NO.34

编程狂人

Programming Madman

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:
http://www.tuicool.com/mags/53ccd678d91b1457ac000088

欢迎下载推酷客户端体验更多阅读乐趣

目录

01.PHP NG (PHP 5.7) 性能比 PHP 5.6 提升近 1 倍

02.颜海镜的博客： JavaScript原型之路

03.说说最近Google:safebrowsing引发页面加载阻塞的问题

04.七牛首席布道师：GO不是在颠覆，就是在逆袭

05.聊聊JVM的年轻代

06.整理对Spark SQL的理解

07.TSRC：主流WAF架构分析与探索

08.近距离端详Android ART运行时库

09.利用QuincyKit + KSCrash构建自己的Crash Log收集与管理系统

10.冯森林：手机淘宝中的那些Web技术

PHP NG (PHP 5.7) 性能比 PHP 5.6 提升近 1 倍

作者：oschina

PHP NG （你要愿意的话叫 PHP 5.7 也行）目前还在 alpha 开发阶段，但已经显示出惊人的性能提升。关键是仍保持对 PHP 5.6 的兼容性。

Dmitry Stogov 在今年1月中旬的 首次发布 以及5月初的 里程碑更新 后对 PHP 速度的提升有着越来越多的思路（特别贡献者来自 Xinchen Hui @雪候鸟 , Nikita Popov 等）.

到了7月中旬这些努力终于有了结果，测试表明开发中的版本性能对比 PHP 5.6 有着近乎 1 倍的提升。测试是在渲染 WordPress 3.6 前端页面上进行的。

同样的页面，PHP 5.6 渲染 1000 次耗时 26.756 秒，而 PHP NG 耗时 14.810 秒。这还没结束，你可以通过 http://wiki.php.net/phpng 了解更多目标和备注。

此次性能提升的秘诀在于将近 60% 的 CPU 指令被替换成更高效的代码。PHP 5.6 执行 100 次渲染需要 9,413,106,833 个 CPU 指令，而 PHP NG 只需 3,627,440,773 指令。

因为多数扩展都可支持 PHP NG，因此你可以轻松的构建自己的环境进行测试。因为有大量的计划，因此 PHP NG 今年无望发布稳定版本，希望 2015 年能有稳定的 Beta 甚至是 RC 版本发布。

原文链接：http://www.oschina.net/news/53677/php-5-7-twice-as-fast

JavaScript原型之路

译者：颜海镜

简介

最近我在学习Frontend Masters 上的高级JavaScript系列教程，Kyle 带来了他的“OLOO”（对象链接其他对象）概念。这让我想起了Keith Peters几年前发表的一篇博文，关于学习没有“new”的世界，其中解释了使用原型继承代替构造函数。两者都是纯粹的原型编码。

标准方法（The Standard Way）

一直以来，我们学习的在 JavaScript 里创建对象的方法都是创建一个构造函数，然后为函数的原型对象添加方法。

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.getName = function() {
  return this.name;
};
```

对于子类的解决方案是，创建一个新的构造函数，并且设置其原型为其父类的原型。调用父类的构造函数，并将this设置为其上下文对象。

```
function Dog(name) {
  Animal.call(this, name);
}
```

```
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.speak = function() {
  return "woof";
};

var dog = new Dog("Scamp");
console.log(dog.getName() + ' says ' + dog.speak());
```

原型方法（The Prototypal Way）

如果你接触过任何原型语言，你会觉得上面的例子看起来很奇怪。我尝试过 IO 语言——一门基于原型的语言。在原型语言中，可以通过克隆对象并添加属性和方法的方式创建一个原型。然后你能克隆刚才创建的原型，从而创建一个可以使用的实例，或者克隆它来创建另一个原型。上面的例子在 IO 里，看起来像下面这样：

```
Animal := Object clone
Animal getName := method(name)

Dog := Animal clone
Dog speak := method("woof")

dog := Dog clone
dog name := "Scamp"
writeln(dog getName(), " says ", dog speak())
```

好消息（The Good News）

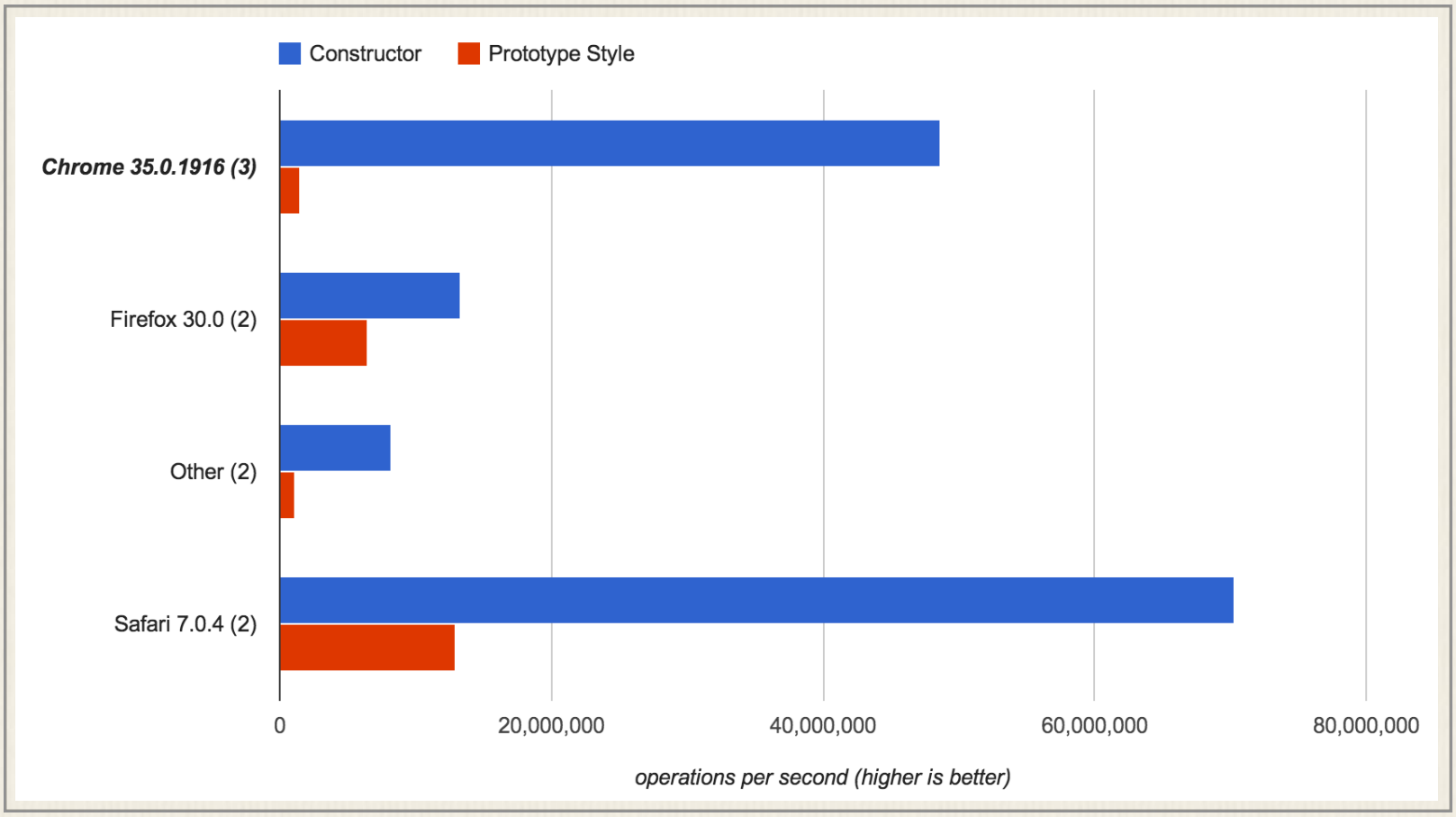
在JavaScript中，也可以使用这种编码方式！Object.create 函数和 IO 里的 clone 类似。下面是在JavaScript 中，纯原型的实现。除了语法不同之外，和 IO 版本一样。

```
Animal = Object.create(Object);

Animal.getName = function() {

  return this.name;

};


Dog = Object.create(Animal);

Dog.speak = function() {

  return "woof";

};


var dog = Object.create(Dog);

dog.name = "Scamp";

console.log(dog.getName() + ' says ' + dog.speak());
```

坏消息（The Bad News）

当使用构造函数时，JavaScript 引擎会进行优化。在 JSPerf 上测试两个不同的操作，显示基于原型的实现比使用构造函数的方式最多慢90多倍。

另外，如果你使用类似 Angular 的框架，当创建控制器和服务时，必须使用构造函数。



引入类（Enter Classes）

ES6带来了新的 class 语法。但其只是标准构造函数方法的语法糖。新的语法看起来更像 Java 或 c#，但其幕后仍然是创建原型对象。这会让来自基于类语言的人感到迷惑，因为当创建原型时，他们希望类和他们的语言有相同的属性。

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  getName() {
    return this.name;
  }
```

```
}

class Dog extends Animal {
  constructor(name) {
    super(name);
  }
  speak() {
    return "woof";
  }
}

var dog = new Dog("Scamp");
console.log(dog.getName() + ' says ' + dog.speak());
```

结论（Conclusion）

如果让我选择，我会用纯原型的风格。这更具有表现力，动态和有趣。由于虚拟机会对构造函数方法进行优化，所有框架都会选择构造函数方法，在产品代码中，我会继续使用构造函数。一旦 ES6 变得流行，我希望使用新的类语法代替古老的构造函数方法。

译文链接：http://yanhaijing.com/javascript/2014/07/18/javascript-prototype/

英文原文链接：http://jurberg.github.io/blog/2014/07/12/javascript-prototype/

说说最近Google:safebrowsing引发页面加载阻塞的问题

作者：理论弟

背景

一个礼拜前，在退款维权的业务中，发现这样一个问题： 在某些Firefox浏览器中，表单的butterfly加载阻塞导致功能异常了。

一开始，我们以为是即将发布的修改点导致的问题。

但再三确认本次的修改点后，确定只是改了文案啊！ 这…

因此，我们首先怀疑是否线上已经有问题？ 经过测试发现，果然，确实是个线上问题。

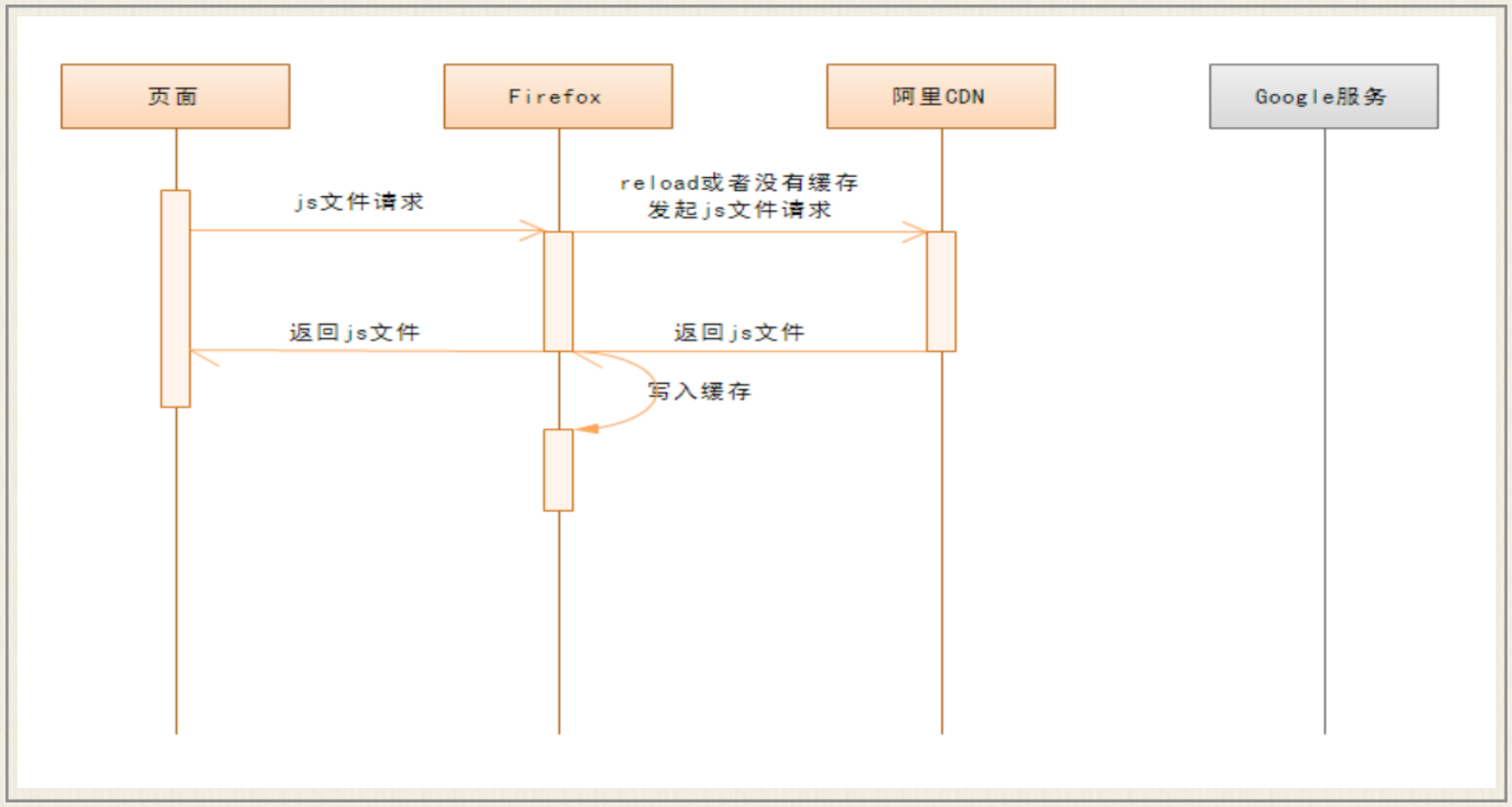
经过并不算麻烦的自测后，发现问题还不小：

影响范围：所有Firefox版本。
（注意：Firefox略有修复，但未完全解决：30.00+官方版Firefox浏览器；原因是该版本携带有“附加组件管理器”1.2.3版本这个扩展）

chrome在一个月前也发现过类似问题，不过近期没有类似案例，可能已经在35中修复。

隐藏在Firefox中的Google:safebrowsing是如何工作的?

印象中，页面在Firefox 中请求一个JS文件的过程，是这样的



图中看到了三个步骤：

1. 页面向Firefox发起一个js的请求；

2. Firefox判断是否为reload请求或者已在缓冲，if true直接读缓存，返回js文件给页面使用；if false则执行步骤3；

3. 向阿里CDN请求，获取到js文件，返回js文件给页面使用，并且写入缓冲；

看起来没什么问题，太符合我们的认知了。

直到有一天，发现页面因为js加载阻塞导致页面功能严重受影响。

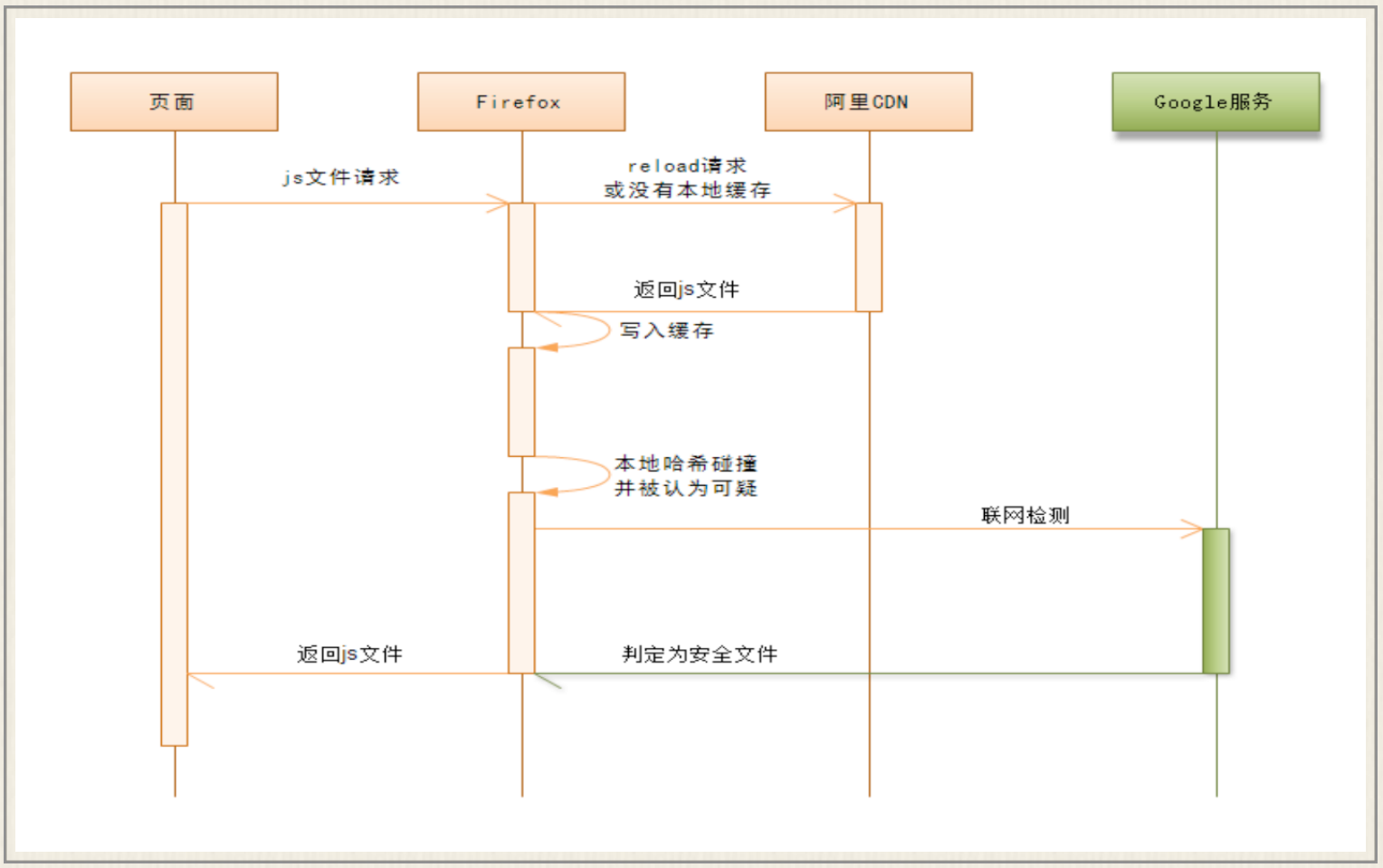
而且怎么强刷、清理缓存，都无果。

才发现....

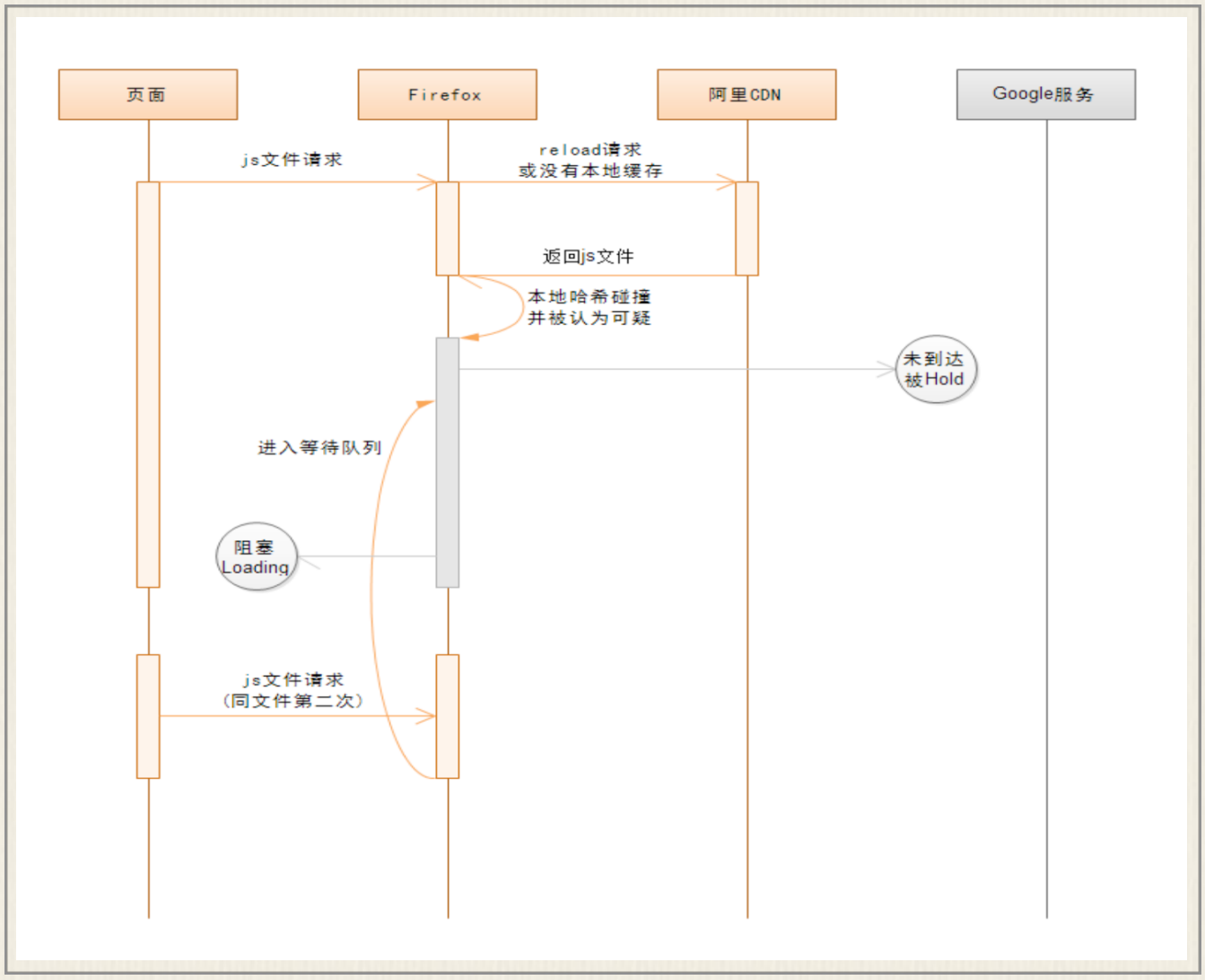
原来，页面在Firefox 中请求一个JS文件的真正过程，是这样的

这里演示下Google服务正常和异常两种情况：

一. Google服务正常时：



二. Google服务不正常时:



图中可以看到多了个本地哈希碰撞和向Google服务发送检测的过程，更多步骤描述如下：

1. cdn返回js给Firefox后，Firefox先在本地safebrowsing哈希库中执行哈希碰撞；
2. 碰撞结果为不通过的话，会向Google服务器（safebrowsing.google.com:443）发起检测通知，并进入等待状态；
3. 此时，页面的直观感受就是js没有请求到，功能异常；
4. 并且，如果再次请求同一个js文件，首先会查看safebrowsing 等待队列；
5. 如果已在safebrowsing等待队列中，则什么都不做。没错！什么都不做！（这就解释了，为什么强刷，甚至清理缓存后的强刷，都毫无作用）

似乎已经定位到了，问题了

第一： Firefox 把我们的js判定为可疑文件（也就是哈希碰撞结果为可疑）；

第二： Google服务被墙了， 哎~~

另外：

1. safebrowsing会对所有静态资源执行哈希碰撞；
2. 哈希碰撞，只针对静态资源的名称；（也就是说与js文件的文件内容无关）

于是，向Firefox的同学请求帮助

无辜的受害者：“为什么Firefox把我们的js认定为可疑文件？”

Firefox的解释是：“我也不知道为什么啊！”；

Firefox：“因为：Firefox浏览器会从Google下来一份safebrowsing哈希库，这个哈希库会对所有静态资源请求执行哈希碰撞。 因此，具体算法，Firefox方面，其实也是不知道的呢。”

无辜的受害者：“哦！ 原来是被检查请求阻塞了呢？那要不你们添加一个阻塞行为的监控吧？”

Firefox：“对啊，对啊，在最新版本：30.00中，我们加了阻塞情况下，只进行本地检查的处理了， 你下载最新的版本就可以了。”

… 几个小时后 …

无辜的受害者：“我下更新到最新版本了，可还是不行呢？ ”

Firefox：“要怪就怪gfw吧，为了伪造成是Google自己服务器的问题，而不是国内封锁，现在gfw对Google的封锁采用hold连接，不释放，不重置，一直不返回数据，造成现在火狐以为连接一直在等待数据”

