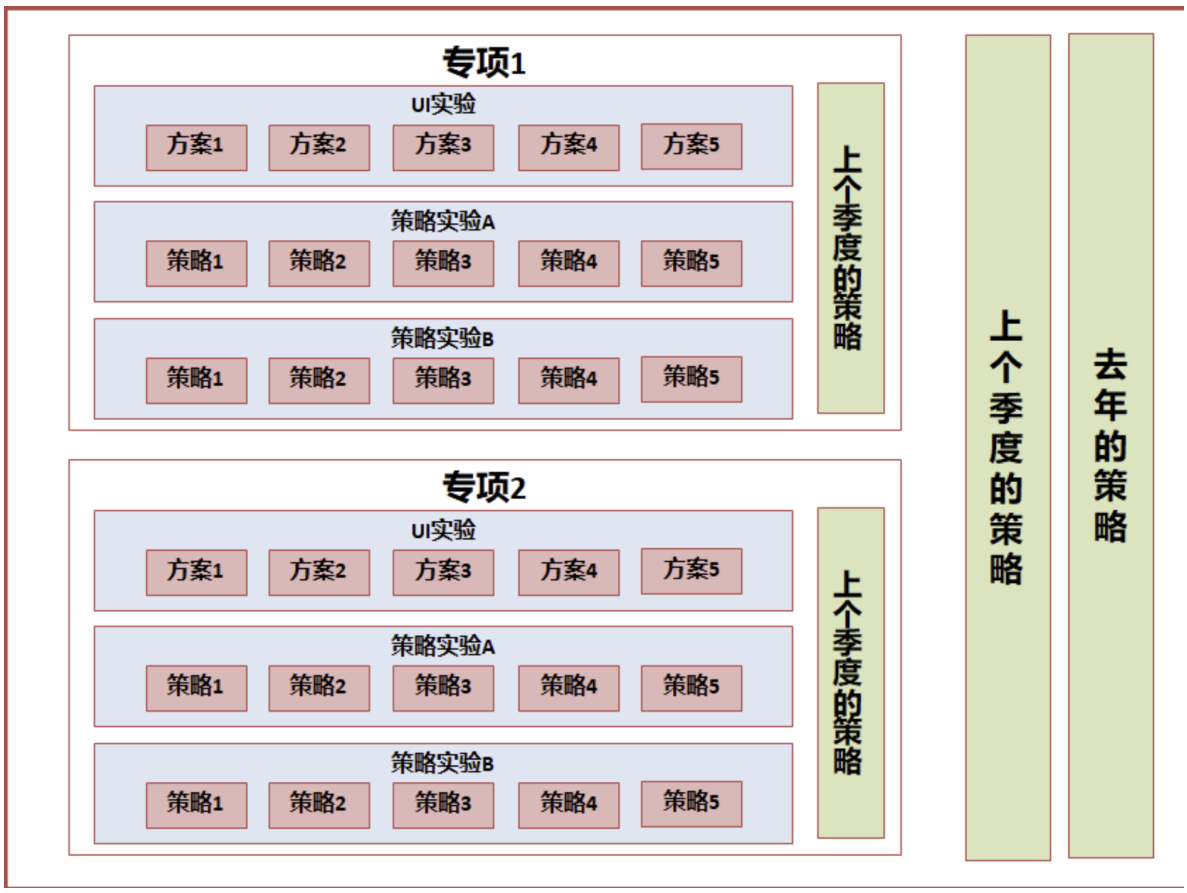
实验结束后需要：

- 反馈实验结论，包括直接效果（渗透、留存、人均时长等）、ROI
- 充分利用实验数据，进一步探索分析不同用户群体，不同场景下的差异，提出探索性分析
- 对于发现的现象，进一步提出假设，进一步实验论证

更高级的实验

对于长线业务，可能同时有数十个实验同时进行，不但对比每项小迭代的差异，同时对比专项对大盘的贡献量、部门整体对大盘的贡献量，这样就需要运用到了实验的层域管理模型。

- 对比每个产品细节迭代的结果
- 对比每个专项在一个阶段的贡献
- 对比整个项目在一个阶段的贡献



多个活动交集量化的实验设计

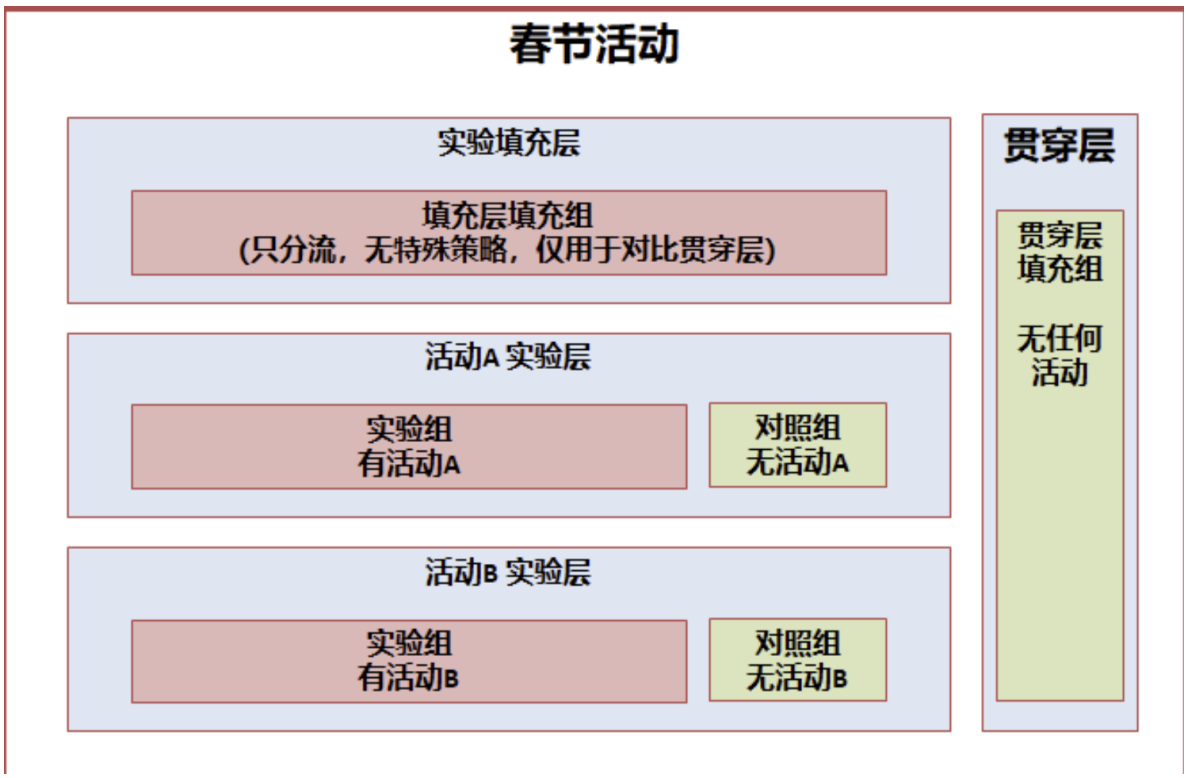
作为数据分析师，多团队合作中，经常遇到多业务交集的问题，以我近期主要负责的春节活动为例，老板会问：

- 春节活动-明星红包子活动贡献了多少 DAU？春节活动-家乡卡子活动贡献了多少 DAU？
- 春节活动总共贡献了多少 DAU？

严谨一点，我们采用了 AB 实验的方式核算，最终可能会发现一个问题：**春节活动各个子活动的贡献之和，不等于春节活动的贡献，为什么呢？**

- 有的时候，活动 A 和活动 B，有着相互放大的作用，这个时候就会 1+1 > 2
- 还有的时候，活动 A 和活动 B，本质上是在做相同的事情，这个时候就会 1+1 < 2

这个时候，我们准确量化春节活动的贡献，就需要一个【贯穿】所有活动的对照组，在 AB 实验系统中通俗称作**贯穿层。**



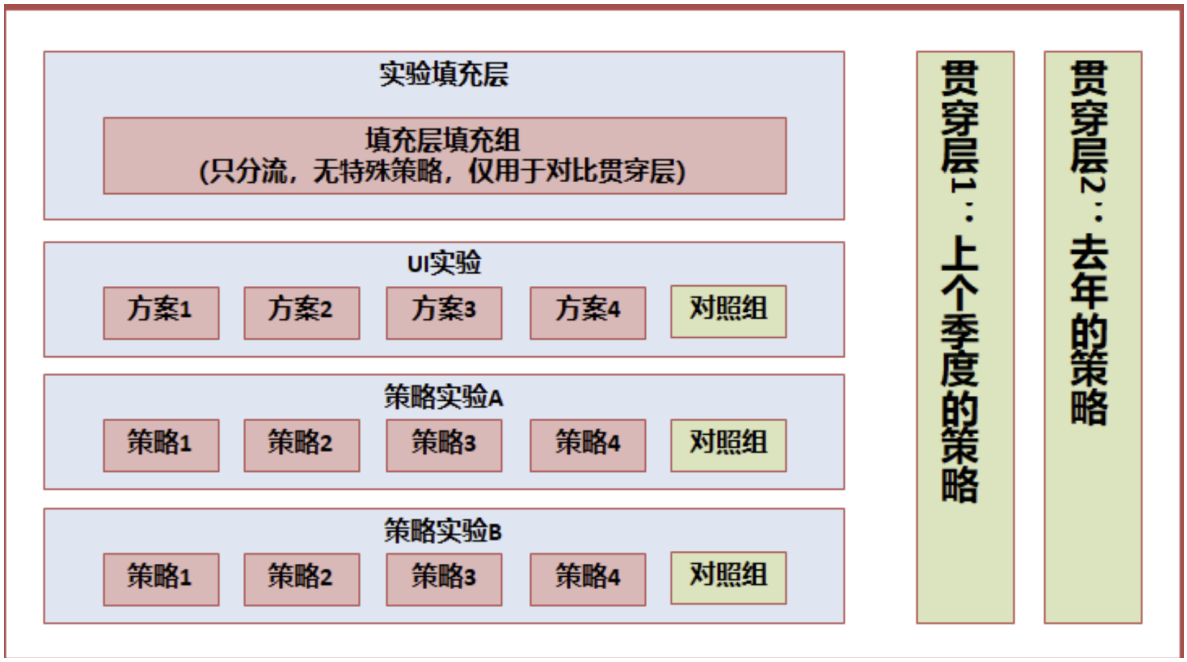
（说明：实验中，各层的**流量是正交的**，简单理解，例如，A 层的分流采用用户 ID 的倒数第 1 位，B 层的分流采用用户 ID 的倒数第 2 位，在用户 ID 随机的情况下，倒数第 1 位和倒数第 2 位是没有关系的，也称作相互**独立**，我们称作**正交**。当然，AB
Test 实验系统真实的分流逻辑，是采用了复杂的 hash 函数、正交表，能够保证正交性。）

这样分层后，我们可以按照**如下的方式量化贡献**：

- 计算春节活动的**整体**贡献：实验填充层-填充层填充组 **VS** 贯穿层-贯穿层填充组
- 计算活动 A 的贡献：活动 A 实验层中，实验组 VS 对照组
- 计算活动 B 的贡献：活动 B 实验层中，实验组 VS 对照组

业务迭代的同时，如何与自身的过去比较

上面谈到了【贯穿层】的设计，贯穿层的设计其实不但可以应用在多个活动的场景，有些场景，我们的**业务需要和去年或上个季度的自身对比，同时业务还不断在多个方面运用 AB Test 迭代。**



类似与上面这种层次设计，在推荐系统中较为常见，**在某一些产品或系统中，贯穿层不能够完全没有策略，那么采用去年或上个季度的策略，代表着基准值，从而量化新一个周期的增量贡献**

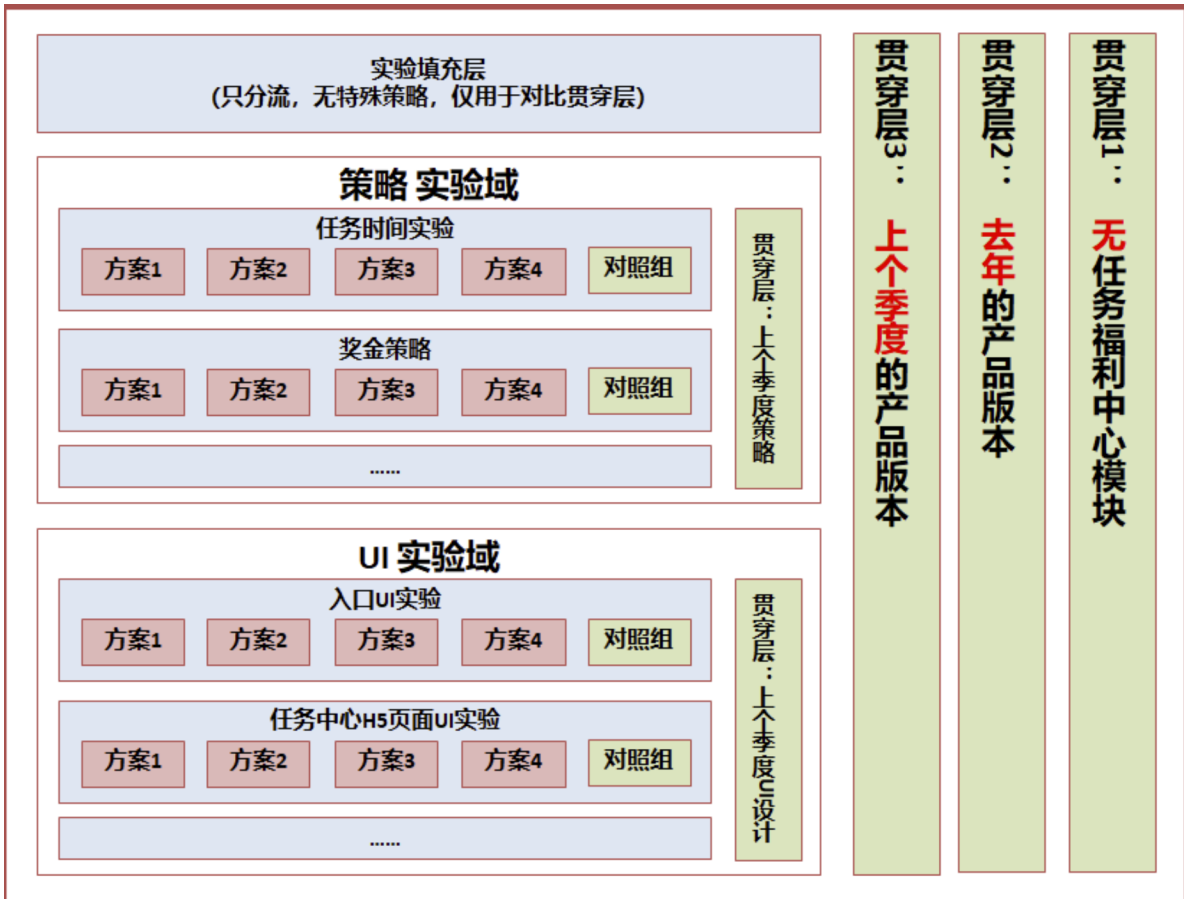
我们可以量化：

- 每个小迭代对整个系统的贡献：实验层中的实验组 VS 对照组
- 周期内，系统全部迭代与上个周期的比较：实验填充层 VS 贯穿层 1（或贯穿层 2）
- 同时，可以量化去年策略的自然增长或下降，以衡量旧有系统是否具有**长期的适用性**（作为系统设计者，更应鼓励设计具有长期适应性的系统）：贯穿层 1（上个季度的策略）VS 贯穿层 2（去年的策略）

更为复杂的实验设计

我以我目前负责的业务，微视任务福利中心的实验设计为例，举例一个更复杂的实验系统设计，综合了上面提到的 2 个目的：

- 量化**每一个实验迭代**为系统带来的增量贡献
- 量化**每一类迭代**（如 UI 迭代、策略迭代），在一个阶段的增量贡献
- 量化系统整体在上一个**周期**（季度、年）的**增量贡献**
- 量化任务福利中心的**整体 ROI**（本质上，是给用户一些激励，促进用户活跃，获得更多商业化收益，所以和推荐系统不同的是，需要有完全没有任务福利中心的对照组，用户量化 ROI）



数据库

深入理解 Mysql 索引底层原理

作者：junshili，腾讯 QQ 后台开发工程师

> 一步一步推导出 Mysql 索引的底层数据结构。

Mysql 作为互联网中非常热门的数据库，其底层的存储引擎和数据检索引擎的设计非常重要，尤其是 Mysql 数据的存储形式以及索引的设计，决定了 Mysql 整体的数据检索性能。

我们知道，索引的作用是做数据的快速检索，而快速检索的实现的本质是数据结构。通过不同数据结构的选择，实现各种数据快速检索。在数据库中，高效的查找算法是非常重要的，因为数据库中存储了大量数据，一个高效的索引能节省巨大的时间。比如下面这个数据表，如果 Mysql 没有实现索引算法，那么查找 id=7 这个数据，那么只能采取暴力顺序遍历查找，找到 id=7 这个数据需要比较 7 次，如果这个表存储的是 1000W 个数据，查找 id=1000W 这个数据那就要比较 1000W 次，这种速度是不能接受的。

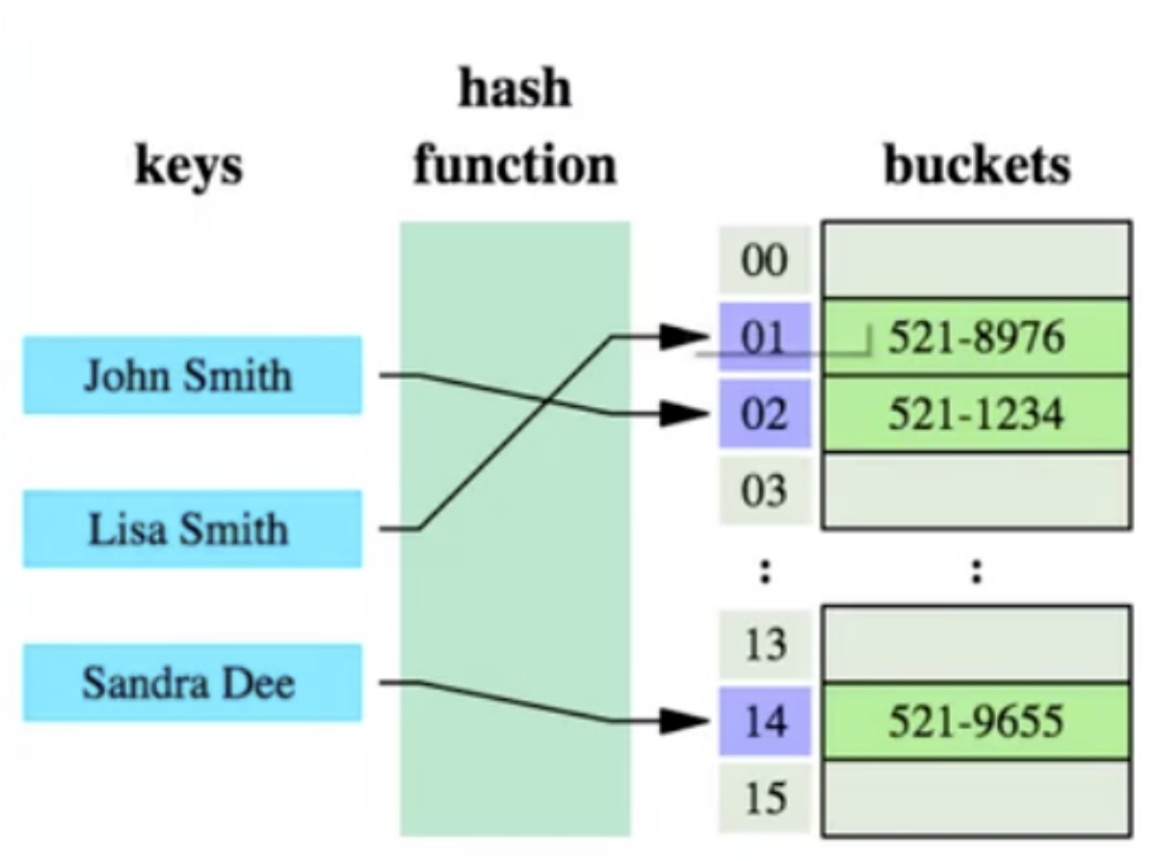
id(pk)	user_name
1	a
2	b
3	c
4	d
5	e
6	f
7	g

一、Mysql 索引底层数据结构选型

1. 哈希表（Hash）

哈希表是做数据快速检索的有效利器。

哈希算法：也叫散列算法，就是把任意值(key)通过哈希函数变换为固定长度的 key 地址，通过这个地址进行具体数据的数据结构。

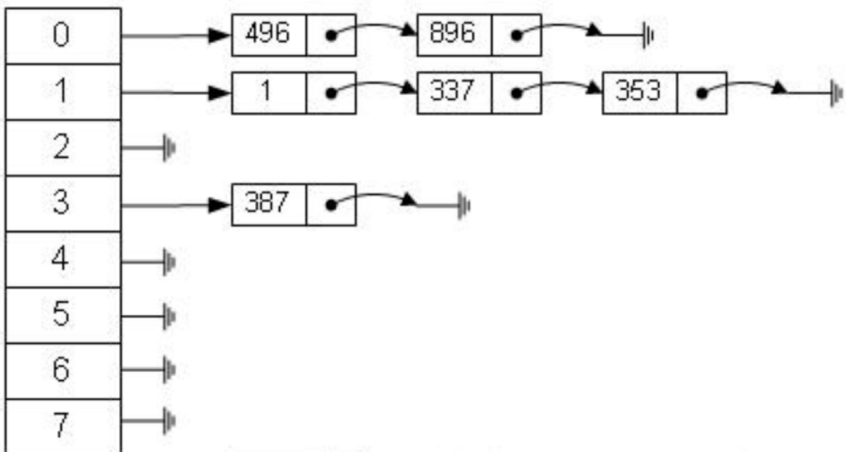


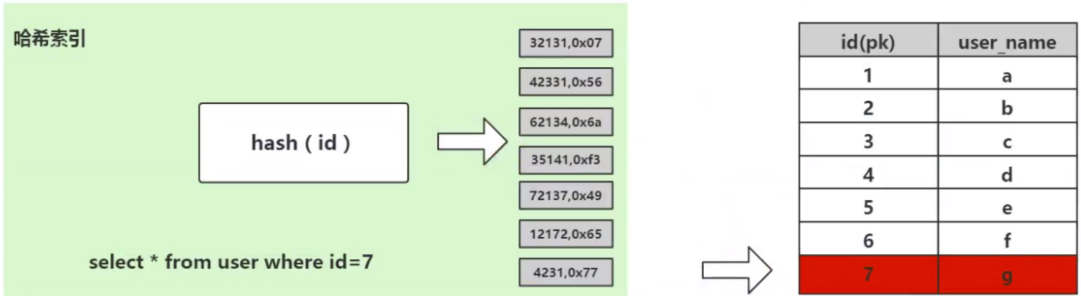
考虑这个数据库表 user，表中一共有 7 个数据，我们需要检索 id=7 的数据，SQL 语法是：

```
select \* from user where id=7;
```

哈希算法首先计算存储 id=7 的数据的物理地址 addr=hash(7)=4231，而 4231 映射的物理地址是 0x77，0x77 就是 id=7 存储的额数据的物理地址，通过该独立地址可以找到对应 user_name='g'这个数据。这就是哈希算法快速检索数据的计算过程。

但是哈希算法有个数据碰撞的问题，也就是哈希函数可能对不同的 key 会计算出同一个结果，比如 hash(7)可能跟 hash(199)计算出来的结果一样，也就是不同的 key 映射到同一个结果了，这就是碰撞问题。解决碰撞问题的一个常见处理方式就是链地址法，即用链表把碰撞的数据接连起来。计算哈希值之后，还需要检查该哈希值是否存在碰撞数据链表，有则一直遍历到链表尾，直达找到真正的 key 对应的数据为止。





从算法时间复杂度分析来看，哈希算法时间复杂度为 O（1），检索速度非常快。比如查找 id=7 的数据，哈希索引只需要计算一次就可以获取到对应的数据，检索速度非常快。但是 Mysql 并没有采取哈希作为其底层算法，这是为什么呢?

因为考虑到数据检索有一个常用手段就是范围查找，比如以下这个 SQL 语句：

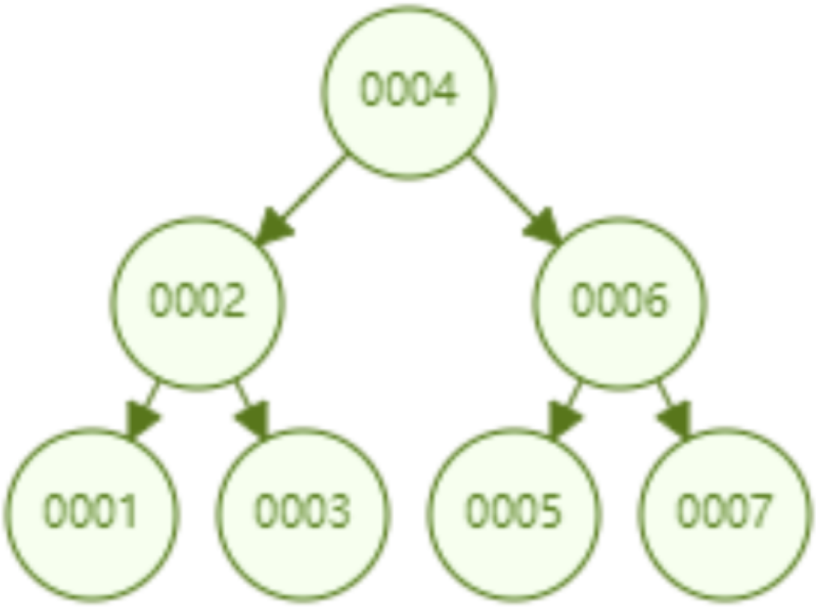
```
select \* from user where id \>3;
```

针对以上这个语句，我们希望做的是找出 id>3 的数据，这是很典型的范围查找。如果使用哈希算法实现的索引，范围查找怎么做呢？一个简单的思路就是一次把所有数据找出来加载到内存，然后再在内存里筛选筛选目标范围内的数据。但是这个范围查找的方法也太笨重了，没有一点效率而言。

所以，使用哈希算法实现的索引虽然可以做到快速检索数据，但是没办法做数据高效范围查找，因此哈希索引是不适合作为 Mysql 的底层索引的数据结构。

2. 二叉查找树(BST)

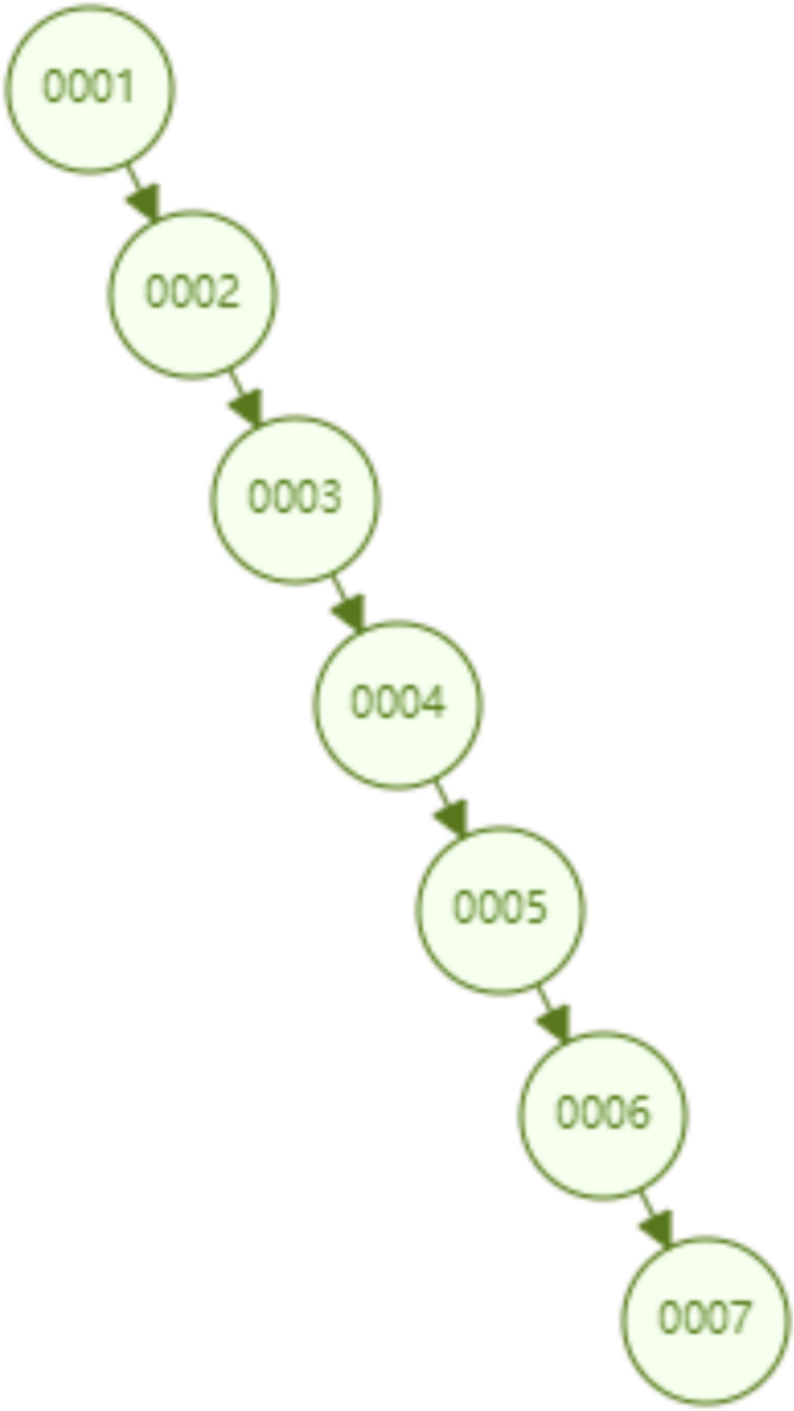
二叉查找树是一种支持数据快速查找的数据结构，如图下所示:



二叉查找树的时间复杂度是 O(lgn)，比如针对上面这个二叉树结构，我们需要计算比较 3 次就可以检索到 id=7 的数据，相对于直接遍历查询省了一半的时间，从检索效率上看来是能做到高速检索的。此外二叉树的结构能不能解决哈希索引不能提供的范围查找功能呢?