无辜的受害者：“这….”, 尼玛，不是坑爹吗？ “好的谢谢， 保持联系”。

因此，又抛出了两个新的问题：

1. 为什么升级了Firefox还是在本地哈希碰撞中被fail了？
2. 为什么Firefox已经添加了防阻塞处理，结果却还是被阻塞了？

问题的答案：

1. Firefox升级是升级了，但是本地哈希库，并没有被更新（因为Google服务调不到啊~）； 自测发现哈希库被保存在Firefox 的应用程序中，而非个人配置目录中；
2. Firefox添加了阻塞处理没错，但是gfw似乎做了一件奇葩的事情：gfw对Google的封锁采用hold连接，不释放，不重置，一直不返回数据，造成现在火狐以为连接一直在等待数据； 看来是Fire-
fox的解决方案不够彻底啊；

最后的问题变成：

1. Firefox没有处理到gfw对Google服务请求采取hold连接的情况；

2. 没有得到能够让用户一次操作（如更新插件，更新浏览器）就能解决的方案，难以形成终极解决方案。

以后怎么办

首先：考虑到网络安全敏感问题，能拿到Google safebrowsing哈希库算法的可能性几乎不可能；尝试通过hash算法扫描本地代码库的思路不太可行。

看来：还是得靠Firefox那边，能够自己修复这个问题。 后续我们会持续跟进这一类的问题，并就这个问题跟Firefox 方面保持反馈和测试结果的同步。

临时解决方案

js的出异常的解决方案是： 修改静态资源文件的文件名（注意如果是js文件必须修改非参数部分如： g.tbcdn.cn/aaa/bbb/ccc/ddd-min.js部分，而非t=xxxxx.js）

另外：

由于safebrowsing检测的是所有静态资源，目前已发现的出现问题的资源文件类型和解决方案是：

/	Type	Solution
1	ajax	修改请求名
2	js, css	修改目录时间戳
3	页面请求	修改参数名或添加时间戳

最后

说多了都是泪， 庆幸本次问题基本只在Firefox中被发现，并且没有大面积爆发。（应该多向chrome团队快速精准解决问题的作风学习呢~）

以后还是需要主动关注浏览器生态圈的动态，避免类似情况发生，以及对某些风险做好提前预防工作， 保证每一个使用Fire-
fox的朋友有一个良好用户体验。

特别感谢Firefox的同学积极的帮助！ 后续还会继续骚扰！

原文链接：http://ued.taobao.com/blog/2014/07/google_safebrowsing%E5%BC%95%E5%8F%91%E9%A1%B5%E9%9D%A2%E5%8A%A0%E8%BD%BD%E9%98%BB%E5%A1%9E%E9%97%AE%E9%A2%98/

七牛首席布道师：GO不是在颠覆，就是在逆袭

作者：徐立

Go 语言是谷歌 2009 年首次推出并在 2012 年正式发布的一种全新的编程语言，可以在不损失应用程序性能的情况下降低代码的复杂性。谷歌首席软件工程师罗布派克(Rob Pike)说：我们之所以开发 Go，是因为过去10多年间软件开发的难度令人沮丧。Google 对 Go 寄予厚望，其设计是让软件充分发挥多核心处理器同步多工的优点，并可解决面向对象程序设计的麻烦。它具有现代的程序语言特色，如垃圾回收，帮助开发者处理 琐碎但重要的内存管理问题。Go 的速度也非常快，几乎和 C 或 C++ 程序一样快，且能够快速开发应用程序。

7月30日的在线培训《Go语言编程》，七牛云存储联合创始人兼首席布道师徐立（@飞天急速徐倒立）将带来 Go 的前世今生与 Go 语言编程的基础教学，以及七牛云存储应用 Go 的实践分享。报名地址：http://huiyi.csdn.net/activity/product/goods_list?project_id=1202

在课程开始前，CSDN 对徐立进行了简单的采访。七牛云存储是用 Go 语言开发实现的，而七牛可以说是全球第一个最早用 Go 吃螃蟹还吃得很香的大玩家。徐立认为 Go 是划时代的，唯有 Go 能成为史诗之绝唱！Go 不是在颠覆，就是在逆袭！Go 在当下必然大有可为，大有作为，事在人为。

请您先简单和大家介绍一下自己，能谈谈您的技术成长历程吗?

Hi，大家好！我叫徐立，默默无闻至今做了十 年码农，必须感谢漫长的码农岁月在侵蚀我花样年华的同时又栽培了我，如今依然还是一个充满技术理想情怀的热血青年！同时，也是一个想用更好的技术创造出一 些更加美好体验的互联网创业者。当前，我与我的团队们正在一起努力铸造 “计算机

网络世界里边同时也是我们这个星球上最具特色并还可以更加惊艳的数据仓储及物流系统” ——七牛云存储，一款为广大应用开发者一站式解决文件加速上传和加速下载以及个性化云端数据处理的云服务，目标是让天下没有难写的代码和难搞的运维，以此 可以让广大应用开发者过上 “Lazy coding, happy life!” 的幸福美好生活。

我在中学年间开始接触编程，那时从网 上偶尔接点活儿挣点游戏点卡和零花钱，从早前比较偏向实用功能的业务层应用开发（比如软件和 Web ）做到后来比较底层的系统技术（比如分布式计算和存储），对于互联网技术日新月异的发展一直保持着敬畏与追随。在中国互联网发展演进的这十几年间，快速经 历了桌面软件，PC 互联网，移动互联网，互联网硬件，云计算服务这样一个快速变革的时代，算是一个在互联网高速成长的时代背景下土生土长出来外表粉嫩内心沧桑的程序员。如果 要采访我和电脑摩擦相爱了至少十年为什么还是这么年轻粉嫩？我会毫不迟疑地告诉你能够抵挡岁月无情摧残的唯有激情与才华，还有梦想。由于现阶段的工作需 要，不少精力都投放在与互联网前沿技术相关的学习与分享交流，比如最近分享的 Docker 和 Go 的话题。我喜欢把经过实践检验后沉淀下来的美好事物分享给大家，俗称布道师，当前也是七牛的首席布道师。如果你同样也是一个有着技术理想情怀的热血青年， 相信我在当下这样一个技术驱动变革的时代你不是一个人在骚动，如果你已经按耐不住心中的喜悦和热情，欢迎联系我（xl@qiniu.com），欢迎来到七 牛。七牛，不只是几头牛，你会发现更多！

您是从何时开始关注 Go 语言？是什么原因促使你们团队决定使用 Go 语言？以及是什么原因促成了《Go语言编程》这本书呢？

2009 年 11 月 Google 首次对外公开透露 Go 的存在后，就一直有保持关注。但真正开始使用 Go 语言大规模投产是在 2011 年上半年，当时是在和我们团队的早期成员使用 Go 语言研发分布式对象存储系统。

与 此同时，市面上几乎没有发现和我们一样这么大规模玩 Go 语言的个体或组织。客观地说，我们这样做的确很冒险，连 Google 官方当时也没有

这么干，且 Go 语言官方正式版都尚未发布，Google 发布 Go 1.0 是在过了一年之后。记得项目启动早期，我们对于要不要使用这么一门还在萌芽生长状态的编程语言来构建线上的大规模系统，内部成员曾各抒己见略有争议。后来 根据我们团队成员的经验自我剖析判断下来，确定这事可行 。原本存储系统我们做过好几遍，实际也跑过几个线上大规模运营的自研存储系统，且成员资历都是十几年的资深研发工程师（只有我个人资历最小当时还不到十 年）。且早前我们在 2006 年尝试使用 C++ 开发分布式存储系统，后来又有尝试使用 Erlang 替代，直到后来我们注意到问世不久的 Go 让我们是既兴奋惊喜又相见恨晚。在被 C++ 实现并发编程框架折腾得够呛之后，又遇到天生为并发而生的 Erlang，但实际上又被 Erlang 不能满足我们预期的高性能计算而堪折；而当遇到 Go 并实际写了一些程序测试检验过之后，发现不但用 Go 编程写代码很顺溜很开心，关键在执行效率和性能上也是非一般地惊艳。一个把 C++ 的性能优势以及 Erlang 天生的并发特性相融合的产物，且语法语义上要比 C++ 或 Erlang 简约不止几个数量级的编程语言，同时还是一帮世界上顶尖大牛的巅峰之作以及 Google 的大力投入支持，还有经过我们这群爱折腾的码农呆瓜们充分的测试和检验得以确认，何乐而不为。常言道实践出真知，并不是我们盲目冒险决策用 Go，只不过是我们早在几年之前第一个吃了螃蟹且经实践检验得出了的明确的结论而已。

当初使用 Go 语言编程的时候，市面上除了官方网站公开可查阅的文档以外，可以参考的资料读物相当甚少，团队成员也是现学现用彼此交流互补。早期其实并没有诞生写书的想 法，只是觉得公司团队壮大起来以后一定需要有份系统的文档手册方便新人上手学习，对我们自己来讲这本身就是一个强烈的刚需，没有任何理由不去执行。然而， 后来内容越写越丰富，微博上等技术圈子里也知道七牛用 Go 且很吃香，就有出版社机构慕名前来找到我们，再后来就有了出书这回事儿。但在当时，我们只有为数不多的几个人，都是研发重活揽了一堆事情，是写代码赶工期 还是出书是个很纠结的态度取向问题。在经过团队成员一致达成共识后，秉着 “技术驱动创新”，“美好的事物就该分享”，“独乐乐不如众乐乐”，“一个人走得虽快但一群人才能走得更快更稳” 等一系列等推动人类文明向前发展的各种鸡汤洗礼陶醉之后，我们自发地默同接受新增任务并开始组织团队协作完善《Go语言编程》一书，开放分享技术经验心得 的同时收获反馈和喜悦，希望有更多的人和组织能够参与进来用 Go 去谱写他们的故事和美妙篇章。后来，事实也的确证明：越来越多的个人和组织以及大大小小的互联网公司都开始使用

Go 语言去承载他们的海量业务，以及听闻个别开发者终于习得 Go 心经之后实现了他们内心积压已久的技术理想。海外甚至都开始有长篇大论开始分析探讨 “Why is Golang popular in China?”，用 Google 搜索 Golang 热度最高的至今一直是中国。放到当时去看，这块大陆是有多么荒芜；今天再回过头来看，江山又是如此多娇。尽管当时白天要忙写代码晚上要忙写书最终仓促出版 留下了些许审校上的遗憾，但不管怎样，那都是一段义无反悔的铿锵岁月。

Go 语言的哪些特点最吸引您?

并发

Go 最大的特色就是在语言层面天生支持并发，不需要像其他大多数编程语言那样需要开发者自行实现或借助第三方类库实现并发编程，Go 在语言级别支持的并发编程，其逻辑简化得通俗易懂简单好上手。

性能

不同于大多数脚本或解释性的高阶编程语言，用 Go 编写的代码直接了当编译成机器码高效执行。

简洁

25 个关键字即表达你能想到的所有招式，没有也不需要有任何多余，想干啥事就 go 一下。

跨平台

x86、AMD64 (x86_64)、ARM；Linux、Windows、Darwin (OSX)、FreeBSD、Android (计划Go 1.4) 几乎全平台支持，真正做到一份源码，随处编译，到处运行。

Go 语言都有哪些常见的应用场景?

作为一个 Go 语言的重度用户来看，当前除了不适合拿来造操作系统以外在操作系统之上应用级的事情都能干。再更具体一点，比如说适用于这样一些使用场景：

系统应用

以前要用 C/C++ 做的系统应用，现在都可以用 Go 来写，事半功倍，而且 Go 完美包容 C 源代码，两者互相调用还可以混合编译从而无缝集成。

网络应用

包含了常见的服务端编程比如 Web 和 API Service，以前用 PHP / Python / Ruby / Java 干的事情现在都可以用 Go 更加简单清晰的来写。再比如还可以拿来做一些 Proxy（代理）如网络穿透软件等，你懂的。

分布式系统

基于 Go 强大的系统编程加网络编程，打造各种跨网络的分布式系统服务，Go 社区有不少和分布式系统相关的开源产物。

各种形态的存储和数据库应用

比如 groupcache，influxdb 等。

客户端应用

包括带界面的桌面软件，以及后续可以想像的移动端应用（比如对 Android 的支持）。

云服务（PaaS）

如基于 Go 打造的七牛云存储（分布式对象存储系统），比如基于 Go 编写的 Docker（一款开源的容器虚拟化产物）。

Go 能做的事情，包含但不限于以上罗列的使用场景。

Go 语言在七牛中都开发了些什么服务？在七牛的代码量中，Go 语言使用占多少比例?

我们主要使用 Go 开发了以下服务和工具：

- 分布式存储系统 (Distributed Key/Value Storage)
- 数据处理服务 (Data Processing)
- 网络接口服务 (RESTful API Service)
- 消息队列服务（Message Queue Service）
- 日志处理系统 (Log Service)

- Web 网站 (不含前端 JavaScript)
- CLI 命令行和 GUI 图形界面工具
- 其他辅助工具

总的来讲，Go 在我们七牛的工程中代码覆盖率超过 90% 。还有 10% 不能覆盖的原因是我们给开发者自助使用的 Web 界面需要用 JavaScript 编程来实现酷炫的前端，以及我们为开发者准备了多达超过 10 种编程语言的 SDK 。

Go 有哪些成功的开源项目？都有哪些公司在使用?

Go 比较热门的开源项目，不完全罗列举例:

- docker - 基于 Linux 容器技术的虚拟化实现，能够轻易实现 PaaS 平台的搭建
- packer - vagrant 的作者开源的用来生成不同平台的镜像文件，例如 QEMU、KVM、Xen、VM、vbox、AWS 等
- drone - 基于 docker 构建的持续集成测试平台，类似 jenkins-ci
- libcontainer - docker 官方开源用 Go 实现的 Linux Containers
- tsuru - 开源的 PaaS 平台，类似 GAE、SAE
- groupcache - Memcached 作者(Brad Fitzpatrick) 写的用于 dl.google.com 线上使用的缓存系统
- nsq - bit.ly 开源的高性能消息队列系统，用以每天处理数十亿条的消息
- influxdb - 开源分布式时序、事件和指标数据库

- heka - Mozilla 开源的日志处理系统
- doozer - 分布式同步工具，类似 ZooKeeper
- etcd - 高可用的 Key/Value 存储系统，主要用于分享配置和服务发现。灵感来自于 ZooKeeper 和 Doozer
- goandroid - 使之用 Go 编写动态库，在原生的 Android 应用中运行
- mandala - 基于 goandroid 的工具链，用 Go 编写原生的 Android 应用的一个便捷框架
- beego - 国内 Go 开发者开发的 Web 开发框架
- revel - 另一个高产的 Web 开发框架，类似 Java Play Framework

更多: https://code.google.com/p/go-wiki/wiki/Projects

用 Go 的公司，不完全罗列举例:

国外:

- Google、 YouTube、Dropbox、dotCloud、10gen、Apcera、Mozilla、Heroku、Github、Bitbucket、 Bitly、CloudFlare、Cloud Foundry、Flipboard、Disqus、SendGrid、Tumblr、Zynga、Soundcloud
- 更多: https://code.google.com/p/go-wiki/wiki/GoUsers

国内:

- 七牛云存储、京东云平台、盛大云CDN、仙侠道、金山微看、Weico、西山居、美团、豆瓣、小米商城、360
- 更多: https://github.com/qiniu/go/issues/15

在本年度 2014 Androiday.Org 开发者大会上，您有个主题演讲提到了 Go 对 Android 的支持，能否与我们分享一下这块的研究心得?

这 是一个很有趣的话题，尤其是在今年6月份相继召开了 APPLE WWDC 2014 和 Google I/O 2014 两场举世闻名的互联网科技盛会后，APPLE 发布了用于 iOS 下一代编程的 Swift 语言，这给关注 Android 的开发者们留下了无限的遐想，尽管后来惊喜并没有如人们预期一样出现。作为一个纯洁的 Android 程序员，大概不会关注到 Go 将要支持 Android 的消息；作为一个无邪的 Go 程序员，大概也不会去关心 Android 开发和未来。但戏剧性的就是，我们当下就是处于这样一个各种跨界的大融合时代，没有什么不可能。

由于 Go 是跨平台编译的，早前就有在 ARM 上编译 Go 并成功运行的尝试。这个尝试是直接将用 Go 编写的源代码在 ARM 环境下编译，然后调用 adb shell 装入 Android 里边作为 Linux 下的可执行文件运行，但是没法关联支持 JNI(Java Native Interface)，只能作为一个独立的进程运行，然后通过 RPC、TCP 等协议方式通信，相当于是一个 App 在运行方式上分成了两部分，这样非常不利于 App 的状态管理，所以此方式无法用 Go 编写出无缝结合的 Android 应用。

然 而，单方面想要用 Go 语言封装 Android SDK 更是难以行通的：Android 原本用 Java 封装的 SDK，包含了海量的 API 接口。如果是手工封装会导致功能上的欠缺，自动封装会让 Go 语言变得丑陋不堪。不管用哪种方式，都很难快速实现。若是用 Go 再实现一遍 Android SDK 且还想期望能与 Java 等效，这几乎是不可能的事情。

但实际上，Android 系统提供了两种开发包：SDK 和 NDK。用 Go 移植 Android SDK 不通，还可以尝试走 Android NDK 。

Android NDK（Native Development Kit ）是一套开发工具集合，允许开发者用像 C/C++ 语言那样实现应用程序的一部分。然后可以通过 JNI 实现 Java 代码与 NDK 动态库的无缝集成。

Go 内置 Cgo，使得 Go 和 C 之间可以无缝地互相转换和调用，以及代码混合编译。仅此一点，就可以看出至少也为 Go 进行 Android NDK 开发奠定了基础。当然，Go 可以进行的扩展尝试可可以更多。

后 续的尝试是另辟蹊径走 Android NDK，借助其可以将动态库 .so 和 Java 代码一起打包成 apk 的机制，实现在 Android 上的无缝加载和运行。所以，问题简化成只要 Go 能够编译出 .so 动态库再通过 Android NDK 就可以实现开发 Android 应用程序了。

Go 是一门纯粹静态类型的编程语言，编译出来的二进制是静态的，如何构建动态库，这是一道坎。不过 Go 社区的欣欣向荣完全超出你我的想像，比如 Go 社区里边有人发布了个开源项目叫 Goandroid，一个扩展了 Go 的工具链和运行库来支持将 Go 代码创建动态库（*.so）的工具。另外一个开源项目 Mandala 则是在基于 Goandroid 的工具链构建了一套完善的用 Go 开发原生 Android 应用的开发框架。

Goandroid 和 Mandala 的出现已经让 Go 开发原生 Android 应用成为现实。 尽管这两个框架都是 Go 社区由个人发起的开源项目，Go 官方尚未参与，不过 Go 已经打开了 Android 的潘多拉魔盒。

另外，可以再来扒一扒和技术无关的业界新闻。

2014 年 5 月，一场已经持续 4 年的官司：关于“Google 在 Android 平台使用 Java 侵犯知识产权”一案，联邦法院判定 Oracle 获胜。

2014 年 6 月，APPLE 公司在 WWDC 2014 上发布 Swift 编程语言，用来替代 Objective-C 更高效地编写 iOS 和 OS X 程序。

2014 年 6 月，Google 公司 Go 语言开发团队成员 David Crawshaw 提议下一个 Go 版本 (Go 1.4) 支持 Android 平台。这算是一个关于 Go 支持 Android 而言相当掷地有声的宣称。文献详见：http://golang.org/s/go14android

就目前而言，Go 支持 Android 后可以干啥。简单来讲，Go 以支持 Android NDK 编程作为切入点，自然是可以在 NDK 这层注入 Go 所能带来的新鲜活力，比如在异步的并发编程上用 channels 而不再是 callbacks 通讯，比如为游戏引擎提供底层的高性能支撑，比如跟多媒体相关的更丰富地

处理。所以，可以想像的到 Go 是可以引领 Android NDK 迎来一片生机盎然的春天。

Go 支持 Android 这事当前看起来是 Go 的一厢情愿，而 Android 对此结合似乎还无动于衷。然而， Android 背负的包袱毕竟过重。Go 跃出的一小步，开启的必然会是 Android 海阔天空的一大步。

谈谈您个人对 Go 语言的理解和看法?

尽 管 Go 是一门问世不久显得很新的编程语言，有很多质疑的声音认为新生的事物就是不成熟就此望而却步。但在我个人看来，考量成熟的因素并不完全取决于它所经历和走 过的岁月，而是在于它是否可以被当下这个时代所需要并赋予厚望，以及其心智和能力是否能够承载并担当得起这份荣耀与责任。

在我们今天这个时 代，是一个包含了 PC 互联网、移动互联网、物联网和云计算 “四世同堂” 的技术多元盛世。在不曾久远的过去，此前的数十年间，没有哪一门编程语言能够经得起此般岁月的几经摧残而长生不老。70 年代就开始基业长青的 C，书写了整个单机时代的辉煌，在硬件更新换代多核一度再翻成指数级更迭又如此瞬息万变的今天，C 那副认真憔悴的注目神情所表达的静静默守和激流勇退才终归得以明白它曾有过的卓绝。而 Java / Python / PHP / Ruby 此前彼后所纵横的 PC Web 时代，以当下之势在此不可逆转的时代背景下正似岁月如梭般地离我们渐行老去。而当移动互联网悄然踏至的那一时刻，Android 势如破竹般地野蛮生长搭救了 Java 一命并在其身上倾覆了所有，换来的却不过是一份难以割舍又无法言痛的沉重爱情，在被包养换主后的 Java 擦伤了贞洁之后不得不忍辱负重潸然泪行，而 Java 也从此难以走下一座叫作 “节操” 的断背深山。而和 Java 长得很像极帅的那位 JavaScript，意气风发一竿子捅到底逆袭了整个 Web 后端，讨得所有 Web 前端开发者的拥簇和狂欢，眼看高举全栈工程大旗就要翻手为云覆手为雨，却优秀得不懂克己错失风向与当下这个多核时代格格不入，曾一度与 HTML5 结队合唱 “轻应用就是未来” 这一出双簧，不料 Apple Swift 以迅雷不及掩耳之势以逆袭之道还治其逆袭之身；如今既生 Go，又何生 NodeJS，感慨 Node 君真是生不逢时。总之，问君能有几多愁，掐指一算，不知诸君该何去何从……

虽群雄逐鹿，现百家争鸣，然乱世必有新生。Go 有如清水出芙蓉，一个曾在温柔的岁月里深沉熟睡，尔后随着换季时节的轻声絮语呼唤苏醒，然后起身刹那之间不经意惊艳了时光，此后就在此应时之季娇艳绽放，随后波光花影，激荡涟漪，满是春意盎然，勃勃生机。

时 光往回倒流近二十年，那是 1995 年，有几位计算机泰斗，在白纸上画了一个圈，开启了分布式系统编程时代的春天。那时还在贝尔实验室参与九号计划（Plan9）的 Robert C. Pike（Plan9 操作系统和 UTF-8 的共同设计者，分布式编程语言 Limbo 作者） 和 Kenneth Lane Thompson （1983 年图灵奖得主，创建了 UNIX 和 Plan9 操作系统，B 语言和 C 语言的共同设计者，UTF-8 的共同设计者），在此段工作经历中开发了分布式网络操作系统 Inferno，并在此之上实现了分布式编程语言 Limbo，这是能够追溯到最早和后来的 Go 在功能特性上比较相似的前身。随后这两位泰斗相继都加入了 Google 公司，在 2007 年 9 月，这两位 Unix 和 C 还有 UTF-8 的始祖成员，再加上 Javascript V8 引擎和 Java HotSpot 编译器的作者 Robert Griesemer 一起，这 3 人小组设计了最初的 Go 语言雏形。此后，Go 语言逐渐吸引了一些业界优秀人物比如 80 后程序员 Brad Fitzpatrick（Memcached 作者，OpenID 协定者）的加入，Go 的团队阵营逐渐壮大。从 2007 年 9 月作为一个 Google 20% 自由时间的一个实验项目；到 2008 年 5 月发展为 Google 100% 支持的全时项目；再到 2009 年 11 月；Google 首次对外公开透露 Go 的存在；以及 2012 年 3 月，Go 1.0 官方正式版问世；Go 的演变和发展简直始料未及，从开始到现在都是一如既往地突飞猛进，一发不可收拾。

如果说到 Java 曾经的流行，我们会联想到 SSH（Struts + Spring + Hibernate）；如果提到 Python，也会联想到 Django；如果提到 Ruby，会联想到 Ruby on Rails；如果提到 JavaScript，会联想到 NodeJS；如果提到 PHP，更是一堆长江后浪推前浪前浪死在沙滩上的 Web 开发框架。这些编程语言社区的繁荣昌盛无一例外都和 Web 开发息息相关，且最终沉淀下来的都是各种五花八门各有千秋的众多 Web 开发框架。可以说，我们当前所面临和 Web 开发的技术选型，从未有过如此的繁荣。繁荣的背后，衬托的是一个大江东流去不复还的 PC Web 时代。

我 们再来看看 Go 的社区，是多么地非同寻常和多样丰盛。我们之前列举过 Go 社区里边比较流行热门的开源项目：比如 Docker ，是时下最流行

和容器虚拟化相关的技术产物；比如 GroupCache ，是一个类似或代替 Memcached 的分布式内存缓存系统；比如 nsq，是一款处理海量并发的消息队列系统；再比如 etcd，是一套用于配置同步管理的分布式键值存储系统；以上这些都是和网络和系统服务以及分布式相关。再比如 Goandroid ，是一个开发 Android NDK 应用的工具，是和移动开发相关的…… 总之，有太多创新的开源产物，而 Web 开发框架都不是主要重点。Go 几乎涵盖了和编程领域相关的所有点和面，在 Go 1.0 正式版出现后的两年时间里，基于 Go 可以枚举全面覆盖的开源项目超过了 1000 个。可谓是随风潜入夜，润物细无声。

从 Go 诞生的时代背景和团队阵容以及开源社区的繁荣来讲，Go 并不是新生不熟，而是后生可畏，且极有可能成为一统天下的集大成者。

我 们再来看一些实际使用场景，比如 Web 开发。大多数编程语言之上的 Web 开发框架都是遵照 MVC 的处理流程去开发 Web 应用：Model 部分封装数据，Controller 部分处理业务逻辑，View 部分植入变量填充模板页面。而大部分 Web 框架关于 MVC 的三部分都是在 Server-side 处理，比如对 View 部分的处理都是在 Server-side 通过程序动态对模版变量求值后再拼接组装成 HTML 页面输出给浏览器去呈现。而 Go 开发 Web 应用，并不依赖任何 Web 开发框架，用内置的标准库就可以轻而易举地实现：比如使用 net/http 标准库就可以数行代码构建一个完整的 Web 骨架应用；再比如，通过关键字 struct 封 装一个数据结构就可以表达原本 MVC 框架中需要用厚重的 ORM (Object-Relational Mapping) 才能表达的那部分。大道至简，这可以说是 Go 的哲学。在 View 这一层，Go 也有相应标准库提供支持，但更推荐的做法，是当下比较流行的 MVVM (Model-View-ViewModel)：Server-side 只输出 JSON，浏览器 DOM 作为 View 层，前端 JavaScript 充当 Contoller 部分；这样，不仅减少了 Server-side 的资源消耗还有中间传输的网络流量，而且前端可以更灵活和更丰富，后端也可以更轻盈和更高效，也更有利于项目的分工和协作。 而 Go 对 JSON 的生成和输出，有数据测试表明异常高效（在 i7-2600K 处理器上针对所有编程语言包含开发框架总计约100个测试对比中，Go 的性能指标稳居第一，详见：http://t.cn/RvZHyKI）。以我个人喜好之见，后端用 Go 前端用 AngularJS 可以说是现今流行 MVVM 方式的 Web 开发之绝唱组合。所以，如有疑问 Go 适不

适合用来做 Web 开发，我想答案很肯定的：不但可以而且更简单同时做得更出色甚至还可以做的更多。

我 们再来看看当下硬件突飞猛进的多核时代，软件层面的编程语言都有哪些支持和作为。“人生苦短，我用 Python”，被喊了这么多年的标语，是有多少有追求和品味的程序员都热衷于 Python。我想大家可以做个实际测验：比如用 Python 写一个死循环程序，然后运行起来，过一小段时间看看 CPU 占用率是多少。测试结果会是什么呢，CPU 会占用 100% 吗？不好意思如果是那应该是单核，且还得是没有超线程支持的骨灰级 CPU 才行！在当下普遍的双核（甚至 4 核或 8 核）CPU 上，这个死循环跑满最多只会占用不超过 50% 的 CPU 资源。好吧，你会说我开多个线程并行来跑这个死循环不就可以吃满 CPU 了吗？我想测试结论可以很明确的告诉你，多线程的方式并行跑这个死循环可是连 50% 的 CPU 都吃不到，不信你可以试试，呵呵。无需我解释为什么，大家可以自行 Google 搜索看看 GIL (Global Interpreter Lock) 机理。也不是我特别拿 Python 举例，PHP / Ruby / NodeJS 的程序员们也可以来做做同样的测试，呵呵。总之，一个简单得不能再简单的死循环，程序上再怎么优化，跑起来都吃不满 CPU，你说我们都是这么有品味的程序员，到底都是在追求些什么呢？这些 PC Web 时代下的脚本语言或者字节码解释型的语言，几乎都由于线程安全问题而在语言级的并发机制上裹足不前。尽管 Ruby 实现了很华丽的纤程，NodeJS 实现了很光线的非阻塞 IO，但始终逃离不了单线程简单可依赖原则抑或多线程效率提升但伴随着各种问题困扰从而不得不折中取舍的桎梏。所以，大部分时候，语言层面没有根本性地解 决并发问题，转而使用传统的多进程这一外援策略去解决并发需求以及变相地迎合这个多核时代。然而，硬件很快，软件很慢，摩尔定律在硬件行业的应验带给传统 软件行业的红利已经走到了尽头。

我们再来看看 Go 在当下这个多核时代的作为。不得不说，Go 最大的特色就是在语言层面天然支持并发，在 Go 程序里边，你可以通过在一个函数调用前使用关键字 go 即可让该函数 func 运行成为一个独立的 goroutine，goroutine 可以理解成一种比线程更加轻盈更省开销的轻量级协程。Go 的并发模型就是通过系统的线程来多路派遣这些独立函数的执行，使得每个用关键字 go 执行调用的函数可以运行成为一个单位协程。当一个协程阻塞的时候，调度器就会自动把其他协程安排到另外的线程中去执行，从而实现

程序的无等待并行化运行。 且调度的开销非常小，单核 CPU 调度的规模不下于每秒百万次，这使得我们能够创建大量的 goroutines，从而可以很轻松地编写并发程序达到我们想要的目的。

同时，Go 在语言层面还引入了 channel 这一内置类型来实现并发执行体 goroutines 之间的消息传递，通信靠 channels 来传递消息。Go 遵循 CSP(Communicating sequential processes) 并发模型，通过通信来共享内存而不是用共享内存的方式进行通信。Go 的并发里边没有共享内存，更没有内存锁，这一切都有利于进行更为安全和简单的并行程序编写。

终有一日，你会感慨：“人生苦短，说 Go 就 Go”！

时 代在快速跃迁，尤其是现在言必及多核和并发，我们所看到的那些 PC Web 时代下的脚本语言所堆砌的华丽不过是一些延续性追随，而且这种延续性的步伐会越来越沉重，最终还是苦不堪言。唯有 Go，在语言层面对并发和多核乃是纯天然的支持，原本就是一场应时而生顺时而为的土生土长，Go 所代表的是一种破坏性创新。而我们所经历的这个时代，正是处于一个时代被时代所颠覆的时代，在这样一个转折点，所有的延续性创新不过是杯水车薪，唯有破坏 性创新，与时代共舞，才会产生革命性的颠覆。

我们还可以回顾下之前提到 Go 对 Android 移动端非侵入式的支持，几乎就是一场悄无声息的逆袭。

Go 很年轻，却已健壮成年。Go 被设计得简洁高效，Go 在语法层面有着清晰简洁却又高效的表达能力，是一个让开发者编写程序很开心同时又更有生产力的系统编程语言。Go 在语言层面有着良好的并发支持，使得用 Go 语言编写多核和分布式网络的应用程序简单容易许多。Go 内置新颖灵活的类型系统还可以很方便地编写和构建模块化程序。Go 是一门需要编译源码才能运行应用的纯静态强类型语言，这点保证的了程序的安全性和执行效率，且 Go 从程序源代码编译成机器码非常快。Go 的跨平台支持，使得用 Go 语言编写的程序可以在现今大多数操作系统上编译运行。Go 还自带垃圾回收的内存管理机制，并且支持强大的运行时反射。Go 是动静相宜的，在性能和安全性方面保留了静态语言的优点，在编程写法上，却有着动态语言的灵活与优雅。

Go 是划时代的，唯有 Go 能成为史诗之绝唱！Go 不是在颠覆，就是在逆袭！

原文链接：http://www.csdn.net/article/2014-07-21/2820743

聊聊JVM的年轻代

作者：郭蕾

1.为什么会有年轻代

我们先来屡屡，为什么需要把堆分代？不分代不能完成他所做的事情么？其实不分代完全可以，分代的 唯一理由就是优化GC性能。你先想想，如果没有分代，那我们所有的对象都在一块，GC的时候我们要找到哪些对象没用，这样就会对堆的所有区域进行扫描。而 我们的很多对象都是朝生夕死的，如果分代的话，我们把新创建的对象放到某一地方，当GC的时候先把这块存“朝生夕死”对象的区域进行回收，这样就会腾出很 大的空间出来。

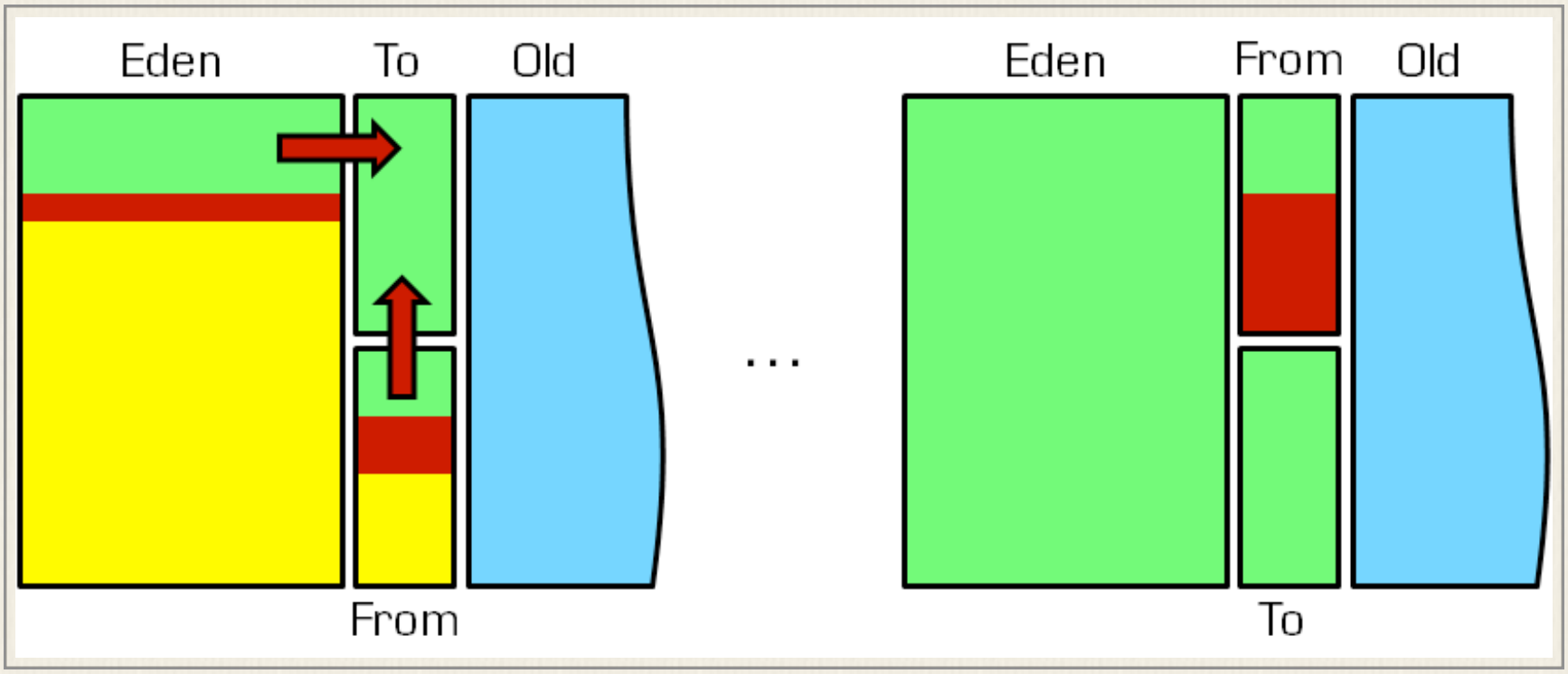
2.年轻代中的GC

HotSpot JVM把年轻代分为了三部分：1个Eden区和2个Survivor区（分别叫from和to）。默认比例为8：1,为啥默认会是这个比例，接下来我们会 聊到。一般情况下，新创建的对象都会被分配到Eden区(一些大对象特殊处理),这些对象经过第一次Minor GC后，如果仍然存活，将会被移到Survivor区。对象在Survivor区中每熬过一次Minor GC，年龄就会增加1岁，当它的年龄增加到一定程度时，就会被移动到年老代中。

因为年轻代中的对象基本都是朝生夕死的(80%以上)，所以在年轻代的垃圾回收算法使用的是复制算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

在GC开始的时候，对象只会存在于Eden区和名为“From”的Survivor 区，Survivor区“To”是空的。紧接着进行GC，Eden区中所有存活的对象都会被复制到“To”，而在“From”区中，仍存活的对象会根据他 们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过-XX:MaxTenuringThreshold来设

置)的对象会被移动到年老代中，没有达 到阈值的对象会被复制到“To”区域。经过这次GC后，Eden区和From区已经被清空。这个时候，“From”和“To”会交换他们的角色，也就是新 的“To”就是上次GC前的“From”，新的“From”就是上次GC前的“To”。不管怎样，都会保证名为To的Survivor区域是空的。 Minor GC会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。



3.一个对象的这一辈子

我是一个普通的java对象，我出生在Eden区，在Eden区我还看到和我长的很像的小兄弟， 我们在Eden区中玩了挺长时间。有一天Eden区中的人实在是太多了，我就被迫去了Survivor区的“From”区，自从去了Survivor区， 我就开始漂了，有时候在Survivor的“From”区，有时候在Survivor的“To”区，居无定所。直到我18岁的时候，爸爸说我成人了，该去 社会上闯闯了。于是我就去了年老代那边，年老代里，人很多，并且年龄都挺大的，我在这里也认识了很多人。在年老代里，我生活了20年(每次GC加一岁)， 然后被回收。

4.有关年轻代的JVM参数

1)-XX:NewSize和-XX:MaxNewSize

用于设置年轻代的大小，建议设为整个堆大小的1/3或者1/4,两个值设为一样大。

2)-XX:SurvivorRatio

用于设置Eden和其中一个Survivor的比值，这个值也比较重要。

3)-XX:+PrintTenuringDistribution

这个参数用于显示每次Minor GC时Survivor区中各个年龄段的对象的大小。

4).-XX:InitialTenuringThreshol和-XX:MaxTenuringThreshold

用于设置晋升到老年代的对象年龄的最小值和最大值，每个对象在坚持过一次Minor GC之后，年龄就加1。

原文链接：http://ifeve.com/jvm-yong-generation/

整理对Spark SQL的理解

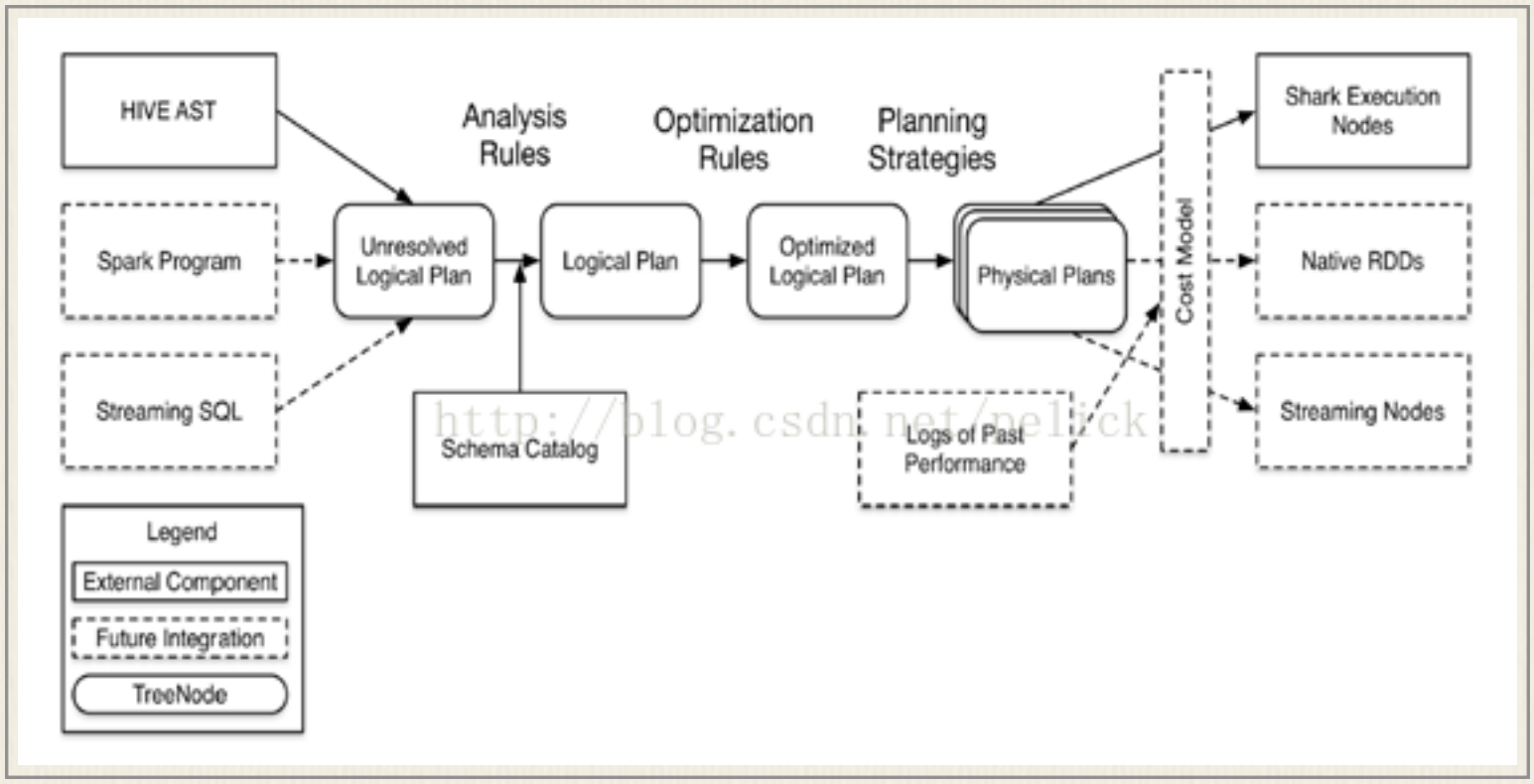
作者：张包峰

Catalyst

Catalyst是与Spark解耦的一个独立库，是一个impl-free的执行计划的生成和优化框架。

目前与Spark Core还是耦合的，对此user 邮件组里有人对此提出疑问，见mail。

以下是Catalyst较早时候的架构图，展示的是代码结构和处理流程。



Catalyst定位

其他系统如果想基于Spark做一些类sql、标准sql甚至其他查询语言的查询，需要基于Catalyst提供的解析器、执行计划树结构、逻辑执行计划的处理规则体系等类体系来实现执行计划的解析、生成、优化、映射工作。

对应上图中，主要是左侧的TreeNodelib及中间三次转化过程中涉及到的类结构都是Catalyst提供的。至于右侧物理执行计划映射生成过程，物理执行计划基于成本的优化模型，具体物理算子的执行都由系统自己实现。

Catalyst现状

在解析器方面提供的是一个简单的scala写的sql parser，支持语义有限，而且应该是标准sql的。

在规则方面，提供的优化规则是比较基础的(和Pig/Hive比没有那么丰富)，不过一些优化规则其实是要涉及到具体物理算子的，所以部分规则需要在系统方那自己制定和实现(如spark-sql里的SparkStrategy)。

Catalyst也有自己的一套数据类型。

下面介绍Catalyst里几套重要的类结构。

TreeNode体系

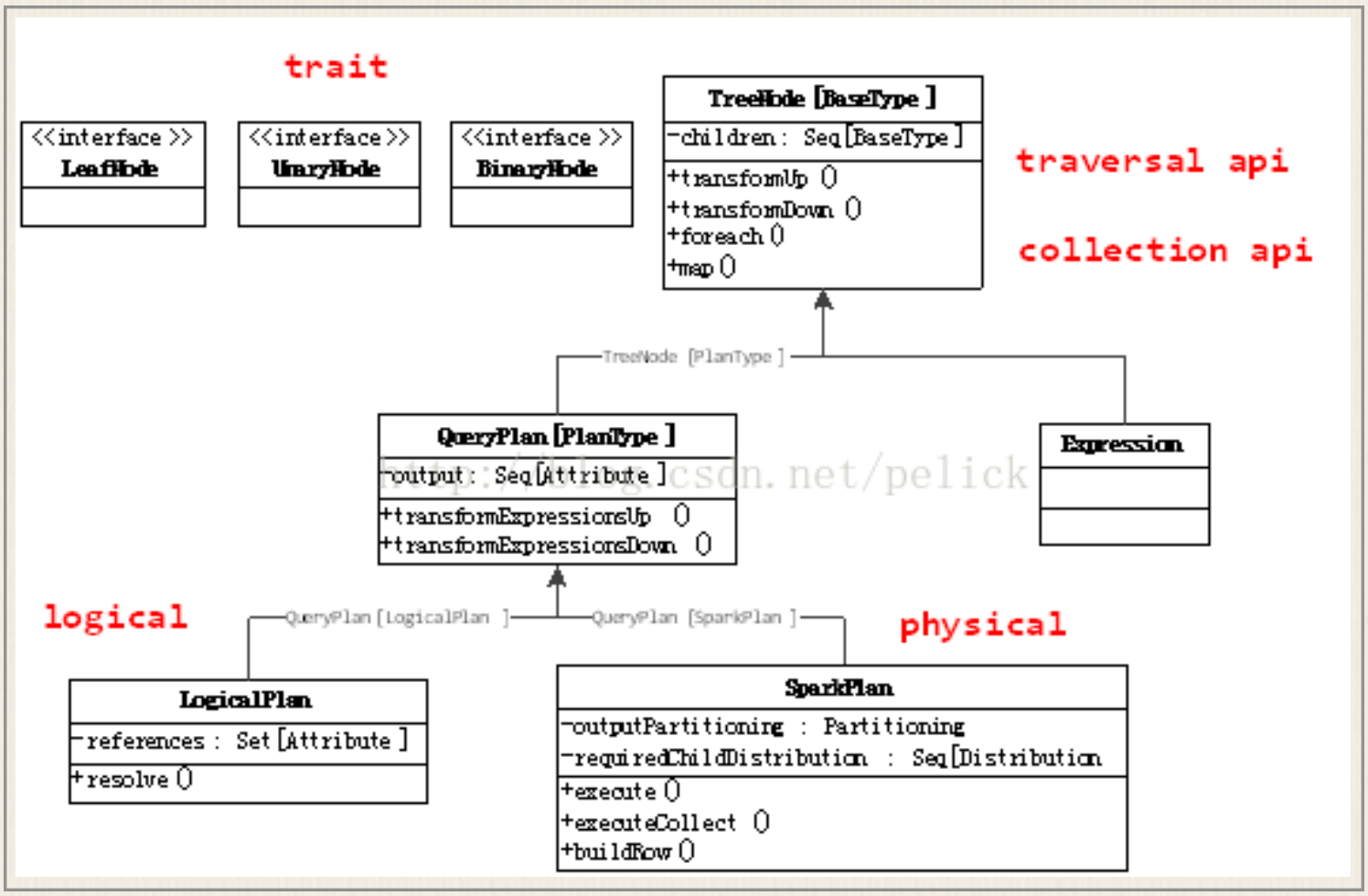
TreeNode是Catalyst 执行计划表示的数据结构，是一个树结构，具备一些 scala collection的操作能力和树遍历能力。这棵树一直在内存里维护，不会dump到磁盘以某种格式的文件存在，且无论在映射逻辑执行计划阶段还是优化 逻辑执行计划阶段，树的修改是以替换已有节点的方式进行的。