答案是可以的。观察上面的图，二叉树的叶子节点都是按序排列的，从左到右依次升序排列，如果我们需要找 id>5 的数据，那我们取出节点为 6 的节点以及其右子树就可以了，范围查找也算是比较容易实现。

但是普通的二叉查找树有个致命缺点：极端情况下会退化为线性链表，二分查找也会退化为遍历查找，时间复杂退化为 O（N），检索性能急剧下降。比如以下这个情况，二叉树已经极度不平衡了，已经退化为链表了，检索速度大大降低。此时检索 id=7 的数据的所需要计算的次数已经变为 7 了。



在数据库中，数据的自增是一个很常见的形式，比如一个表的主键是 id，而主键一般默认都是自增的，如果采取二叉树这种数据结构作为索引，那上面介绍到的不平衡状态导致的线性查找的问题必然出现。因此，简单的二叉查找树存在不平衡导致的检索性能降低的问题，是不能直接用于实现 Mysql 底层索引的。

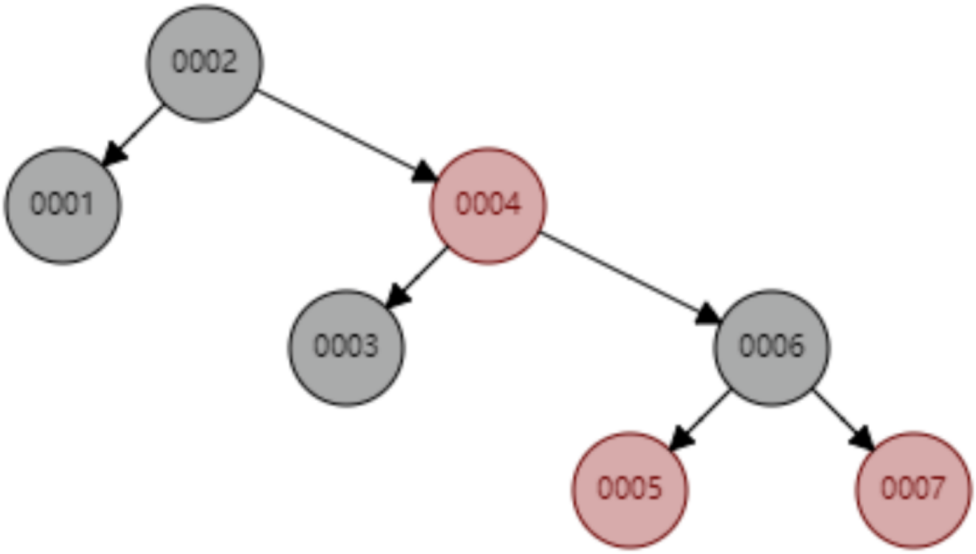
3. AVL 树和红黑树

二叉查找树存在不平衡问题，因此学者提出通过树节点的自动旋转和调整，让二叉树始终保持基本平衡的状态，就能保持二叉查找树的最佳查找性能了。基于这种思路的自调整平衡状态的二叉树有 AVL 树和红黑树。

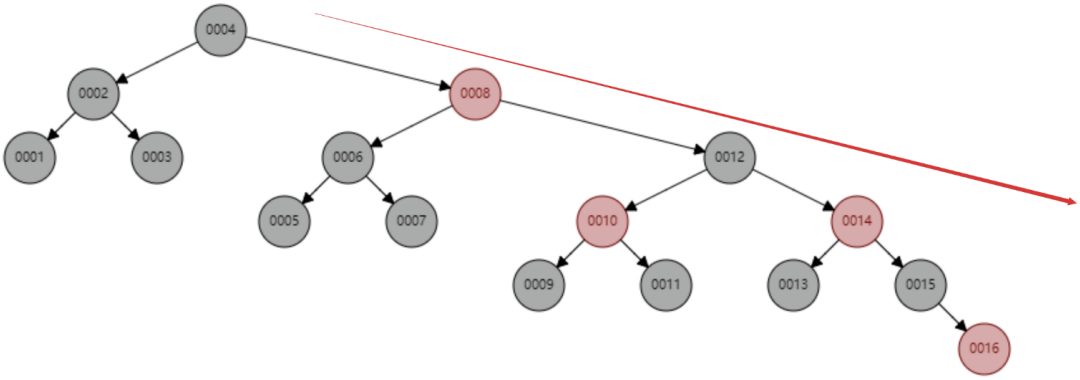
首先简单介绍红黑树，这是一颗会自动调整树形态的树结构，比如当二叉树处于一个不平衡状态时，红黑树就会自动左旋右旋节点以及节点变色，调整树的形态，使其保持基本的平衡状态（时间复杂度为 O（logn）），也就保证了查找效率不会明显减低。比如从 1 到 7 升序插入数据节点，如果是普通的二叉查找树则会退化成链表，但是红黑树则会不断调整树的形态，使其保持基本平衡状态，如下图所示。下面这个红黑树下查找 id=7 的所要比较的节点数为 4，依然保持二叉树不错的查找效率。

红黑树拥有不错的平均查找效率，也不存在极端的 O(n)情况，那红黑树作为 Mysql 底层索引实现是否可以呢？其实红黑树也存在一些问题，观察下面这个例子。

红黑树顺序插入 1~7 个节点，查找 id=7 时需要计算的节点数为 4。

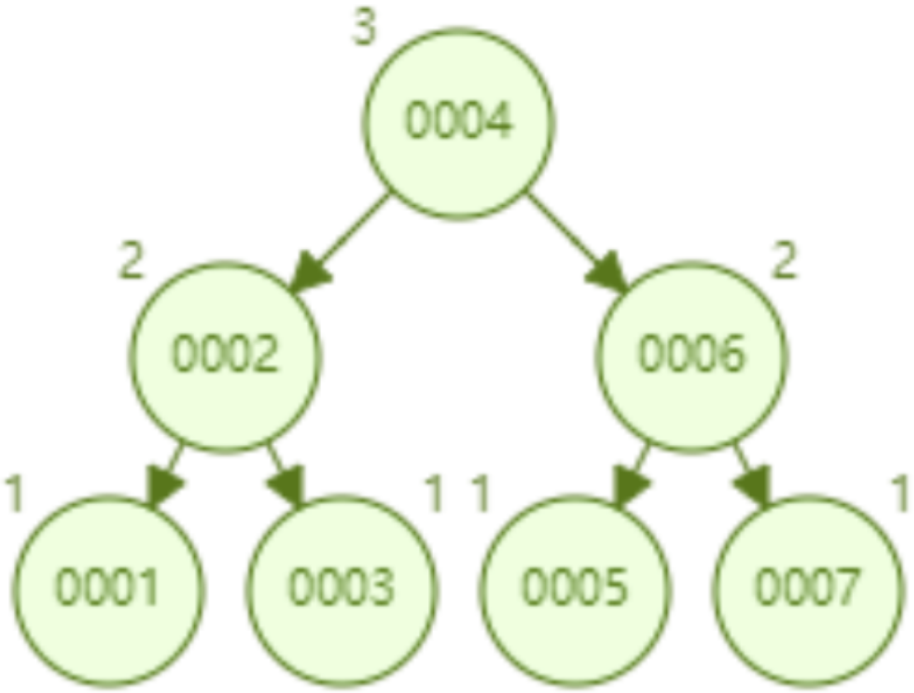


红黑树顺序插入 1~16 个节点，查找 id=16 需要比较的节点数为 6 次。观察一下这个树的形态，是不是当数据是顺序插入时，树的形态一直处于“右倾”的趋势呢？从根本上上看，红黑树并没有完全解决二叉查找树虽然这个“右倾”趋势远没有二叉查找树退化为线性链表那么夸张，但是数据库中的基本主键自增操作，主键一般都是数百万数千万的，如果红黑树存在这种问题，对于查找性能而言也是巨大的消耗，我们数据库不可能忍受这种无意义的等待的。

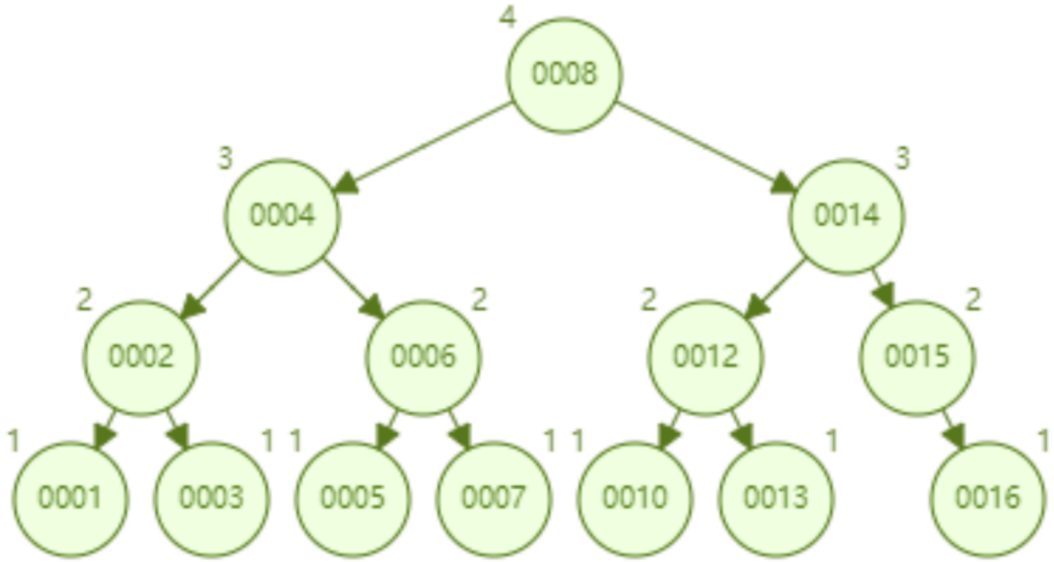


现在考虑另一种更为严格的自平衡二叉树 AVL 树。因为 AVL 树是个绝对平衡的二叉树，因此他在调整二叉树的形态上消耗的性能会更多。

AVL 树顺序插入 1~7 个节点，查找 id=7 所要比较节点的次数为 3。



AVL 树顺序插入 1~16 个节点，查找 id=16 需要比较的节点数为 4。从查找效率而言，AVL 树查找的速度要高于红黑树的查找效率（AVL 树是 4 次比较，红黑树是 6 次比较）。从树的形态看来，AVL 树不存在红黑树的“右倾”问题。也就是说，大量的顺序插入不会导致查询性能的降低，这从根本上解决了红黑树的问题。



总结一下 AVL 树的优点：

1. 不错的查找性能（O（logn）），不存在极端的低效查找的情况。
2. 可以实现范围查找、数据排序。

看起来 AVL 树作为数据查找的数据结构确实很不错，但是 AVL 树并不适合做 Mysql 数据库的索引数据结构，因为考虑一下这个问题：

数据库查询数据的瓶颈在于磁盘 IO，如果使用的是 AVL 树，我们每一个树节点只存储了一个数据，我们一次磁盘 IO 只能取出来一个节点上的数据加载到内存里，那比如查询 id=7 这个数据我们就要进行磁盘 IO 三次，这是多么消耗时间的。所以我们设计数据库索引时需要首先考虑怎么尽可能减少磁盘 IO 的次数。

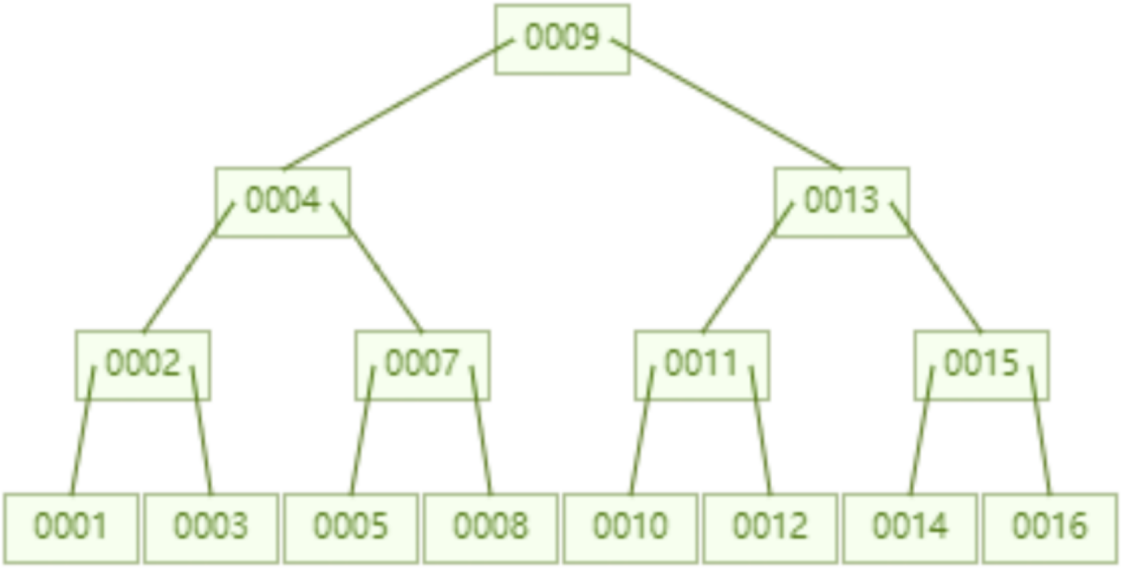
磁盘 IO 有个有个特点，就是从磁盘读取 1B 数据和 1KB 数据所消耗的时间是基本一样的，我们就可以根据这个思路，我们可以在一个树节点上尽可能多地存储数据，一次磁盘 IO 就多加载点数据到内存，这就是 B 树，B+树的的设计原理了。

4. B 树

下面这个 B 树，每个节点限制最多存储两个 key，一个节点如果超过两个 key 就会自动分裂。比如下面这个存储了 7 个数据 B 树，只需要查询两个节点就可以知道 id=7 这数据的具体位置，也就是两次磁盘 IO 就可以查询到指定数据，优于 AVL 树。



下面是一个存储了 16 个数据的 B 树，同样每个节点最多存储 2 个 key，查询 id=16 这个数据需要查询比较 4 个节点，也就是经过 4 次磁盘 IO。看起来查询性能与 AVL 树一样。



但是考虑到磁盘 IO 读一个数据和读 100 个数据消耗的时间基本一致，那我们的优化思路就可以改为：尽可能在一次磁盘 IO 中多读一点数据到内存。这个直接反映到树的结构就是，每个节点能存储的 key 可以适当增加。

当我们把单个节点限制的 key 个数设置为 6 之后，一个存储了 7 个数据的 B 树，查询 id=7 这个数据所要进行的磁盘 IO 为 2 次。



一个存储了 16 个数据的 B 树，查询 id=7 这个数据所要进行的磁盘 IO 为 2 次。相对于 AVL 树而言磁盘 IO 次数降低为一半。



所以数据库索引数据结构的选型而言，B 树是一个很不错的选择。总结来说，B 树用作数据库索引有以下优点：

1. 优秀检索速度，时间复杂度：B 树的查找性能等于 O（h*logn），其中 h 为树高，n 为每个节点关键词的个数；
2. 尽可能少的磁盘 IO，加快了检索速度；
3. 可以支持范围查找。
4. B+树

B 树和 B+树有什么不同呢?

第一，B 树一个节点里存的是数据，而 B+树存储的是索引（地址），所以 B 树里一个节点存不了很多个数据，但是 B+树一个节点能存很多索引，B+树叶子节点存所有的数据。

第二，B+树的叶子节点是数据阶段用了一个链表串联起来，便于范围查找。



通过 B 树和 B+树的对比我们看出，B+树节点存储的是索引，在单个节点存储容量有限的情况下，单节点也能存储大量索引，使得整个 B+树高度降低，减少了磁盘 IO。其次，B+树的叶子节点是真正数据存储的地方，叶子节点用了链表连接起来，这个链表本身就是有序的，在数据范围查找时，更具备效率。因此 Mysql 的索引用的就是 B+树，B+树在查找效率、范围查找中都有着非常不错的性能。

二、Innodb 引擎和 Myisam 引擎的实现

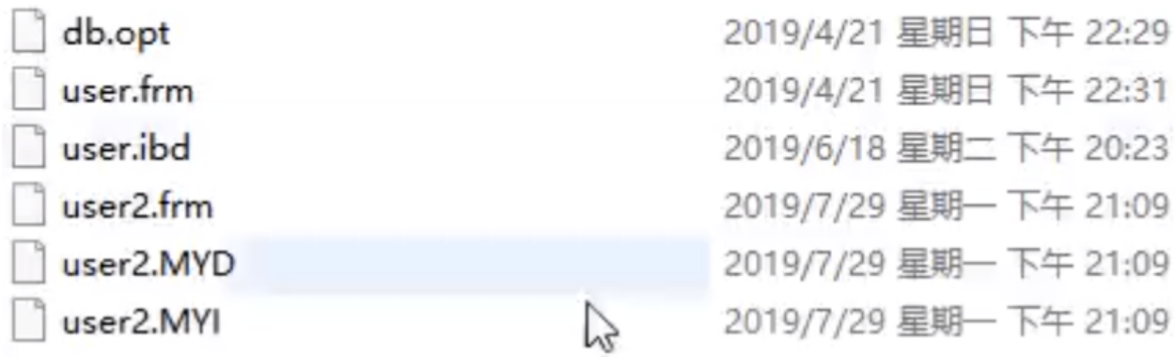
Mysql 底层数据引擎以插件形式设计，最常见的是 Innodb 引擎和 Myisam 引擎，用户可以根据个人需求选择不同的引擎作为 Mysql 数据表的底层引擎。我们刚分析了，B+树作为 Mysql 的索引的数据结构非常合适，但是数据和索引到底怎么组织起来也是需要一番设计，设计理念的不同也导致了 Innodb 和 Myisam 的出现，各自呈现独特的性能。