TreeNode，内部带一个children: Seq[BaseType]表示孩子节点，具备foreach、map、collect等针对节点操作的方法，以及transformDown(默认，前 序遍历)、transformUp这样的遍历树上节点，对匹配节点实施变化的方法。

提供UnaryNode,BinaryNode, LeafNode三种trait，即非叶子节点允许有一个或两个子节点。

TreeNode提供的是范型。

TreeNode有两个子类继承体系，QueryPlan和Expression。QueryPlan下面是逻辑和物理执行计划两个体系，前者在Catalyst里有详细实现，后者需要在系统自己实现。Expression是表达式体系，后面章节都会展开介绍。



Tree的transformation实现：

传入PartialFunction[TreeType,TreeType]，如果与操作符匹配，则节点会被结果替换掉，否则节点不会变动。整个过程是对children递归执行的。

执行计划表示模型

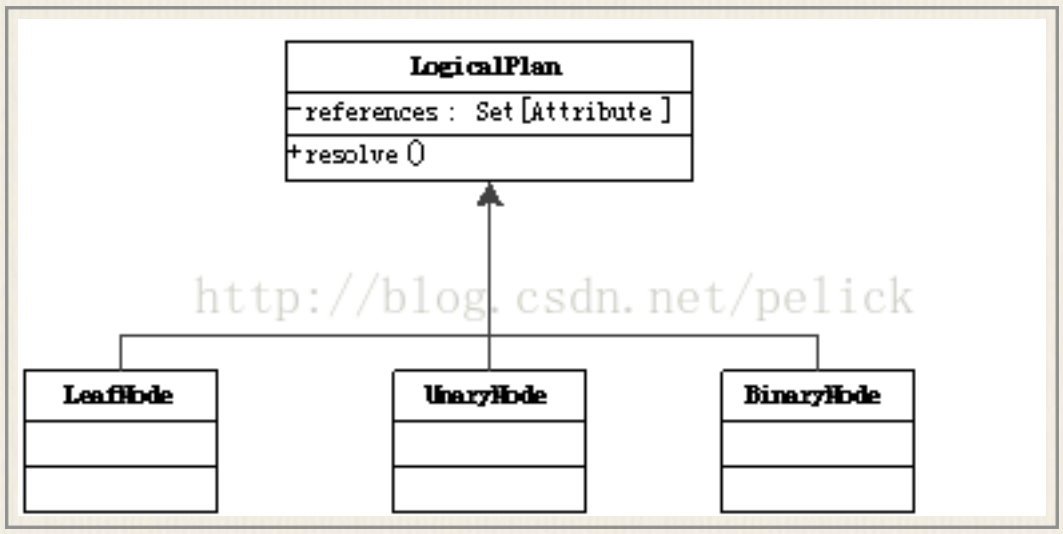
逻辑执行计划

QueryPlan继承自TreeNode，内部带一个output: Seq[Attribute],具备transformExpressionDown、transformExpressionUp方法。

在Catalyst中，QueryPlan的主要子类体系是LogicalPlan，即逻辑执行计划表示。其物理执行计划表示由使用方实现(spark-sql项目中)。

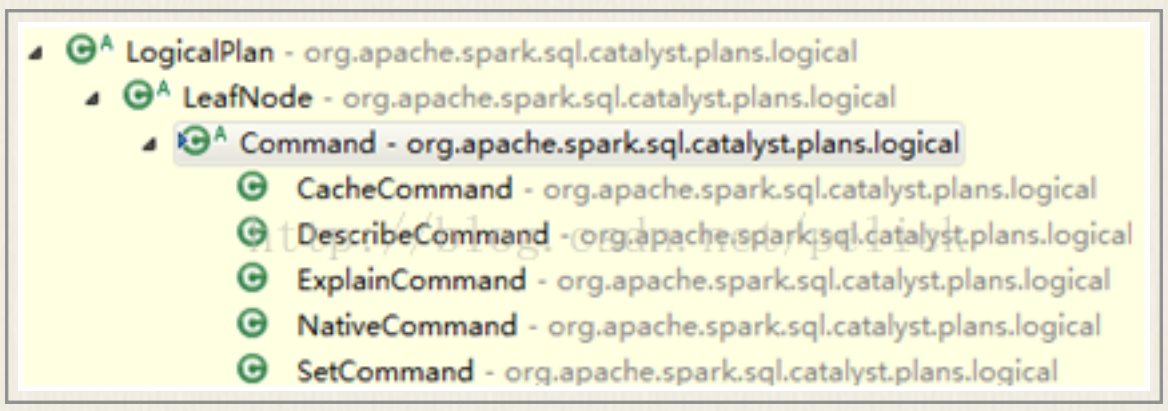
LogicalPlan 继承自QueryPlan，内部带一个reference:Set[Attribute]，主要方法为resolve(name:String): Option[NamedeExpression]，用于分析生成对应的NamedExpression。

LogicalPlan有许多具体子类，也分为UnaryNode, BinaryNode, LeafNode三类，具体在org.apache.spark.sql.catalyst.plans.logical路径下。



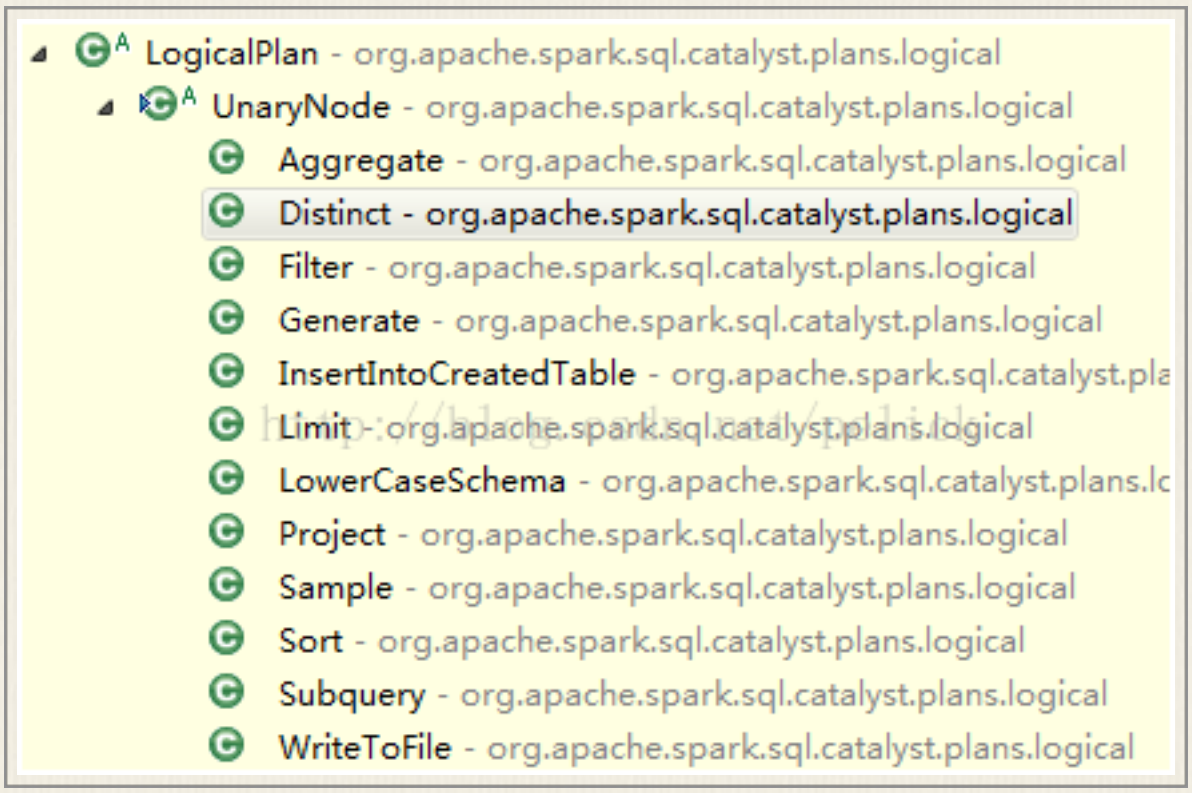
逻辑执行计划实现

LeafNode主要子类是Command体系：



各command的语义可以从子类名字看出，代表的是系统可以执行的non-query 命令，如DDL。

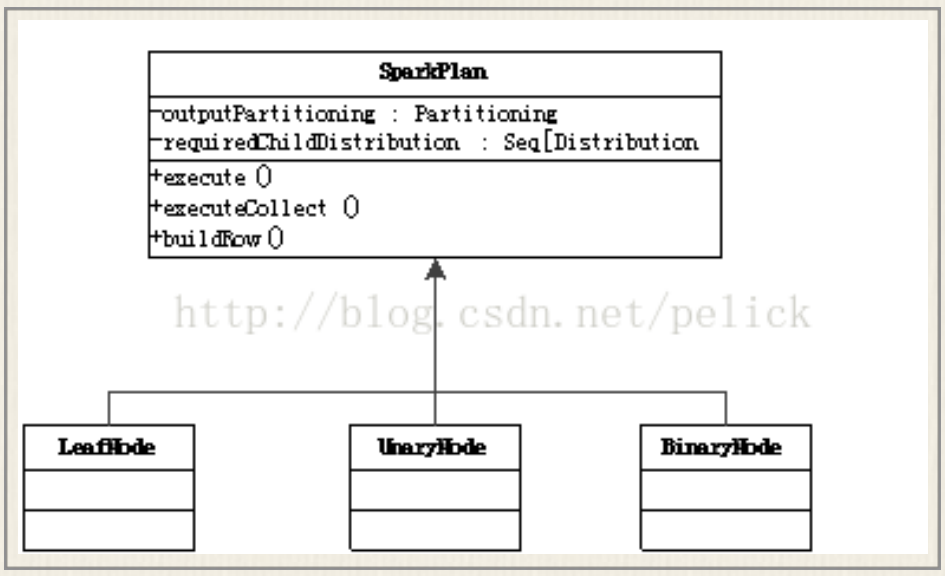
UnaryNode的子类：



BinaryNode的子类：

物理执行计划

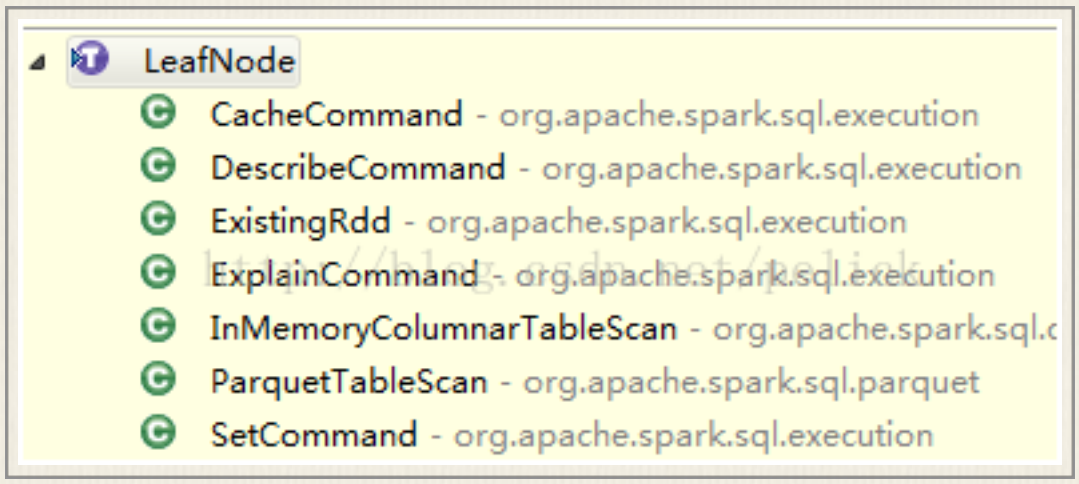
另一方面，物理执行计划节点在具体系统里实现，比如spark-sql工程里的SparkPlan继承体系。



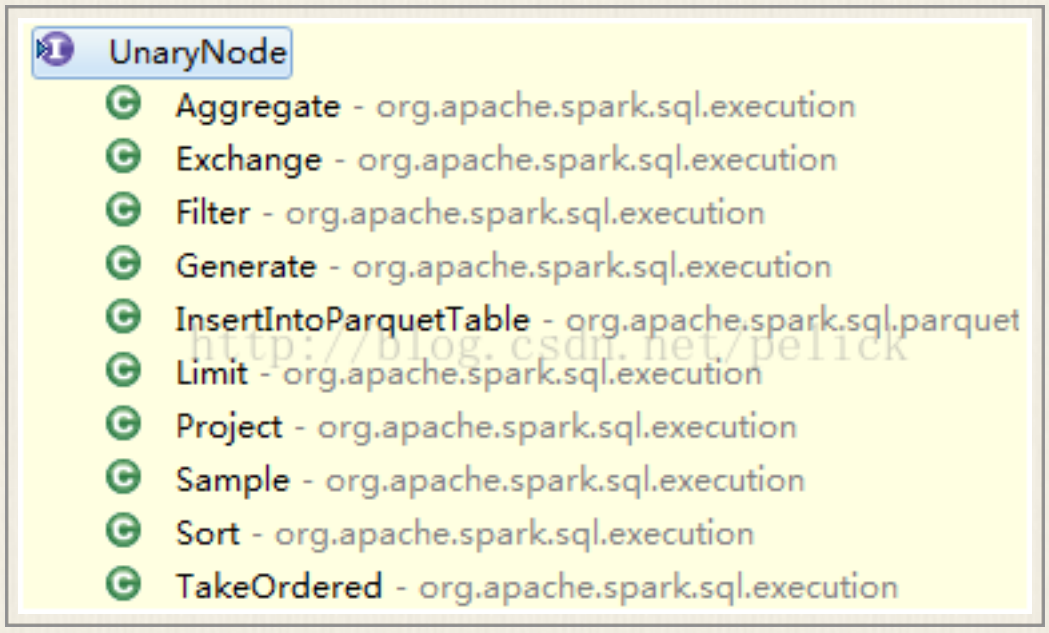
物理执行计划实现

每个子类都要实现execute()方法，大致有以下实现子类(不全)。

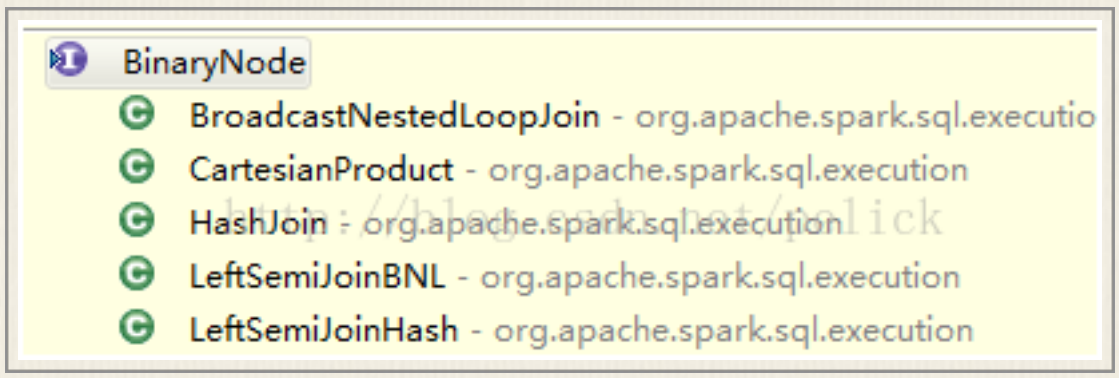
LeadNode的子类：



UnaryNode的子类：



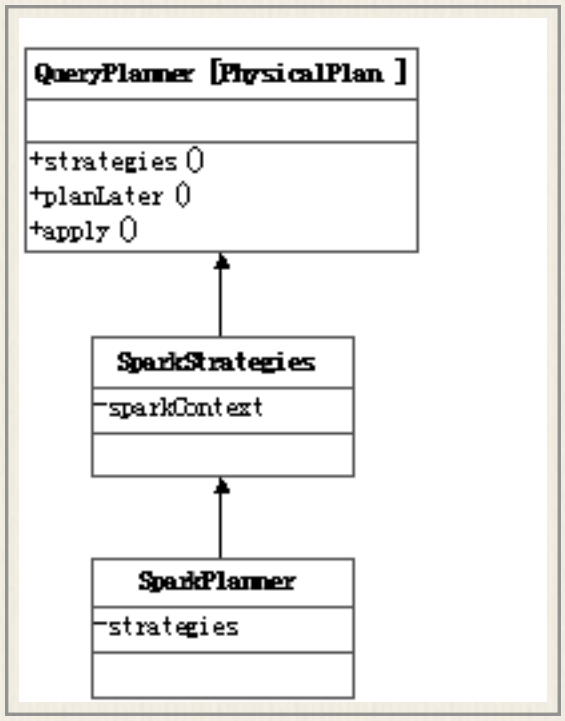
BinaryNode的子类：



提到物理执行计划，还要提一下Catalyst提供的分区表示模型。

执行计划映射

Catalyst还提供了一个QueryPlanner[Physical <: TreeNode[PhysicalPlan]]抽象类，需要子类制定一批strategies: Seq[Strategy]，其apply方法也是类似根据制定的具体策略来把逻辑执行计划算子映射成物理执行计划算子。由于物理执行计划的节点是在具体 系统里实现的，所以QueryPlanner及里面的strategies也需要在具体系统里实现。



在spark-sql 项目中，SparkStrategies继承了QueryPlanner[SparkPlan]，内部制定了 LeftSemiJoin, HashJoin,PartialAggregation, BroadcastNestedLoopJoin, CartesianProduct等几种策略，每种策略接受的都是一个LogicalPlan，生成的是Seq[SparkPlan]，每个 SparkPlan理解为具体RDD的算子操作。

比如在BasicOperators这个Strategy里，以match-case匹配的方式处理了很多基本算子(可以一对一直接映射成RDD算子)，如下:

```
1.  case logical.Project(projectList, child) =>
2.    execution.Project(projectList, planLater(child)) :: Nil
3.  case logical.Filter(condition, child) =>
4.    execution.Filter(condition, planLater(child)) :: Nil
5.  case logical.Aggregate(group, agg, child) =>
```

6. *execution.Aggregate(partial = false, group, agg, planLater(child))(sqlContext) :: Nil*

7. *case logical.Sample(fraction, withReplacement, seed, child) =>*

8. *execution.Sample(fraction, withReplacement, seed, planLater(child)) :: Nil*

Expression体系

Expression，即表达式，指不需要执行引擎计算，而可以直接计算或处理的节点，包括Cast操作，Projection操作，四则运算，逻辑操作符运算等。

具体可以参考org.apache.spark.sql.expressionspackage下的类。

Rules体系

凡是需要处理执行计划树(Analyze过程，Optimize过程，SparkStrategy过程)，实施规则匹配和节点处理的，都需要继承RuleExecutor[TreeType]抽象类。

RuleExecutor内部提供了一个Seq[Batch]，里面定义的是该RuleExecutor的处理步骤。每个Batch代表着一套规则，配备一个策略，该策略说明了迭代次数(一次还是多次)。

[java] view plaincopy

protected case class Batch(name: String, strategy: Strategy, rules: Rule[TreeType])*

Rule[TreeType <: TreeNode[_]]是一个抽象类，子类需要复写apply(plan: TreeType)方法来制定处理逻辑。

RuleExecutor的apply(plan: TreeType): TreeType方法会按照batches顺序和batch内的Rules顺序，对传入的plan里的节点迭代处理，处理逻辑为由具体Rule子类实现。

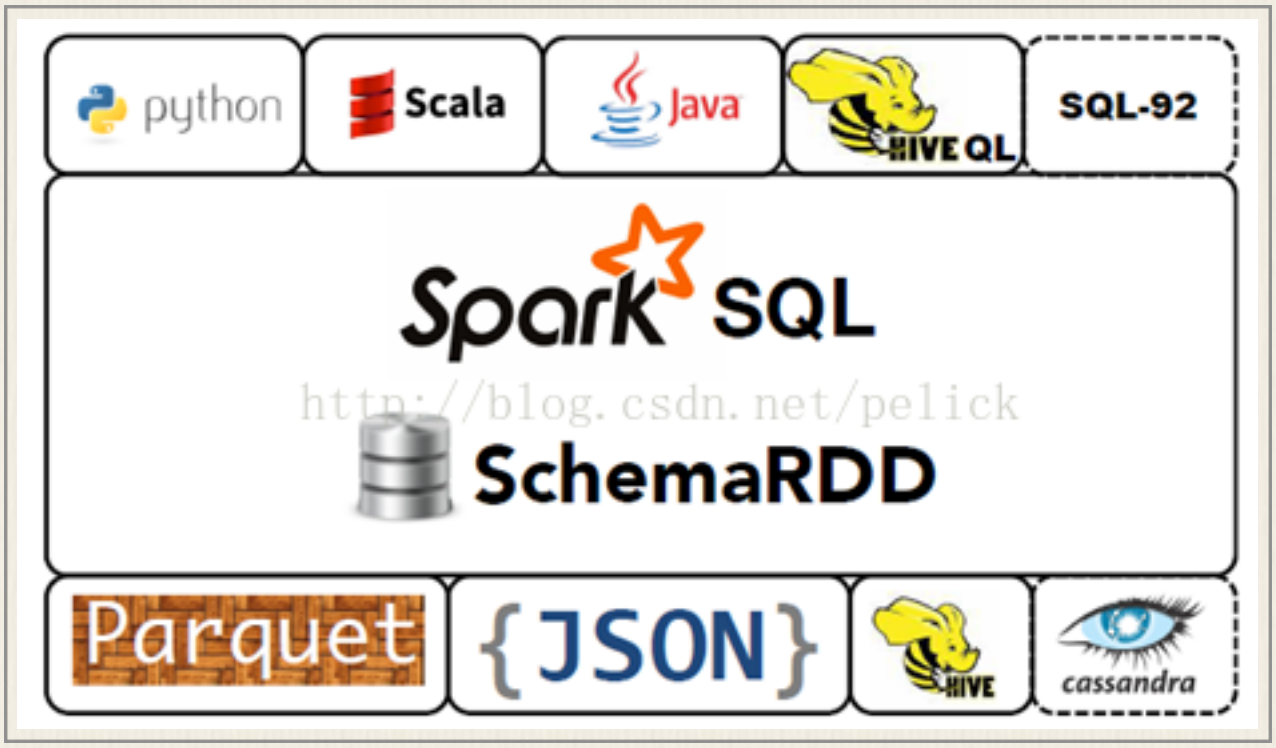
Hive相关

Hive支持方式

Spark SQL对hive的支持是单独的spark-hive项目，对Hive的支持包括HQL查询、hive metaStore信息、hive SerDes、hive UDFs/UDAFs/ UDTFs，类似Shark。

只有在HiveContext下通过hive api获得的数据集，才可以使用hql进行查询，其hql的解析依赖的是org.apache.hadoop.hive.ql.parse.ParseDriver类的parse方法，生成Hive AST。

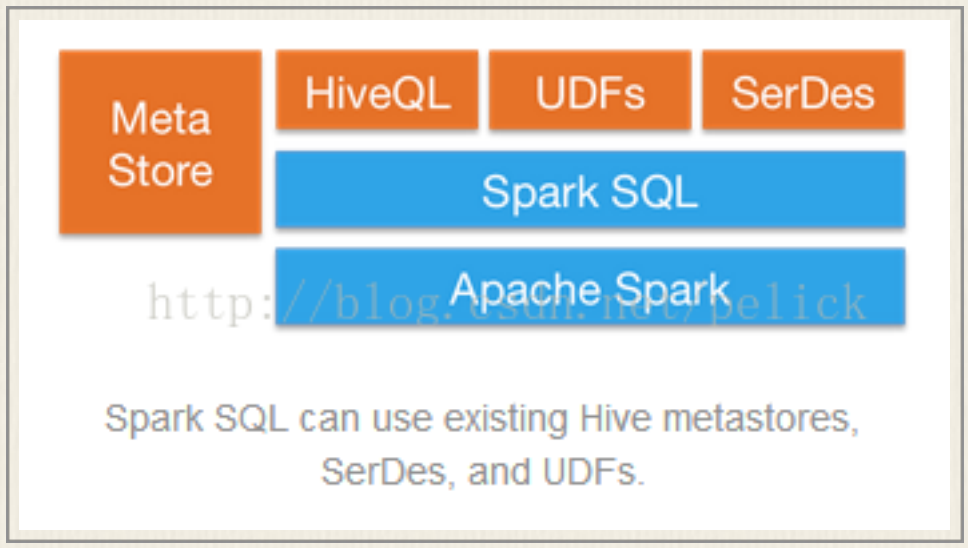
实际上sql和hql，并不是一起支持的。可以理解为hql是独立支持的，能被hql查询的数据集必须读取自hive api。下图中的parquet、json等其他文件支持只发生在sql环境下(SQLContext)。