MyISAM 虽然数据查找性能极佳，但是不支持事务处理。Innodb 最大的特色就是支持了 ACID 兼容的事务功能，而且他支持行级锁。Mysql 建立表的时候就可以指定引擎，比如下面的例子，就是分别指定了 Myisam 和 Innodb 作为 user 表和 user2 表的数据引擎。

```
CREATE TABLE `user2` (
  `id` int(11) NOT NULL DEFAULT '0',
  `username` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=myisam DEFAULT CHARSET=utf8;

CREATE TABLE `user` (
  `id` int(11) NOT NULL DEFAULT '0',
  `username` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

执行这两个指令后，系统出现了以下的文件，说明这两个引擎数据和索引的组织方式是不一样的。



Innodb 创建表后生成的文件有：

- frm:创建表的语句
- idb:表里面的数据+索引文件

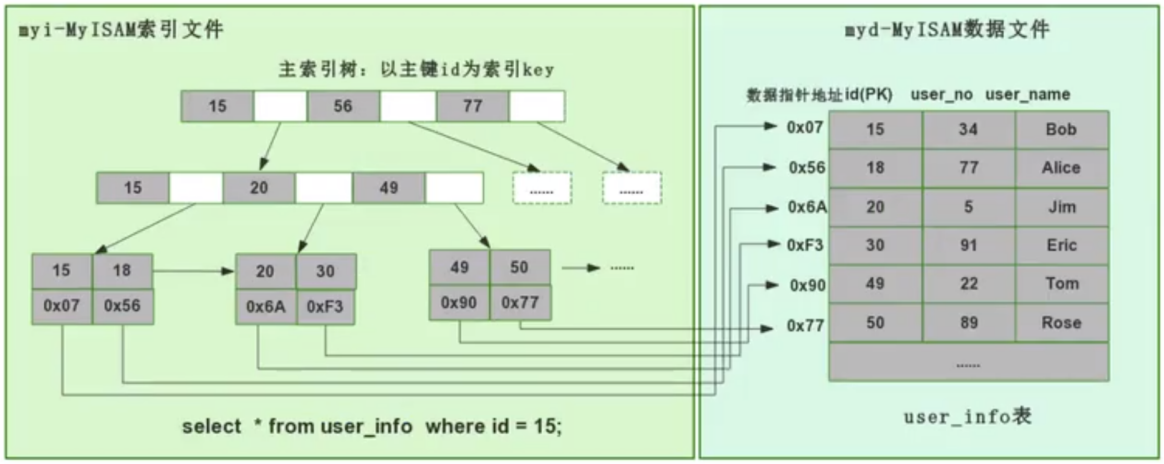
Myisam 创建表后生成的文件有

- frm:创建表的语句
- MYD:表里面的数据文件（myisam data）
- MYI:表里面的索引文件（myisam index）

从生成的文件看来，这两个引擎底层数据和索引的组织方式并不一样，MyISAM 引擎把数据和索引分开了，一人一个文件，这叫做非聚集索引方式；Innodb 引擎把数据和索引放在同一个文件里了，这叫做聚集索引方式。下面将从底层实现角度分析这两个引擎是怎么依靠 B+树这个数据结构来组织引擎实现的。

1. MyISAM 引擎的底层实现（非聚集索引方式）

MyISAM 用的是非聚集索引方式，即数据和索引落在不同的两个文件上。MyISAM 在建表时以主键作为 KEY 来建立主索引 B+树，树的叶子节点存的是对应数据的物理地址。我们拿到这个物理地址后，就可以到 MyISAM 数据文件中直接定位到具体的数据记录了。

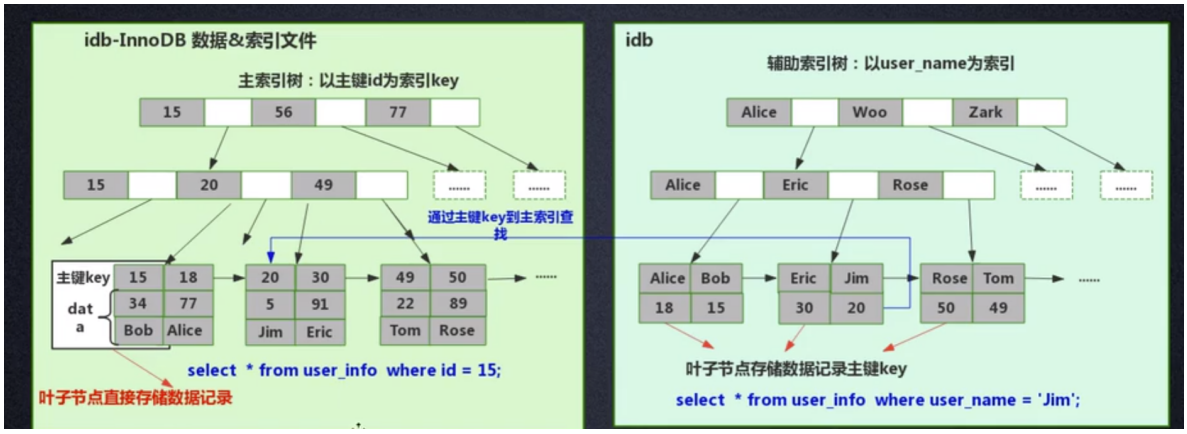


当我们为某个字段添加索引时，我们同样会生成对应字段的索引树，该字段的索引树的叶子节点同样是记录了对应数据的物理地址，然后也是拿着这个物理地址去数据文件里定位到具体的数据记录。

2. Innodb 引擎的底层实现（聚集索引方式）

InnoDB 是聚集索引方式，因此数据和索引都存储在同一个文件里。首先 InnoDB 会根据主键 ID 作为 KEY 建立索引 B+树，如左下图所示，而 B+树的叶子节点存储的是主键 ID 对应的数据，比如在执行 select * from user_info where
id=15 这个语句时，InnoDB 就会查询这颗主键 ID 索引 B+树，找到对应的 user_name='Bob'。

这是建表的时候 InnoDB 就会自动建立好主键 ID 索引树，这也是为什么 Mysql 在建表时要求必须指定主键的原因。当我们为表里某个字段加索引时 InnoDB 会怎么建立索引树呢？比如我们要给 user_name 这个字段加索引，那么 InnoDB 就会建立 user_name 索引 B+树，节点里存的是 user_name 这个 KEY，叶子节点存储的数据的是主键 KEY。注意，叶子存储的是主键 KEY！拿到主键 KEY 后，InnoDB 才会去主键索引树里根据刚在 user_name 索引树找到的主键 KEY 查找到对应的数据。



问题来了，为什么 InnoDB 只在主键索引树的叶子节点存储了具体数据，但是其他索引树却不存具体数据呢，而要多此一举先找到主键，再在主键索引树找到对应的数据呢?

其实很简单，因为 InnoDB 需要节省存储空间。一个表里可能有很多个索引，InnoDB 都会给每个加了索引的字段生成索引树，如果每个字段的索引树都存储了具体数据，那么这个表的索引数据文件就变得非常巨大（数据极度冗余了）。从节约磁盘空间的角度来说，真的没有必要每个字段索引树都存具体数据，通过这种看似"多此一举"的步骤，在牺牲较少查询的性能下节省了巨大的磁盘空间，这是非常有值得的。

在进行 InnoDB 和 MyISAM 特点对比时谈到，MyISAM 查询性能更好，从上面索引文件数据文件的设计来看也可以看出原因：MyISAM 直接找到物理地址后就可以直接定位到数据记录，但是 InnoDB 查询到叶子节点后，还需要再查询一次主键索引树，才可以定位到具体数据。等于 MyISAM 一步就查到了数据，但是 InnoDB 要两步，那当然 MyISAM 查询性能更高。

本文首先探讨了哪种数据结构更适合作为 Mysql 底层索引的实现，然后再介绍了 Mysql 两种经典数据引擎 MyISAM 和 InnoDB 的底层实现。最后再总结一下什么时候需要给你的表里的字段加索引吧：

1. 较频繁的作为查询条件的字段应该创建索引；
2. 唯一性太差的字段不适合单独创建索引，即使该字段频繁作为查询条件；
3. 更新非常频繁的字段不适合创建索引。

Clickhouse 在腾讯的应用实践

作者： shawnding 、brianxiong

今天给大家介绍 Clickhouse 在腾讯的应用实践， 主要分为两部分内容：

- Clickhouse 的部署和管理
- Clickhouse 在腾讯业务线的应用实践今天给大家介绍 Clickhouse 在腾讯的应用实践， 主要分为两部分内容：
    - Clickhouse 的部署和管理
    - Clickhouse 在腾讯业务线的应用实践

Clickhouse 它自身是一个非常强大的数据处理的引擎，因为它非常专注数据处理的计算效率这一块，因此它周边的一些管理插件，其实还是比较弱的。

全文链接：https://mp.weixin.qq.com/s/nwBaChd-PQvKHzVtE6GcSw

深入浅出百亿请求高可用Redis(codis)分布式集群揭秘

作者：jackiej

作为noSql中的kv数据库的王者，redis以其高性能，低时延，丰富的数据结构备受开发者青睐，但是由于redis在水平伸缩性上受限，如何做到能够水平扩容，同时对业务无侵入性是很多使用redis的开发人员都会面临的问题，而redis分布式解决方案的一个开源产品【codis】较好的弥补了这一弱势，本文主要讲解codis是如何做到对业务无感知，平滑迁移，迁移性能高，迁移异常处理，高可用以及常见的redis的避坑指南，虽然codis目前随着公司的nosql产品越来越成熟，生命周期也即将结束，不过鉴于还有很多同学对codis的原理比较感兴趣，于是将以前的分享的内容重新整理，当然codis在公司外应用目前依旧还是相对比较广泛。作为noSql中的kv数据库的王者，redis以其高性能，低时延，丰富的数据结构备受开发者青睐，但是由于redis在水平伸缩性上受限，如何做到能够水平扩容，同时对业务无侵入性是很多使用redis的开发人员都会面临的问题，而redis分布式解决方案的一个开源产品【codis】较好的弥补了这一弱势，本文主要讲解codis是如何做到对业务无感知，平滑迁移，迁移性能高，迁移异常处理，高可用以及常见的redis的避坑指南，虽然codis目前随着公司的nosql产品越来越成熟，生命周期也即将结束，不过鉴于还有很多同学对codis的原理比较感兴趣，于是将以前的分享的内容重新整理，当然codis在公司外应用目前依旧还是相对比较广泛。

全文链接：https://mp.weixin.qq.com/s/F68-e2umTQIq0aGfif58ow

SuperSQL：跨数据源、跨DC、跨执行引擎的高性能大数据SQL中间件

SuperSQL是一款腾讯自研的**跨数据源、跨数据中心、跨执行引擎**的高性能大数据SQL中间件，满足对位于不同数据中心的不同类型数据源的数据联合分析/即时查询的需求。SuperSQL的目标是成为公司内部统一的SQL分析中间件，实现以下三点的价值：

- 解决业务数据孤岛，最大化数据的使用价值
- 执行引擎最优选择，提升业务使用数据效率
- 优化集群资源使用，解决业务资源使用瓶颈

全文链接：https://mp.weixin.qq.com/s/ck7roZFZ9_u7fUf8TDH4eA

效率工具

把 VSCode 打造成 C++ 开发利器

作者： denniszhu ， 腾讯 CDG 开发工程师

> 面对大量代码，在开发任务繁重场景下，VSCode 绝对是一把利器。本文虽以 C++为引，但在 python、php、java 等中后台语言方面支持也很强大，对 js、css 等前端技术支持上更加厉害。简言之，只需安装一个 IDE（VSCode），就能解决你在大型项目多语言、任务繁重、跨平台场景下的开发需求。
>
> 本文以介绍配置方法为主，虽看似长，但实际在 30 分钟内绝对完成。一招在手，天下我有。

1. 为什么是 VSCode

1.1. 日常开发对 IED 功能的需求

- **支持多语言，比如 C++、JAVA、PYTHON、JS、PHP 等等；基本日常用到的希望不用开多种 IDE 就能解决；**
- 支持 vim、sublime、emacs、eclipse 等各种其他编辑器的快捷键。
- 支持多种编码方式。代码有 gbk 和 utf8 两种，希望 ide 能自动识别。
- 代码自动补全、函数列表、函数引用、函数定义、查到、函数跳转、函数跳回 都比较好用。
- 支持上传到 sftp
- 支持 svn、git
- 支持 markdown
- ide 能显示文件全部路径（近期接触到的代码，不同文件夹下同名文件太多）
- 主题好看点，时不时换个主题换个心情
- ide 上手快，配置简单，跨平台；
- 占用内存少，不卡。轻便灵活
- 支持调试代码。
- 支持跨平台使用，windows、ubuntu 等操作系统
- 使用人数？我们研发中心都在使用 VSCode

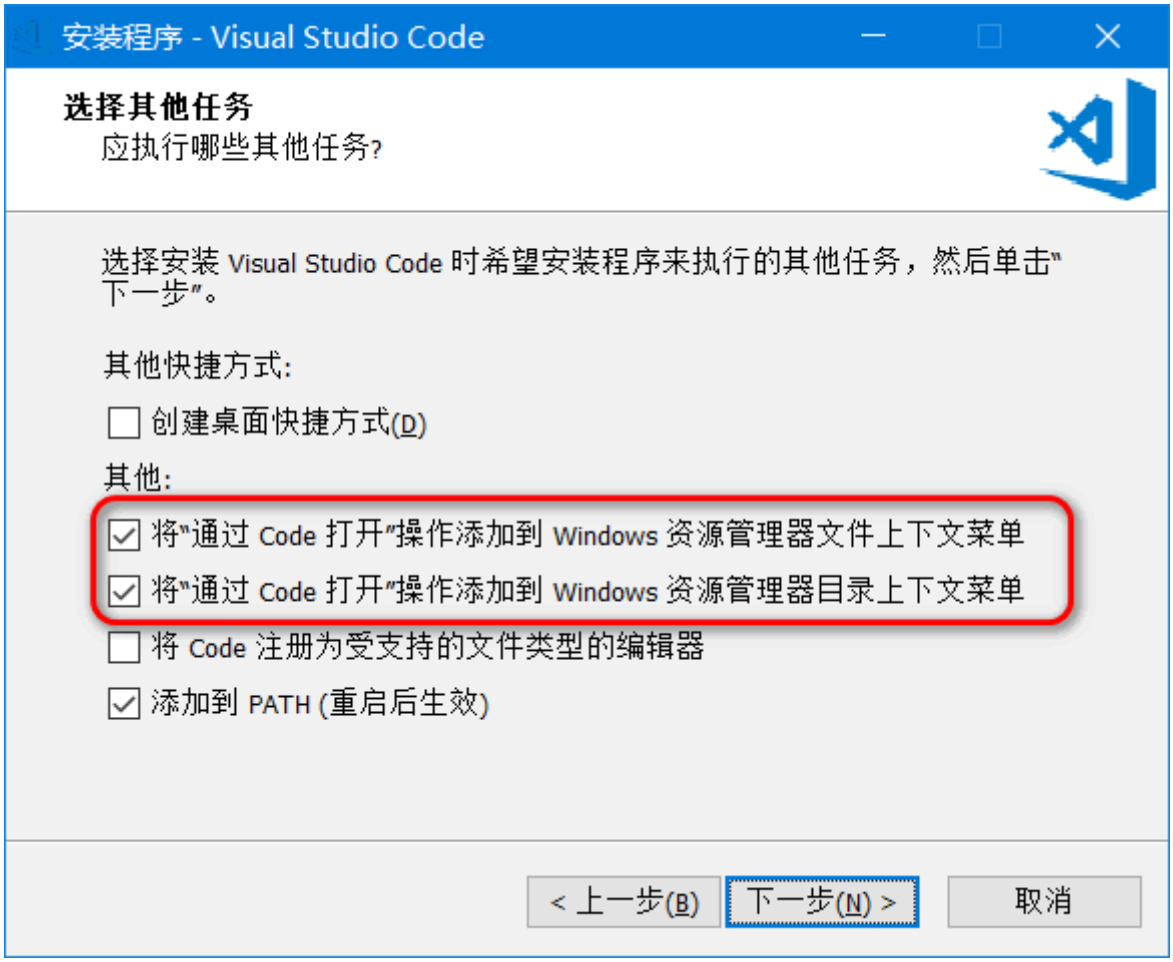
使用过的各种 IDE 对比

- sourceinsight，不支持 gbk
- sourceinsight + beyond compare.sourceinsight 上述功能在 4 支持不好，对于长路径会被省略掉。对于同名文件很多的项目，看代码很困难。
- vim 对于大规模写代码，操作还是不很熟练。
- eclipse 等虽然也能配置 C++，JAVA，但太过笨重，多开几个窗口后，机器基本就废了。
- VSCode 完全满足需求。√

2.安装 VSCode

2.1. 下载安装包

去 VSCode 官网 下载 windows 版本即可，安装时记得勾选将 VSCode 添加至右键菜单选项，**最好下面的勾全部勾上。**



如果忘记勾选，也可参考下文设置，但最好建议 **重装 VSCode!!!**

参考：VSCode 添加右键快捷打开方式

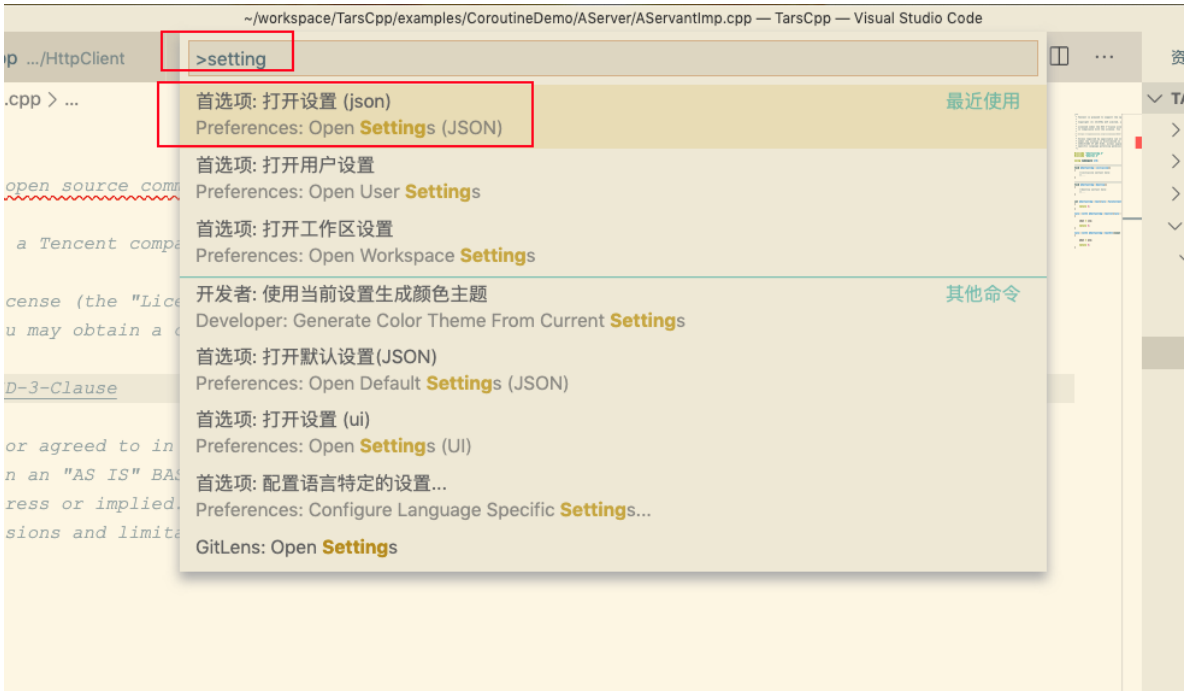
3. 设置 settings.json 文件

3.1. 打开设置 json 文件

VSCode 的设置采用设置面板的形式，避免大家修改时出现错误。但对于开发人员来讲，直接设置 json 配置文件方式更方便容易。操作路径是：用户首选项-设置。但不推荐。

目前最新版本（1.40.2+）有意隐藏了 settings.json 配置文件的打开方式，需要手动打开。

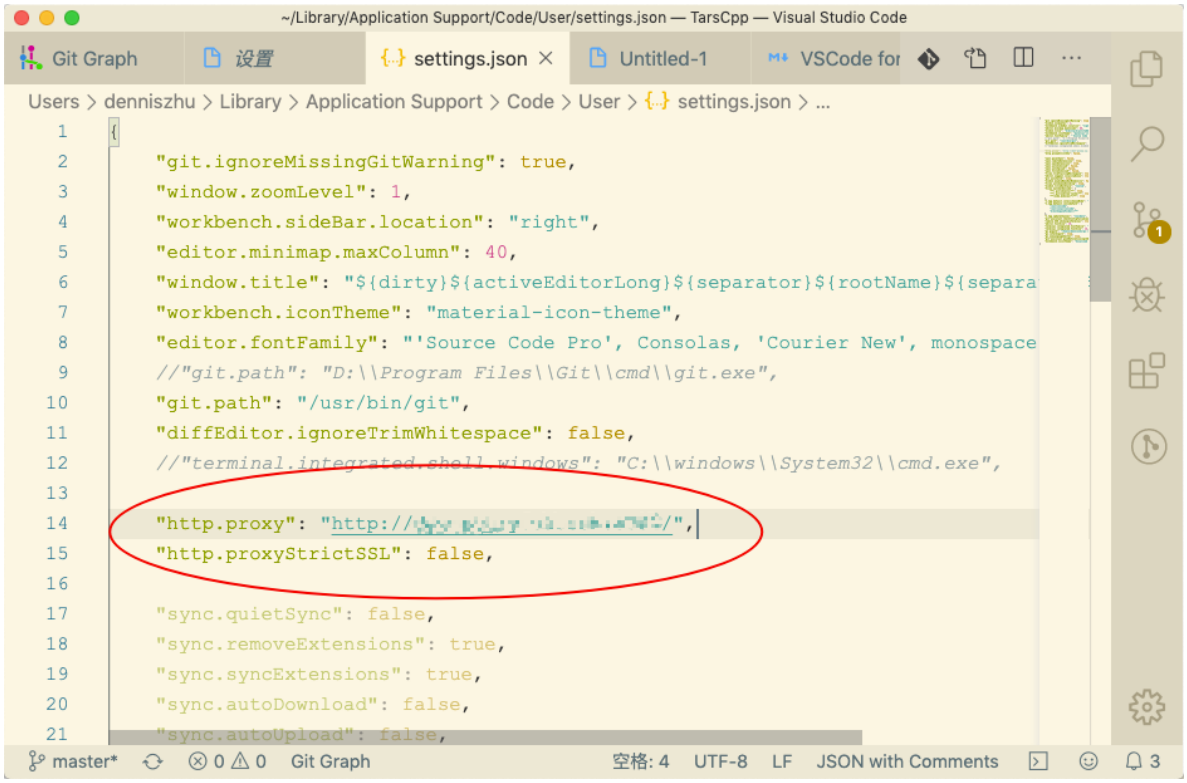
在 VSCode 中使用 `Ctrl+Shift+P` 打开命令面板，简单输入 `settings`，选择**首选项：打开设置 (json)**，即可打开 settings.json 文件：



3.2 设置公司代理

开发网下为了能下载插件，需要设置代理，设置方式是在`settings.json`文件中配置如下内容
方式如下：

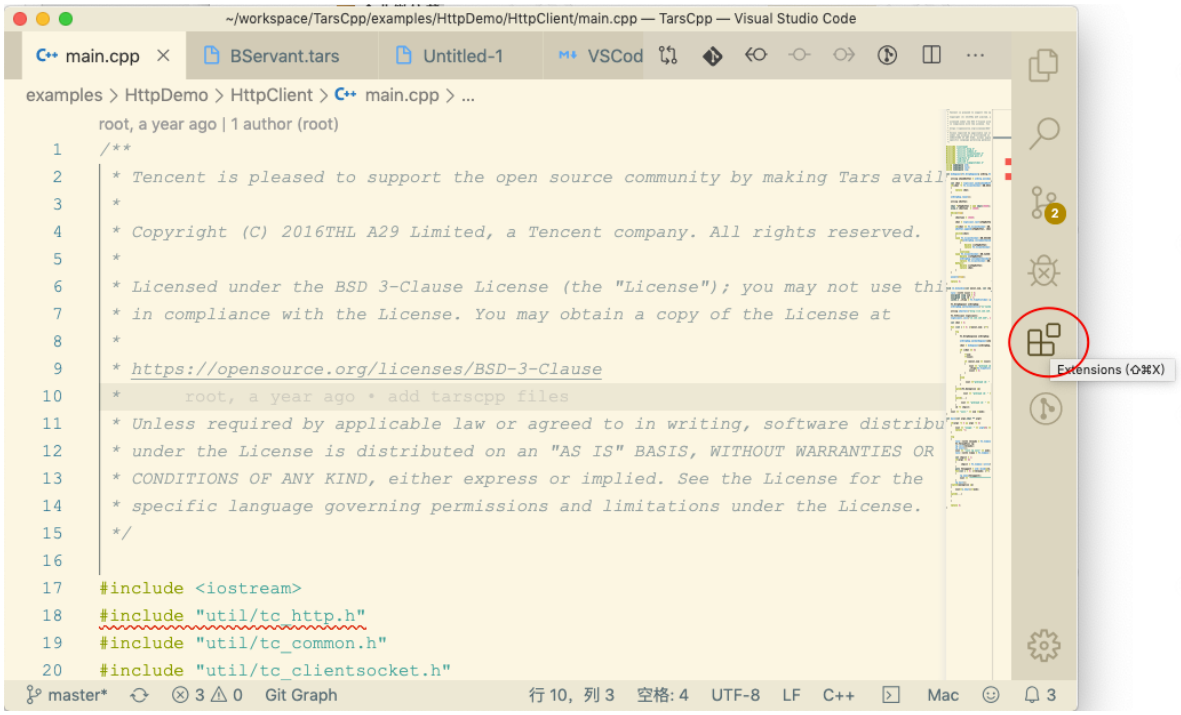
```
    "http.proxy": "http://你的代理服务器IP:端口/",
    "http.proxyStrictSSL": false,
```



4. 安装插件

4.1 安装 C++开发相关插件(必装)

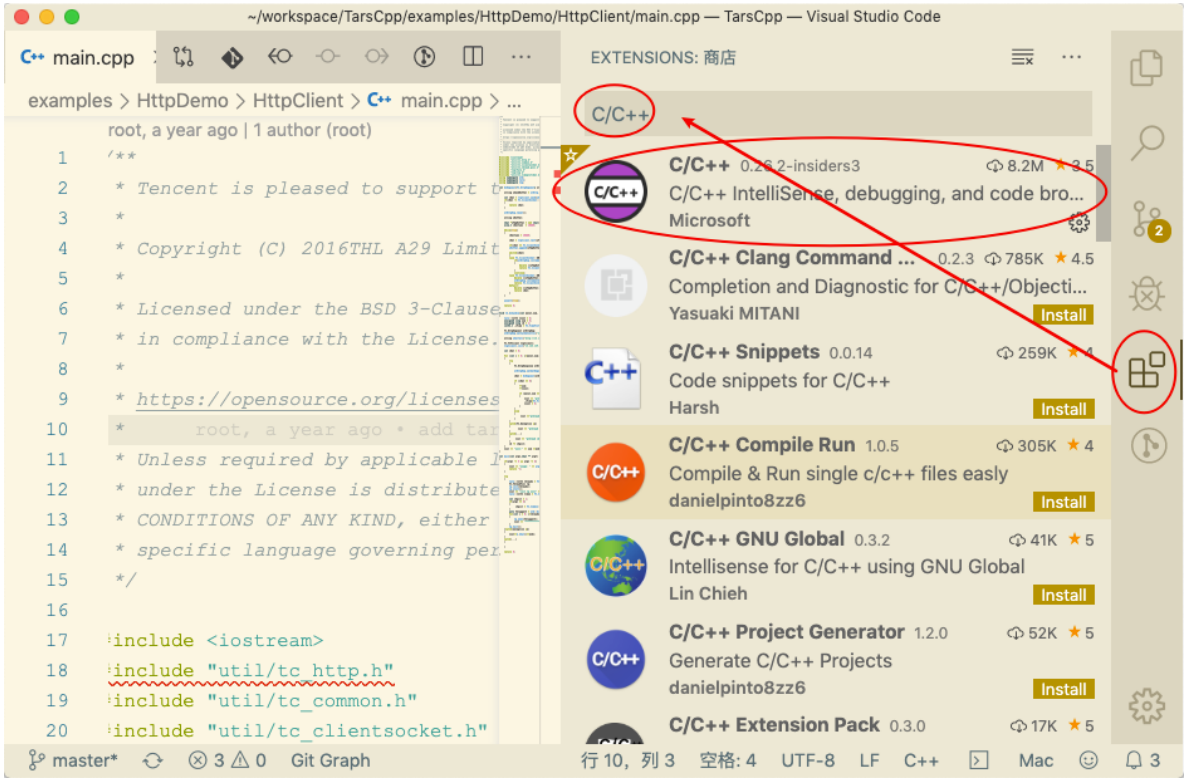
> 主要功能：C++语法识别、智能补全、代码跳转、调用依赖识别等，注意，下文中描述如果没有指明是需要单独手动下载的话，一般来说都是指通过 VSCode 侧边栏上的扩展按钮来安装插件。



> C++ 插件仅推荐装这一个，日常开发已足够用

4.1.1 安装 C/C++ 插件

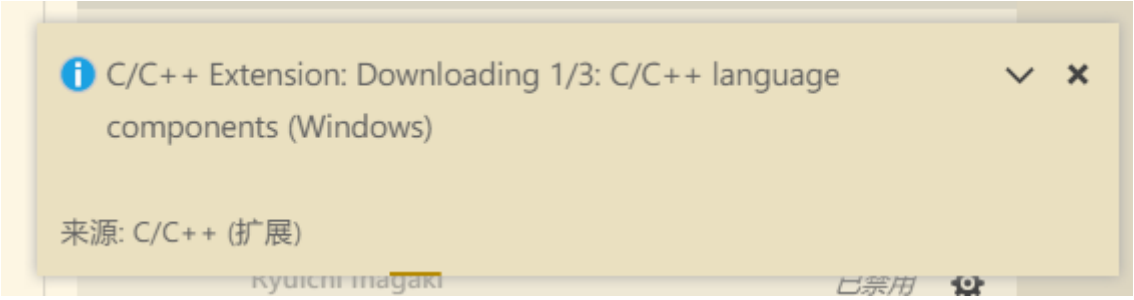
首先安装 C/C++ 插件：



安装完此插件后，再安装下 VSCode_cpptools 工具。它是由微软 VSCode C++团队负责维护的。

> **↓ 下面这段描述，如果你没遇到，也没关系。继续按照下面（2）小节内容操作即可。 ↓**

安装完成后，一般 VSCode 弹窗还会提示下载 C/C++其他依赖，内容如下：



一般都无法下载成功，状态栏栏里的输出内容文本内容如下：



此时，就需要按照下面（2）小节内容操作。

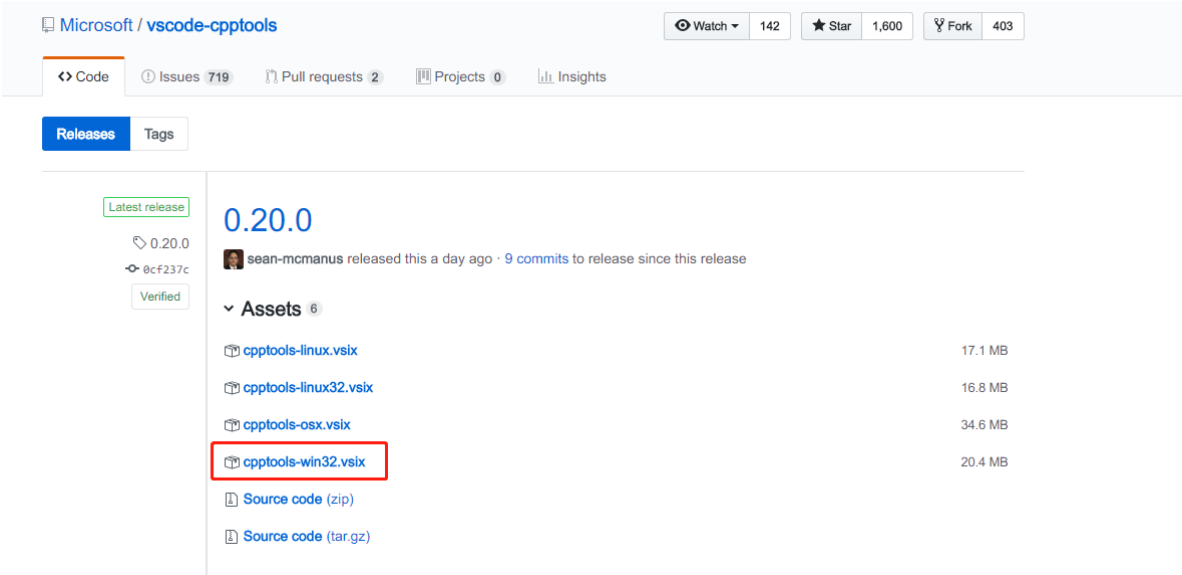
> **↑上面这段描述，如果你没遇到，也没关系。继续按照下面（2）小节内容操作即可。↑**

4.1.2 手动下载 VSCode_cpptools（C/C++ language components (Windows)）

有时在公司开发环境下，由于网络策略等问题 VSCode 无法自动下载，所以我们采取手动下载的方式。打开官方下载页面：

https://github.com/Microsoft/VSCode-cpptools/releases

页面中选择最新版本 `cpptools-win32.vsix` 点击下载到本地，（注意选择最新版本，本文档写作时插件是 0.20.0,你只要选最新版就可以，理论上你的版本一定不能比截图中的小）如下图：



官网支持 win、linux、mac，选择对应的下载即可。

4.1.3 在 VSCode 中安装插件

在任意一个 VSCode 打开的界面中，按住`Ctrl+Shfit+P`，在弹窗的输入框中输入`Install from VSIX`（或简单输入`vsix`），再选择刚才下载好的`cpptools-win32.vsix`，VSCode 会自动安装，安装好后，重启 VSCode 即可。



4.1.4 快捷键

这里主要介绍下最主要的快捷键：

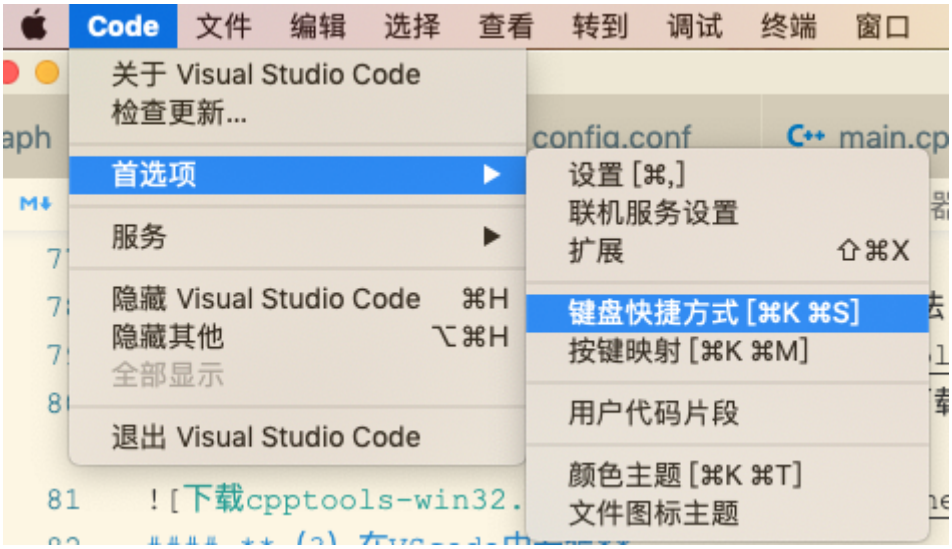
- `Ctrl+鼠标左键`，是文件、函数等跳转。
- `Alt + ←`，是跳转后返回原处。
- `Ctrl + Shift + O`，列出函数名
- `Ctrl + P`，列出近期打开的文件名
- `Ctrl + Tab`，可以列出最近打开的文件，在开发时，两个文件间切换时效率很高。

上述是最常用的功能。其他功能请参考VSCode 官网快捷键

要注意下，在`windows`下，上述快捷是默认的.但在`linux`、`mac`下，`Alt + ←`跳转返回原处 这一快捷键是无法使用的，原因是快捷键冲突。笔者在这三个 os 间都长时间做过开发工作，因此建议三种 os 下用统一快捷键，以减少认知成本。`linux`、`mac`下，如何绑定`Alt + ←`为`跳转返回原处`。操作如下：

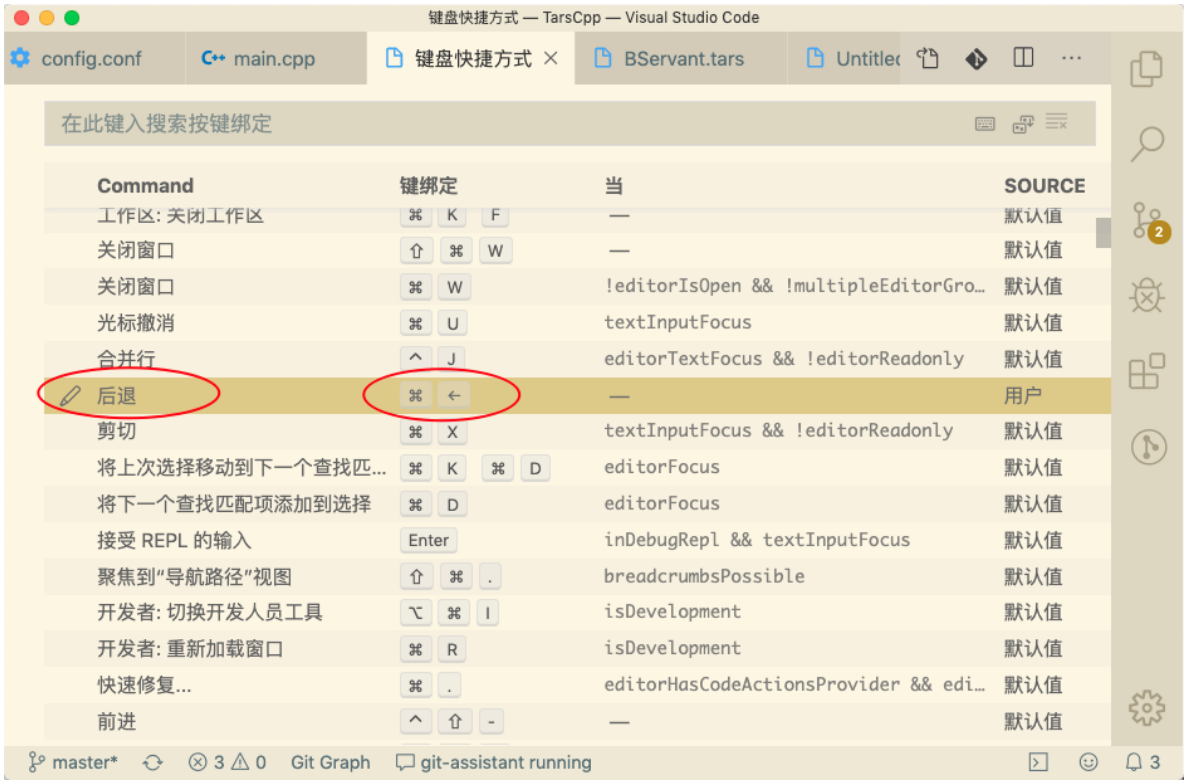
找到快捷键

操作路径：`【首选项】`-`【键盘快捷方式】`



注意：截图中笔者使用的`mac`，但实际上`linux`、`win`都是同样路径。

找到`后退`功能

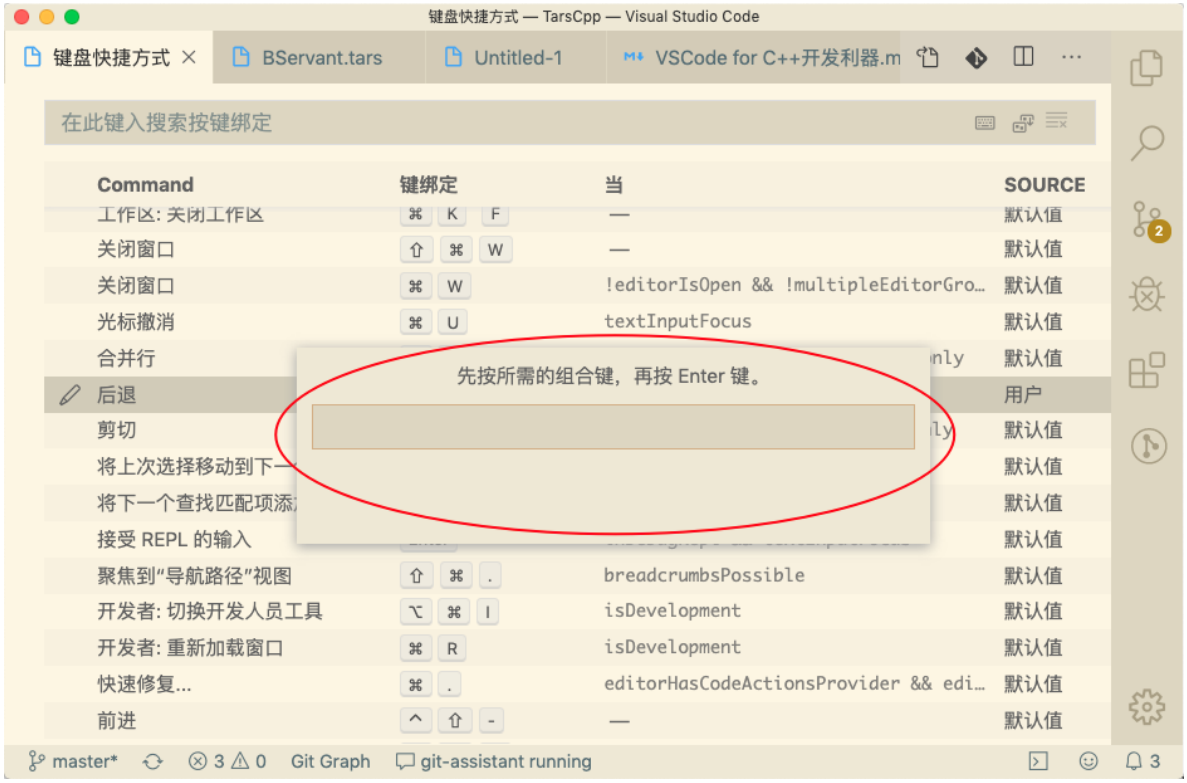


寻找时可在搜索框中输入`后退`查找，截图是为了方便大家定位，手动上下翻动找的。

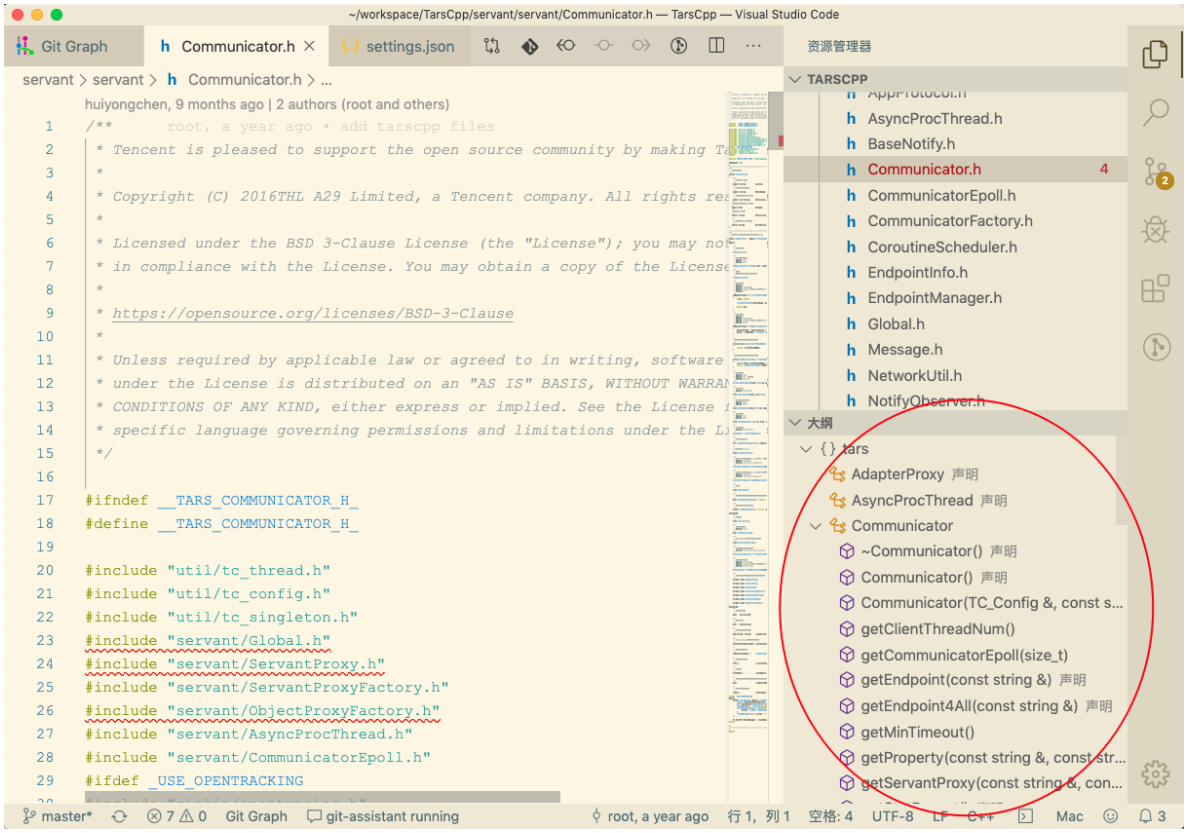
键绑定

点击 上图中的第二个圈位置（`键绑定`那一列），然后按下键盘上的组合键。我在`win`、`linux`下使用普通键盘时，采用的是`Alt + ←`，`mac`是使用的`command + ←`。

注意，实际按键时只要同时按下`Alt`和`←`（或者`command`和`←`）两个键即可，不需要`+`键。是为了方便大家看到是需要按两个键才中间增加了`+`。



4.1.5 大纲功能



大纲功能比较使用，直接列出了代码中所有的函数列表，且支持搜索。

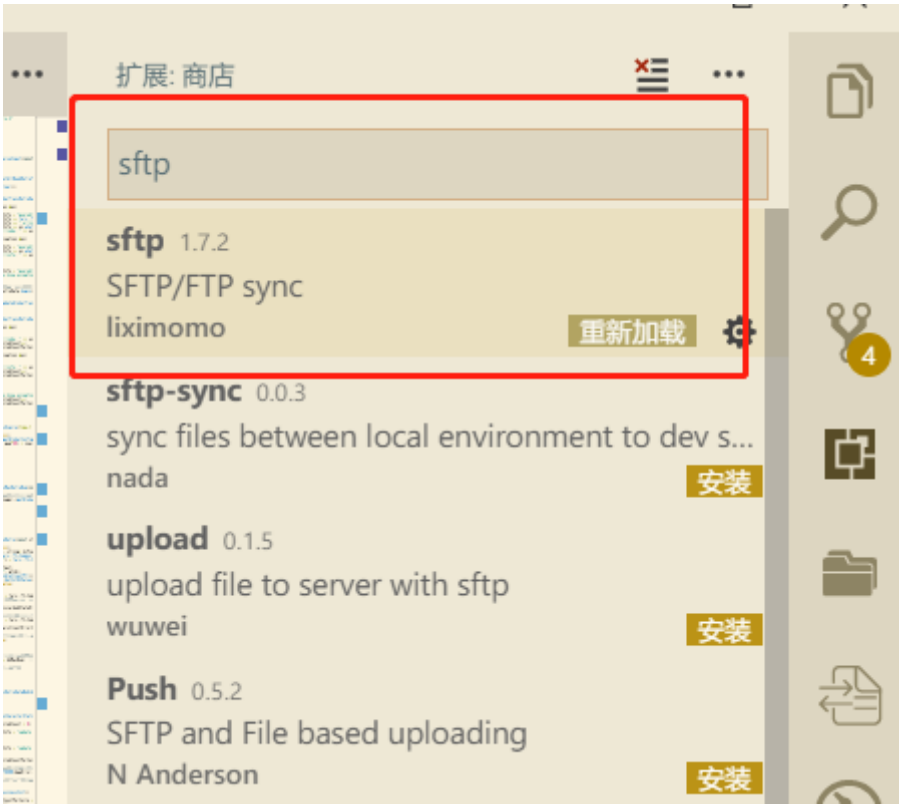
4.2 安装汉化插件

安装`Chinese (Simplified) Language Pack for Visual Studio Code`插件，相当于汉化插件。

4.3 安装代码同步插件

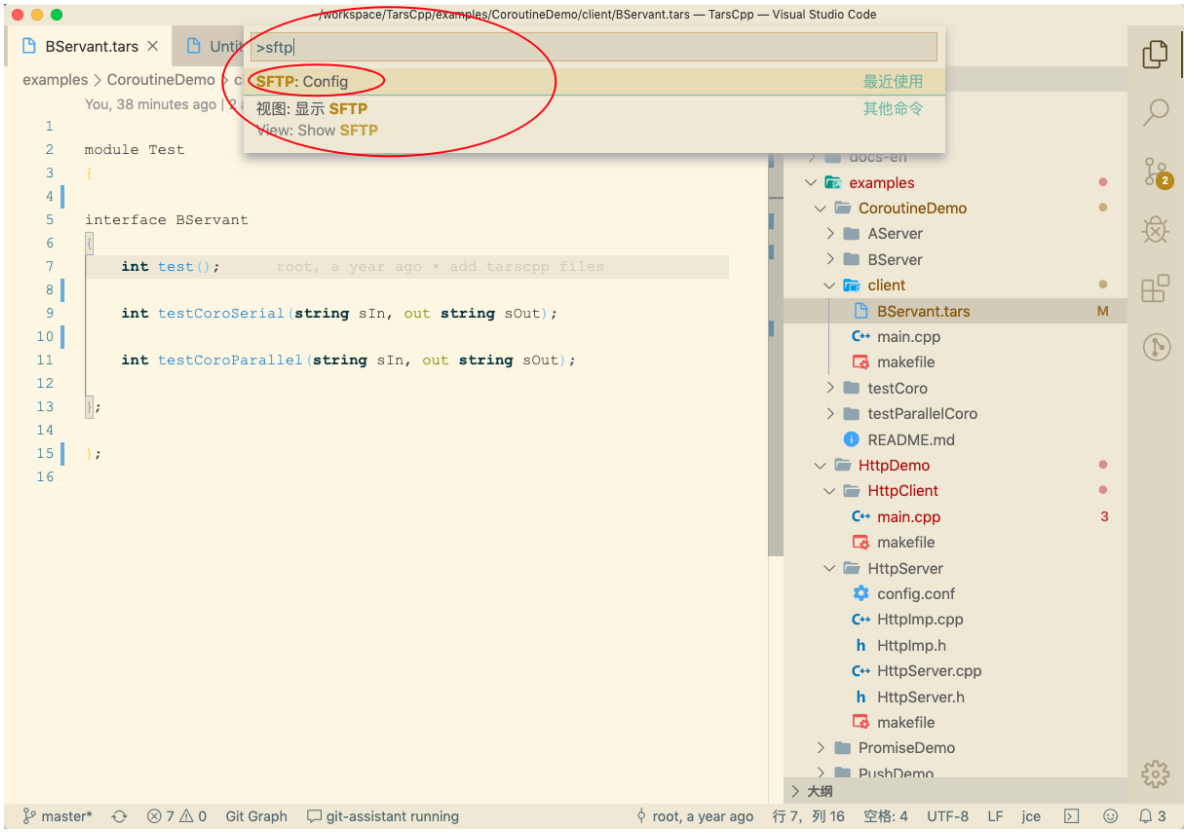
推荐安装 SFTP。

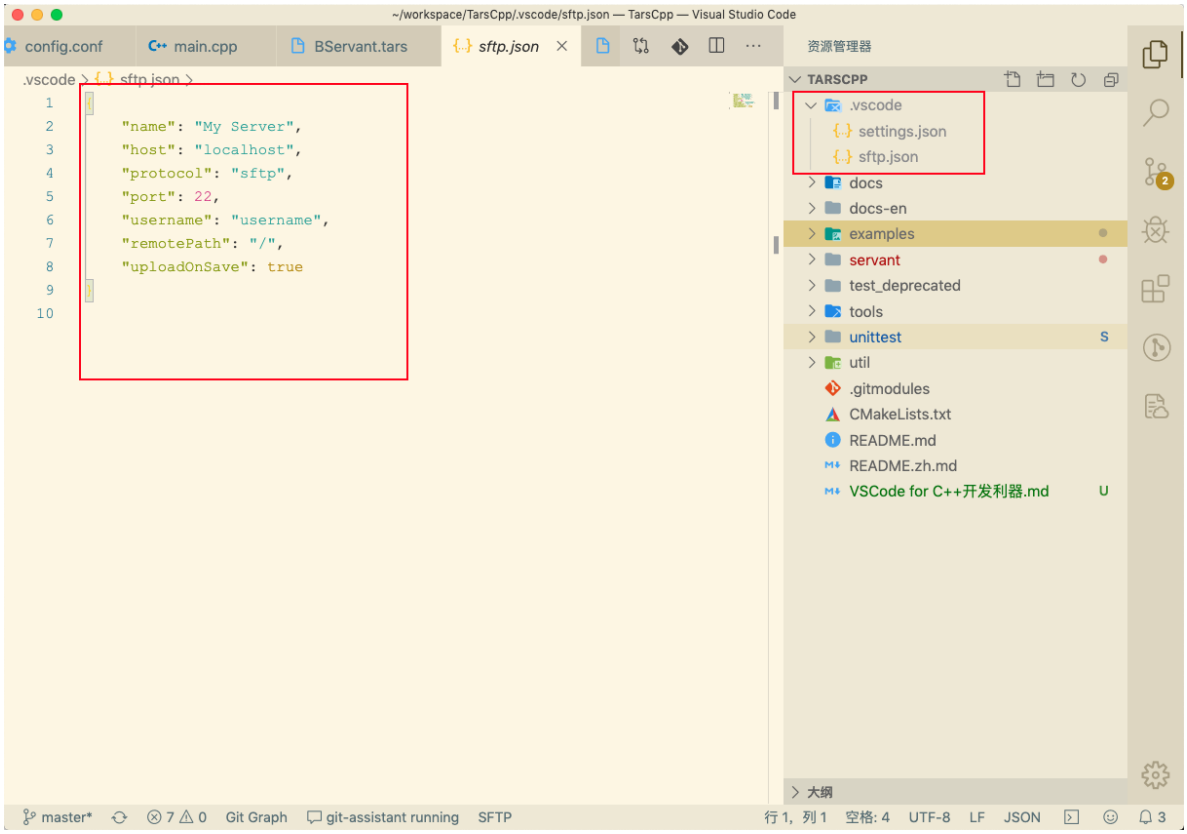
4.3.1 VSCode 中安装 sftp



4.3.2 新工程中首先配置 sftp

Ctrl+Shift+P 后，输入 `sftp` ，选择 Config 功能，然后 VSCode 会在当前目录下生成.VSCode 目录，并创建 sftp.json 文件，即为 sftp 要同步文件到其他机器的配置信息。





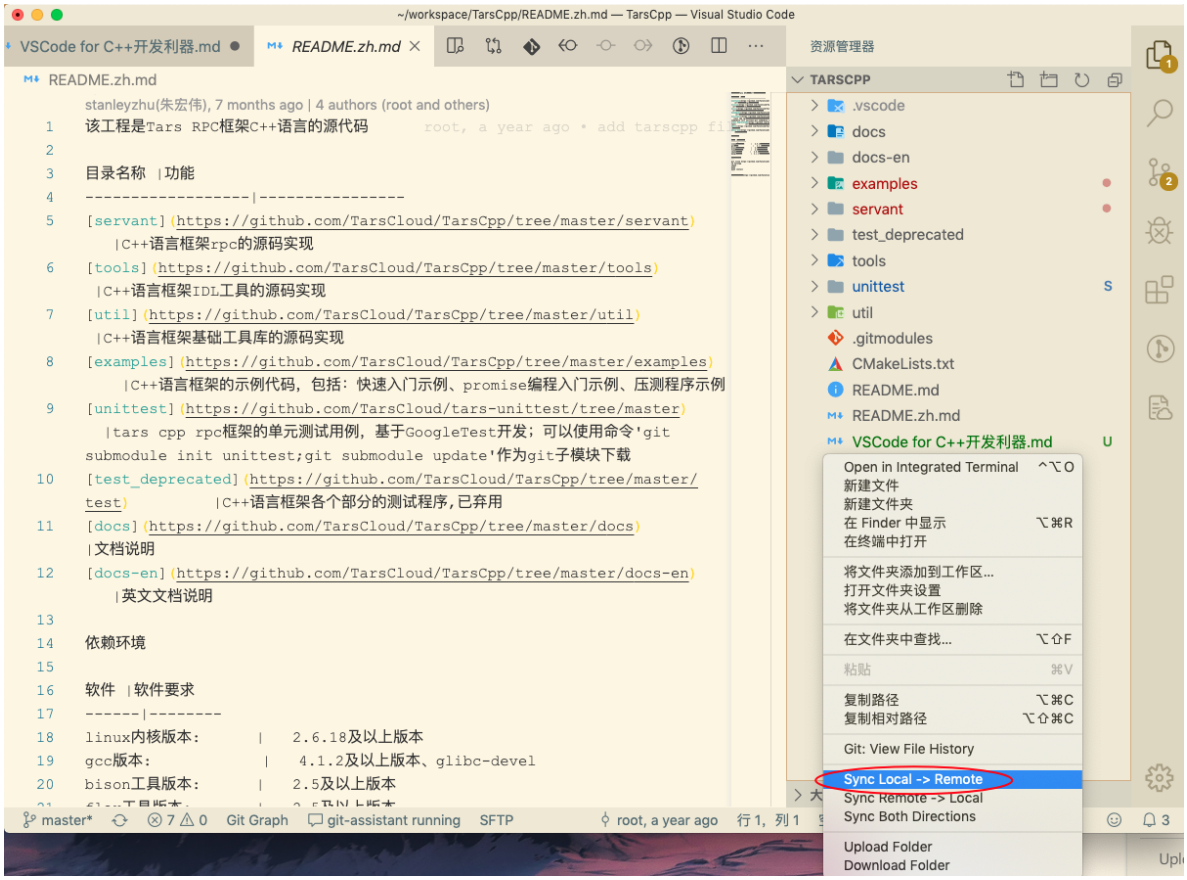
上图中内容是默认创建的 sftp.json 文件。读者可以直接拷贝下面这段配置，并修改对应内容为你自己的服务器的信息