Hive on Spark

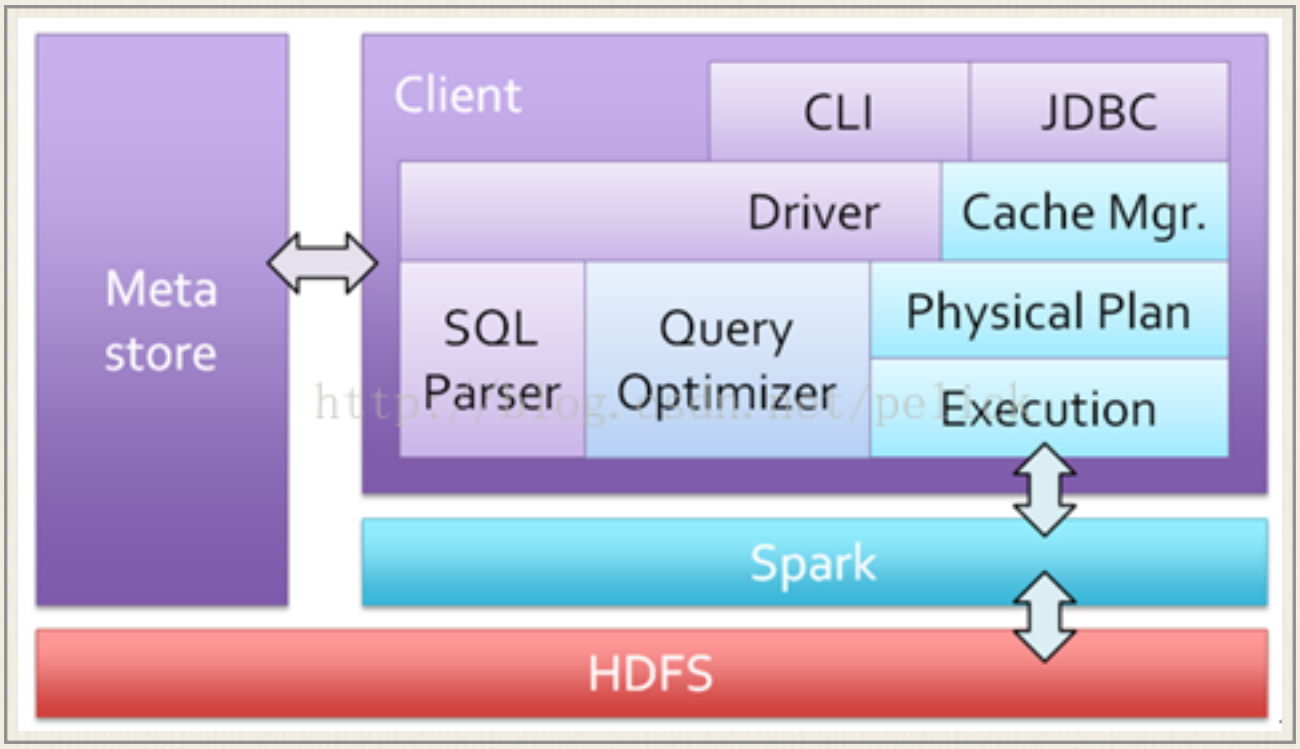
Hive官方提出了Hive onSpark的JIRA。Shark结束之后，拆分为两个方向：

Spark SQL里现在对Hive的支持，体现在复用了Hive的meta store数据、hql解析、UDFs、SerDes，在执行DDL和某些简单命令的时候，调的是hive客户端。hql翻译前会处理一些与query主 体无关的set, cache, addfile等命令，然后调用ParserDriver翻译hql，并把AST转换成Catalyst的LogicalPlan，后续优化、物理执行计 划翻译及执行过程，与Sql一样使用的是Catalyst提供的内容，执行引擎是Spark。在整个结合过程中，ASTNode映射成 LogicalPlan是重点。



而Hive社区的Hive on Spark会怎样实现，具体参考jira里的设计文档。

与Shark对比，



Shark多依赖了Hive的执行计划相关模块以及CLI。CLI和JDBC部分是Spark SQL后续打算支持的。Shark额外提供的对Table数据行转列、序列化、压缩存内存的模块，也被拿到了Spark Sql的sql工程里。

以上说明了Shark与Spark SQL Hive的区别，对Shark这个项目继承性的理解。

而Spark SQL Hive与Hive社区 Hive on Spark的区别需要具体参考jira里的设计文档，我也还没有读过。

spark-hive工程

	Spark SQL	Hive Support
Parser *词法语法解析，产出语法树*	Catalyst SqlParser	HiveQl.parseSql
Analyzer *解析出初步的逻辑执行计划*	Catalyst Analyzer	Hive MetaStore & Hive UDFs
Optimizer *逻辑执行计划优化*	Catalyst Optimizer	Catalyst Optimizer
QueryPlanner *逻辑执行计划映射物理执行计划*	spark-sql SparkPlanner & SparkStrategy	HivePlanner
Execute *物理执行计划树触发计算*	spark-sql SparkPlan.execute()	SparkPlan.execute()

解析过程

HiveQl.parseSql()把hql解析成logicalPlan。解析过程，提取出一些command，包括：

- set key=value
- cache table
- uncache table
- add jar
- add file
- dfs
- source

然后由Hive的ParseDriver把hql解析成AST，得到ASTNode，

[java] view plaincopy

def getAst(sql: String): ASTNode = ParseUtils.findRootNonNullToken((new ParseDriver).parse(sql))

把Node转化为Catalyst的LogicalPlan，转化逻辑较复杂，也是Sparksql对hql支持的最关键部分。详见HiveQl.nodeToPlan(node: Node):LogicalPlan方法。

大致转换逻辑包括：

处理TOK_EXPLAIN和TOK_DESCTABLE

处理TOK_CREATETABLE，包括创建表时候一系列表的设置TOK_XXX

处理TOK_QUERY，包括TOK_SELECT，TOK_WHERE，TOK_GROUPBY，TOK_HAVING，TOK_SORTEDBY，TOK_LIMIT等等，对FROM后面跟的语句进行nodeToRelation处理。

处理TOK_UNION

对Hive AST树结构和表示不熟悉，所以此处略过。

Analyze过程

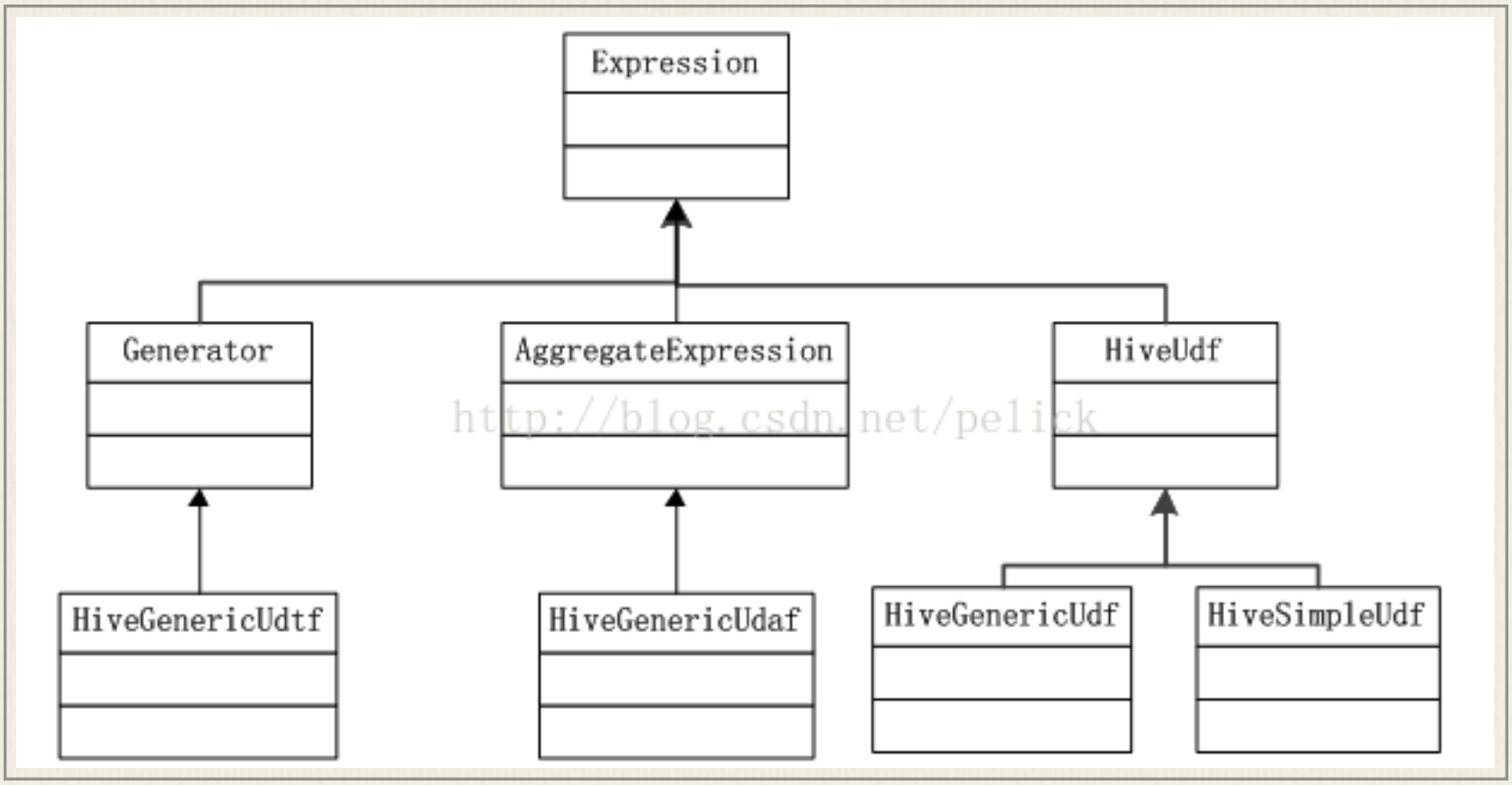
metadata交互

Catalog 类为HiveMetastoreCatalog，通过hive的conf生成client(org.apache.hadoop.hive.ql.metadata.Hive，用于与MetaStore通信，获得metadata 以及进行DDL操作)，catalog的lookupRelation方法里面，client.getTable()得到表信 息，client.getAddPartitionsForPruner()得到分区信息。

udf相关

FunctionRegistry 类为HiveFunctionRegistry，根据方法名，通过hive的相关类去查询该方法，检查是否具有该方法，是UDF，还是UDAF(aggregation)，或是UDTF(table)。这里只做已有udf的查询，不做新方法的include。

与Catalyst的Expression对应继承关系如下：



Inspector相关

HiveInspectors提供了几个映射数据类型和ObjectInspetor子类 的方法，包括 PrimitiveObjectInspector,ListObjectInspector,MapObjectInspector,StructObjectInspector 四种

Optimizer过程

在做优化前，会尝试对之前生成的逻辑执行计划进行 createtabl操作，因为执行的hql可能是“CREATE TABLE XXX”，这部分处理在HiveMetastoreCatalog的CreateTables单例里，继承了Rule[LogicalPlan]。

以及PreInsertionCasts处理，也是HiveMetastoreCatalog里的单例，继承了Rule[LogicalPlan]。

之后的optimizer过程同SQLContext里，用的是同一个Catalyst提供的Optimizer类。

Planner及执行过程

HiveContext继承自SQLContext，其QueryExecution也继承自SQLContext的QueryExecution。后续执行计划优化、物理执行计划翻译、处理及执行过程同SQL的处理逻辑是一致的。

翻译物理执行计划的时候，hive planner里制定了些特定的策略，与SparkPlanner稍有不同。

```
override val strategies: Seq[Strategy] = Seq(
  CommandStrategy(self),
  HiveCommandStrategy(self),
  TakeOrdered,
  ParquetOperations,
  InMemoryScans,
  HiveTableScans,
  DataSinks,
  Scripts,
  PartialAggregation,
  LeftSemiJoin,
  HashJoin,
  BasicOperators,
  CartesianProduct,
  BroadcastNestedLoopJoin
)
```

```
val strategies: Seq[Strategy] =
  CommandStrategy(self) ::
  TakeOrdered ::
  PartialAggregation ::
  LeftSemiJoin ::
  HashJoin ::
  InMemoryScans ::
  ParquetOperations ::
  BasicOperators ::
  CartesianProduct ::
  BroadcastNestedLoopJoin :: Nil
```


多了Scripts，DataSinks，HiveTableScans和HiveCommandStrategy四种处理物理执行计划的策略(见HiveStrategies)。

1. Scripts，用于处理那种hive命令行执行脚本的情况。实现方式是使用ProcessBuilder新起一个JVM进程的方式，用”/bin/bash –c scripts”的方式执行脚本并获取输出流数据，转化为Catalyst Row数据格式。

2. DataSinks，用于把数据写入到Hive表的情况。里面涉及到一些hive读写的数据格式转化、序列化、读配置等工作，最后通过SparkContext的runJob接口，提交作业。

3. HiveTableScans，用于对hive table进行扫描，支持使用谓词的分区裁剪(Partition pruning predicates are detected and applied)。

4. HiveCommandStrategy，用于执行native command和describe command。我理解是这种命令是直接调hive客户端单机执行的，因为可能只与meta data打交道。

toRDD: RDD[Row]处理也有少许区别，返回RDD[Row]的时候，对每个元素做了一次拷贝。

SQL Core

Spark SQL的核心是把已有的RDD，带上Schema信息，然后注册成类似sql里的”Table”，对其进行sql查询。这里面主要分两部分，一是生成SchemaRD，二是执行查询。

生成SchemaRDD

如果是spark-hive项目，那么读取metadata信息作为Schema、读取hdfs上数据的过程交给Hive完成，然后根据这俩部分生成SchemaRDD，在HiveContext下进行hql()查询。

对于Spark SQL来说，

数据方面，RDD可以来自任何已有的RDD，也可以来自支持的第三方格式，如json file、parquet file。

SQLContext下会把带case class的RDD隐式转化为SchemaRDD

[java] view plaincopy

```
implicit def createSchemaRDD[A <: Product: TypeTag](rdd: RDD[A]) =
new SchemaRDD(this,
SparkLogicalPlan(ExistingRdd.fromProductRdd(rdd)))
```

ExsitingRdd 单例里会反射出case class的attributes，并把RDD的数据转化成Catalyst的GenericRow，最后返回RDD[Row]，即一个 SchemaRDD。

这里的具体转化逻辑可以参考ExsitingRdd的productToRowRdd和convertToCatalyst方法。

之后可以进行SchemaRDD提供的注册table操作、针对Schema复写的部分RDD转化操作、DSL操作、saveAs操作等等。

Row和GenericRow是Catalyst里的行表示模型

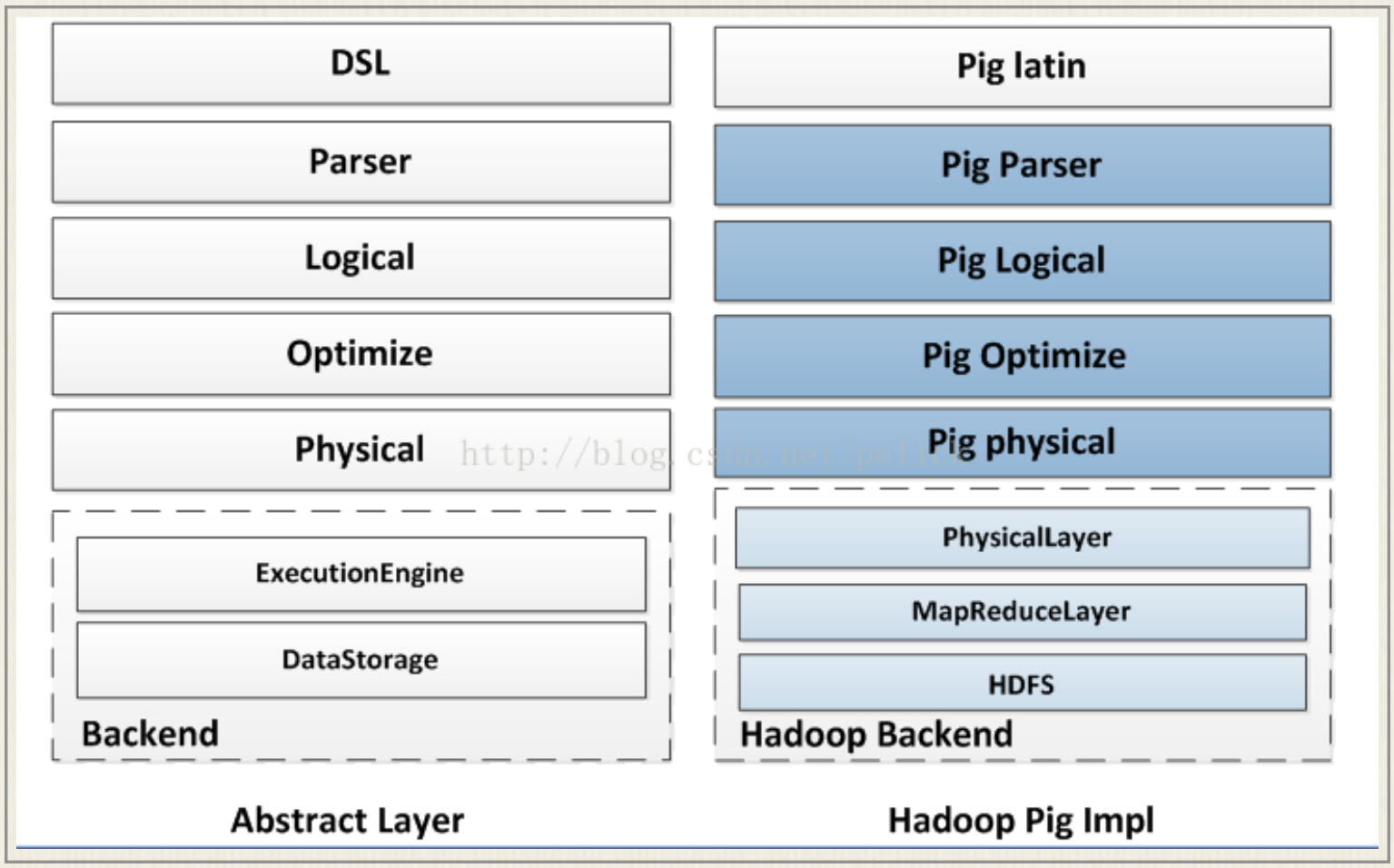
Row 用Seq[Any]来表示values，GenericRow是Row的子类，用数组表示values。Row支持数据类型包括Int, Long, Double, Float, Boolean, Short, Byte, String。支持按序数(ordinal)读取某一个列的值。读取前需要做isNullAt(i: Int)的判断。

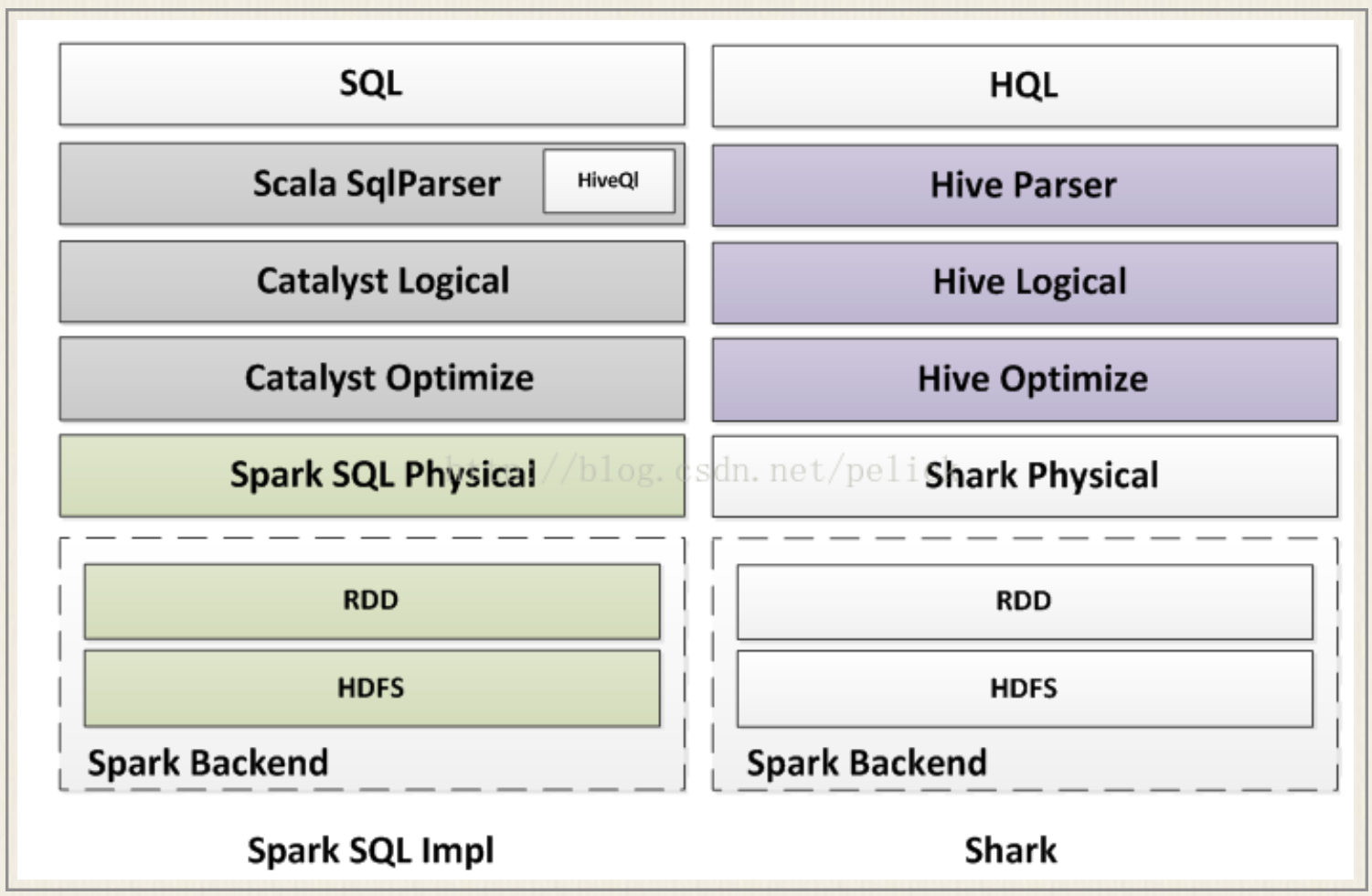
各自都有Mutable类，提供setXXX(i: int, value: Any)修改某序数上的值。

层次结构

	Spark SQL	Pig/Hive
Parser *词法语法解析，产出语法树*	Catalyst SqlParser	Antlr
Analyzer *解析出初步的逻辑执行计划*	Catalyst Analyzer	
Optimizer *逻辑执行计划优化*	Catalyst Optimizer	
QueryPlanner *逻辑执行计划映射物理执行计划*	spark-sql SparkPlanner & SparkStrategy	
Execute *物理执行计划树触发计算*	spark-sql SparkPlan.execute()	

下图大致对比了Pig，Spark SQL，Shark在实现层次上的区别，仅做参考。


DSL
Parser
Logical
Optimize
Physical
ExecutionEngine
DataStorage
Backend
Abstract Layer
Pig latin
Pig Parser
Pig Logical
Pig Optimize
Pig physical
PhysicalLayer
MapReduceLayer
HDFS
Hadoop Backend
Hadoop Pig Impl


SQL
Scala SqlParser
HiveQl
Catalyst Logical
Catalyst Optimize
Spark SQL Physical
RDD
HDFS
Spark Backend
Spark SQL Impl
HQL
Hive Parser
Hive Logical
Hive Optimize
Shark Physical
RDD
HDFS
Spark Backend
Shark

查询流程

SQLContext里对sql的一个解析和执行流程：

1. 第一步parseSql(sql: String)，simple sql parser做词法语法解析，生成LogicalPlan。

2. 第二步analyzer(logicalPlan)，把做完词法语法解析的执行计划进行初步分析和映射，

目前SQLContext内的Analyzer由Catalyst提供，定义如下：

new Analyzer(catalog, EmptyFunctionRegistry, caseSensitive =true)

catalog为SimpleCatalog，catalog是用来注册table和查询relation的。

而这里的FunctionRegistry不支持lookupFunction方法，所以该analyzer不支持Function注册，即UDF。

Analyzer内定义了几批规则：

```
1.  al batches: Seq[Batch] = Seq(
2.    Batch("MultiInstanceRelations", Once,
3.      NewRelationInstances),
4.    Batch("CaseInsensitiveAttributeReferences", Once,
5.
(if (caseSensitive) Nil else LowercaseAttributeReferences :: Nil) : _*),
6.    Batch("Resolution", fixedPoint,
7.      ResolveReferences ::
8.      ResolveRelations ::
9.      NewRelationInstances ::
10.     ImplicitGenerate ::
11.     StarExpansion ::
12.     ResolveFunctions ::
```

```
13.      GlobalAggregates ::
14.      typeCoercionRules :_*),
15.    Batch("Check Analysis", Once,
16.      CheckResolution),
17.    Batch("AnalysisOperators", fixedPoint,
18.      EliminateAnalysisOperators)
19.  )
```

3. 从第二步得到的是初步的logicalPlan，接下来第三步是optimizer(plan)。

Optimizer里面也是定义了几批规则，会按序对执行计划进行优化操作。

[java] view plaincopy

```
1.    val batches =
2.      Batch("Combine Limits", FixedPoint(100),
3.        CombineLimits) ::
4.      Batch("ConstantFolding", FixedPoint(100),
5.        NullPropagation,
6.        ConstantFolding,
7.        LikeSimplification,
8.        BooleanSimplification,
9.        SimplifyFilters,
10.       SimplifyCasts,
11.       SimplifyCaseConversionExpressions) ::
```

```
12.  Batch("Filter Pushdown", FixedPoint(100),
13.    CombineFilters,
14.    PushPredicateThroughProject,
15.    PushPredicateThroughJoin,
16.    ColumnPruning) :: Nil
```

4. 优化后的执行计划，还要丢给SparkPlanner处理，里面定义了一些策略，目的是根据逻辑执行计划树生成最后可以执行的物理执行计划树，即得到SparkPlan。

[java] view plaincopy

```
1.  val strategies: Seq[Strategy] =
2.    CommandStrategy(self) ::
3.    TakeOrdered ::
4.    PartialAggregation ::
5.    LeftSemiJoin ::
6.    HashJoin ::
7.    InMemoryScans ::
8.    ParquetOperations ::
9.    BasicOperators ::
10.   CartesianProduct ::
11.   BroadcastNestedLoopJoin :: Nil
```

5. 在最终真正执行物理执行计划前，最后还要进行两次规则，SQLContext里定义这个过程叫prepareForExecution，这个步骤是额外增加的，直接new RuleExecutor[SparkPlan]进行的。

[java] view plaincopy

```
1.  val batches =
2.    Batch("Add exchange", Once, AddExchange(self)) ::
3.
 Batch("Prepare Expressions", Once, new BindReferences[SparkPlan]) ::
Nil
```

6. 最后调用SparkPlan的execute()执行计算。这个execute()在每种SparkPlan的实现里定义，一般都会递归调用children的execute()方法，所以会触发整棵Tree的计算。

其他特性

内存列存储

SQLContext下cache/uncache table的时候会调用列存储模块。

该模块借鉴自Shark，目的是当把表数据cache在内存的时候做行转列操作，以便压缩。

实现类

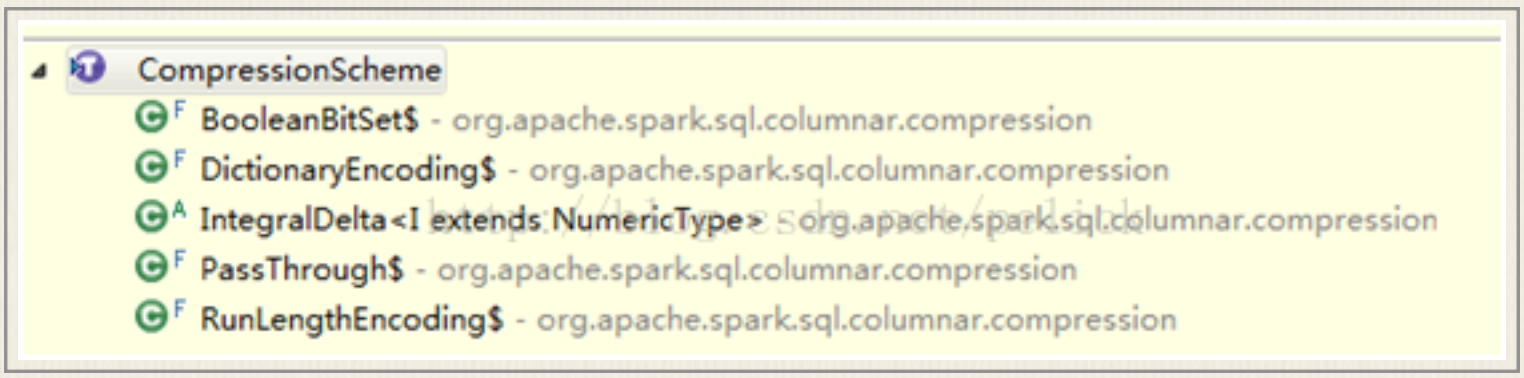
InMemoryColumnarTableScan类是SparkPlan LeafNode的实现，即是一个物理执行计划。传入一个SparkPlan(确认了的物理执行计)和一个属性序列，内部包含一个行转列、触发计算并cache的过程(且是lazy的)。

ColumnBuilder 针对不同的数据类型(boolean, byte, double, float, int, long, short, string)由不同的子类把数据写到ByteBuffer里，即包装Row的每个field，生成Columns。与其对应的 ColumnAccessor是访问column，将其转回Row。

CompressibleColumnBuilder和CompressibleColumnAccessor是带压缩的行列转换builder，其ByteBuffer内部存储结构如下

```
[java]
01. *    .--------------------------- Column type ID (4 bytes)
02. *    |   .----------------------- Null count N (4 bytes)
03. *    |   |   .------------------- Null positions (4 x N bytes, empty if null count is zero)
04. *    |   |   |     .------------- Compression scheme ID (4 bytes)
05. *    |   |   |     |   .--------- Compressed non-null elements
06. *    V   V   V     V   V
07. *   +---+---+-----+---+---------+
08. *   |   |   | ... |   | ... ... |
09. *  +---+---+-----+---+---------+
10. *  \-----------/ \-----------/
11. *       header         body
```

CompressionScheme子类是不同的压缩实现



都是scala实现的，未借助第三方库。不同的实现，指定了支持的column data类型。在build()的时候，会比较每种压缩，选择压缩率最小的（若仍大于0.8就不压缩了）。

这里的估算逻辑，来自子类实现的gatherCompressibilityStats方法。

Cache逻辑

cache之前，需要先把本次cache的table的物理执行计划生成出来。

在cache这个过程里，InMemoryColumnarTableScan并没有触发执行，但是生成了以InMemoryColumnarTableScan为物理执行计划的SparkLogicalPlan，并存成table的plan。

其 实在cache的时候，首先去catalog里寻找这个table的信息和table的执行计划，然后会进行执行（执行到物理执行计划生成），然后把这个table再放回catalog里维护起来，这个时候的执行计划已经是最终要执行的物理执行计划了。但是此时Columner模块相关的转换等操作都是没 有触发的。

真正的触发还是在execute()的时候，同其他SparkPlan的execute()方法触发场景是一样的。

Uncache逻辑

UncacheTable 的时候，除了删除catalog里的table信息之外，还调用了InMemoryColumnarTableScan的 cacheColumnBuffers方法，得到RDD集合，并进行了unpersist()操作。cacheColumnBuffers主要做了把 RDD每个partition里的ROW的每个Field存到了ColumnBuilder内。

UDF(暂不支持)

如前面对SQLContext里Analyzer的分析，其FunctionRegistry没有实现lookupFunction。

在spark-hive项目里，HiveContext里是实现了FunctionRegistry这个trait的，其实现为HiveFunctionRegistry，实现逻辑见org.apache.spark.sql.hive.hiveUdfs

Parquet支持

待整理

http://parquet.io/

Specific Docs and Codes:

https://github.com/apache/incubator-parquet-format

https://github.com/apache/incubator-parquet-mr

http://www.slideshare.net/julienledem/parquet-hadoop-summit-2013

JSON支持

SQLContext 下，增加了json-File的读取方法，而且目前看，代码里实现的是hadoop textfile的读取，也就是这份json文件应该是在HDFS上的。具体这份json文件的载入，InputFormat是 TextInputFormat，key class是LongWritable，value class是Text，最后得到的是value部分的那段String内容，即RDD[String]。

除了jsonFile，还支持jsonRDD，例子：

http://spark.apache.org/docs/latest/sql-programming-guide.html#json-datasets

读取json文件之后，转换成SchemaRDD。JsonRDD.inferSchema(RDD[String])里有详细的解析json和映射出schema的过程，最后得到该json的LogicalPlan。

Json的解析使用的是FasterXML/jackson-databind库，GitHub地址，wiki

把数据映射成Map[String, Any]

Json的支持丰富了Spark SQL数据接入场景。

JDBC支持

Jdbc support branchis under going

SQL92

Spark SQL目前的SQL语法支持情况见SqlPars-er类。目标是支持SQL92？？

1. 基本应用上，sql server 和oracle都遵循sql 92语法标准。

2. 实际应用中大家都会超出以上标准，使用各家数据库厂商都提供的丰富的自定义标准函数库和语法。

3. 微软sql server的sql 扩展叫T-SQL(Transcate SQL).

4. Oracle 的sql 扩展叫PL-SQL.

存在问题

大家可以跟进社区邮件列表，后续待整理。

http://apache-spark-developers-list.1001551.n3.nabble.com/sparkSQL-thread-safe-td7263.html

http://apache-spark-user-list.1001560.n3.nabble.com/Supported-SQL-syntax-in-Spark-SQL-td9538.html

总结

以上整理了对Spark SQL各个模块的实现情况，代码结构，执行流程以及自己对Spark SQL的理解。

理解有偏差的地方欢迎交流讨论 :)

全文完 :)

原文链接：http://blog.csdn.net/pelick/article/details/37809681

TSRC：主流WAF架构分析与探索

作者：腾讯安全应急响应中心 那个谁

摘要: 背景： 网站安全漏洞在被发现后，需要修复，而修复需要时间，有些厂商可能无能力修复，甚至部分网站主可能连是否存在漏洞都无法感知（尤其是攻击者使用0day的 情况下）。或者刚公开的1day漏洞，厂商来不及修复，而众多黑客已经掌握利用方法四面出击，防不胜防。这个时候如...

背景：

网站安全漏洞在被发现后，需要修复，而修复需要时间，有些厂商可能无能力修复，甚至部分网站主可能连是否存在漏洞都无法感知（尤其是攻击者使用 0day的情况下）。或者刚公开的1day漏洞，厂商来不及修复，而众多黑客已经掌握利用方法四面出击，防不胜防。这个时候如果有统一集中的网站防护系统 （WAF）来防护，则漏洞被利用的风险将大大降低，同时也为漏洞修复争取时间！

笔者所在公司的业务较多，光域名就成千上万，同时网络流量巨大，动辄成百上千G。而这些业务部署在数十万台服务器上，管理运维职能又分散在数十个业 务部门，安全团队不能直接的管理运维。在日常的安全管理和对抗中，还会经常性或者突发性的遇到0day需要响应，比如更新检测引擎的功能和规则，这些都是 比较现实的挑战。

对于WAF的实现，业界知名的网站安全防护产品也各有特色。本文将对主流的WAF架构做一个简单的介绍，和大家分享下。同时结合笔者所在公司业务的实际情况，采用了一种改进方案进行探索，希望本文能达到抛砖引玉的效果。(不足之处请各位大牛多多指教)

特别声明：以下方案无优劣之分，仅有适合不适合业务场景的区别。

业界常见网站安全防护方案

方案A：本机服务器模块模式

通过在Apache，IIS等Web服务器内嵌实现检测引擎，所有请求的出入流量均先经过检测引擎的检测，如果请求无问题则调用CGI处理业务逻 辑，如果请求发现攻击特征，再根据配置进行相应的动作。以此对运行于Web服务器上的网站进行安全防护。著名的安全开源项目ModSecurity及 nax-si防火墙就是此种模式。

优点：

1、 网络结构简单，只需要部署Web服务器的安全模块

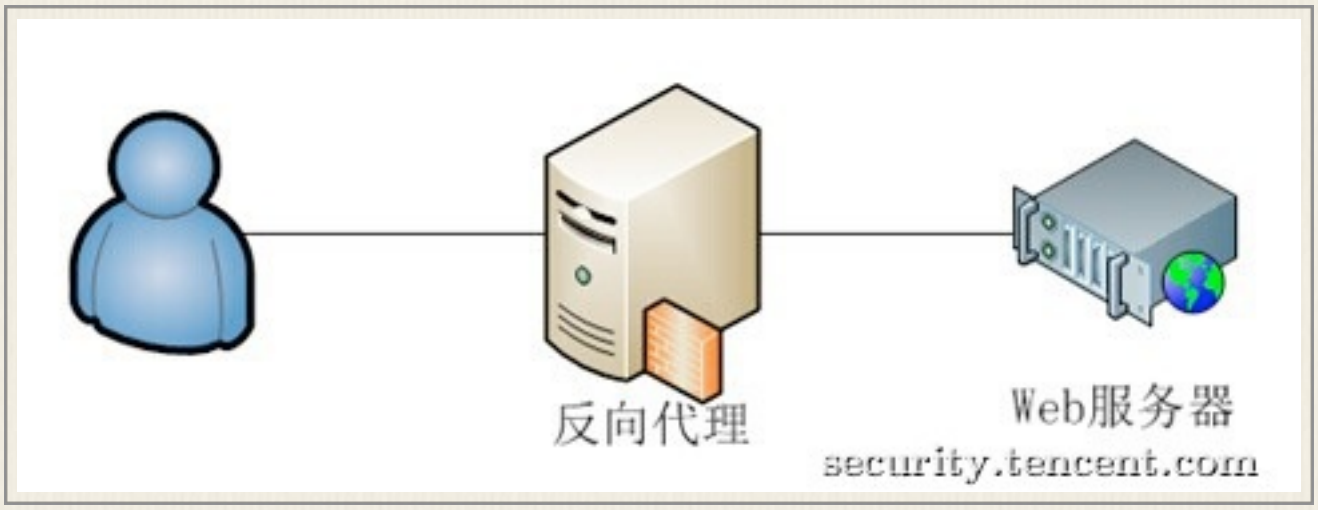
挑战：

1、 维护困难。当有大规模的服务器集群时，任何更新都涉及到多台服务器。

2、 需要部署操作，在面临大规模部署时成本较高。

3、 无集中化的数据中心。针对安全事件的分析往往需要有集中式的数据汇总，而此种模式下用户请求数据分散在各个Web服务器上。

方案B：反向代理模式



使用这种模式的方案需要修改DNS，让域名解析到反向代理服务器。当用户向某个域名发起请求时，请求会先经过反向代理进行检测，检测无问题之后再转 发给后端的Web服务器。这种模式下，反向代理除了能提供Web安全防护之外，还能充当抗DDoS攻击，内容加速（CDN）等功能。云安全厂商 CloudFlare采用这种模式。

优点：

1、 集中式的流量出入口。可以针对性地进行大数据分析。

2、 部署方便。可多地部署，附带提供CDN功能。

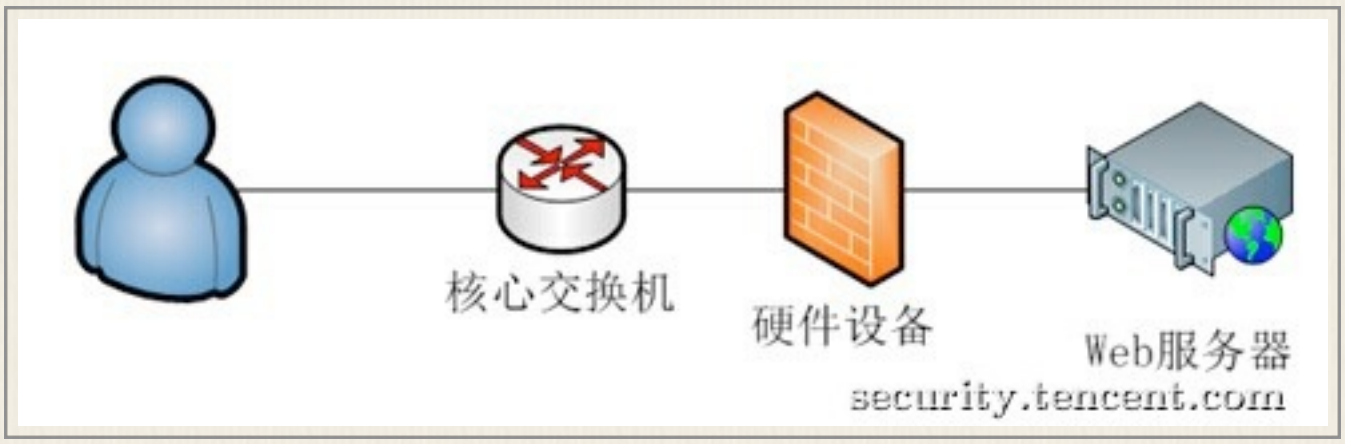
挑战：

1、 动态的额外增加一层。会带来用户请求的网络开销等。

2、 站点和后端Web服务器较多的话，转发规则等配置较复杂。

3、 流量都被捕捉，涉及到敏感数据保护问题，可能无法被接受。

方案C：硬件防护设备



这种模式下，硬件防护设备串在网络链路中，所有的流量经过核心交换机引流到防护设备中，在防护设备中对请求进行检测，合法的请求会把流量发送给 Web服务器。当发现攻击行为时，会阻断该请求，后端Web服务器无感知到任何请求。防护设备厂商如imperva等使用这种模式。

优点：

1、 对后端完全透明。

挑战：

1、 部署需改变网络架构，额外的硬件采购成本。

2、 如Web服务器分布在多个IDC，需在多个IDC进行部署。

3、 流量一直在增加，需考虑大流量处理问题。以及流量自然增长后的升级维护成本。

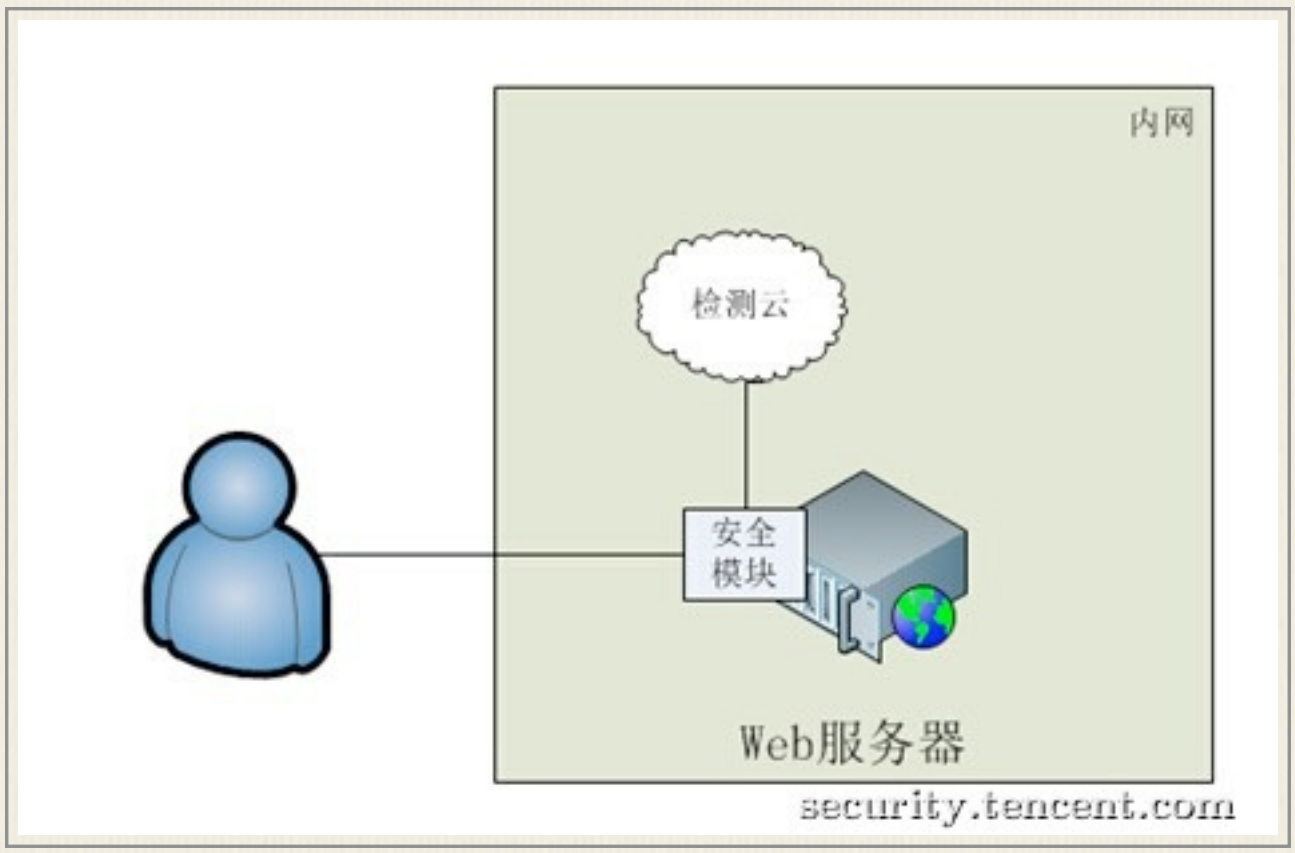
4、 规则依赖于厂商，无法定制化，不够灵活。

我们的探索

笔者所在的公司在不同的场景下，也近似的采纳了一种或者多种方案的原型加以改进使用运营。（比如方案C，其实我们也有给力的小伙伴做了很棒的努力，通过自研的方式解决了不少挑战，以后有机会或许也会和大家分享一些细节）

今天给大家介绍的，是我们有别于业界常见模型的一种思路：

方案D：服务器模块+检测云模式



这种模式其实是方案A的增强版，也会在Web服务器上实现安全模块。不同点在于，安全模块的逻辑非常简单，只是充当桥梁的作用。检测云则承担着所有 的检测发现任务。当安全模块接收到用户的请求时，会通过UDP或者TCP的方式，把用户请求的HTTP文本封装后，发送到检测云进行检测。当检测无问题 时，告知安全模块把请求交给CGI处理。当请求中检测到攻击特征时，则检测云会告知安全模块阻断请求。这样所有的逻辑、策略都在检测云端。

我们之所以选择这个改进方案来实现防护系统，主要是出于以下几个方面的考虑：

1、 维护问题

假如使用A方案，当面临更新时，无法得到及时的响应。同时，由于安全逻辑是嵌入到Web服务器中的，任何变更都存在影响业务的风险，这是不能容忍的。

2、 网络架构

如果使用方案B，则需要调动大量的流量，同时需要提供一个超大规模的统一接入集群。而为了用户就近访问提高访问速度，接入集群还需要在全国各地均有部署，对于安全团队来说，成本和维护难度难以想象。

使用该方案时，需要考虑如下几个主要的挑战：

1、 网络延时

采用把检测逻辑均放在检测云的方式，相对于A来说，会增加一定的网络开销。不过，如果检测云放在内网里，这个问题就不大，99%的情况下，同城内网发送和接收一个UDP包只需要1ms。

2、 性能问题：

由于是把全量流量均交给集中的检测云进行检测，大规模的请求可能会带来检测云性能的问题。这样在实现的时候就需要设计一个好的后端架构，必须充分考虑到负载均衡，流量调度等问题。

3、 部署问题：

该方案依然需要业务进行1次部署，可能会涉及到重编译web服务器等工作量，有一定的成本。并且当涉及到数千个域名时，问题变的更为复杂。可能需要区分出高危业务来对部署有一个前后顺序，并适时的通过一些事件来驱动部署。