```
{
    "name": "tfae.dtx",
    "protocol": "sftp",
    "host": "10.xxxxxxxx.22",
    "port": 36000,
    "username": "xxxxxxxx",
    "password": "xxxxxx",
    "uploadOnSave": true,
    "ignore": [
        "\\.VSCode",
        "\\.git",
        "\\.DS_Store",
        "\\.svn",
        "\\.history",
        "\\.IAB",
        "\\.IAD",
        "\\.IMB",
        "\\.IMD",
        "\\.PFI",
        "\\.PO",
        "\\.PR",
        "\\.PRI",
        "\\.PS",
        "\\.WK3"
    ],
    "remotePath": "/root/home/denniszhu/tarsCpp"
}
```

- host、port、username、password 和你是用 xshell 时需要配置的信息一样。

- `uploadOnSave`参数意思是，你本地编写的代码，在你是用 VSCode 保存时，sftp 会自动帮你同步到远程。这个功能建议开启，开启后，就不需要像 beyond compare 每次都需要手动同步那样麻烦了。
- ignore，意思是你同步时，忽略哪些文件，一般可以固定的把.VSCode、.git、sourceinsight 自动生成的文件都忽略掉，这样同步到远程的文件都是你关注的代码。
- remotePath 就是你远程的代码放置的目录

4.3.3 同步代码到单台远程机器



注意，可以在代码列表的空白处右键，同步所有代码；也可以在某个文件，某个文件夹上右键，则仅同步选中的内容

4.4 安装 git 相关插件

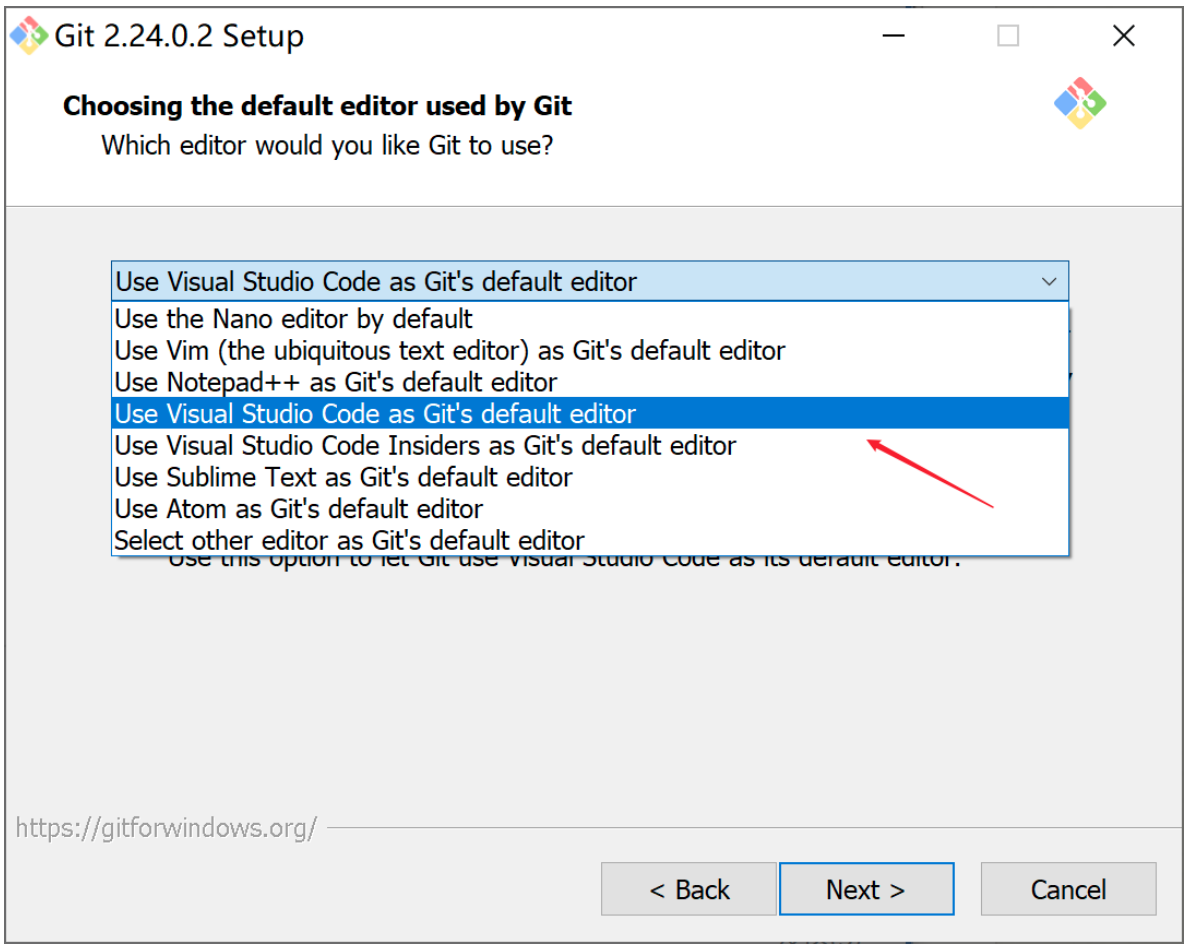
4.4.1 安装 git

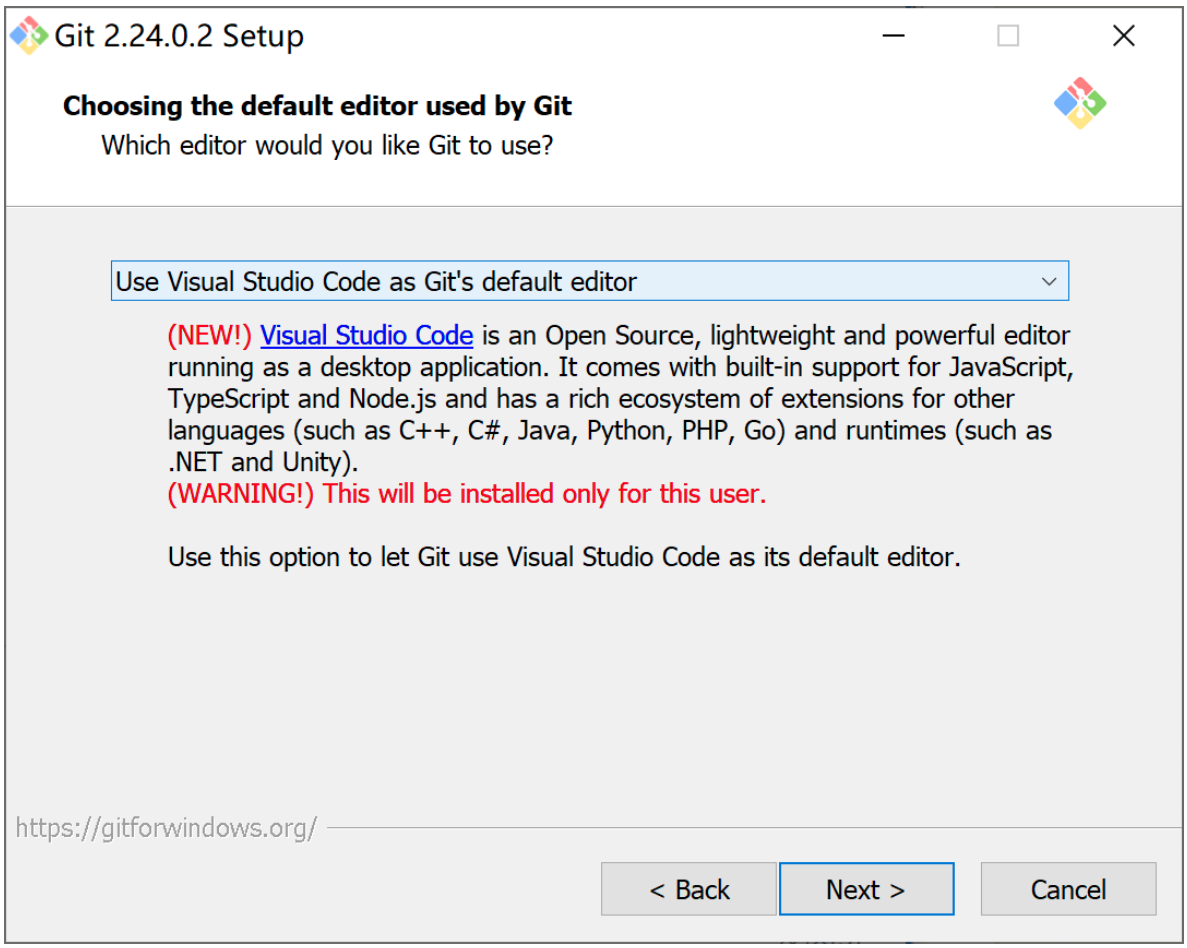
git 是 VSCode 自带的功能，不过 windows 下需要我们自己先安装 git。然后 VSCode 才能识别。
安装 git 可参考： https://www.jianshu.com/p/414ccd423efc

linux、macos 下如何安装 git 请自行查阅。

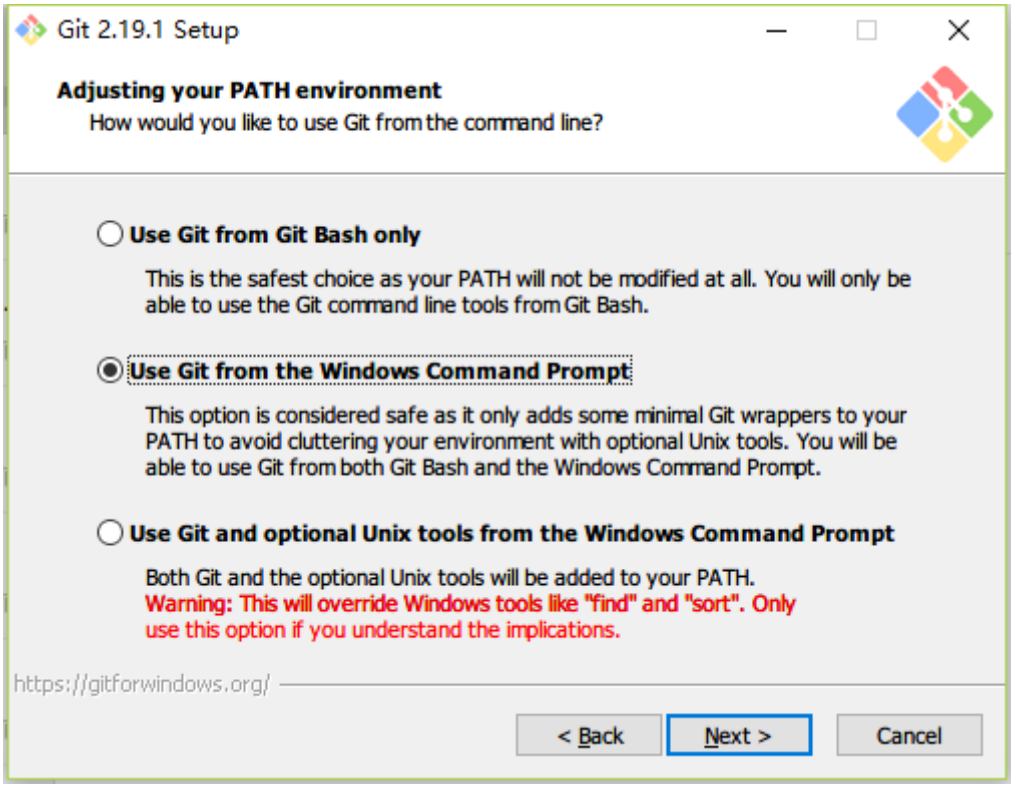
git 安装时，需要在关键几步注意下：请按照图中选项设置

git 安装 0:

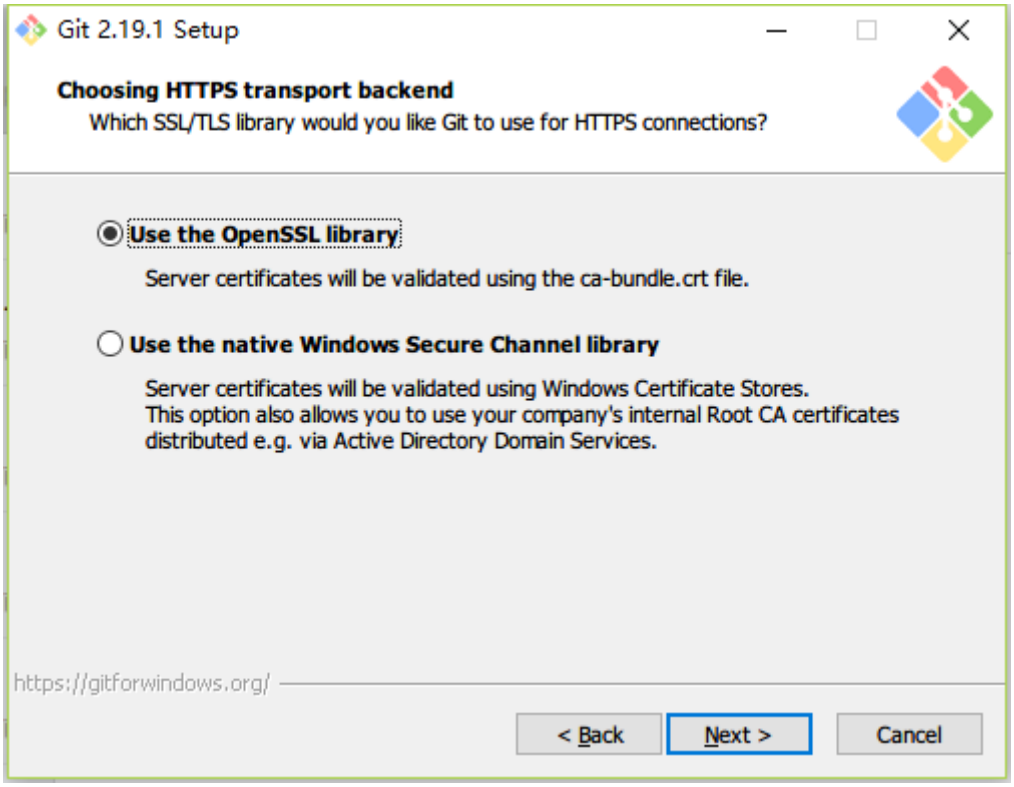

Git 2.24.0.2 Setup
Choosing the default editor used by Git
Which editor would you like Git to use?
Use Visual Studio Code as Git's default editor
Use the Nano editor by default
Use Vim (the ubiquitous text editor) as Git's default editor
Use Notepad++ as Git's default editor
Use Visual Studio Code as Git's default editor
Use Visual Studio Code Insiders as Git's default editor
Use Sublime Text as Git's default editor
Use Atom as Git's default editor
Select other editor as Git's default editor
https://gitforwindows.org/
< Back
Next >
Cancel


Git 2.24.0.2 Setup
Choosing the default editor used by Git
Which editor would you like Git to use?
Use Visual Studio Code as Git's default editor
(NEW!) Visual Studio Code is an Open Source, lightweight and powerful editor running as a desktop application. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages (such as C++, C#, Java, Python, PHP, Go) and runtimes (such as .NET and Unity).
(WARNING!) This will be installed only for this user.
Use this option to let Git use Visual Studio Code as its default editor.
https://gitforwindows.org/
< Back
Next >
Cancel

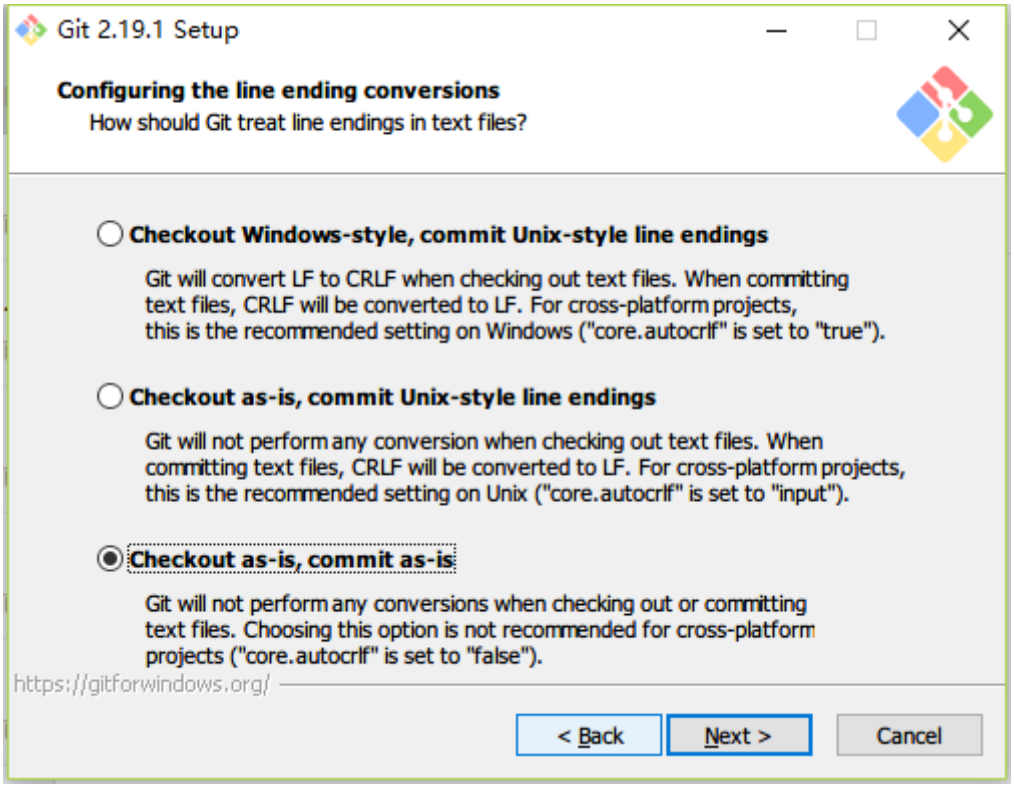
git 安装 1:



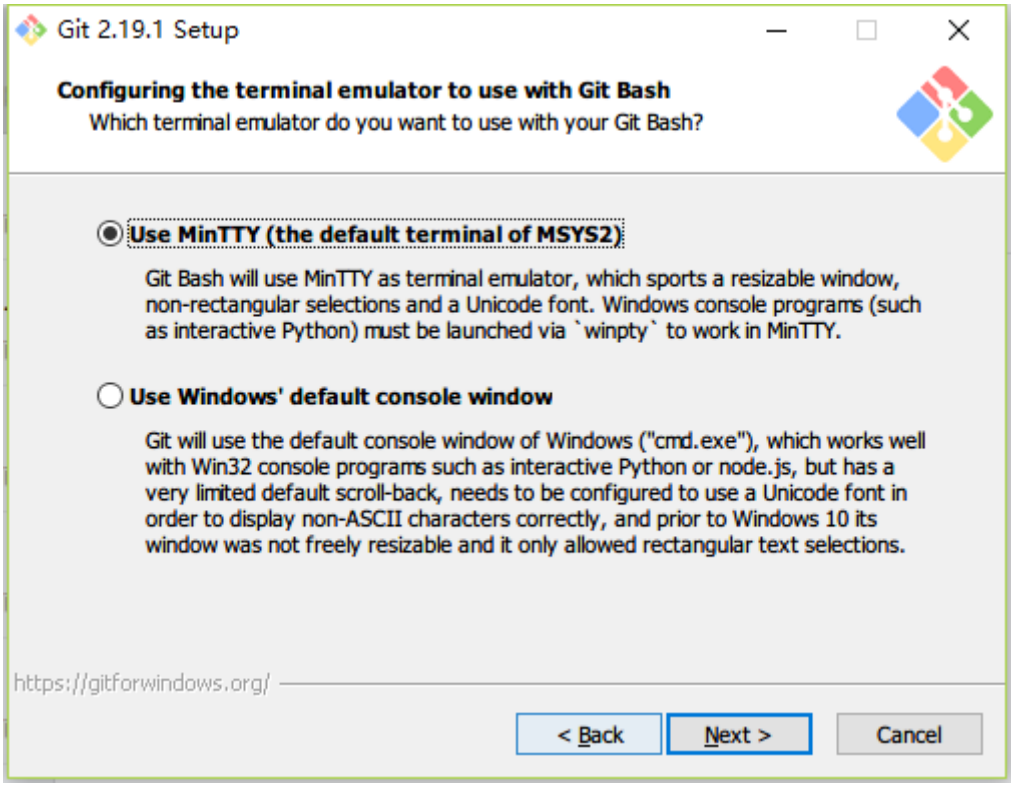
git 安装 2:



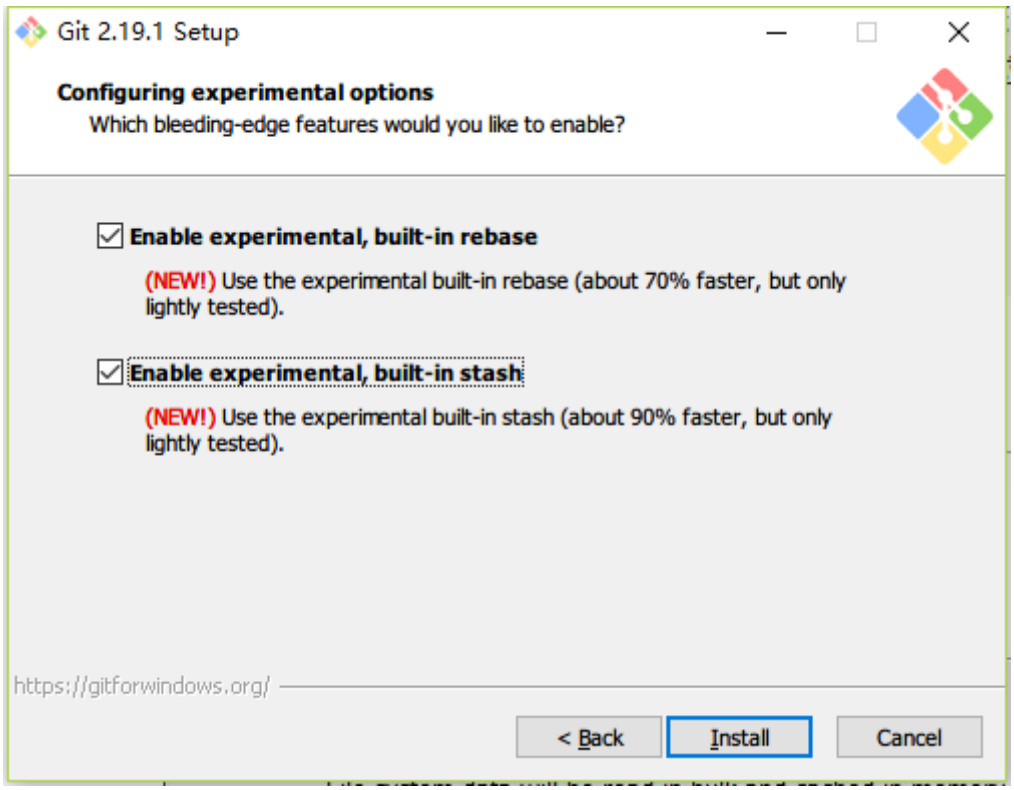
git 安装 3:



git 安装 4:



git 安装 5:



4.4.2 VSCode 中配置 git 安装路径

安装 git 时要关注添加环境变量到系统中即可。安装完成后，需要在 VSCode 中配置 git 路径。VSCode 中配置 git 安装路径如下：win 下，仅供参考：

```
    "git.path": "D:\\Program Files\\Git\\cmd\\git.exe",
```

linux、mac 下，仅供参考：