最后，我们目前已灰度上线了这套防御方案，覆盖重要以及高危的业务站点。目前日均处理量约数百亿的规模，且正在快速增长阶段。

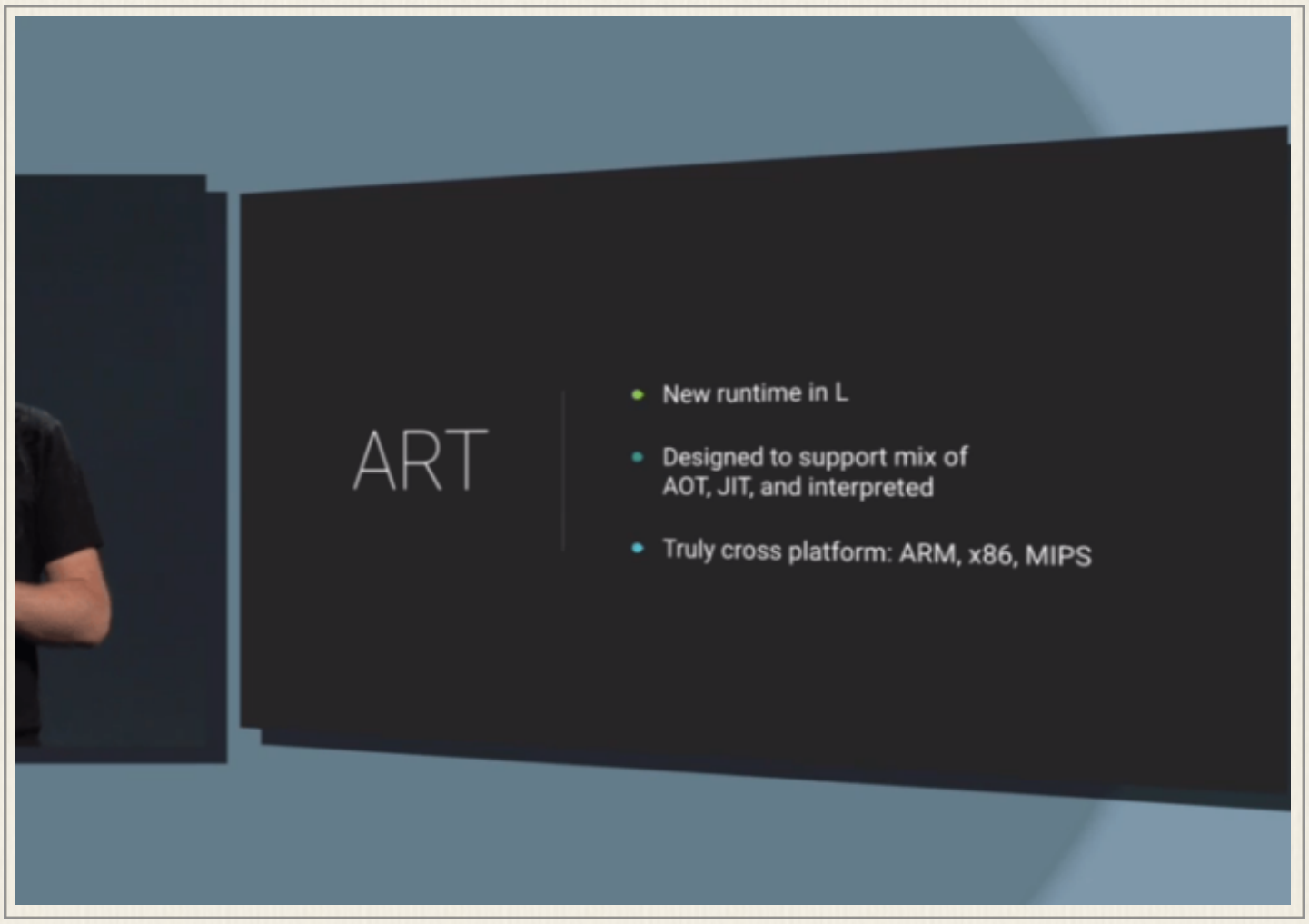
本文只是一个粗略的简介，我们在建设过程中也遇到了远高于想象的挑战和技术难题，后续将会有系列文章来深入剖析和介绍，希望对大家有所帮助。

原文链接：http://www.youxia.org/tsrc-waf.html

近距离端详Android ART运行时库

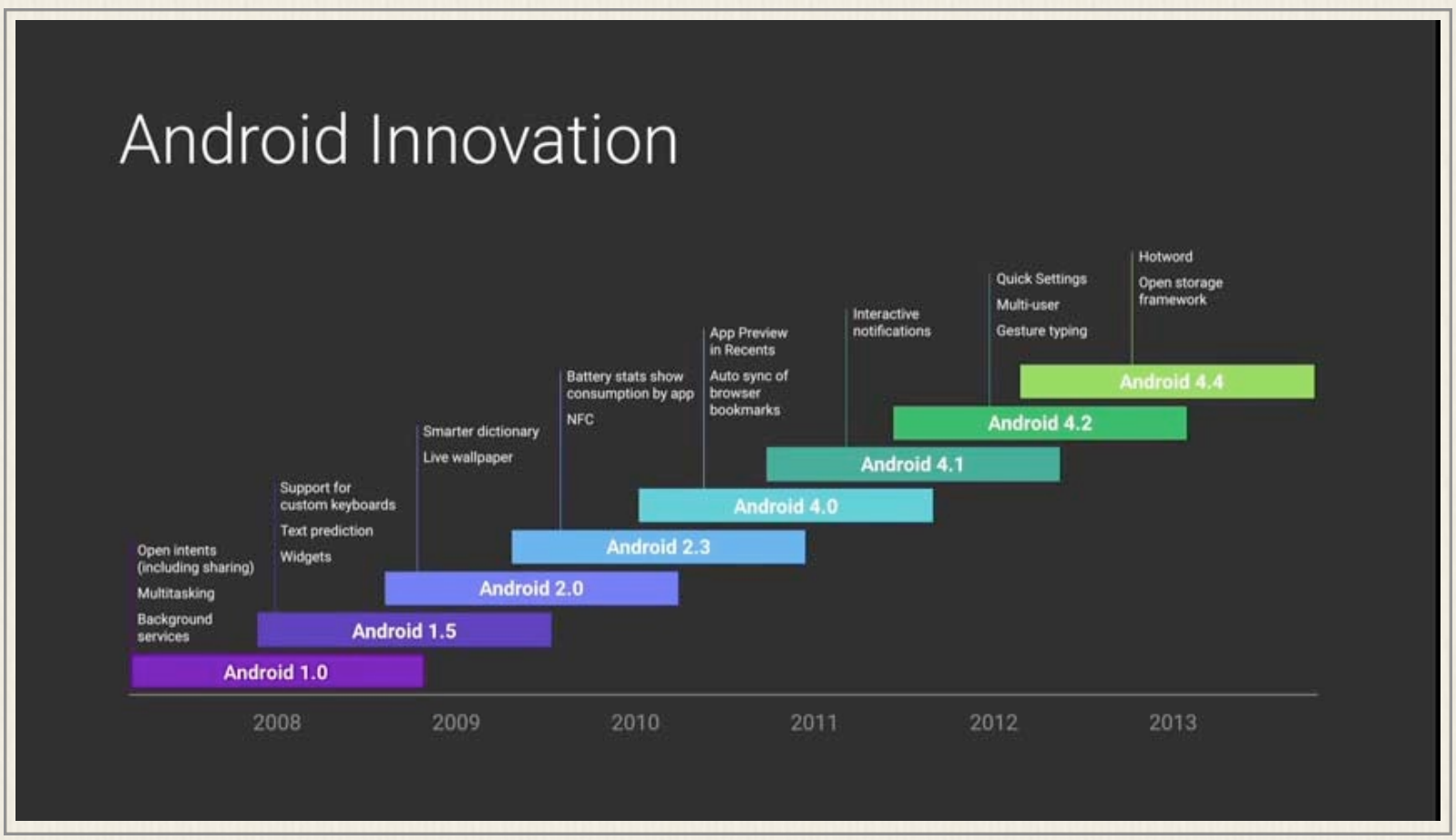
作者：Lingcc

在最新的Google I/O大会上，Google 发布了关于Android上最新的运行时库的情况。这就是Android RunTime （ART). ART 将会取代Dalvik虚拟机，成为Android平台上Java代码的执行工具。虽然自从Android KitKat，就有了一些关于ART的消息，但是基本都是一些新闻性质的，缺乏具体技术细节方面的介绍。本文尝试综合目前已有的各种消息，以及最新放出的 Android L 预览版本的ROM的情况，对ART运行时库做个详细的分析。



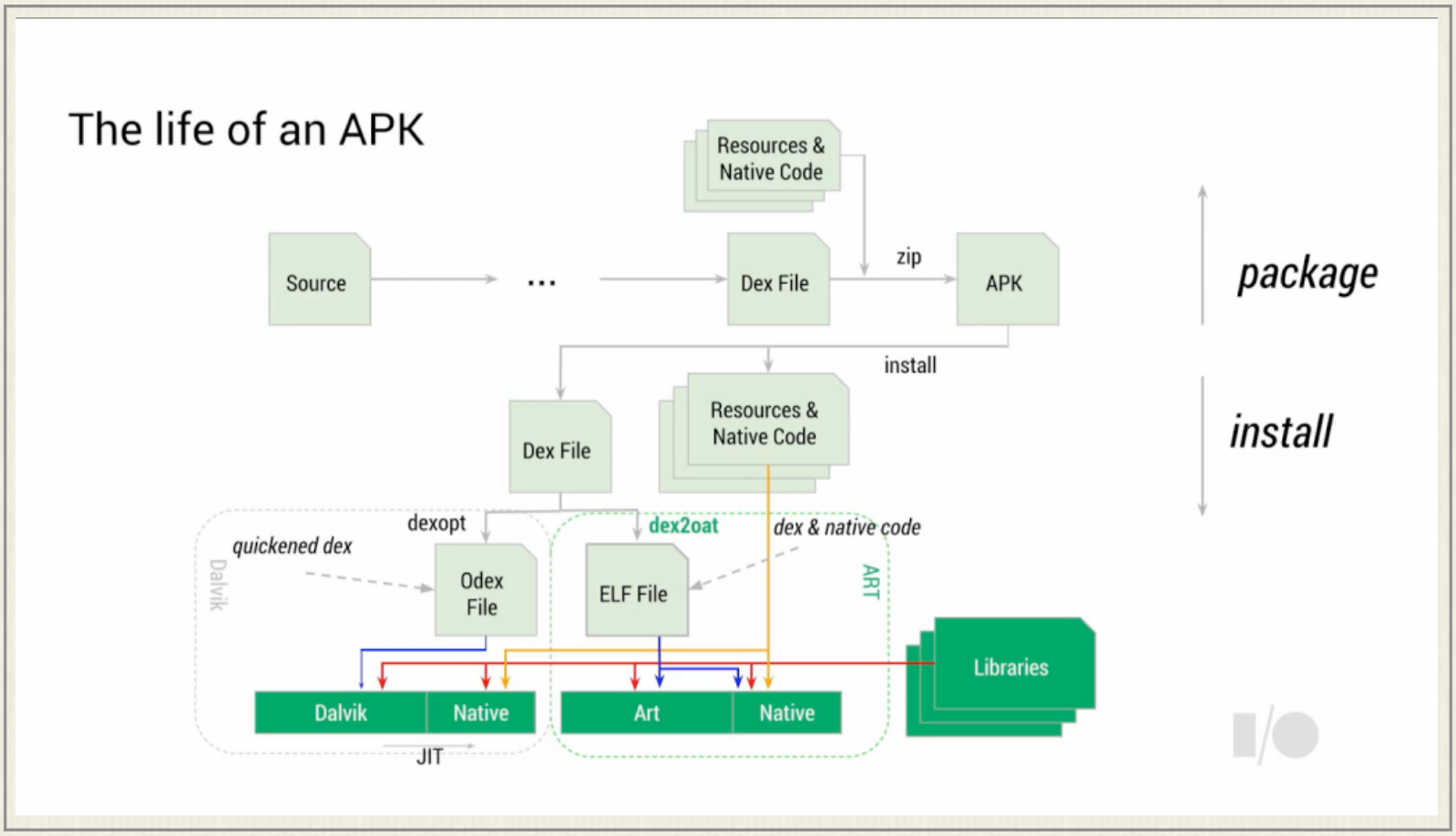
和IOS，Windows，Tizen之类的移动平台直接将软件编译成能够直接运行在特定硬件平台上的本地代码不同。Android平台上的软件会被编译器首先编译成通用的“byte-code”，然后再在具体的移动设备上被转换成本地指令执行。

从Android诞生至今的十几年时间里，Dalvik从开始时非常简单的Java Byte-Code 执行虚拟机，逐渐增加各种新的特性，满足应用程序对性能的需求，以及与硬件设备协同演进。这其中包括在Android 2.2版本中引入的即时编译器(JIT-Compiler), 以及随后的多线程支持，以及其他一些优化。



不过，在近两年里，Android整个生态系统的进步对Android虚拟机的需求，目前的Dalvik虚拟机的开发已经无法满足了。Dalvik最初设 计时，处理器的性能很弱，移动设备的内存空间非常有限，而且都是32位的系统。于是Google开始构建一个新的虚拟机来更好的面对未来的发展趋势。这种 虚拟机的性能能够在目前的多核处理器，甚至未来的8核处理上轻松扩展，能够满足对大容量存储的支持，以及大容量内存的支持。 于是乎，ART出现了。

1 架构介绍



首先，ART的首要设计需求就是完全兼容能在Dalvik上运行的byte-code，即dex(Dalvik executable)。这样的话，对于程序员来说，就不需要对重新编译已有的程序，直接拿APK就可以在Dalvik和ART虚拟机上运行。ART带来 的最大的变化，就是使用预编译技术（Ahead-of-Time compile)取代Dalvik中的即时编译技术(Just-In-Time compile)。之前，在应用程序每次执行的时候，虚拟机需要将bytecode编译成本地码执行，而在ART中这种编译操作只需执行一次，随后对该应 用程序的执行都可以通过直接执行保存下来的本地码完成。当然，这种预编译技术，需要占用额外的存储空间来存储本地码。正是因为现在移动设备的存储空间越来 越大，这种技术才得以应用。

这种预编译技术使得很多原来无法执行的编译优化技术在新的Android平台上成为可能。因为代码只被编译和优化一次，因此值得花费更多的时间在这 次编译上，以便进行更多的优化。Google表示，现在可以在应用程序的整体代码技术上进行更高层次的优化，因为编译器现在能够看到应用程序的

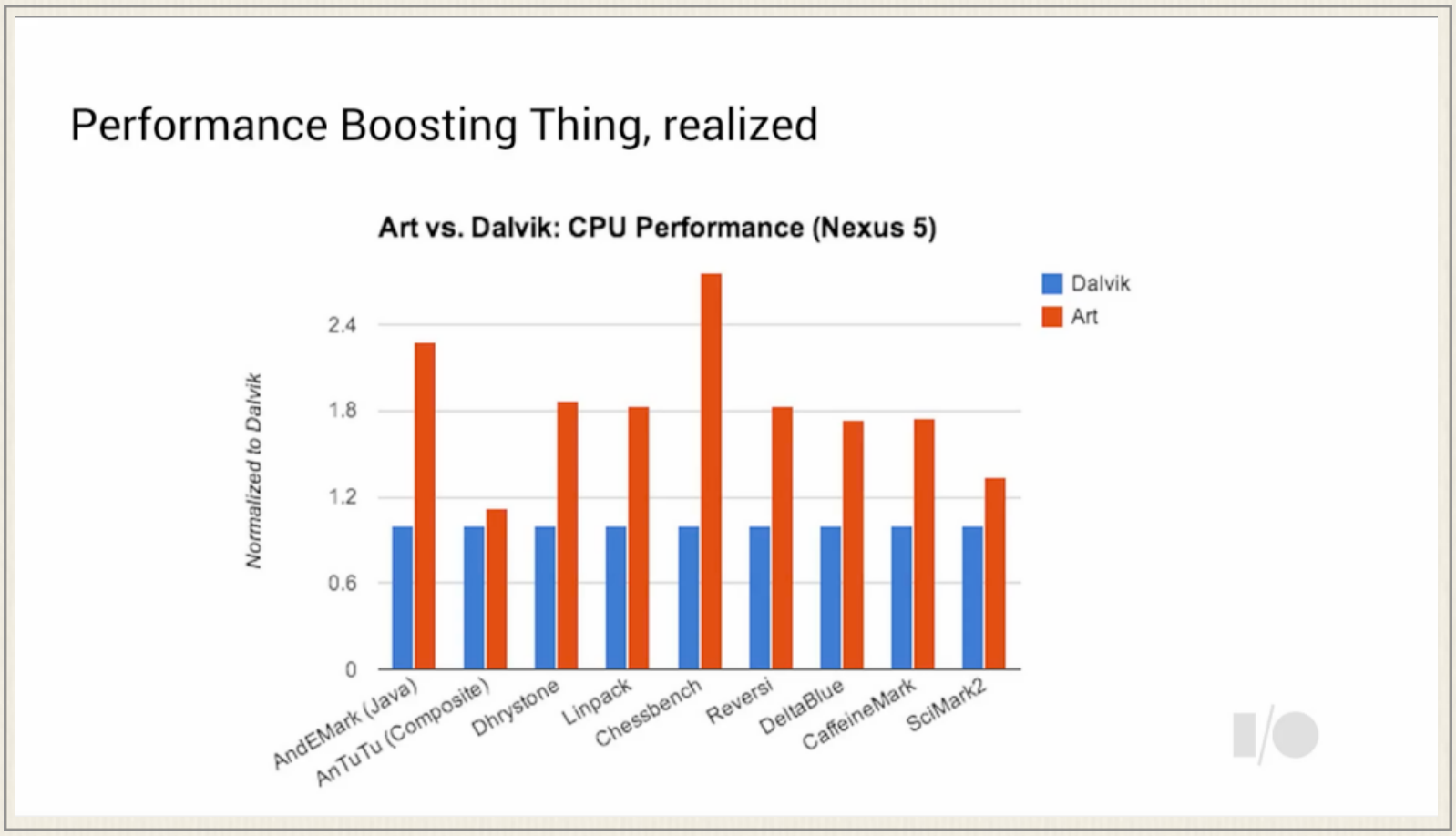
整体代码， 而之前的即时编译，编译器只能看到并优化应用程序中某个函数或者非常小的一部分代码。采用ART后，代码中异常检查带来的开销绝大部分可以避免，对方法和 接口的调用也加快了很多。完成这部分功能的是新添加的“dex2oat”组件，用来替代Dalvik中对应的“dexopt”组件。Dalvik中的 Odex文件(优化后的dex)文件，在ART中也用ELF文件代替了。

因为ART目前编译生成ELF可执行文件，内核就可以直接对载入内存中的代码进行分页管理，这也会带来更加高效的内存管理，以及更少的内存占用。说 到这里，我非常好奇内核中的KSM(Kernel same-page merging)在ART中会有什么样的影响，应该能带来不错的效果吧。我们拭目以待。

ART对续航时间的影响也是非常显著的。因为不再需要解释执行，JIT也不用在程序运行时工作，这样会直接节省CPU需要执行的指令数，因而耗电降低。

因为预编译时引入了更多分析和优化，编译的时间会变长，这是ART可能会带来的一个副作用。因此相比Dalvik虚拟机，当设备首次启动及应用程序 第一次安装时，需要花费的时间更久。Google声称，这种时间上的增加并不那么恐怖。他们希望并预期日后ART上完成上述动作的时间会和目前的Dalvik差不多，甚至更短些。

下面的图显示，ART带来的性能提升是非常明显的。对于同样的代码，性能提升约2倍左右。Google称，将Android L最终发布的时候，可以预计的性能提升将会像Chessbench一样，有3x的加速。



2 垃圾收集：理论和实践

Android虚拟机依赖自动化的内存管理机制，即自动垃圾收集。这一Java语言编程模式的基石也是Android系统自诞生之日起，非常重要的 一部分。这里向不太了解垃圾收集概念的朋友解释一下，所谓自动垃圾收集，就是说程序员在编程过程中，不需要自己负责物理内存的存储的分配和释放。只需要使 用固定的模式创建你需要的变量或者对象，然后直接利用该变量或对象即可。程序的运行环境会自动在内存中分配相应的内存空间存储该变量或者对象， 并在该变量或者对象失效后，自动释放所分配的内存。这是和其他需要人工进行存储管理的较低层次语言最大的区别。自动垃圾收集的好处是，程序员不必再在编程 时担心内存管理的问题，当然，这也是有代价的，那就是程序员无法控制内存何时分配和释放，因而无法在需要时进行优化（Java语言有一些编程接口可以供程 序员手工优化程序，但控制方式和粒度有限).

Android曾经被Dalvik的垃圾收集机制折腾了很久。Android平台的内存普遍较小，每次应用程序需要分配内存，当堆空间（分配给应用 程序的一块内存空间）不能提供如此大小的空间时，Dalvik的垃圾收集器就会启动。垃圾收集器会遍历整个堆空间，查看每一个应用程序分配的对象，并对所 有

可到达的对象（即还会被使用的对象）标记，并将那些没有标记的对象空间释放掉。

在Dalvik虚拟机中，垃圾收集器执行的过程将导致两次应用程序的停顿：

- 一是在遍历堆地址空间阶段，
- 另一个是标记阶段。

所谓停顿，即应用程序所有正在执行的进程将暂停。如果停顿时间过长，将会导致应用程序在渲染时出现丢帧现象，进而导致应用程序的卡顿现象，大大降低用户体验。

Google声称，在Nexus 5手机上，这种停顿的平均长度在54ms。这个停顿时间将导致平均每次垃圾收集会导致在应用程序渲染显式时丢掉4帧的。

我自己的经验和测试表明，根据应用程序的不同，停顿的时间可能会增大很多。比如，在官方的FIFA应用程序这一典型程序中，垃圾收集的停顿会非常厉害。

07-01 15:56:14.275: D/dalvikvm(30615): GCFORALLOC freed 4442K, 25% free 20183K/26856K, paused 24ms, total 24ms

07-01 15:56:16.785: I/dalvikvm-heap(30615): Grow heap (frag case) to 38.179MB for 8294416-byte allocation

07-01 15:56:17.225: I/dalvikvm-heap(30615): Grow heap (frag case) to 48.279MB for 7361296-byte allocation

07-01 15:56:17.625: I/Choreographer(30615): Skipped 35 frames! The application may be doing too much work on its main thread.

07-01 15:56:19.035: D/dalvikvm(30615): GCCONCURRENT freed 35838K, 43% free 51351K/89052K, paused 3ms+5ms, total 106ms

07-01 15:56:19.035: D/dalvikvm(30615): WAITFORCONCURRENTGC blocked 96ms

07-01 15:56:19.815: D/dalvikvm(30615): GCCONCURRENT freed 7078K, 42% free 52464K/89052K, paused 14ms+4ms, total 96ms

07-01 15:56:19.815: D/dalvikvm(30615): WAITFORCONCURRENTGC blocked 74ms

07-01 15:56:20.035: I/Choreographer(30615): Skipped 141 frames! The application may be doing too much work on its main thread.

07-01 15:56:20.275: D/dalvikvm(30615): GCFORALLOC freed 4774K, 45% free 49801K/89052K, paused 168ms, total 168ms

07-01 15:56:20.295: I/dalvikvm-heap(30615): Grow heap (frag case) to 56.900MB for 4665616-byte allocation

07-01 15:56:21.315: D/dalvikvm(30615): GCFORALLOC freed 1359K, 42% free 55045K/93612K, paused 95ms, total 95ms

07-01 15:56:21.965: D/dalvikvm(30615): GCCONCURRENT freed 6376K, 40% free 56861K/93612K, paused 16ms+8ms, total 126ms

07-01 15:56:21.965: D/dalvikvm(30615): WAITFORCONCURRENTGC blocked 111ms

07-01 15:56:21.965: D/dalvikvm(30615): WAITFORCONCURRENTGC blocked 97ms

07-01 15:56:22.085: I/Choreographer(30615): Skipped 38 frames! The application may be doing too much work on its main thread.