```
    "git.path": "/usr/bin/git",
```

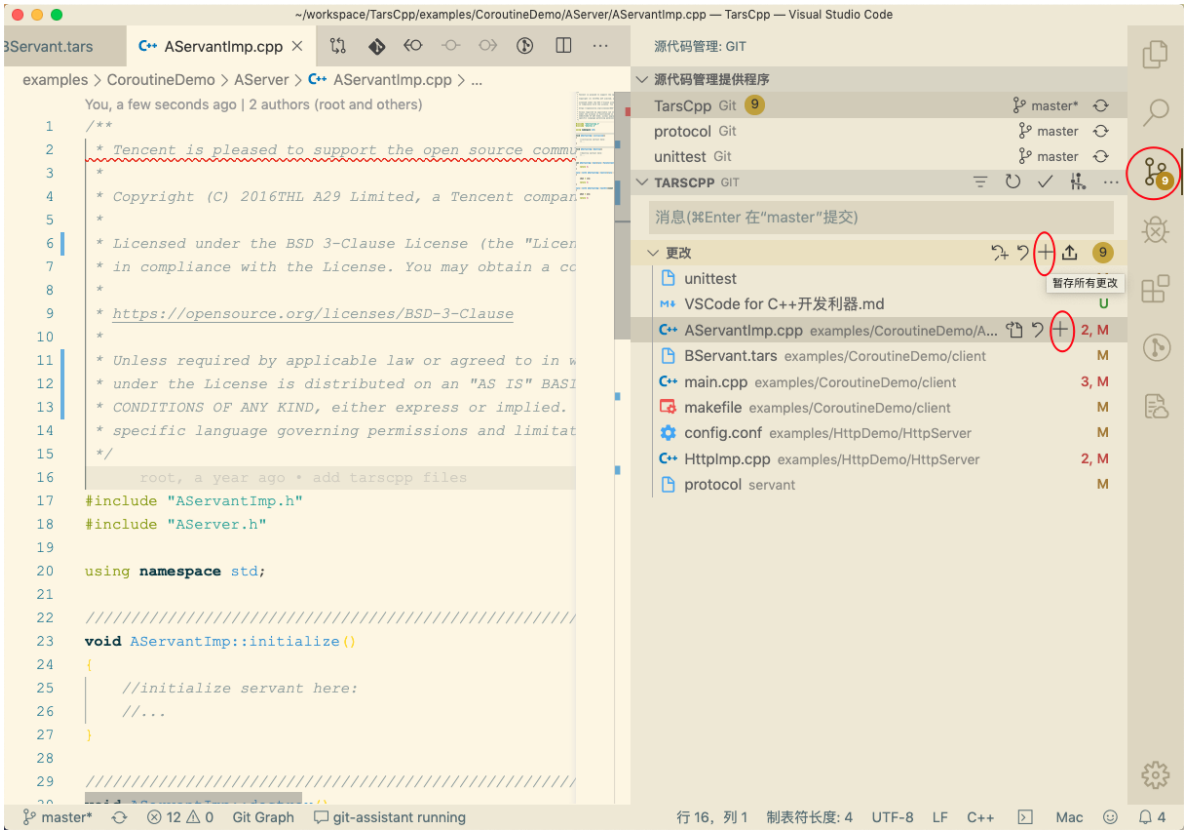
4.4.3 VSCode 中使用 git 日常操作

> 注：本文列举的是如何在 VSCode 中使用 git，但还是要求开发人员对 git 的命令比较熟悉，VSCode 仅仅是省去了你敲命令的工作，但并不代表如果你不懂 git 就能上手，毕竟下面中菜单列出的选项都是和 git 知识强相关的。

> 注：VSCode 中所有的 git 功能你也可以不用，所有操作使用 git 命令行进行也可以，随你的习惯。笔者 git 日常工作基本全在 VSCode 中进行了。

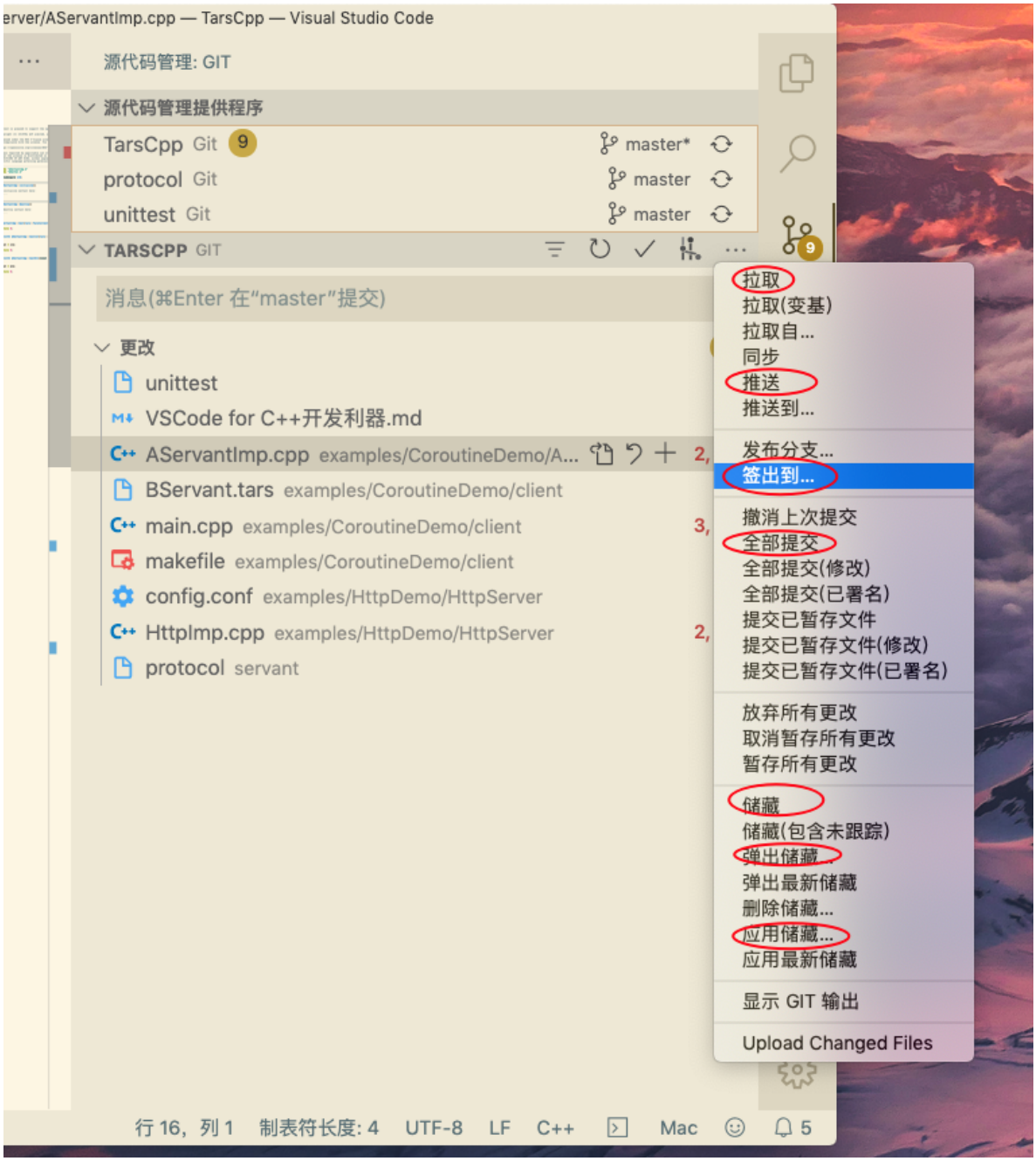
> 注：VSCode 暂不支持 git clone 自动拉去仓库，因此是需要先手动执行命令行 git clone 下仓库到本地目录中，才能使用 VSCode 进行后续 git 操作。

（1）暂存更改



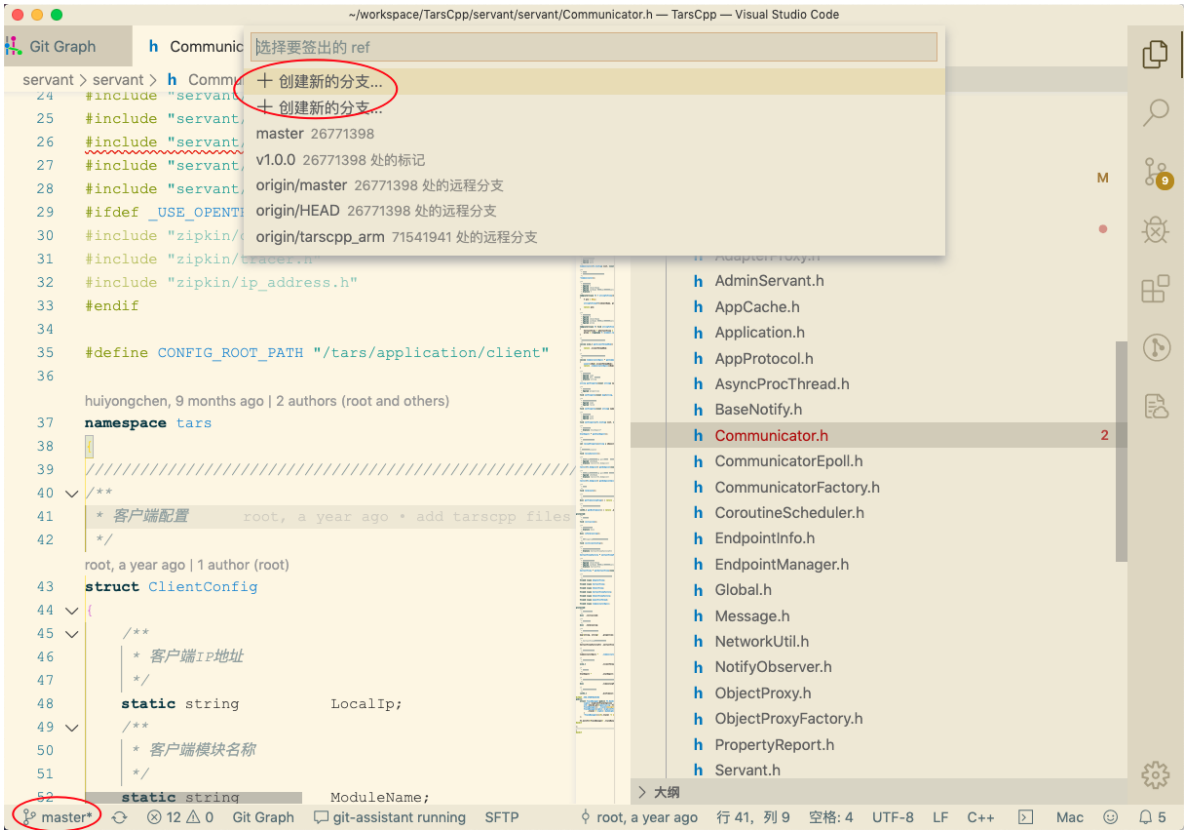
暂存更改可以指定某个文件按+暂存，可一个全选，按图中的加号。

（2） 提交暂存、推送、拉取等功能



`签出到` 主要是首次使用 git 命令 glone 仓库下，用 VSCode 打开后这里不会展示分支名，而是一个一个 commitid，所以要收到签出到对应的分支名，一般来说都是迁出到 master。如上图。

(3) 切换分支、拉取分支

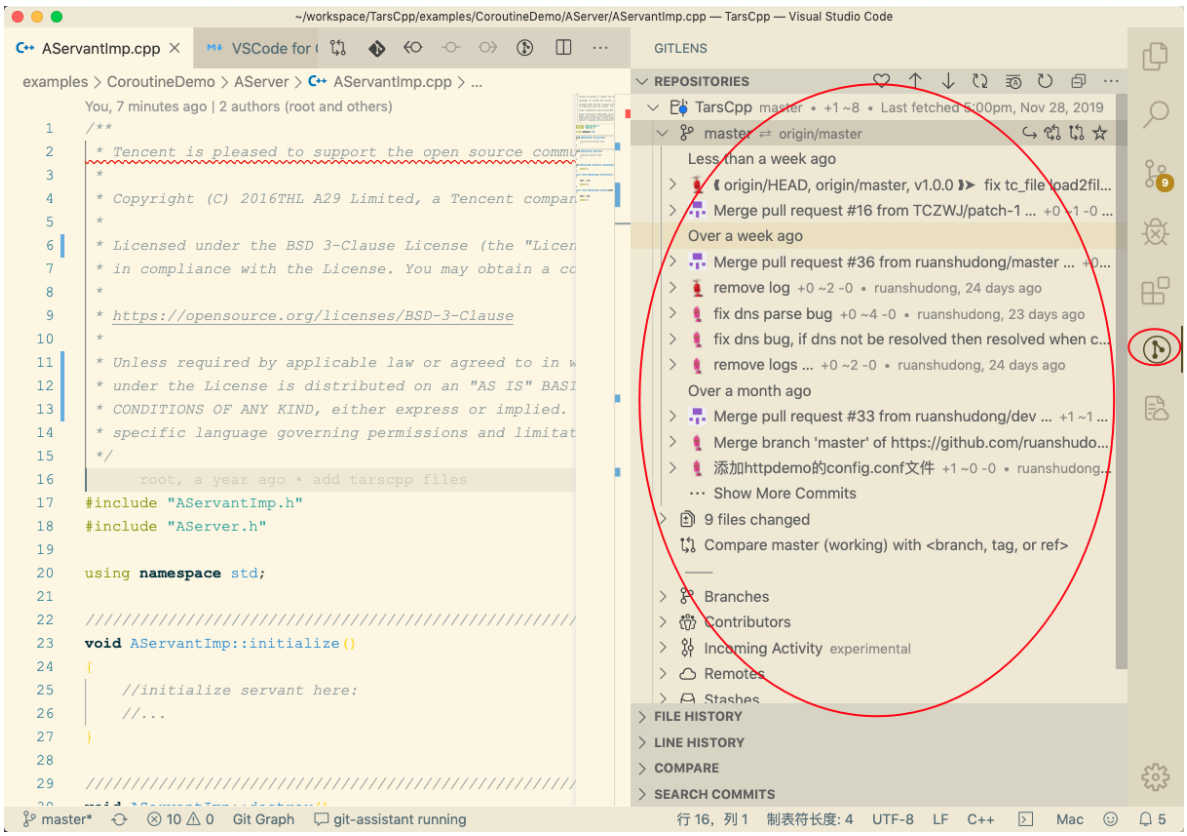


等等 git 的各种功能都在 VSCode 中有所集成。

上述仅列举了常用高频操作，很多功能需要自行探索。

4.4.4 git 相关其他插件

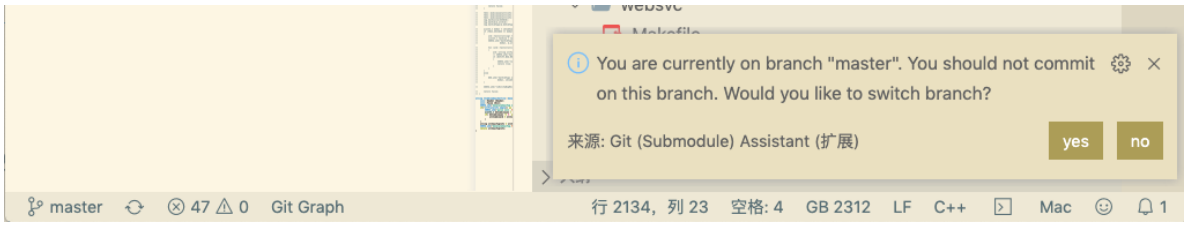
- **GitLens**
  它会显示有 git 有关的所有代码提交信息，是在 sidebar 上的二叉树图标



- **Git Branch Warnings**
  Git Branch Warnings 比较有意思，当你切到 master 分支上时会在最下面的状态栏和弹窗提示。

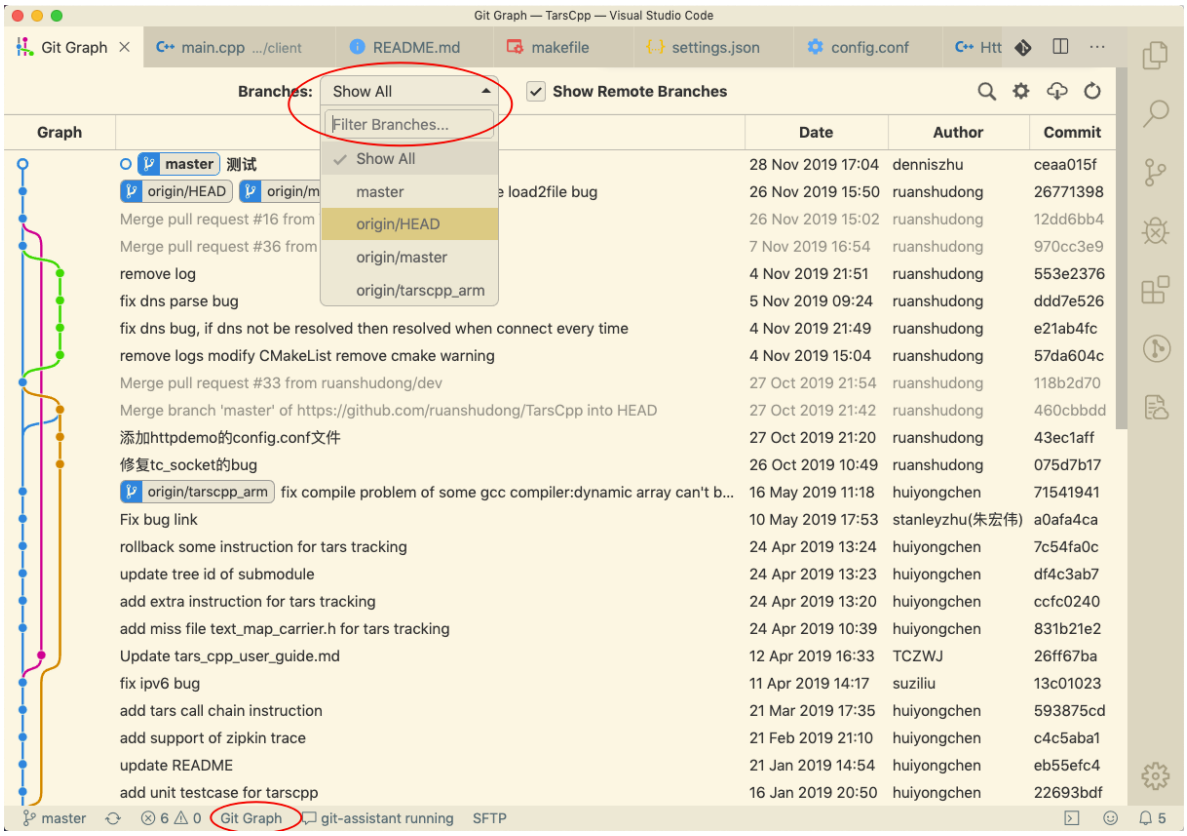
以防你误切 master 分支
需要在 User Setting 上加一行配置：【】里是你想提醒的分支

```
"branchwarnings.protectedBranches": [ "master" ]
```



- **git graph**

它会显示有 git 有关的所有代码提交信息，可以方便查看本地分支、远程分支、主干的 commit、push、pull 情况，尤其在版本提测、提发时确认分支代码是否提交很有帮助；具体功能需自行探索，只能说很强大。

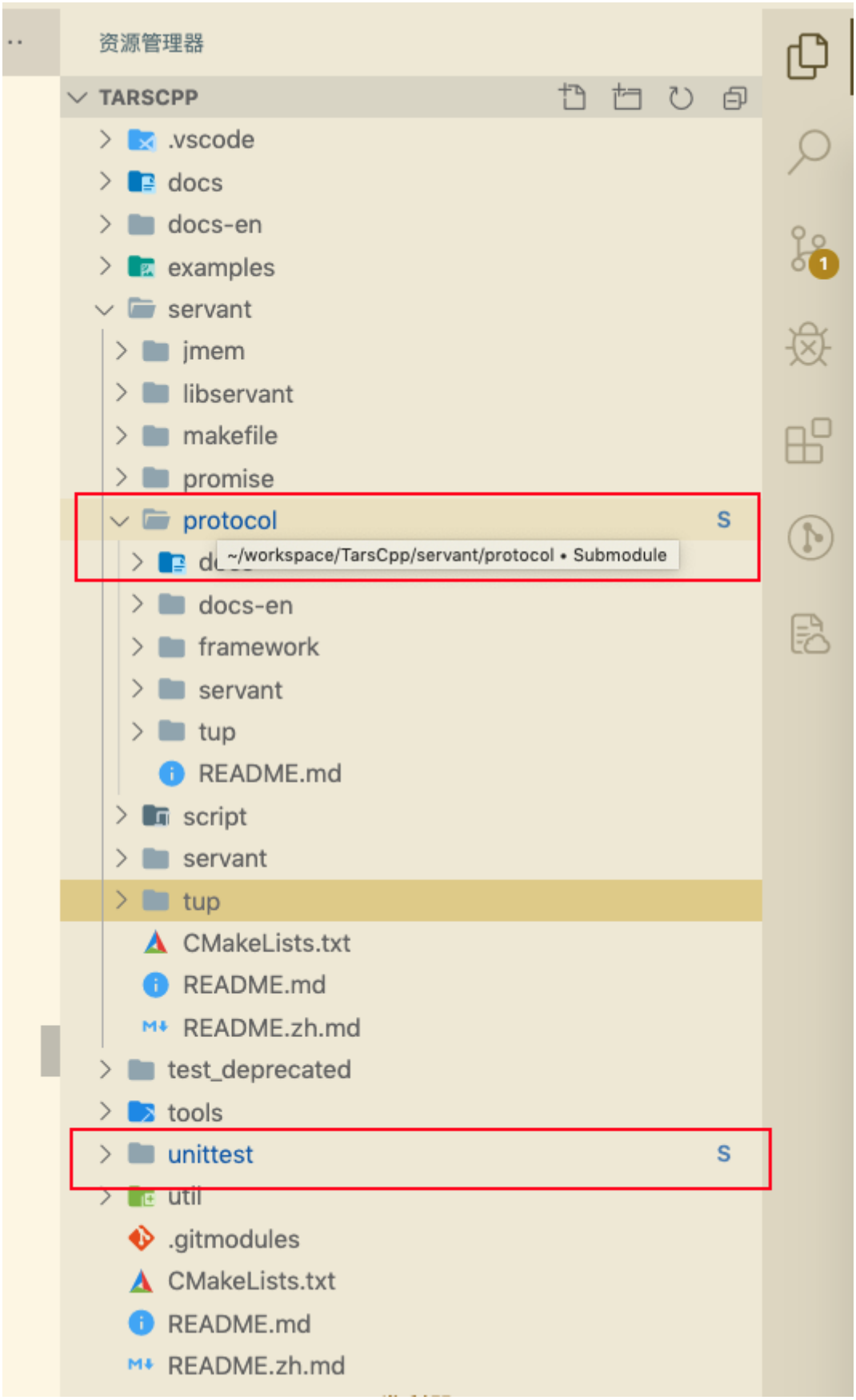


- **Git (Submodule) Assistant**

有些工程使用 submodule 方式引入其他库。如果想在 VSCode 的 git 也能自动拉取这些 submodule 的代码，则需要该插件，对于看代码会十分方便。git 默认是不拉取 submodule 的；

以腾讯开源的 tarscpp 框架为例：
`https://github.com/TarsCloud/TarsCpp.git`

git clone 到本地后(clone 时可不需要--recursive 选项)，通过 VSCode 打开后，`Git (Submodule) Assistant` 插件会自动识别并下载所有 submodule。当然如果你账户没有权限 clone 子模块，则会下载失败，会有详细提示。



上图中的红框圈起来的目录均为子模块，可以看到右侧还有个`s`（`submodule`）的标志。直接 git clone 工程时本地是不会 pull 这些 submodules 的。安装该插件后可以自动下载。

4.5. 其他提高效率的插件（可选）

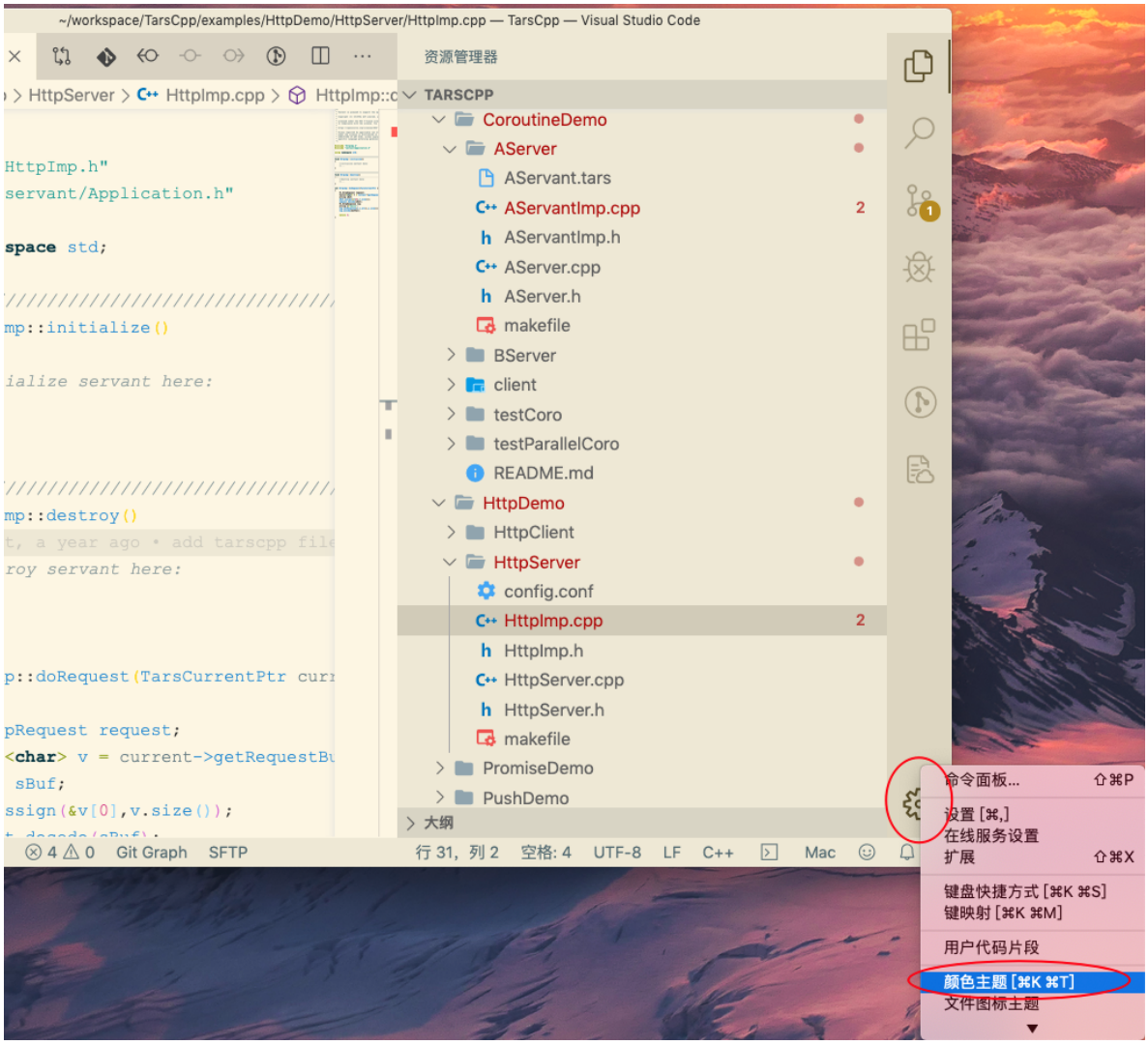
（1） 自动识别文件编码(必做)

针对不同编码的文件，VSCode 也能自动识别。

```
    "files.autoGuessEncoding": true,
```

(2) 安装图标、颜色主题

推荐使用`material icon theme`主题，它提供的文件图标比较漂亮。
推荐使用`solarized Light`颜色主题。更多主题可以通过 VSCode 右下角的`颜色主题`和`文件图标主体`来选择更多：



之后可以选择喜欢的主题。本小节推荐的主题和颜色主题均可以在 sidebar 的 extensions 中下载。

(3) 缩小小地图

VSCode 自带的小地图太宽，写代码时太占用可编辑区域，所以可以调小

```
"editor.minimap.maxColumn": 40,
```

(4) 字符分割

editor.wordSeparators，是指在写代码时，VSCode如何区分每个词（token）。加入以下设置，在双击选中一串字符序列时，VSCode就会依此来区分词，此配置包含了很多中文字符。

```
"editor.wordSeparators": "`~!@#$%^&*()-=+[{]}\\|;:'\",.<>/?·～！￥…（）—
【】、；：‘’“”，。《》？ ",
```

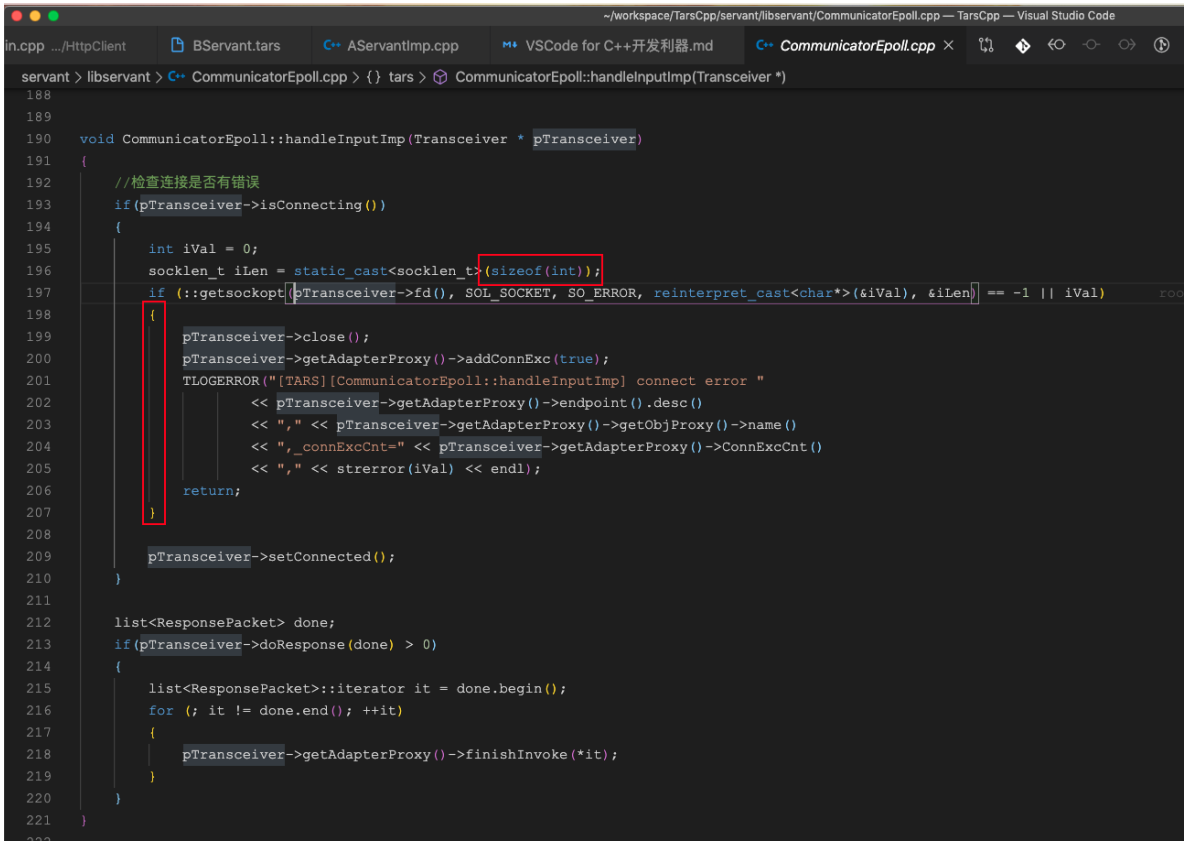
(5) 让标题能够显示完整的文件路径名

有时候代码中有些项目文件名相同，默认的标题只显示文件名，看不出是哪个目录的，最好设置让他全显示

```
 "window.title":
"${dirty}${activeEditorLong}${separator}${rootName}${separator}${appName}",
```

（6） 括号通过不同颜色区分

安装`Bracket Pair Colorizer2`插件，会通过不同颜色来展示匹配的括号，这样代码不容易一楼括号。



例如上图中`if`有多重括号，分别用蓝色和黄色标记，这样很容易看出代码有没有遗漏括号

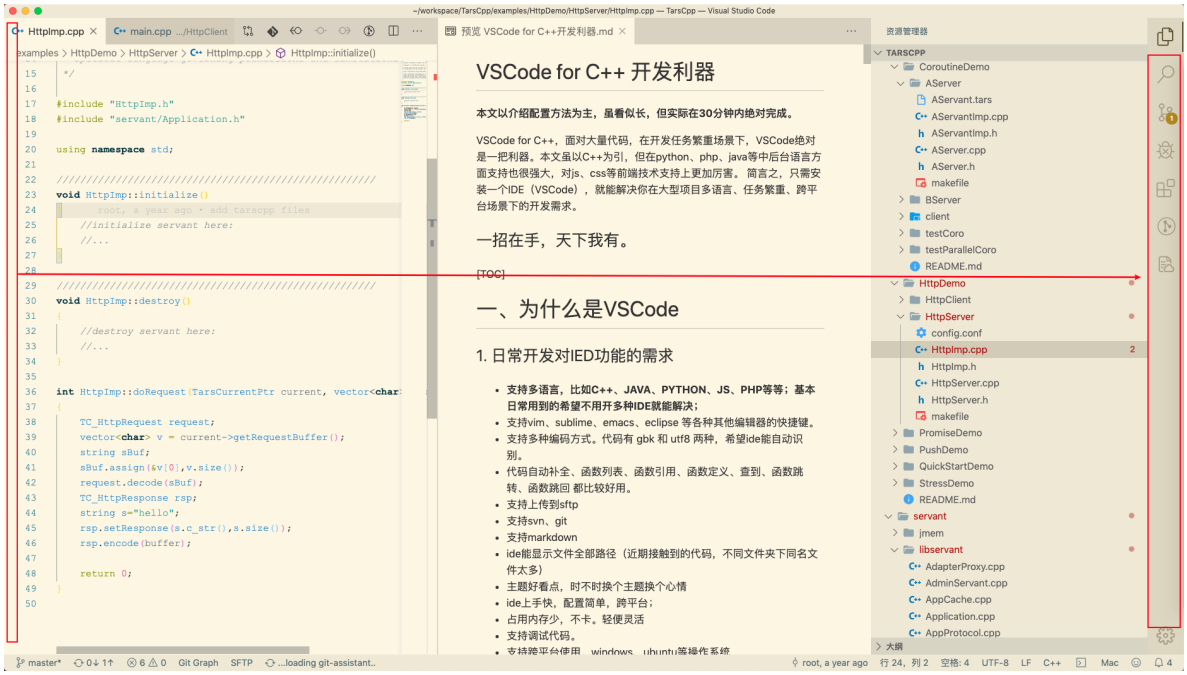
（7） 展示对齐层级深度

安装`Indenticator`插件，会将同一级的深度用竖线对齐展示，这样很容易看出来块结构，比如下图中就`for`语句块内的代码就有一条竖线，而内部的`if`和`else`则分别还有一条竖线。这些竖线只是帮助你看代码更容易。
上一小节的截图中外层 if、内层 if 间都有竖线，便于观察。

（8） 移动控制栏的位置

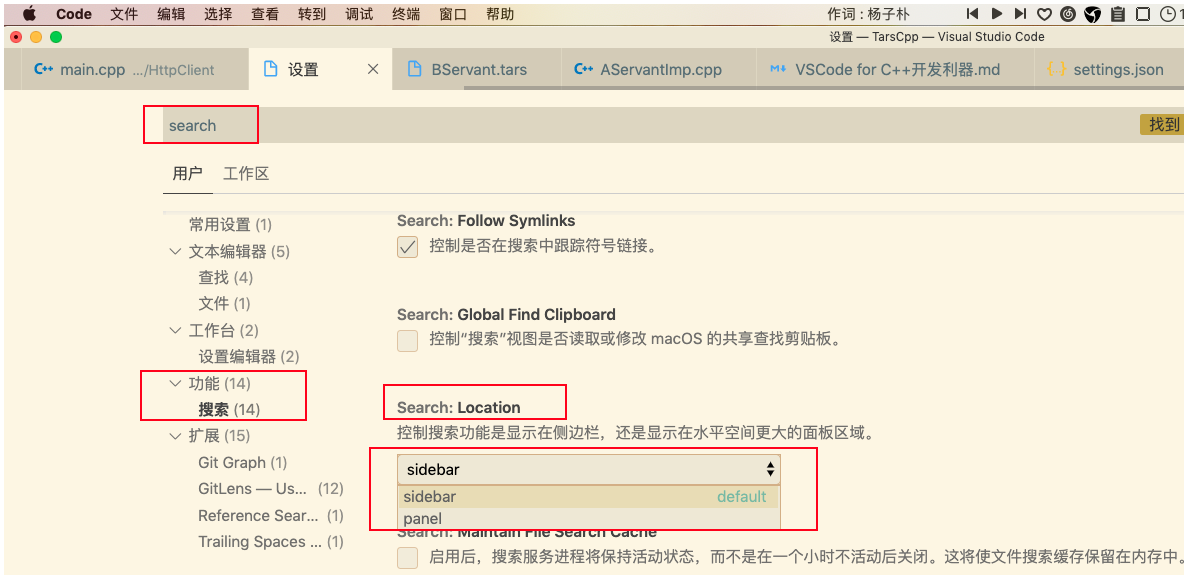
由于编辑的习惯，右手持鼠标时，最好把控制栏也放置到右边，这样操作控制栏时头和眼睛不用来回转。

```
"workbench.sideBar.location": "right",
```



（9） 移动全局搜索框的位置

vscode 老版本的全局搜索框是放在 sidebar 上的，和 git、扩展等并列，新版本后，放在了底部，官方理由是搜索结果视野更加宽阔，不过不太习惯，可以配置成老板的方式。选择 sidebar，而不是 panel：





（10） 自动删除文件行尾空格

写代码时，免不了会导致文件行尾出现很多空格，该命令可在文件保存时，自动去掉行尾所有空格。

```
"files.trimTrailingWhitespace": true,
```

该命令还可配合插件`Trailing Spaces`，该插件会将行尾空格高亮。



（11）TabNine

TabNine 是一个智能的代码自动补全工具。最终要的是，零配置，开箱即用（不用购买 license，也可以使用）。

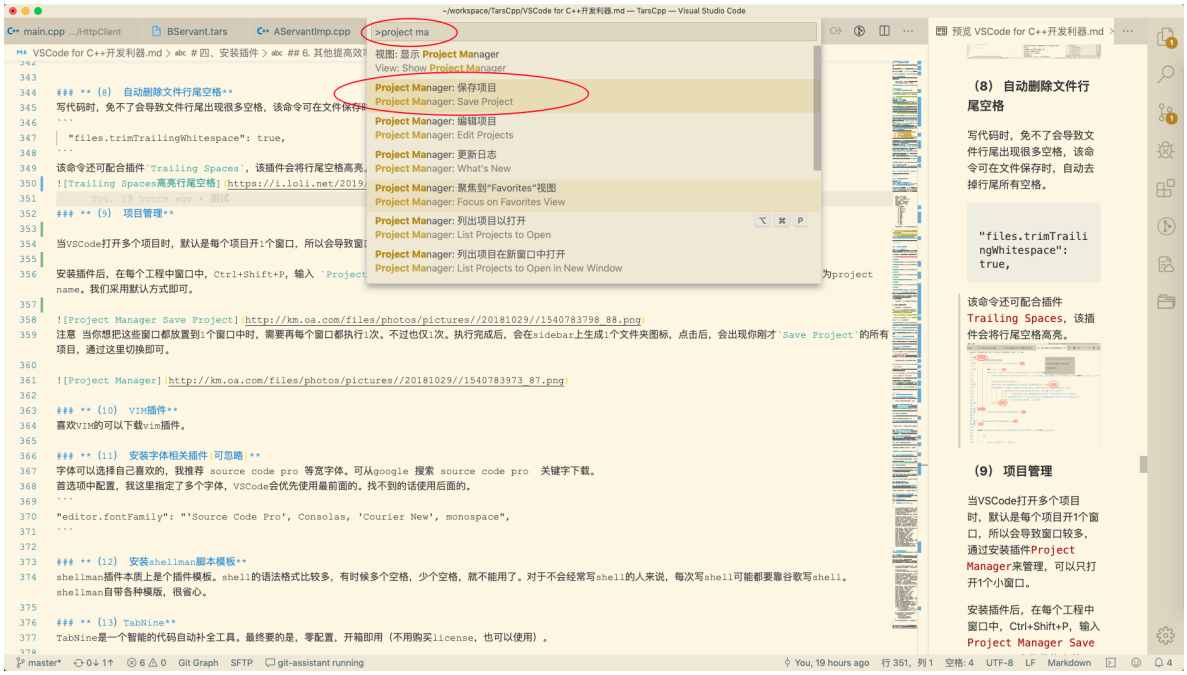
（12）terminal

terminal 是一个能够在 vscode 内自动打开终端命令该行的插件，大家可以尝试。

（13） 项目管理

当 VSCode 打开多个项目时，默认是每个项目开 1 个窗口，所以会导致窗口较多，通过安装插件 `Project Manager` 来管理，可以只打开 1 个小窗口。

安装插件后，在每个工程中窗口中，Ctrl+Shift+P，输入 `Project Manager Save Project` ,意思是将当前工程命名成 1 个 project，它会默认取当前根目录的文件名为 project name。我们采用默认方式即可。



注意 当你想把这些窗口都放置到 1 个窗口中时，需要再每个窗口都执行 1 次。不过也仅 1 次。执行完成后，会在 sidebar 上生成 1 个文件夹图标，点击后，会出现你刚才`Save Project`的所有项目，通过这里切换即可。

（14） VIM 插件

喜欢 VIM 的可以下载 vim 插件。

（15） 安装字体相关插件(可忽略)

字体可以选择自己喜欢的，我推荐 source code pro 等宽字体。可从 google 搜索 source code pro 关键字下载。
首选项中配置，我这里指定了多个字体，VSCode 会优先使用最前面的。找不到的话使用后面的。

```
"editor.fontFamily": "'Source Code Pro', Consolas, 'Courier New', monospace",
```

（16） 安装 shellman 脚本模板

shellman 插件本质上是个插件模板。shell 的语法格式比较多，有时候多个空格，少个空格，就不能用了。对于不会经常写 shell 的人来说，每次写 shell 可能都要靠谷歌写 shell。shellman 自带各种模版，很省心。

5. 附上我的 User Settings.json

```
{
    "editor.fontSize": 13,
    "editor.minimap.maxColumn": 40,
    "editor.wordSeparators": "`~!@#$%^&*()-=+[{]}\\|;:'\",.<>/?·～！￥…（）－
【】、；：‘’“”，。《》？ ",
    "editor.fontFamily": "'Source Code Pro', Consolas, 'Courier New',
monospace",
    "editor.suggest.localityBonus": true,

    "window.title":
"${dirty}${activeEditorLong}${separator}${rootName}${separator}${appName}",
    "window.zoomLevel": 1,
    "window.titleBarStyle": "custom",
```

```
    "workbench.iconTheme": "material-icon-theme",
    "workbench.colorTheme": "Solarized Light",
    "workbench.sideBar.location": "right",

    //"git.path": "D:\\Program Files\\Git\\cmd\\git.exe",
    "git.path": "/usr/bin/git",
    "git.autofetch": true,
    "git.ignoreMissingGitWarning": true,

    "git-assistant.checkConfigVariables": "disabled",
    "git-assistant.pushBeforeClosingIDE": "disabled",

    "diffEditor.ignoreTrimWhitespace": false,
    //"terminal.integrated.shell.windows": "C:\\windows\\System32\\cmd.exe",

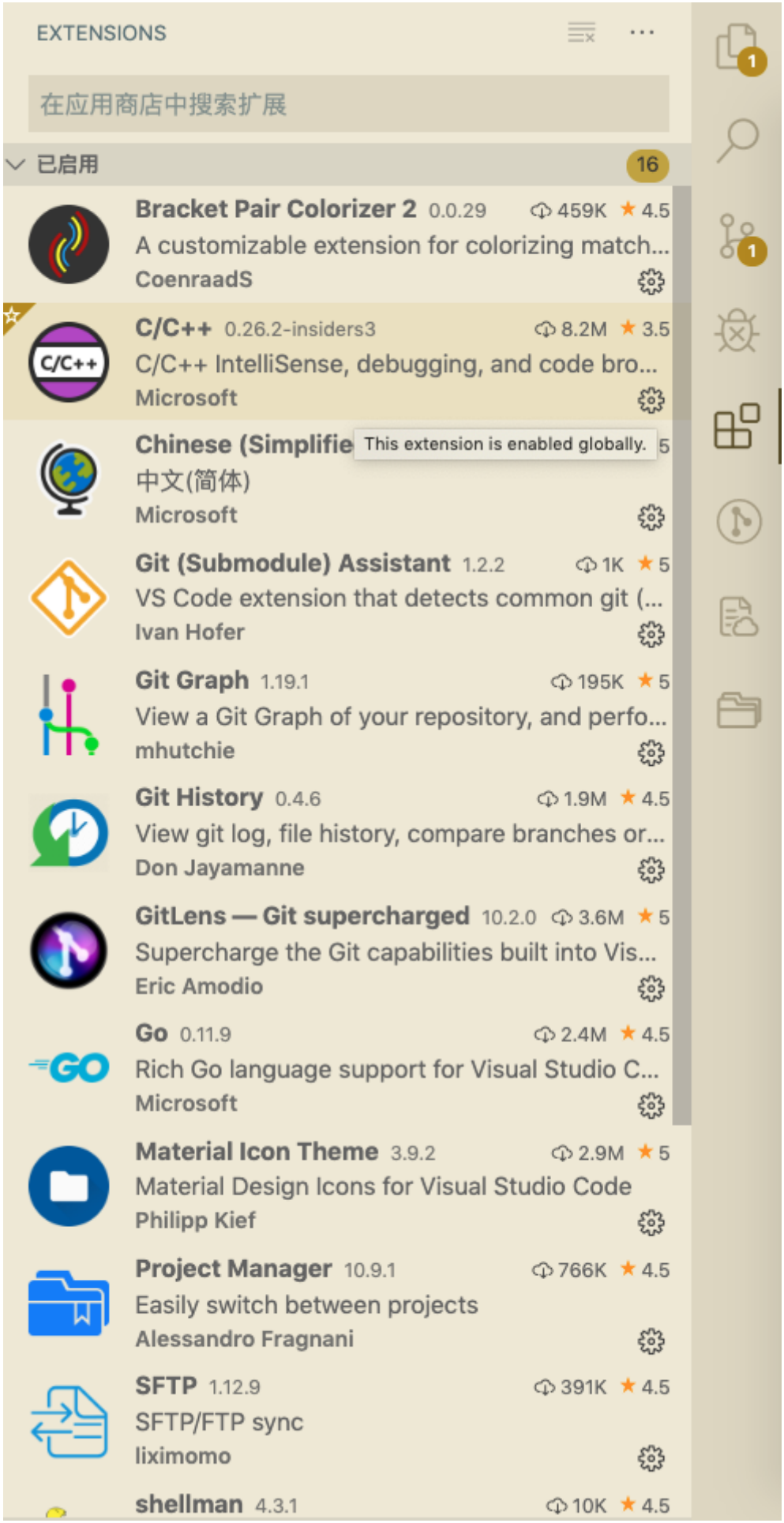
    "http.proxy": "http://公司代理地址IP:端口/",
    "http.proxyStrictSSL": false,

    "files.trimTrailingWhitespace": true,
    "files.autoGuessEncoding": true,
    "files.watcherExclude": {
        "**/.git/objects/**": true,
        "**/.git/subtree-cache/**": true,
        "**/node_modules/*/**": true
    },

    "C_Cpp.updateChannel": "Insiders",
    "C_Cpp.default.intelliSenseMode": "gcc-x64",
    "C_Cpp.default.includePath": [
        "/usr/include",
        "/usr/local/include",
        "${workspaceFolder}/**"
    ],

    "terminal.integrated.fontFamily": "Hack Nerd Font",
    "terminal.integrated.fontSize": 14,
}
```

6. 附上我的插件列表


EXTENSIONS
在应用商店中搜索扩展
已启用
16
Bracket Pair Colorizer 2 0.0.29 459K 4.5
A customizable extension for colorizing match...
CoenraadS
C/C++ 0.26.2-insiders3 8.2M 3.5
C/C++ IntelliSense, debugging, and code bro...
Microsoft
Chinese (Simplifie
This extension is enabled globally.
5
中文(简体)
Microsoft
Git (Submodule) Assistant 1.2.2 1K 5
VS Code extension that detects common git (...
Ivan Hofer
Git Graph 1.19.1 195K 5
View a Git Graph of your repository, and perfo...
mhutchie
Git History 0.4.6 1.9M 4.5
View git log, file history, compare branches or...
Don Jayamanne
GitLens — Git supercharged 10.2.0 3.6M 5
Supercharge the Git capabilities built into Vis...
Eric Amodio
Go 0.11.9 2.4M 4.5
Rich Go language support for Visual Studio C...
Microsoft
Material Icon Theme 3.9.2 2.9M 5
Material Design Icons for Visual Studio Code
Philipp Kief
Project Manager 10.9.1 766K 4.5
Easily switch between projects
Alessandro Fragnani
SFTP 1.12.9 391K 4.5
SFTP/FTP sync
liximomo
shellman 4.3.1 10K 4.5

7. 常见问题

Q：VSCode 不知道为啥一直提示未找到 xxx 的任何定义，之前都是好好的，[流泪]，CTRL+左键或者 F12 都不能跳转。
A：VSCode_cpptools 插件官网有更新，需要重新执行 3.1 操作，下载最新版，更新插件。https://github.com/Microsoft/VSCode-cpptools/releases

Q： VSCode 在 windows10（7）、linux、mac 上体验如何?
A：VSCode 在 win 上使用时，主要受限于 git 是，操作较慢，且对于 C++的系统库文件跳转无法正常解析，还需另外通过 wsl 或者安装其他工具来设置，但在 linux、mac，则可以使用系统自带的 git 和系统库，系统库文件跳转容易很多，且 git 速度很快。

从零把 Mac 打造成开发利器 | 附软件推荐

本文作者：guoweiye，腾讯 PCG 开发工程师

一、基本设置

1. 升级到最新版本操作系统(macOS Mojave 10.14.3)
2. 关闭菜单栏效果, 减少资源占用和产生的热量

> 系统偏好设置/辅助功能/显示, 勾选 (减弱动态效果、减少透明度)

3. 配置睡眠保护

> 系统偏好设置/安全性与隐私/通用, 勾选(进入睡眠或开始保护程序 `立即` 要求输入密码)

4. 配置触发角

> 系统偏好设置/屏幕保护程序/触发角, 选择(右上桌面，左下启动台，右下启动屏幕保护)

二、工具安装

1. Xcode Command Line Tools

> xcode-select --install

2. Homebrew

```
# 这里必须设置 代理地址，否则无法安装brew
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install
```

3. brew cask list
4. brew 安装常用工具

工具名	安装命令	备注
git	brew install git	版本管理工具
wget	brew install wget	下载工具
curl	brew install curl	下载工具
tmux	brew install tmux	终端复用神器
vim	brew install vim	升级 vim 到 8.x 版本，替换系 统自带 vim
JDK	brew cask install java	open jdk java
tree	brew install tree	目录展示工具
lua	brew install lua	lua 编程语言
coreutils	brew install coreutils	gnu 核心工具
bin	brew install binutils	gnu bin 工具
diff	brew install diffutils	gnu 比较工具
find	brew install findutils	gnu 查找工具
ag	brew install the_silver_searcher	高效内容查找 工具, 比 grep、ack 更快的递归搜索文件内容
rzsz	brew install lrzsz	ssh 上传下载文件, 适用于跳 板机环境下, 但和 expect 有冲突
go	brew install go	go 语言
node	brew install node	node
jq	brew install jq	命令行 json 处理工具 与python -m json.tool 功能类似
htop	brew install htop	代替 top
axel	brew install axel	多线程下载工具
cloc	brew install cloc	代码统计工具
shellcheck	brew install shellcheck	shell 脚本检查工具
tldr	brew install tldr	命令行示例
ncdu	brew install ncdu	磁盘空间占用分
glances	brew install glances	监控工具

工具名	安装命令	备注
figlet	brew install figlet	艺术字转换
screenFetch	brew install screenFetch	系统信息
nmap	brew install nmap	网络扫描工具
ctop	brew install ctop	docker 容器监控工具
pstree	brew install pstree	进程树查看
bash-completion	brew install bash-completion	bash补全
lolcat	brew install lolcat	彩虹文字
peco	brew install peco	go 写的极简过滤工具
cowsay	brew install cowsay	ascii 图片
graphviz	brew install graphviz	拓扑图绘制工具,很对关系 绘制工具底层依赖

5. brew cask 安装常用工具

工具名	安装命令	备注
cask	brew tap caskroom/cask	cask
cask update	brew update && brew upgrade brew-cask && brew cleanup	-
iterm2	brew cask install iterm2	终端神器
google-chrome	brew cask install google-chrome	chrome
firefox	brew cask install firefox	firefox
alfred	brew cask install alfred	快捷键工具
dash	brew install dash	文档工具
vscode	brew cask install visual-studio-code	文本编辑器，微软出品，首推
sublimetext	brew cask install sublime-text	文本编辑器， 不再推荐
atom	brew cask install atom	github 出品，太卡, 不再推荐

工具名	安装命令	备注
postman	brew install postman	http 模拟 app
wireshark	brew cask install wireshark	网络抓包 工具
drawio	brew cask install drawio	流程图绘制工具, 可替代 visio
xmind	brew cask install xmind	思维导图工具
virtualbox	brew cask install virtualbox	虚拟机工具，不好用，建议用 vmware fusion 或 PD 代替
macpass	brew cask install macpass	免费的密码管理 工具
beyond-compare	brew cask install beyond-compare	老牌图形化比较工具
youdaodict	brew cask install youdaodict	有道词典
typora	brew cask install typora	markdown 工具，首推
macdown	brew cask install macdown	markdown 工具
docker	brew cask install docker	docker 必备
moom	brew cask install moom	窗口大小调整工具,收费
snip	brew cask install snip	截图工具
the-unarchiver	brew cask install the-unarchiver	解压工具, 可解 rar
keka	brew cask install keka	解压工具
fonts	brew tap caskroom/fonts	安装字体
source-code-pro	brew cask install font-source-code-pro	Adobe 出品编程字体
android-file-transfer	brew cask install android-file-transfer	安卓文件传输工具
licecap	brew cask install licecap	录屏软件
kap	brew cask install kap	视频录屏
sougou	brew cask install sogouinput	搜狗拼音
scroll-reverser	brew cask install scroll-reverser	触摸板滑动反转
go2shell	brew install go2shell	go2shell finder 直接跳转至命令行
rdm	brew install rdm	rdm redis desktop manager
imagemagick	brew install imagemagick	图片编辑工具

6. 其他工具
    - 下载工具 motrix

- 截图工具 jietu
- 看图工具 jietu
- brew cask tools list
- 快速查看工具 Nicer plugins for QuickLook

  > brew cask install qlcolorcode qlstephen qlmarkdown quicklook-json qlimagesize webpquicklook suspicious-package quicklookase qlvideo

- 微软的字体 Consolas

```
   wget -qO- https://raw.githubusercontent.com/yakumioto/YaHei-Consolas-Hybrid-1.12/master/install.sh && sudo install.sh
```

- szrz插件
- graphviz 所用语言 dot 验证

  > dot -h

三、其他配置

1. 设置 iterm2
   - iterm2 配色官网

     > iterm2 > preference > profiles > colors > Color Presets > solarized dark

2. 安装 zsh，oh-my-zsh

```
 # 安装 zsh 及 补全
 brew install zsh zsh-completions

 # 安装 oh-my-zsh
 curl -L https://github.com/robbyrussell/oh-my-zsh/raw/master/tools/install.sh | sh

 # 编辑 /etc/shells
 sudo vim /etc/shells

 # 添加 zsh
 /usr/local/bin/zsh

 # 修改默认shell
 sudo chsh -s /usr/local/bin/zsh

 # 将环境变量移到~/.env.sh
 vim ~/.zshrc

 # 设置主题
 ZSH_THEME=pygmalion
 # 设置插件
 plugins=(git colored-man colorize github jira vagrant virtualenv pip python brew osx zsh-syntax-highlighting)

 # ls 配色生效
```

```
unset LSCOLORS
export CLICOLOR=1
export CLICOLOR_FORCE=1

# 生效
source ~/.env.sh
```

3. solarized 主题配色

```
# clone 之
git clone https://github.com/altercation/solarized

# 配置 vim 主题
cd solarized/vim-colors-solarized/
mkdir -p ~/.vim/colors
cp colors/solarized.vim ~/.vim/colors/

# 配置 vim
vim ~/.vimrc
syntax on
set background=dark
colorscheme solarized
set backspace=2
```

4. vscode 配置命令行中打开

> Open the Command Palette (?? P) and type 'shell command' to find the Shell Command: Install 'code' command in PATH command.

5. vscode 插件安装
    - Chinese language pack(中文语言包)
    - markdownlint(markdown格式检查器)
    - atom one dark theme(ATOM 风格主题)
    - one dark pro(主题)
    - material theme(主题)
    - vscode-icons(图标)
    - python(vscode python扩展包)
    - go(go 扩展包)
6. vscode shell 选择，zsh

> 首选项/设置, 搜索 terminal.integrated.shell.osx ， 设置值为 /usr/local/bin/zsh

7. vscode python 解释器选择

> ??P, 搜索 select interpreter ， 选择一个 env-3.6.8

8. vscode python lint 安装，选择 pylint 即可
9. vscode python 自动格式化

> "python.formatting.provider": "black"

10. python 开发工具

- jetbrains 全家桶
- pycharm Community
- idea Community

四、 python 环境

1. pyenv 安装

```
 # 安装 pyenv 参考, http://einverne.github.io/post/2017/04/pyenv.html
 curl -L https://github.com/pyenv/pyenv-installer/raw/master/bin/pyenv-
installer | bash

 # 添加环境变量
 vim ~/.zshrc
 export PATH="$HOME/.pyenv/bin:$PATH"
 eval "$(pyenv init -)"
 eval "$(pyenv virtualenv-init -)"

 # 常用命令

 # 查看可用版本
 pyenv install -l

 # 安装 python2 最新版本
 pyenv install 2.7.16

 # 安装报错 ERROR: The Python zlib extension was not compiled. Missing the
zlib? https://github.com/pyenv/pyenv/wiki/common-build-problems
 brew install readline xz

 # When running Mojave or higher (10.14+) you will also need to install the
additional SDK headers by downloading them from Apple Developers. You can
also check under /Library/Developer/CommandLineTools/Packages/ as some
versions of Mac OS will have the pkg locally.
 sudo installer -pkg
/Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_macOS_10.
14.pkg -target /

 # 安装 python3 最新版本 3.7.2
 # 报错 ERROR: The Python ssl extension was not compiled. Missing the
OpenSSL lib?
 # https://github.com/pyenv/pyenv/issues/1184#issuecomment-409320318
 CONFIGURE_OPTS="--with-openssl=$(brew --prefix openssl)" pyenv install
3.7.2

 # 安装 python 3.6 , tf 支持的最新版本
 pyenv install 3.6.8

 # 安装 python2 最新版 2.7.16
 pyenv install 2.7.16

 # 创建虚拟环境
 pyenv virtualenv 3.6.4 env-3.6.4

 # 激活虚拟环境
 pyenv activate env-3.6.4

 # 退出虚拟环境
 pyenv deactivate
```

2. 创建 pip 配置文件

```
# 创建 pip 配置目录
mkdir ~/.pip

# 编辑配置
vim ~/.pip/.pip.conf

# 添加以下内容
[list]
format=columns
[global]
download_cache = ~/.cache/pip
index-url = http://mirrors.aliyun.com/pypi/simple/
[install]
trusted-host=mirrors.aliyun.com
```

五、数据库安装

1. 数据库安装

数据库	安装命令	备注
postgresql	brew install postgresql	-
mysql	brew install mysql	-
redis	brew install redis	-

2. 通过 brew service 管理此类后台任务

```
brew tap homebrew/services
brew services start mysql
brew services restart mysql
brew services list
brew services cleanup

# 通过界面管理后台服务
brew cask install launchrocket
```

参考

1. 主要参考
    - awesome-mac
    - open-source-mac-os-apps
    - Best-App
    - mac-setup
    - mac-dev-setup
    - dev-setup

Golang 开发工具必备

本文作者：neonliu，腾讯 CSIG 开发工程师

一. 开发工具

1)sql2go
用于将 sql 语句转换为 golang 的 struct. 使用 ddl 语句即可。
例如对于创建表的语句: show create table xxx. 将输出的语句，直接粘贴进去就行。
http://stming.cn/tool/sql2go.html

2)toml2go
用于将编码后的 toml 文本转换问 golang 的 struct.
https://xuri.me/toml-to-go/

3)curl2go
用来将 curl 命令转化为具体的 golang 代码.
https://mholt.github.io/curl-to-go/

4)json2go
用于将 json 文本转换为 struct.
https://mholt.github.io/json-to-go/

5)mysql 转 ES 工具
http://www.ischoolbar.com/EsParser/

6)golang
模拟模板的工具，在支持泛型之前，可以考虑使用。
https://github.com/cheekybits/genny

7)查看某一个库的依赖情况，类似于 go list 功能
https://github.com/KyleBanks/depth

8)一个好用的文件压缩和解压工具，集成了 zip，tar 等多种功能，主要还有跨平台。
https://github.com/mholt/archiver

9)go 内置命令
go list 可以查看某一个包的依赖关系.
go vet 可以检查代码不符合 golang 规范的地方。

10)热编译工具
https://github.com/silenceper/gowatch

11)revive
golang 代码质量检测工具
https://github.com/mgechev/revive

12)Go Callvis
golang 的代码调用链图工具
https://github.com/TrueFurby/go-callvis

13)Realize
开发流程改进工具
https://github.com/oxequa/realize

14)Gotests
自动生成测试用例工具
https://github.com/cweill/gotests

二.调试工具

1)perf
代理工具，支持内存，cpu，堆栈查看，并支持火焰图.
perf 工具和 go-torch 工具，快捷定位程序问题.
https://github.com/uber-archive/go-torch
https://github.com/google/gops

2)dlv 远程调试
基于 goland+dlv 可以实现远程调式的能力.
https://github.com/go-delve/delve
提供了对 golang 原生的支持，相比 gdb 调试，简单太多。

3)网络代理工具
goproxy 代理，支持多种协议，支持 ssh 穿透和 kcp 协议.
https://github.com/snail007/goproxy

4)抓包工具
go-sniffer 工具，可扩展的抓包工具，可以开发自定义协议的工具包. 现在只支持了 http，mysql，redis，mongodb.
基于这个工具，我们开发了 qapp 协议的抓包。
https://github.com/40t/go-sniffer

5)反向代理工具，快捷开放内网端口供外部使用。
ngrok 可以让内网服务外部调用
https://ngrok.com/
https://github.com/inconshreveable/ngrok

6)配置化生成证书
从根证书，到业务侧证书一键生成.
https://github.com/cloudflare/cfssl

7)免费的证书获取工具
基于 acme 协议，从 letsencrypt 生成免费的证书，有效期 1 年，可自动续期。
https://github.com/Neilpang/acme.sh

8)开发环境管理工具，单机搭建可移植工具的利器。支持多种虚拟机后端。
vagrant常被拿来同 docker 相比，值得拥有。
https://github.com/hashicorp/vagrant

9)轻量级容器调度工具
nomad 可以非常方便的管理容器和传统应用，相比 k8s 来说，简单不要太多.
https://github.com/hashicorp/nomad

10)敏感信息和密钥管理工具
https://github.com/hashicorp/vault

11)高度可配置化的 http 转发工具，基于 etcd 配置。
https://github.com/gojek/weaver

12)进程监控工具 supervisor
https://www.jianshu.com/p/39b476e808d8

13)基于**procFile**进程管理工具. 相比 supervisor 更加简单。
https://github.com/ddollar/foreman

14)基于 http，https，websocket 的**调试代理工具**，配置功能丰富。在线教育的 nohost web 调试工具，基于此开发.
https://github.com/avwo/whistle

15)分布式调度工具
https://github.com/shunfei/cronsun/blob/master/README_ZH.md
https://github.com/ouqiang/gocron

16)自动化运维平台 Gaia
https://github.com/gaia-pipeline/gaia

三. 网络工具

序号	功能分类	命令名	功能描述
1	网络状态	netstat	用来查看网络状态的工具.用来查看网络状态的工具.
2		ping	用来验证机器是否可达和网络延时的工具.
3		tcping	可以代替ping的功能,当对方服务器禁ping的时候,就缺不了这个工具。
4		telnet	用来验证某一个端口是否监听.
5		ss	是可以查看系统中socket的状态的工具,可以替代netstat的部分功能,性能更好。
6		iftop	流量异常飙高,快速排查出哪个进程的锅的工具.
7		nmon	查看内存,网络,cpu占用的综合性工具.
8		route	用于查看和配置机器的路由.
9	链路追踪	traceroute	进行路由查看和检测的工具.
10		mtr	Linux中有一个非常棒的网络连通性判断工具,它结合了ping,traceroute,nslookup的相关特性
11	主机配置	ifconfig	查看和配置机器网卡的功能.
12		ethtool	用于查询及设置网卡参数的命令
13	抓包工具	tcpdump	Linux下功能强大的网络抓包工具.
14	发包工具	nc	有瑞士军刀之称的工具,可以作为tcp|udp服务器,或者作为工具,模拟发送tcp,udp包。作为聊天或者文件传输工具也是没有问题的
15	域名解析	curl	这个就不说了,模拟http请求工具.
16		dig	实现dns解析的功能.
17		nslookup	实现dns解析的功能.
18	流量控制	trickle	是一款轻量级的用户空间带宽控制管理的工具.
19		tc	流量控制控制,可以实现后台网络环境的模拟.
20	端口扫描	hping3	一款路由追踪和端口探测工具。
21		Nmap	主机侦测,端口扫描,类似的工具.

四. 常用网站

go 百科全书: https://awesome-go.com/
json 解析: https://www.json.cn/
出口 IP: https://ipinfo.io/
redis 命令: http://doc.redisfans.com/
ES 命令首页: https://www.elastic.co/guide/cn/elasticsearch/guide/current/index.html
UrlEncode: http://tool.chinaz.com/Tools/urlencode.aspx
Base64: https://tool.oschina.net/encrypt?type=3
Guid: https://www.guidgen.com/
常用工具: http://www.ofmonkey.com/

五. golang 常用库

日志
https://github.com/Sirupsen/logrus
https://github.com/uber-go/zap

配置
兼容 json，toml，yaml，hcl 等格式的日志库.
https://github.com/spf13/viper

存储
mysql https://github.com/go-xorm/xorm
es https://github.com/elastic/elasticsearch
redis https://github.com/gomodule/redigo
mongo https://github.com/mongodb/mongo-go-driver
kafka https://github.com/Shopify/sarama

数据结构
https://github.com/emirpasic/gods

命令行
https://github.com/spf13/cobra

框架
https://github.com/grpc/grpc-go
https://github.com/gin-gonic/gin

并发
https://github.com/Jeffail/tunny
https://github.com/benmanns/goworker
现在我们框架在用的，虽然 star 不多，但是确实好用，当然还可以更好用.
https://github.com/rafaeldias/async

工具
定义了实用的判定类，以及针对结构体的校验逻辑，避免业务侧写复杂的代码.
https://github.com/asaskevich/govalidator
https://github.com/bytedance/go-tagexpr

protobuf 文件动态解析的接口，可以实现反射相关的能力。
https://github.com/jhump/protoreflect

表达式引擎工具
https://github.com/Knetic/govaluate
https://github.com/google/cel-go

字符串处理
https://github.com/huandu/xstrings

ratelimit 工具
https://github.com/uber-go/ratelimit
https://blog.csdn.net/chenchongg/article/details/85342086
https://github.com/juju/ratelimit

golang 熔断的库
熔断除了考虑频率限制，还要考虑 qps，出错率等其他东西.
https://github.com/afex/hystrix-go
https://github.com/sony/gobreaker

表格
https://github.com/chenjiandongx/go-echarts

tail 工具库
https://github.com/hpcloud/taglshi

这才是真正的 Git——Git 内部原理揭秘！

本文作者：lzaneli，腾讯 TEG 前端开发工程师

本文以一个具体例子结合动图介绍了 Git 的内部原理，包括 Git 是什么储存我们的代码和变更历史的、更改一个文件时，Git 内部是怎么变化的、Git 这样实现的好处等等。



通过例子解释清楚上面这张动图，让大家了解 Git 的内部原理。如果你已经能够看懂这张图了，下面的内容可能对你来说会比较基础。

> 本文是 2019/11/24 在深圳腾讯大厦 2 楼多功能厅举办的 FCC 前端分享会（freeCodeConf 2019 深圳站）上分享的文字版。
> 视频：https://www.bilibili.com/video/av77252063
> PPT：https://www.lzane.com/slide/git-under-the-hood

前言

近几年技术发展十分迅猛，让部分同学养成了一种学习知识停留在表面，只会调用一些指令的习惯。我们时常有一种"我会用这个技术、这个框架"的错觉，等到真正遇到问题，才发现事情没有那么简单。

而 Git 也是一个大部分人都知道如何去使用它，知道有哪些命令，却只有少部分人知道具体原理的东西。了解一些底层的东西，可以更好的帮你理清思路，知道你真正在操作什么，不会迷失在 Git 大量的指令和参数上面。

Git 是怎么储存信息的

这里会用一个简单的例子让大家直观感受一下 git 是怎么储存信息的。

首先我们先创建两个文件

```
$ git init
$ echo '111' > a.txt
$ echo '222' > b.txt
$ git add *.txt
```

Git 会将整个数据库储存在 `.git/` 目录下，如果你此时去查看 `.git/objects` 目录，你会发现仓库里面多了两个 object。

```
$ tree .git/objects
.git/objects
├── 58
│   └── c9bdf9d017fcd178dc8c073cbfcbb7ff240d6c
├── c2
│   └── 00906efd24ec5e783bee7f23b5d7c941b0c12c
├── info
└── pack
```

好奇的我们来看一下里面存的是什么东西

```
$ cat .git/objects/58/c9bdf9d017fcd178dc8c073cbfcbb7ff240d6c
xKOR0a044K%
```

怎么是一串乱码？这是因为 Git 将信息压缩成二进制文件。但是不用担心，因为 Git 也提供了一个能够帮助你探索它的 api `git cat-file [-t] [-p]`， `-t` 可以查看 object 的类型，`-p` 可以查看 object 储存的具体内容。

```
$ git cat-file -t 58c9
blob
$ git cat-file -p 58c9
111
```

可以发现这个 object 是一个 blob 类型的节点，他的内容是 111，也就是说这个 object 储存着 a.txt 文件的内容。

这里我们遇到第一种 Git object，blob 类型，它只储存的是一个文件的内容，不包括文件名等其他信息。然后将这些信息经过 SHA1 哈希算法得到对应的哈希值 58c9bdf9d017fcd178dc8c073cbfcbb7ff240d6c，作为这个 object 在 Git 仓库中的唯一身份证。

也就是说，我们此时的 Git 仓库是这样子的：



我们继续探索，我们创建一个 commit。

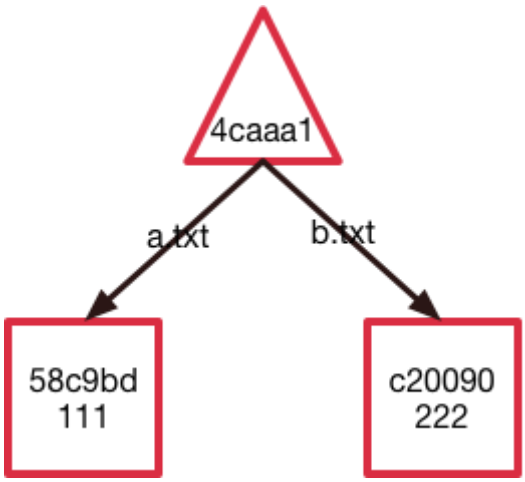
```
$ git commit -am '[+] init'
$ tree .git/objects
.git/objects
├── 0c
|   └── 96bfc59d0f02317d002ebbf8318f46c7e47ab2
├── 4c
|   └── aaa1a9ae0b274fba9e3675f9ef071616e5b209
...
```

我们会发现当我们 commit 完成之后，Git 仓库里面多出来两个 object。同样使用`cat-file`命令，我们看看它们分别是什么类型以及具体的内容是什么。

```
$ git cat-file -t 4caaa1
tree
$ git cat-file -p 4caaa1
100644 blob 58c9bdf9d017fcd178dc8c0...     a.txt
100644 blob c200906efd24ec5e783bee7... b.txt
```

这里我们遇到了第二种 Git object 类型——tree，它将当前的目录结构打了一个快照。从它储存的内容来看可以发现它储存了一个目录结构（类似于文件夹），以及每一个文件（或者子文件夹）的权限、类型、对应的身份证（SHA1 值）、以及文件名。

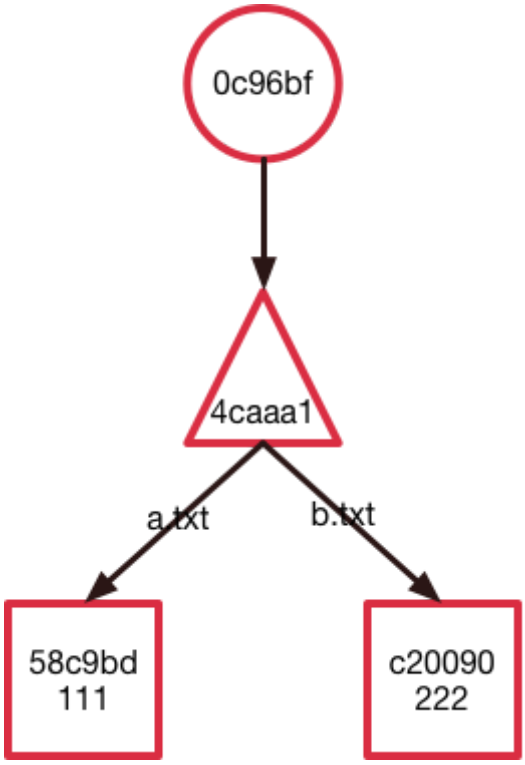
此时的 Git 仓库是这样的：




```
$ git cat-file -t 0c96bf
commit
$ git cat-file -p 0c96bf
tree 4caaa1a9ae0b274fba9e3675f9ef071616e5b209
author lzane 李泽帆  1573302343 +0800
committer lzane 李泽帆  1573302343 +0800
[+] init
```

接着我们发现了第三种 Git object 类型——commit，它储存的是一个提交的信息，包括对应目录结构的快照 tree 的哈希值，上一个提交的哈希值（这里由于是第一个提交，所以没有父节点。在一个 merge 提交中还会出现多个父节点），提交的作者以及提交的具体时间，最后是该提交的信息。

此时我们去看 Git 仓库是这样的：

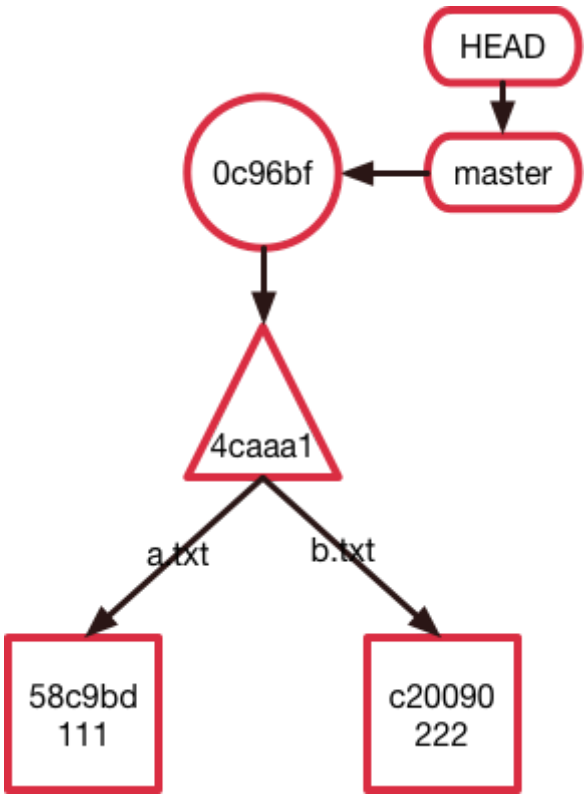


到这里我们就知道 Git 是怎么储存一个提交的信息的了，那有同学就会问，我们平常接触的分支信息储存在哪里呢?

```
$ cat .git/HEAD
ref: refs/heads/master

$ cat .git/refs/heads/master
0c96bfc59d0f02317d002ebbf8318f46c7e47ab2
```

在 Git 仓库里面，HEAD、分支、普通的 Tag 可以简单的理解成是一个指针，指向对应 commit 的 SHA1 值。



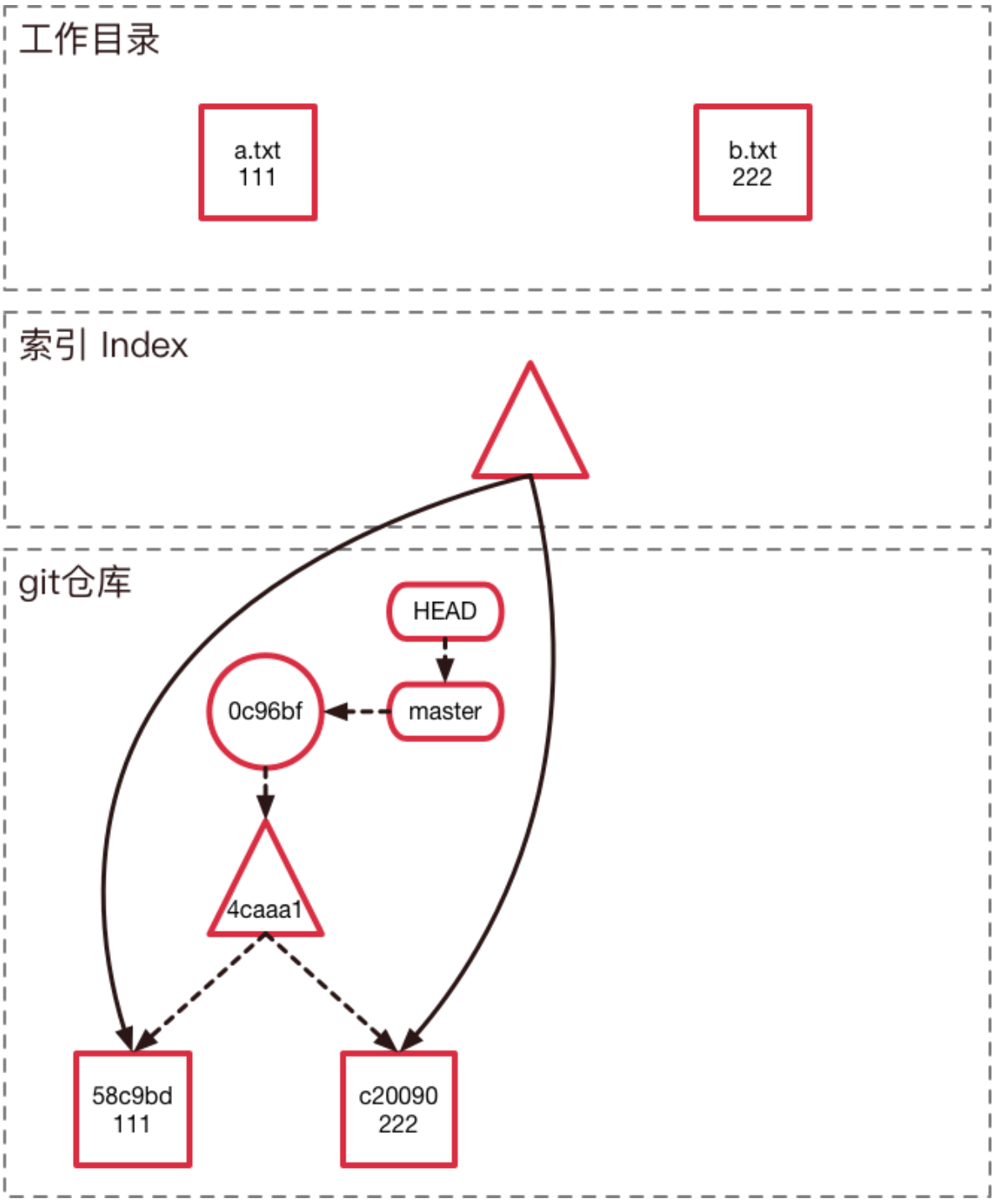
其实还有第四种 Git object，类型是 tag，在添加含附注的 tag（`git tag -a`）的时候会新建，这里不详细介绍，有兴趣的朋友按照上文中的方法可以深入探究。

至此我们知道了 Git 是什么储存一个文件的内容、目录结构、commit 信息和分支的。**其本质上是一个 key-value 的数据库加上默克尔树形成的有向无环图（DAG）**。这里可以蹭一下区块链的热度，区块链的数据结构也使用了默克尔树。

Git 的三个分区

接下来我们来看一下 Git 的三个分区（工作目录、Index 索引区域、Git 仓库），以及 Git 变更记录是怎么形成的。了解这三个分区和 Git 链的内部原理之后可以对 Git 的众多指令有一个“可视化”的理解，不会再经常搞混。

接着上面的例子，目前的仓库状态如下：

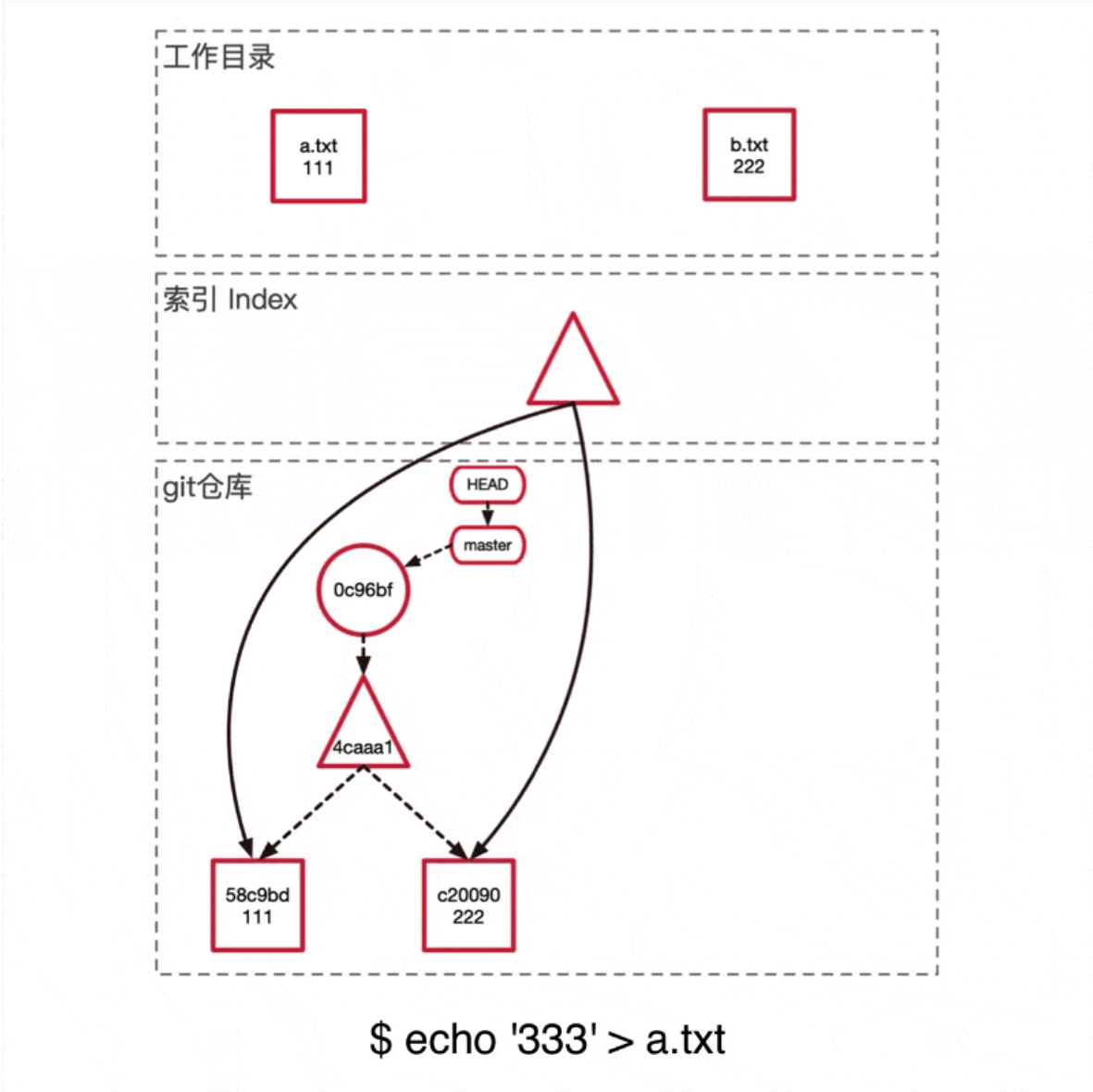


这里有三个区域，他们所储存的信息分别是：

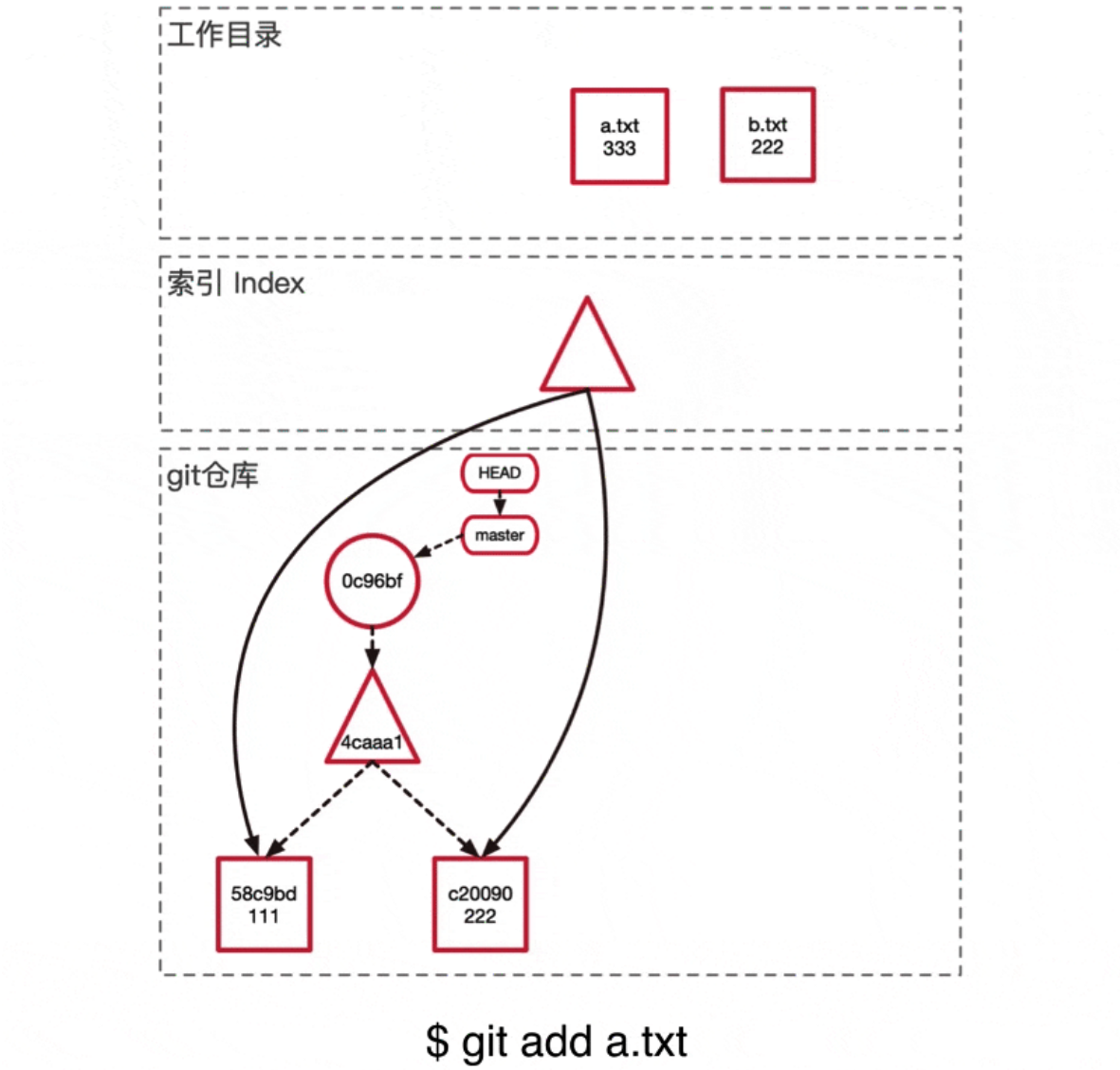
- 工作目录 （ working directory ）：操作系统上的文件，所有代码开发编辑都在这上面完成。

- 索引（index or staging area）：可以理解为一个暂存区域，这里面的代码会在下一次 commit 被提交到 Git 仓库。
- Git 仓库（git repository）：由 Git object 记录着每一次提交的快照，以及链式结构记录的提交变更历史。

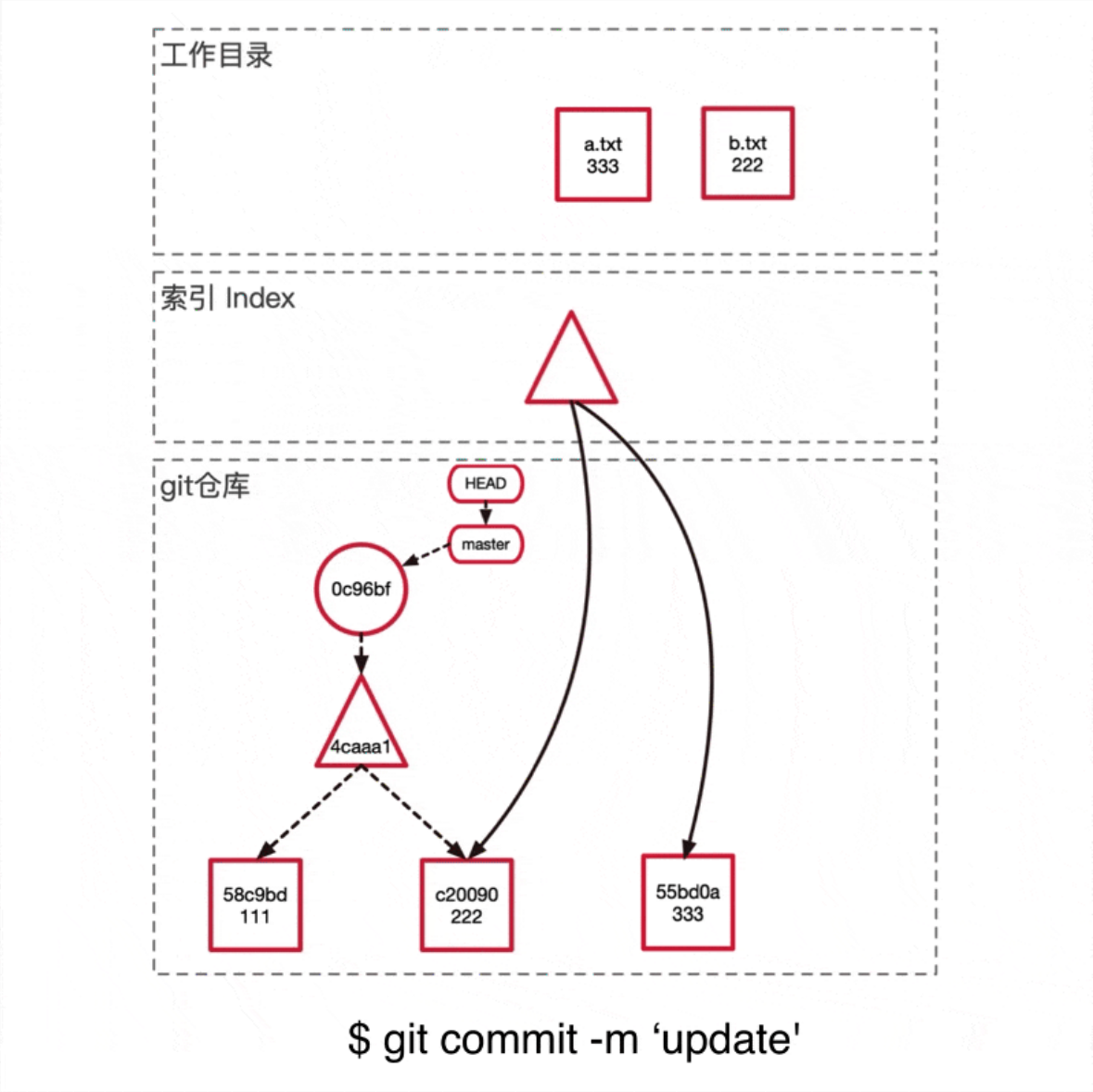
我们来看一下更新一个文件的内容这个过程会发生什么事。



运行`echo "333" > a.txt`将 a.txt 的内容从 111 修改成 333，此时如上图可以看到，此时索引区域和 git 仓库没有任何变化。



运行`git add a.txt`将 a.txt 加入到索引区域，此时如上图所示，git 在仓库里面新建了一个 blob object，储存了新的文件内容。并且更新了索引将 a.txt 指向了新建的 blob object。



运行`git commit -m 'update'`提交这次修改。如上图所示

1. Git 首先根据当前的索引生产一个 tree object，充当新提交的一个快照。
2. 创建一个新的 commit object，将这次 commit 的信息储存起来，并且 parent 指向上一个 commit，组成一条链记录变更历史。
3. 将 master 分支的指针移到新的 commit 结点。

至此我们知道了 Git 的三个分区分别是什么以及他们的作用，以及历史链是怎么被建立起来的。**基本上 Git 的大部分指令就是在操作这三个分区以及这条链。**可以尝试的思考一下 git 的各种命令，试一下你能不能够在上图将它们**“可视化”**出来，这个很重要，建议尝试一下。

如果不能很好的将日常使用的指令“可视化”出来，推荐阅读 图解 Git

一些有趣的问题

有兴趣的同学可以继续阅读，这部分不是文章的主要内容

问题 1：为什么要把文件的权限和文件名储存在 tree object 里面而不是 blob object 呢?

想象一下修改一个文件的命名。

如果将文件名保存在 blob 里面，那么 Git 只能多复制一份原始内容形成一个新的 blob object。而 Git 的实现方法只需要创建一个新的 tree object 将对应的文件名更改成新的即可，原本的 blob object 可以复用，节约了空间。

问题 2：每次 commit，Git 储存的是全新的文件快照还是储存文件的变更部分?

由上面的例子我们可以看到，Git 储存的是全新的文件快照，而不是文件的变更记录。也就是说，就算你只是在文件中添加一行，Git 也会新建一个全新的 blob object。那这样子是不是很浪费空间呢?

这其实是 Git 在空间和时间上的一个取舍，思考一下你要 checkout 一个 commit，或对比两个 commit 之间的差异。如果 Git 储存的是问卷的变更部分，那么为了拿到一个 commit 的内容，Git 都只能从第一个 commit 开始，然后一直计算变更，直到目标 commit，这会花费很长时间。而相反，Git 采用的储存全新文件快照的方法能使这个操作变得很快，直接从快照里面拿取内容就行了。

当然，在涉及网络传输或者 Git 仓库真的体积很大的时候，Git 会有垃圾回收机制 gc，不仅会清除无用的 object，还会把已有的相似 object 打包压缩。

问题 3：Git 怎么保证历史记录不可篡改?

通过 SHA1 哈希算法和哈系树来保证。假设你偷偷修改了历史变更记录上一个文件的内容，那么这个问卷的 blob object 的 SHA1 哈希值就变了，与之相关的 tree object 的 SHA1 也需要改变，commit 的 SHA1 也要变，这个 commit 之后的所有 commit SHA1 值也要跟着改变。又由于 Git 是分布式系统，即所有人都有一份完整历史的 Git 仓库，所以所有人都能很轻松的发现存在问题。

希望大家读完有所收获，下一篇文章会写一些我日常工作中觉得比较实用的 Git 技巧、经常被问到的问题、以及发生一些事故时的处理方法。

参考

- Scott Chacon, Ben Straub - Pro Git-Apress (2014) 免费，有兴趣继续深入的同学推荐阅读这本书
- Jon Loeliger, Matthew McCullough - Version Control with Git, 2nd Edition - O'Reilly Media (2012) 作为上面那本书的补充这才是真正的 Git——Git 内部原理揭秘!

本文作者：lzaneli，腾讯 TEG 前端开发工程师

本文以一个具体例子结合动图介绍了 Git 的内部原理，包括 Git 是什么储存我们的代码和变更历史的、更改一个文件时，Git 内部是怎么变化的、Git 这样实现的好处等等。



通过例子解释清楚上面这张动图，让大家了解 Git 的内部原理。如果你已经能够看懂这张图了，下面的内容可能对你来说会比较基础。

> 本文是 2019/11/24 在深圳腾讯大厦 2 楼多功能厅举办的 FCC 前端分享会（freeCodeConf 2019 深圳站）上分享的文字版。
> 视频：https://www.bilibili.com/video/av77252063
> PPT：https://www.lzane.com/slide/git-under-the-hood

前言

近几年技术发展十分迅猛，让部分同学养成了一种学习知识停留在表面，只会调用一些指令的习惯。我们时常有一种"我会用这个技术、这个框架"的错觉，等到真正遇到问题，才发现事情没有那么简单。

而 Git 也是一个大部分人都知道如何去使用它，知道有哪些命令，却只有少部分人知道具体原理的东西。了解一些底层的东西，可以更好的帮你理清思路，知道你真正在操作什么，不会迷失在 Git 大量的指令和参数上面。

Git 是怎么储存信息的

这里会用一个简单的例子让大家直观感受一下 git 是怎么储存信息的。

首先我们先创建两个文件

```
$ git init
$ echo '111' > a.txt
$ echo '222' > b.txt
$ git add *.txt
```

Git 会将整个数据库储存在`.git/`目录下，如果你此时去查看`.git/objects`目录，你会发现仓库里面多了两个 object。

```
$ tree .git/objects
.git/objects
├── 58
|   └── c9bdf9d017fcd178dc8c073cbfcbb7ff240d6c
├── c2
|   └── 00906efd24ec5e783bee7f23b5d7c941b0c12c
├── info
└── pack
```

好奇的我们来看一下里面存的是什么东西

```
$ cat .git/objects/58/c9bdf9d017fcd178dc8c073cbfcbb7ff240d6c
xKOR0a044K%
```

怎么是一串乱码？这是因为 Git 将信息压缩成二进制文件。但是不用担心，因为 Git 也提供了一个能够帮助你探索它的 api `git cat-file [-t] [-p]`， `-t`可以查看 object 的类型，`-p`可以查看 object 储存的具体内容。

```
$ git cat-file -t 58c9
blob
$ git cat-file -p 58c9
111
```

可以发现这个 object 是一个 blob 类型的节点，他的内容是 111，也就是说这个 object 储存着 a.txt 文件的内容。

这里我们遇到第一种 Git object，blob 类型，它只储存的是一个文件的内容，不包括文件名等其他信息。然后将这些信息经过 SHA1 哈希算法得到对应的哈希值 58c9bdf9d017fcd178dc8c073cbfcbb7ff240d6c，作为这个 object 在 Git 仓库中的唯一身份证。

也就是说，我们此时的 Git 仓库是这样子的：



我们继续探索，我们创建一个 commit。

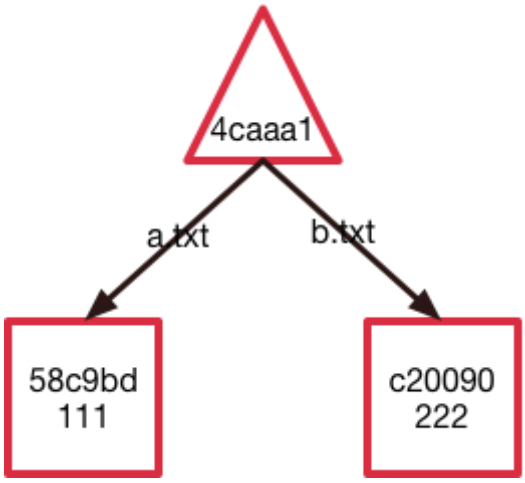
```
$ git commit -am '[+] init'
$ tree .git/objects
.git/objects
├── 0c
|   └── 96bfc59d0f02317d002ebbf8318f46c7e47ab2
├── 4c
|   └── aaa1a9ae0b274fba9e3675f9ef071616e5b209
...
```

我们会发现当我们 commit 完成之后，Git 仓库里面多出来两个 object。同样使用`cat-file`命令，我们看看它们分别是什么类型以及具体的内容是什么。

```
$ git cat-file -t 4caaa1
tree
$ git cat-file -p 4caaa1
100644 blob 58c9bdf9d017fcd178dc8c0...      a.txt
100644 blob c200906efd24ec5e783bee7... b.txt
```

这里我们遇到了第二种 Git object 类型——tree，它将当前的目录结构打了一个快照。从它储存的内容来看可以发现它储存了一个目录结构（类似于文件夹），以及每一个文件（或者子文件夹）的权限、类型、对应的身份证（SHA1 值）、以及文件名。

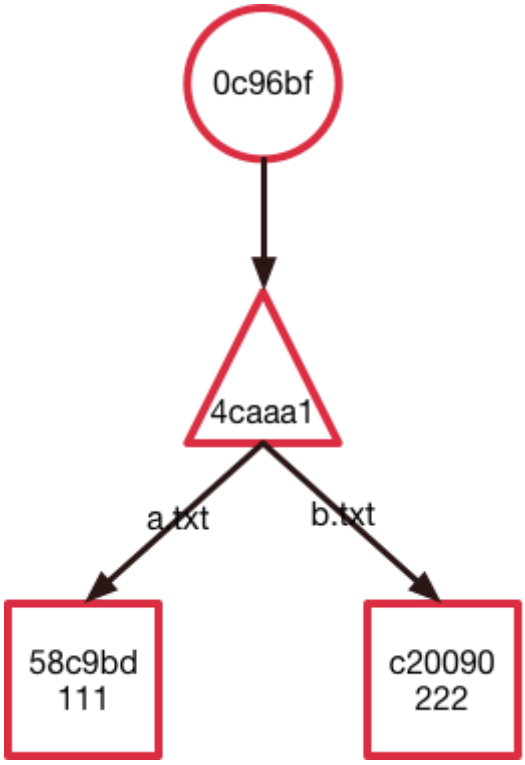
此时的 Git 仓库是这样的：




```
$ git cat-file -t 0c96bf
commit
$ git cat-file -p 0c96bf
tree 4caaa1a9ae0b274fba9e3675f9ef071616e5b209
author lzane 李泽帆  1573302343 +0800
committer lzane 李泽帆  1573302343 +0800
[+] init
```

接着我们发现了第三种 Git object 类型——commit，它储存的是一个提交的信息，包括对应目录结构的快照 tree 的哈希值，上一个提交的哈希值（这里由于是第一个提交，所以没有父节点。在一个 merge 提交中还会出现多个父节点），提交的作者以及提交的具体时间，最后是该提交的信息。

此时我们去看 Git 仓库是这样的：

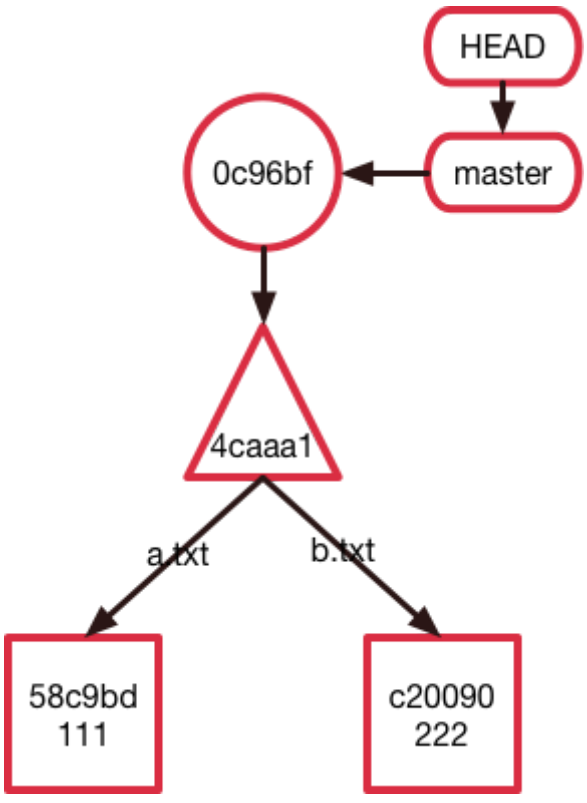


到这里我们就知道 Git 是怎么储存一个提交的信息的了，那有同学就会问，我们平常接触的分支信息储存在哪里呢?

```
$ cat .git/HEAD
ref: refs/heads/master

$ cat .git/refs/heads/master
0c96bfc59d0f02317d002ebbf8318f46c7e47ab2
```

在 Git 仓库里面，HEAD、分支、普通的 Tag 可以简单的理解成是一个指针，指向对应 commit 的 SHA1 值。



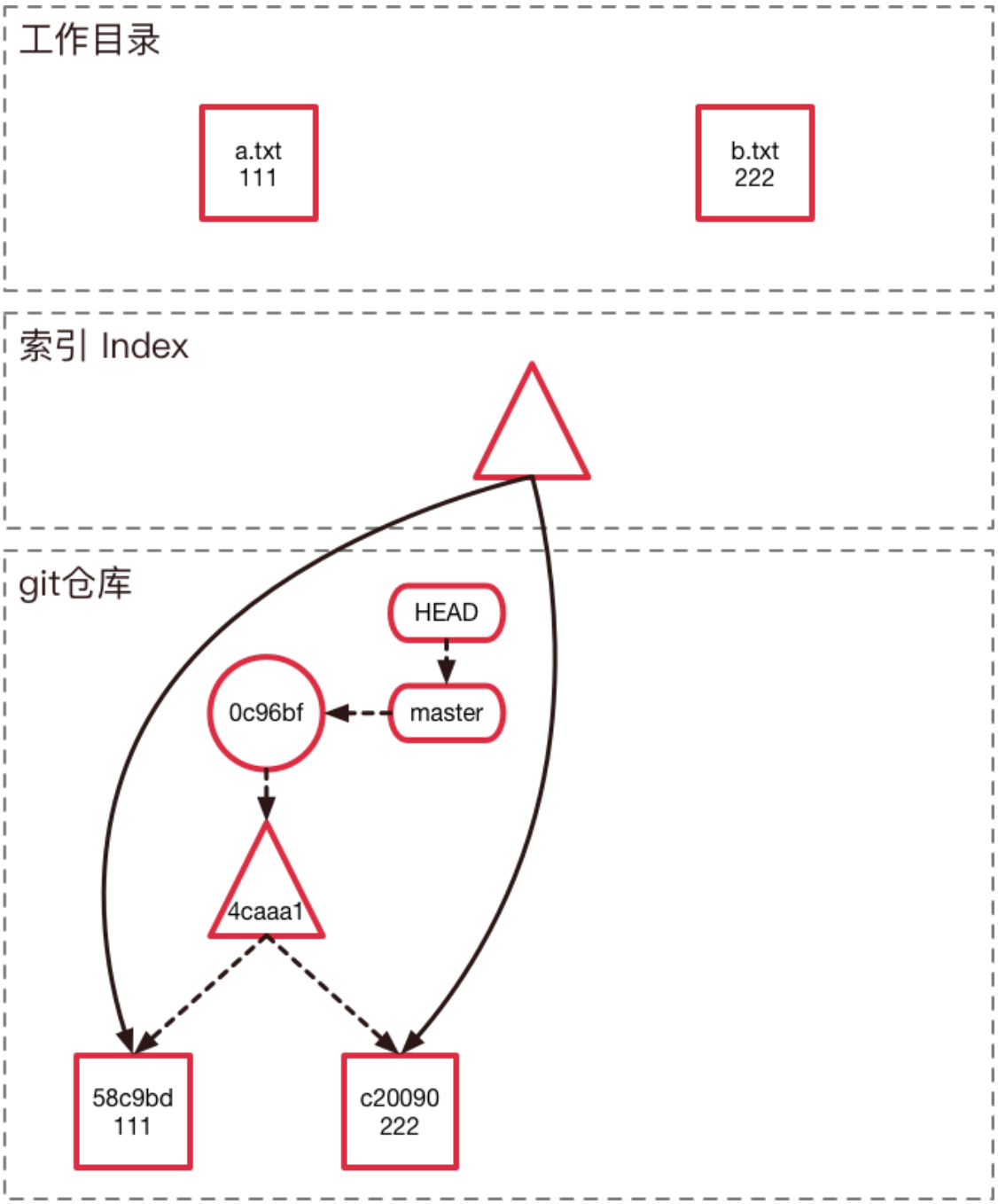
其实还有第四种 Git object，类型是 tag，在添加含附注的 tag（`git tag -a`）的时候会新建，这里不详细介绍，有兴趣的朋友按照上文中的方法可以深入探究。

至此我们知道了 Git 是什么储存一个文件的内容、目录结构、commit 信息和分支的。**其本质上是一个 key-value 的数据库加上默克尔树形成的有向无环图（DAG）**。这里可以蹭一下区块链的热度，区块链的数据结构也使用了默克尔树。

Git 的三个分区

接下来我们来看一下 Git 的三个分区（工作目录、Index 索引区域、Git 仓库），以及 Git 变更记录是怎么形成的。了解这三个分区和 Git 链的内部原理之后可以对 Git 的众多指令有一个“可视化”的理解，不会再经常搞混。

接着上面的例子，目前的仓库状态如下：

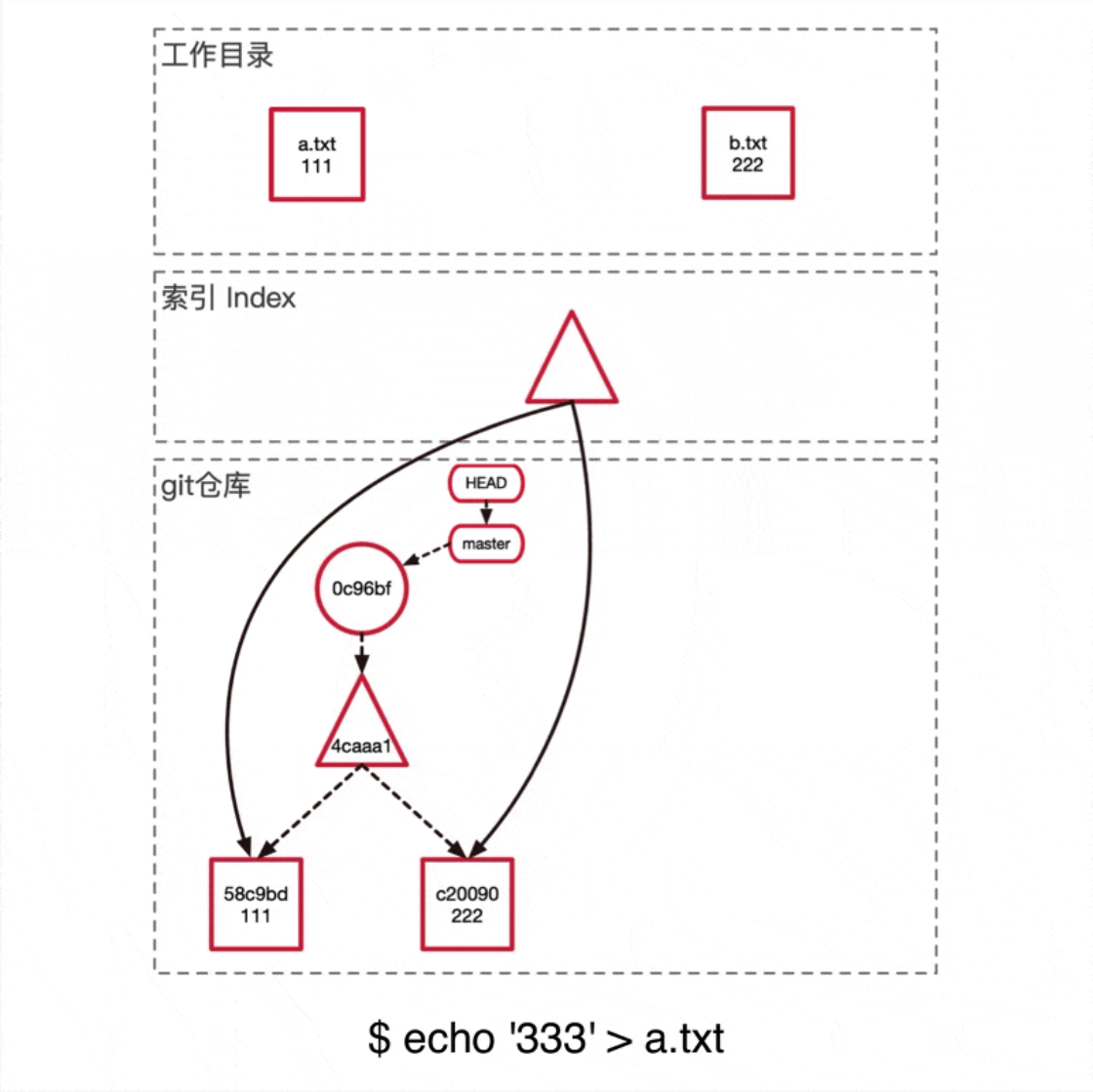


这里有三个区域，他们所储存的信息分别是：

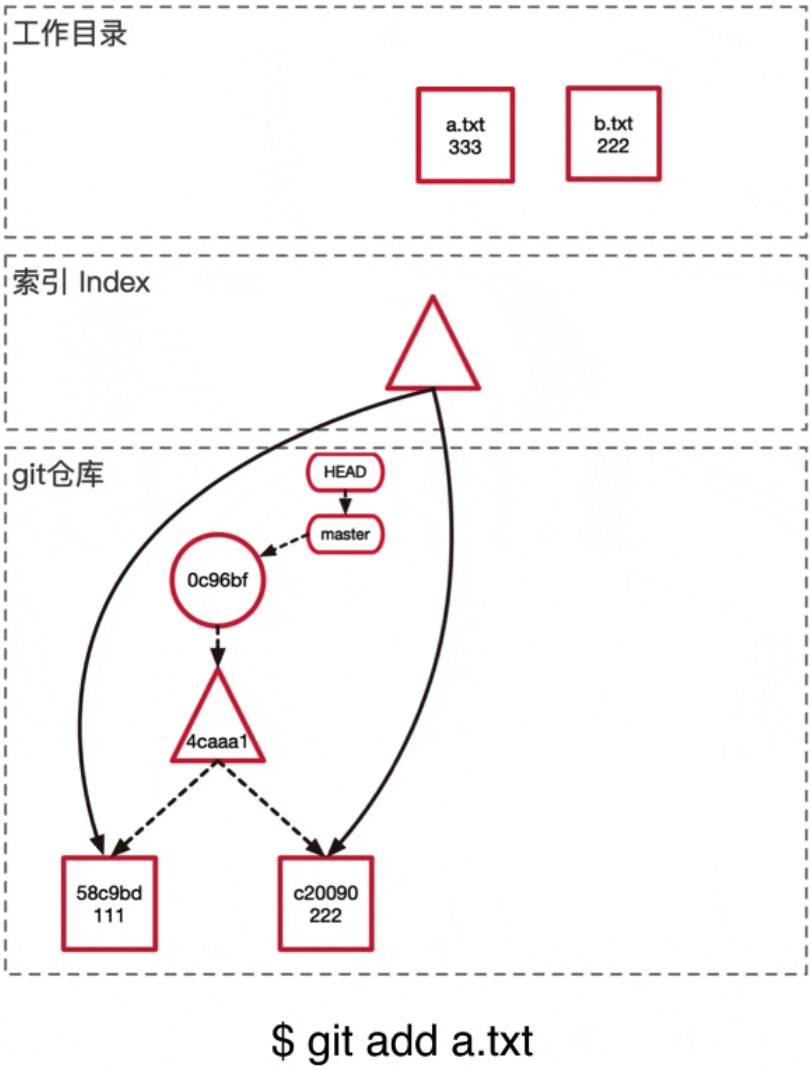
- 工作目录 （ working directory ）：操作系统上的文件，所有代码开发编辑都在这上面完成。
- 索引（ index or staging area ）：可以理解为一个暂存区域，这里面的代码会在下一次 commit 被提交到 Git 仓库。

- Git 仓库（git repository）：由 Git object 记录着每一次提交的快照，以及链式结构记录的提交变更历史。

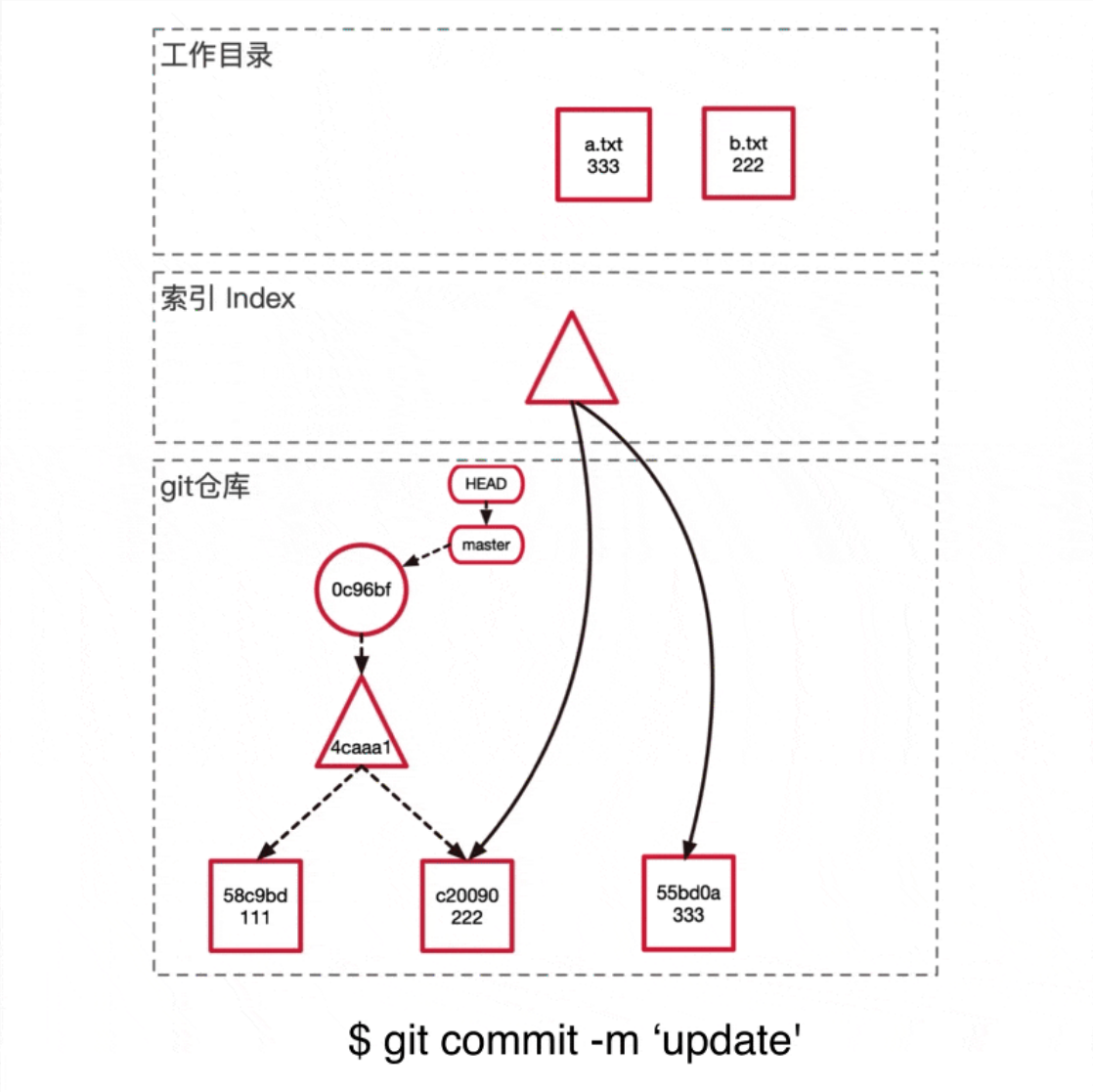
我们来看一下更新一个文件的内容这个过程会发生什么事。



运行`echo "333" > a.txt`将 a.txt 的内容从 111 修改成 333，此时如上图可以看到，此时索引区域和 git 仓库没有任何变化。



运行`git add a.txt`将 a.txt 加入到索引区域，此时如上图所示，git 在仓库里面新建了一个 blob object，储存了新的文件内容。并且更新了索引将 a.txt 指向了新建的 blob object。



运行 `git commit -m 'update'` 提交这次修改。如上图所示

1. Git 首先根据当前的索引生产一个 tree object，充当新提交的一个快照。
2. 创建一个新的 commit object，将这次 commit 的信息储存起来，并且 parent 指向上一个 commit，组成一条链记录变更历史。
3. 将 master 分支的指针移到新的 commit 结点。

至此我们知道了 Git 的三个分区分别是什么以及他们的作用，以及历史链是怎么被建立起来的。**基本上 Git 的大部分指令就是在操作这三个分区以及这条链。**可以尝试的思考一下 git 的各种命令，试一下你能不能够在上图将它们**“可视化”**出来，这个很重要，建议尝试一下。

如果不能很好的将日常使用的指令“可视化”出来，推荐阅读 图解 Git

一些有趣的问题

有兴趣的同学可以继续阅读，这部分不是文章的主要内容

问题 1：为什么要把文件的权限和文件名储存在 tree object 里面而不是 blob object 呢?

想象一下修改一个文件的命名。

如果将文件名保存在 blob 里面，那么 Git 只能多复制一份原始内容形成一个新的 blob object。而 Git 的实现方法只需要创建一个新的 tree object 将对应的文件名更改成新的即可，原本的 blob object 可以复用，节约了空间。

问题 2：每次 commit，Git 储存的是全新的文件快照还是储存文件的变更部分?

由上面的例子我们可以看到，Git 储存的是全新的文件快照，而不是文件的变更记录。也就是说，就算你只是在文件中添加一行，Git 也会新建一个全新的 blob object。那这样子是不是很浪费空间呢?

这其实是 Git 在空间和时间上的一个取舍，思考一下你要 checkout 一个 commit，或对比两个 commit 之间的差异。如果 Git 储存的是问卷的变更部分，那么为了拿到一个 commit 的内容，Git 都只能从第一个 commit 开始，然后一直计算变更，直到目标 commit，这会花费很长时间。而相反，Git 采用的储存全新文件快照的方法能使这个操作变得很快，直接从快照里面拿取内容就行了。

当然，在涉及网络传输或者 Git 仓库真的体积很大的时候，Git 会有垃圾回收机制 gc，不仅会清除无用的 object，还会把已有的相似 object 打包压缩。

问题 3：Git 怎么保证历史记录不可篡改?

通过 SHA1 哈希算法和哈系树来保证。假设你偷偷修改了历史变更记录上一个文件的内容，那么这个问卷的 blob object 的 SHA1 哈希值就变了，与之相关的 tree object 的 SHA1 也需要改变，commit 的 SHA1 也要变，这个 commit 之后的所有 commit SHA1 值也要跟着改变。又由于 Git 是分布式系统，即所有人都有一份完整历史的 Git 仓库，所以所有人都能很轻松的发现存在问题。

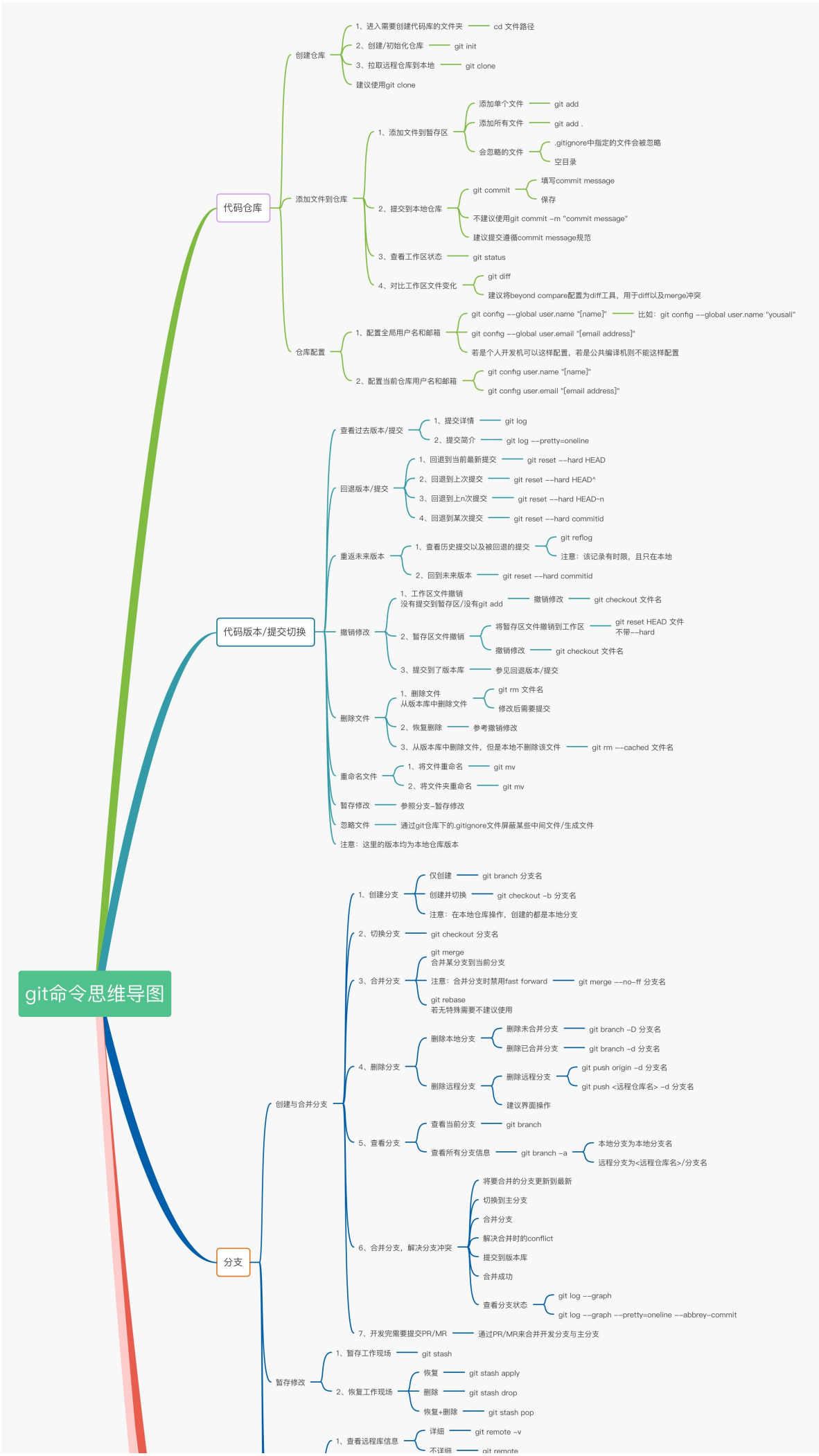
希望大家读完有所收获，下一篇文章会写一些我日常工作中觉得比较实用的 Git 技巧、经常被问到的问题、以及发生一些事故时的处理方法。

参考

- Scott Chacon, Ben Straub - Pro Git-Apress (2014) 免费，有兴趣继续深入的同学推荐阅读这本书
- Jon Loeliger, Matthew McCullough - Version Control with Git, 2nd Edition - O'Reilly Media (2012) 作为上面那本书的补充

如何优雅地使用Git?

作者：yousali，腾讯 CSIG 后台开发工程师


git命令思维导图
代码仓库
创建仓库
1、进入需要创建代码库的文件夹
cd 文件路径
2、创建/初始化仓库
git init
3、拉取远程仓库到本地
git clone
建议使用git clone
添加文件到仓库
1、添加文件到暂存区
添加单个文件
git add
添加所有文件
git add .
会忽略的文件
.gitignore中指定的文件会被忽略
空目录
2、提交到本地仓库
git commit
填写commit message
保存
不建议使用git commit -m "commit message"
建议提交遵循commit message规范
3、查看工作区状态
git status
4、对比工作区文件变化
git diff
建议将beyond compare配置为diff工具，用于diff以及merge冲突
仓库配置
1、配置全局用户名和邮箱
git config --global user.name "[name]"
比如：git config --global user.name "yousali"
git config --global user.email "[email address]"
若是个人开发机可以这样配置，若是公共编译机则不能这样配置
2、配置当前仓库用户名和邮箱
git config user.name "[name]"
git config user.email "[email address]"
代码版本/提交切换
查看过去版本/提交
1、提交详情
git log
2、提交简介
git log --pretty=oneline
回退版本/提交
1、回退到当前最新提交
git reset --hard HEAD
2、回退到上次提交
git reset --hard HEAD^
3、回退到上n次提交
git reset --hard HEAD~n
4、回退到某次提交
git reset --hard commitid
重返未来版本
1、查看历史提交以及被回退的提交
git reflog
注意：该记录有时限，且只在本地
2、回到未来版本
git reset --hard commitid
撤销修改
1、工作区文件撤销
没有提交到暂存区/没有git add
撤销修改
git checkout 文件名
2、暂存区文件撤销
将暂存区文件撤销到工作区
git reset HEAD 文件
不带--hard
撤销修改
git checkout 文件名
3、提交到了版本库
参见回退版本/提交
删除文件
1、删除文件
从版本库中删除文件
git rm 文件名
修改后需要提交
2、恢复删除
参考撤销修改
3、从版本库中删除文件，但是本地不删除该文件
git rm --cached 文件名
重命名文件
1、将文件重命名
git mv
2、将文件夹重命名
git mv
暂存修改
参照分支-暂存修改
忽略文件
通过git仓库下的.gitignore文件屏蔽某些中间文件/生成文件
注意：这里的版本均为本地仓库版本
分支
创建与合并分支
1、创建分支
仅创建
git branch 分支名
创建并切换
git checkout -b 分支名
注意：在本地仓库操作，创建的都是本地分支
2、切换分支
git checkout 分支名
3、合并分支
git merge
合并某分支到当前分支
注意：合并分支时禁用fast forward
git merge --no-ff 分支名
git rebase
若无特殊需要不建议使用
4、删除分支
删除本地分支
删除未合并分支
git branch -D 分支名
删除已合并分支
git branch -d 分支名
删除远程分支
删除远程分支
git push origin -d 分支名
git push <远程仓库名> -d 分支名
建议界面操作
5、查看分支
查看当前分支
git branch
查看所有分支信息
git branch -a
本地分支为本地分支名
远程分支为<远程仓库名>/分支名
6、合并分支，解决分支冲突
将要合并的分支更新到最新
切换到主分支
合并分支
解决合并时的conflict
提交到版本库
合并成功
查看分支状态
git log --graph
git log --graph --pretty=oneline --abbrev-commit
7、开发完需要提交PR/MR
通过PR/MR来合并开发分支与主分支
暂存修改
1、暂存工作现场
git stash
2、恢复工作现场
恢复
git stash apply
删除
git stash drop
恢复+删除
git stash pop
1、查看远程库信息
详细
git remote -v
不详细
git remote


多人协作
2、更新/推送远程库
更新远程库信息
git fetch
将远程库最新修改更新到本地
git pull
git pull可以认为是
git fetch+git merge
将本地修改推送到远程库
git push
git push origin 分支名
3、本地分支与远程分支交互
使用远程分支A创建本地分支
git checkout -b A origin/A
origin是远程仓库名，若名字一样origin/A可以省略
将本地分支与远程分支作关联
git branch --set-upstream A origin/A
提示no tracking information错误
建议开发遵循或者参照git标准工作流，比如git flow、github flow或者gitlab flow
代码版本tag
1、查看tag
本地tag
git tag
远程tag
git tag -r
2、操作tag
添加tag
给当前版本添加tag
git tag 标签名
给历史版本添加tag
git tag 标签名 commitid
删除tag
删除本地标签
git tag -d 标签名
删除远程标签
git push origin -d 标签名
推送到远端仓库
git push origin 标签名
推送所有未提交的tag
git push origin --tags
更新到本地
git pull origin --tags
tag与branch的操作基本一致，因为tag就是一个仅可读的branch
其他生僻命令
git blame
git bisect
过二分查找定位引入 bug 的变更
git relog
...
可以使用git help查看git常用的命令，使用git help -a查看git可用的所有命令

腾讯技术文章合集

本文档内容来自「腾讯技术工程」



腾讯技术工程

扫描上方二维码关注回复「腾讯技术」：获取400+页腾讯技术文章精华合集
腾讯技术官方 QQ 交流群：957411539

2020.5