07-01 15:56:22.195: D/dalvikvm(30615): GCFORALLOC freed 1539K, 40% free 56833K/93612K, paused 87ms, total 87ms

07-01 15:56:22.195: I/dalvikvm-heap(30615): Grow heap (frag case) to 60.588MB for 1331732-byte allocation

07-01 15:56:22.475: D/dalvikvm(30615): GCFORALLOC freed 308K, 39% free 59497K/96216K, paused 84ms, total 84ms

07-01 15:56:22.815: D/dalvikvm(30615): GCFORALLOC freed 287K, 38% free 60878K/97516K, paused 95ms, total 95ms

上面的log是从FIFA应用程序运行后的几秒钟时间里截取的。垃圾收集器在短短的8秒内被执行了9次，导致应用程序总共卡顿了603ms，丢帧达214次。绝大多数的卡顿都来自内存分配请求，在log中以”GC_FOR_ALLOC“标签描述。

ART将整个垃圾收集系统做了重新设计和实现。为了能做些对比，下面给出使用ART运行相同的应用程序，在相同的场景下提取的log：

07-01 16:00:44.531: I/art(198): Explicit concurrent mark sweep GC freed 700(30KB) AllocSpace objects, 0(0B) LOS objects, 792% free, 18MB/21MB, paused 186us total 12.763ms

07-01 16:00:44.545: I/art(198): Explicit concurrent mark sweep GC freed 7(240B) AllocSpace objects, 0(0B) LOS objects, 792% free, 18MB/21MB, paused 198us total 9.465ms

07-01 16:00:44.554: I/art(198): Explicit concurrent mark sweep GC freed 5(160B) AllocSpace objects, 0(0B) LOS objects, 792% free, 18MB/21MB, paused 224us total 9.045ms

07-01 16:00:44.690: I/art(801): Explicit concurrent mark sweep GC freed 65595(3MB) AllocSpace objects, 9(4MB) LOS objects, 810% free, 38MB/58MB, paused 1.195ms total 87.219ms

07-01 16:00:46.517: I/art(29197): Background partial concurrent mark sweep GC freed 74626(3MB) AllocSpace objects, 39(4MB) LOS objects, 1496% free, 25MB/32MB, paused 4.422ms total 1.371747s

07-01 16:00:48.534: I/Choreographer(29197): Skipped 30 frames! The application may be doing too much work on its main thread.

07-01 16:00:48.566: I/art(29197): Background sticky concurrent mark sweep GC freed 70319(3MB) AllocSpace objects, 59(5MB) LOS objects, 825% free, 49MB/56MB, paused 6.139ms total 52.868ms

07-01 16:00:49.282: I/Choreographer(29197): Skipped 33 frames! The application may be doing too much work on its main thread.

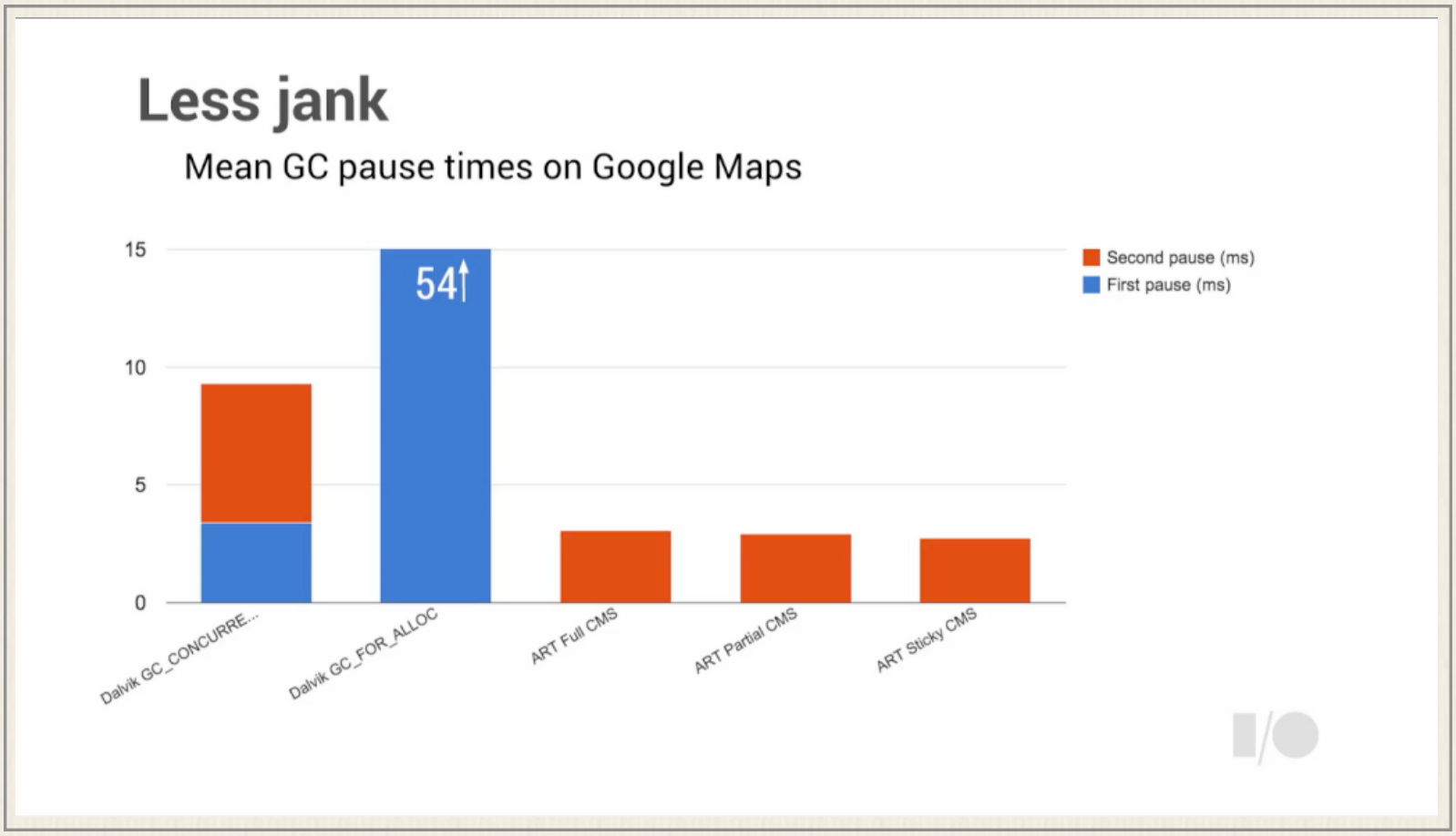
07-01 16:00:49.652: I/art(1287): Heap transition to ProcessStateJankImperceptible took 45.636146ms saved at least 723KB

07-01 16:00:49.660: I/art(1256): Heap transition to ProcessStateJankImperceptible took 52.650677ms saved at least 966KB

ART和Dalvik的差别非常大，新的运行时内存管理仅仅停顿了12.364ms，运行了4次前台垃圾收集，以及2次后台垃圾收集。在应用程序执 行的过程中，应用程序的堆空间大小并没有增加，而Dalvik虚拟机中堆空间共增加了4次。丢帧的个数方面，ART虚拟机也降到了63帧。

上面这段示例，只不过是一个开发并不完善的应用程序中最坏的一个场景。因为即使在ART虚拟机中，这个应用程序还是丢掉了不少帧渲染图像。不过上面 的log对比依然很有参考价值，毕竟牛逼的程序员没几个，大多数的Android程序都没办法开发的很完美。Android需要能hold住这种情况。

ART将一些通常需要垃圾收集器做的工作，拆分给应用程序本身完成。这样，Dalvik中因为遍历堆空间引入的第一次停顿，就被完全消除了。而第二 次停顿也因为一项预清理技术 （packard pre-cleaning）的应用而大大缩短。使用该技术后，只需要在清理完成后，简单的检查和验证时稍微停顿一下即可。Google声称，他们已经设法 将这类停顿的时间缩短到3ms左右，相比Dalvik虚拟机的垃圾收集器来说，基本上是一个多数量级的降低，很不错了。



ART 还引入了一个特殊的超大对象存储空间(large object space，LOS)，这个空间与堆空间是分开的，不过仍然驻留在应用程序内存空间中。这一特殊的设计是为了让ART可以更好的管理较大的对象，比如位图对象(bitmaps)。在对堆空间分段时，这种较大的对象会带来一些问题。比如，在分配一个此类对象时，相比其他普通对象，会导致垃圾收集器启动的次数 增加很多。有了这个超大对象存储空间的支持，垃圾收集器因堆空间分段而引发调用次数将会大大降低，这样垃圾收集器就能做更加合理的内存分配，从而降低运行 时开销。

一个很好的例子，就是运行Hangouts(环聊)应用程序时，在Dalvik虚拟机中，我们能看到数次因为分配内存，运行GC而导致的停顿。

07-01 06:37:13.481: D/dalvikvm(7403): GCEXPLICIT freed 2315K, 46% free 18483K/34016K, paused 3ms+4ms, total 40ms

07-01 06:37:13.901: D/dalvikvm(9871): GCCONCURRENT freed 3779K, 22% free 21193K/26856K, paused 3ms+3ms, total 36ms

07-01 06:37:14.041: D/dalvikvm(9871): GCFORALLOC freed 368K, 21% free 21451K/26856K, paused 25ms, total 25ms

07-01 06:37:14.041: I/dalvikvm-heap(9871): Grow heap (frag case) to 24.907MB for 147472-byte allocation

07-01 06:37:14.071: D/dalvikvm(9871): GCFORALLOC freed 4K, 20% free 22167K/27596K, paused 25ms, total 25ms

07-01 06:37:14.111: D/dalvikvm(9871): GCFORALLOC freed 9K, 19% free 23892K/29372K, paused 27ms, total 28ms

我们从所有的垃圾收集log中截取了上述一段。其中的显式(GC_EXPLICIT)和并发(GC_CONCURRENT)是垃圾收集器中比较通用 的清理和维护性调用。GC_FOR_ALLOC则是在内存分配器尝试分配新的内存空间，但堆空间不够用时，调用的。上面的log中，我们能看到堆空间因为分段操作而扩充了堆空间，但仍然无法装下大对象。在整个大对象分配的过程中，停顿时间长达90ms。

相比之下，下面这段log是从Android L预览版本的ART运行log中提取的。

07-01 06:35:19.718: I/art(10844): Heap transition to ProcessStateJankPerceptible took 17.989063ms saved at least -138KB

07-01 06:35:24.171: I/art(1256): Heap transition to ProcessStateJankImperceptible took 42.936250ms saved at least 258KB

07-01 06:35:24.806: I/art(801): Explicit concurrent mark sweep GC freed 85790(3MB) AllocSpace objects, 4(10MB) LOS objects, 850% free, 35MB/56MB, paused 961us total 83.110ms

我们目前还不知道log中的”Heap Transition”表达的什么意思，不过可以猜测应该是堆空间大小重设机制。在应用程序已经运行之后，唯一的对垃圾收集器的调用仅消耗的961us。 我们并没有在这段截取的log之前，发现任何对垃圾收集器的调用操作。这段log中比较有趣的，就是LOS的统计。能

够看到，在LOS中有4个较大的对 象，共10MB。这块内存并没有分配在堆空间内，否则应该会有类似Dalvik的提示。

ART的内存分配系统本身也被重写了。虽然ART相比Dalvik，在内存分配方面，能够带来大约25%的性能提升，不过Google显然对此不满意，因此引入了一个新的内存分配器来取代当前使用的“malloc”分配器。

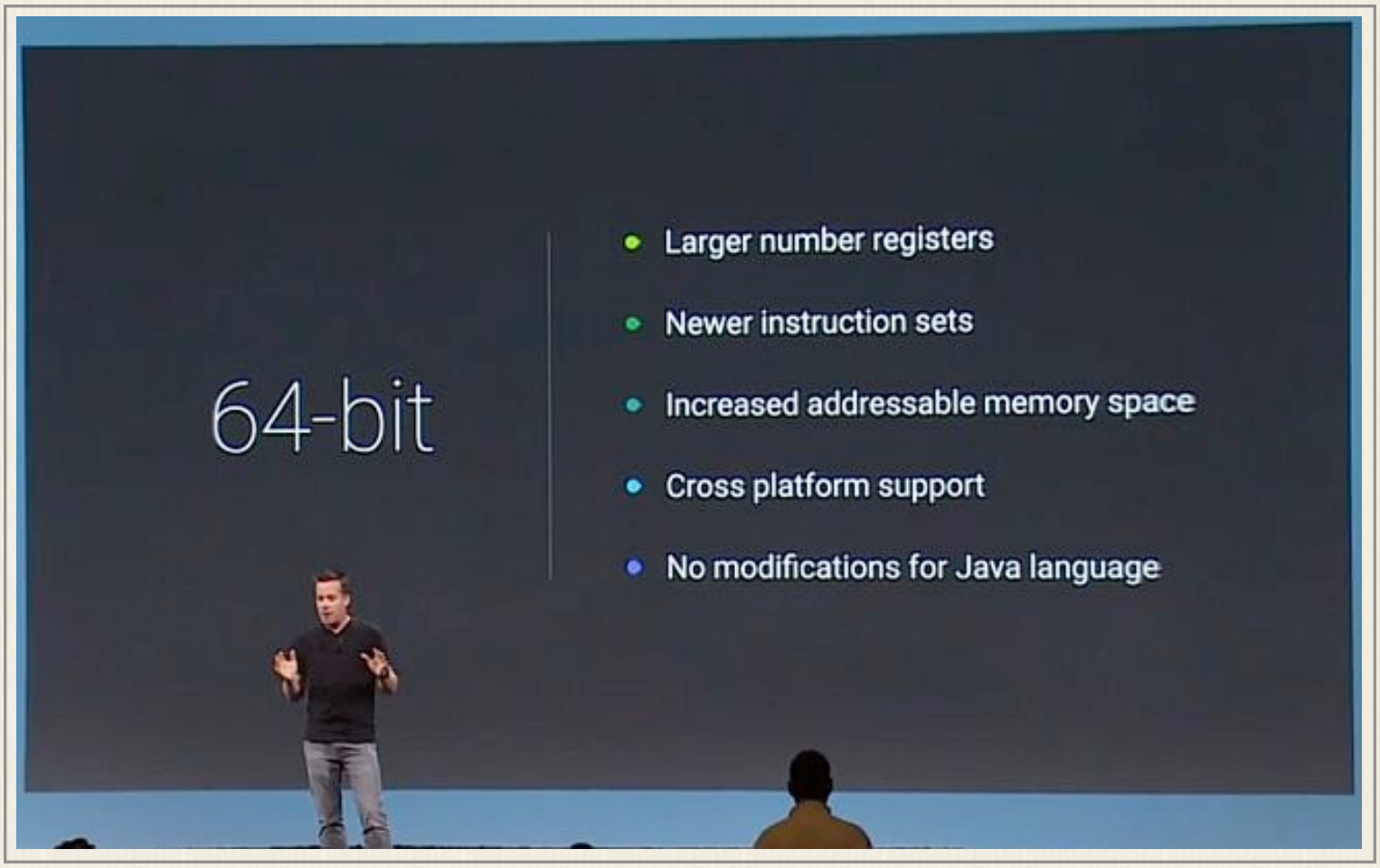
这个新的内存分配器，“rosalloc”(Runs-of-Slots-Allocator)是依据多线程Java应用程序的特点而设计的。此内 存分配器有更细粒度的锁机制，可以直接对独立的对象上锁，而非对整个待分配的内存空间上锁。在线程局部区域中的小对象的分配，完全可以无视锁的存在了。没 有了锁的请求和释放，线程局部小对象的访问速度也就大幅提升了。

这个新的内存分配器大幅提升了内存分配的速度，加速比达到了10x。

同时，ART的垃圾回收算法也做了改进，提升了用户使用体验，避免应用程序的卡顿。这些算法在Google内部目前仍然正在开发中。近 期，Google仅仅介绍了一个新算法，“Moving Garbage Collector”.核心思想是，当应用程序运行在后台时，将程序的堆空间做段合并操作。

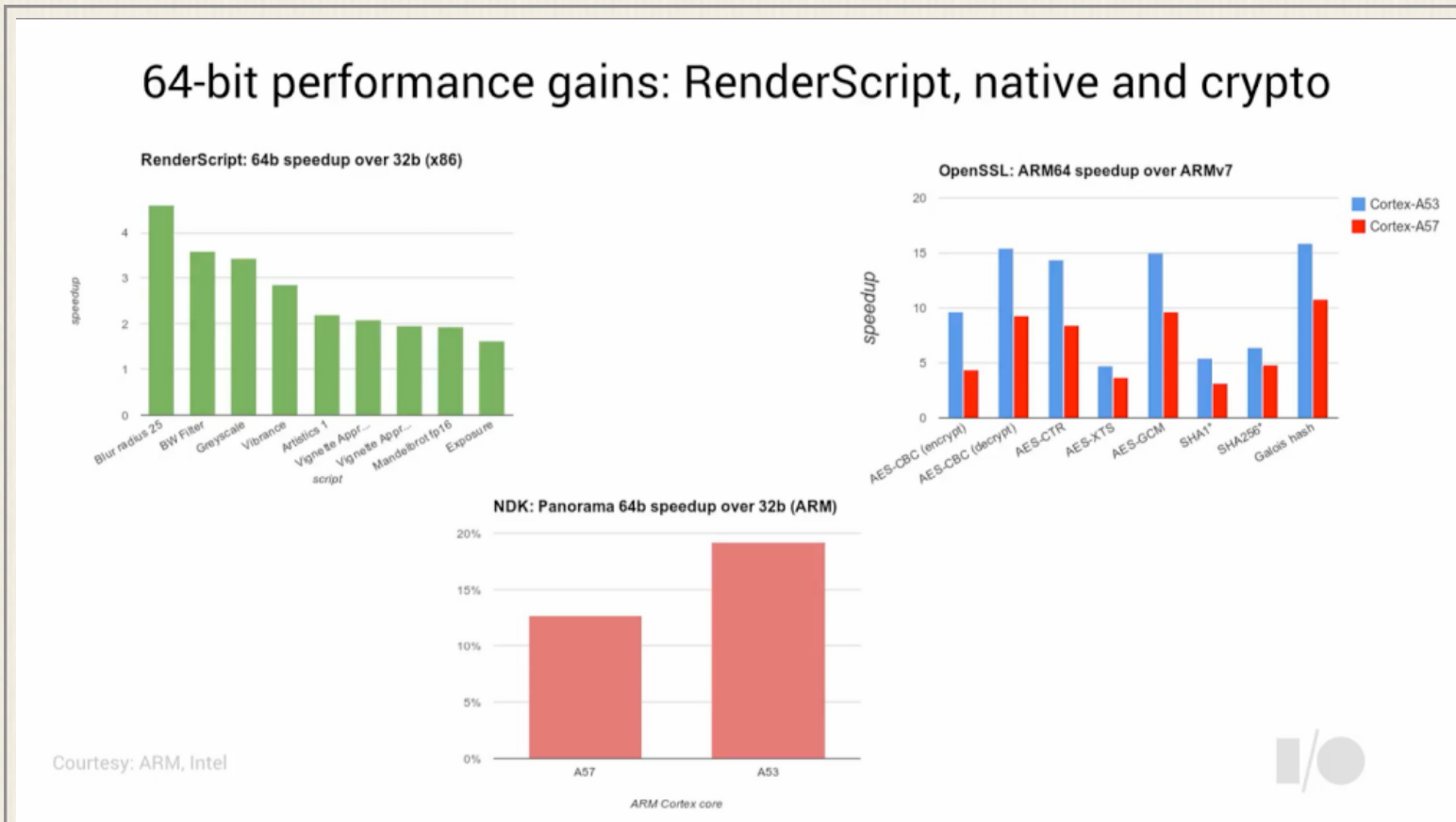
3 64位支持

ART在设计时充分考虑了将日后可能运行的各种平台进行模块化。因此，ART提供了大量的编译器后端，用于生成目前常见的体系结构的代码，例如ARM，X86和MIPS，其中包括对ARM64， X86-64 的支持，以及尚未实现的对MIPS64的支持。



对于ARM的64位系统带来的好处，相比很多朋友都了解了。更大的内存地址空间，普适的性能提升，以及加解密的能力和性能提升，此外还有对已有32位应用程序的兼容。

除此之外，Google还在ART中引入了引用压缩技术，来避免ART堆空间内部因为64位指针的引入导致的内存占用变大问题。其实，就是在执行时，所有的指针都采用32位表示，而非64位系统应该采用的64位指针。



Google 公开了一些ARM和X86平台上应用程序在64位和32位模式下的性能对比。这只是一些预览性质数据。X86的性能测试在Intel的BayTrail系 统上进行，对于不同的RenderScript测试程序，性能提升从2x到4.5x不等。ARM平台方面，分别在A57和A53系统上，对crypto的 性能做了对比。这些数据因为都是针对非常小的例子，所以代表性不大，因此还无法代表实际应用场景的情况。

不过，Google也放出了一些有趣的数据，这些数据是在他们内部使用的系统Panorama上测试的。通过简单的从32位ABI转换为64位 ABI，能够获得13%到19%的性能提升。还有个喜人的结论，那就是ARM的Cortex A53在AArch64模式下能获得性能提升比A57核要多。

Google还声称，目前应用商店中85%的应用程序都可以直接在64位模式下运行，也就是说仅有15%的应用程序在某种程度上使用了本地代码，需要重新为64位平台编译该应用程序。这对Google来说将是一个非常大的优

势。明年，当大多数芯片厂商都开始推64位片上系统的时候，从32位 Android系统到64位Android系统的的切换将会非常快。

4 结论

结合上面介绍的诸多方面，ART是Google发布的一款性能提升大杀器，并且ART也解决了多个数年来困扰Android系统的诸多问题。ART 有效地改进了多个解释执行应用程序面临的问题，也提供了一个自动化的高效的存储管理系统。对于开发者来说，许多过去需要手工添加代码解决的性能问题，现在 都能被ART轻松hold住了。

这也意味着Android系统终于能够在系统平滑度，应用程序性能方面与IOS势均力敌了。对消费者来说，是件喜大普奔的事情。

Google目前仍在，而且在未来一段时间内还将大力改进ART。ART目前的状况，与6个月前已经大不相同了，预计等到Android L真正发布的时候，又会有翻天覆地的变化。前途是光明的，让我们拭目以待，翘首期盼吧。

5 《灵犀志趣》注

• 最新Android ART的代码库可以在 https://android.googlesource.com/platform/art/+/master 找到。ART的代码里，能够看到不少LLVM的影子，见参考链接3.

• 本文大部分内容取材自参考链接1

6 参考

1. http://www.lingcc.com/2014/07/16/12599/www.anandtech.com/print/8231/a-closer-look-at-android-runtime-art-in-android-l
2. https://source.android.com/devices/tech/dalvik/art.html

3. https://android.googlesource.com/platform/art/+/master/compiler/llvm/

原文链接：http://www.lingcc.com/2014/07/16/12599/

利用QuincyKit + KSCrash构建自己的Crash Log收集与管理系统

作者：wonderffee

前言

我们知道，iOS bug定位是极看重crash log的，目前网上提供了不少crash log收集与管理服务，较有名的有Crashlytics, Flurry, 友盟，可能大部分人也就是使用这个。我这里要说的QuincyKit + KSCrash是一对开源组合，可能没有前者各种高大上的功能，基本功能还是有的，但更偏重于以下使用场合：

1）访问外网不太方便，或者大部分情况下在内网测试
2）对出现的crash问题要求快速响应，快速定位
3）需要自己掌控Crash Report，而不是交给别人

显而易见，第1条就足以把Crashlytics, Flurry, 友盟诸如此类的排除在外了；
关于第2条，我所知道的Flurry显示crash report延迟比较大，至少为6小时，Crashlytics稍微好一些，但是它们的服务器在国外，网页打开也比较慢。这里要额外说的是比较讨厌 Crashlytics在程序每次编译时都会上传app binary与dSYM文件，在网络情况较差或app比较大的情况下相当费时。 还有我经历的另一种情况，就是开发与测试人员相隔比较远，比如开发的在5楼，测试的在1楼，在最原始的阶段测试人员发现了崩溃问题，会将测试设备送到5楼 让开发人员用Mac解析，想想这效率，不言自明了吧。
第3点其实比较勉强，crash log又没多少机密可言。

QuincyKit

相信还是有一些人了解QuincyKit的，不过我看到相关的文章比较少。我之前主要参考的是Nico的《QuincyKit的crashReport框架》和《炒冷饭，再提一次QuincyKit》.

QuincyKit，简而言之，是一个为iOS和Mac OS X提供的程序崩溃报告管理解决方案。关于QuincyKit的介绍大家看Nico的文章就了解得差不多了，它对测试期的应用来说确实是很方便的，不需要提 前注册APP Key，不管你有多少个应用，App中集成了QuincyKit的Client端后，只管向Server端发送crash log就行，Server端会自动根据App ID来分类管理。

QuincyKit分Server端和Client端。它的Server端是用php编写的，用一个支持PHP5.2以上，还有Mysql、 Apache的服务器就可以搭建起一个完整的环境，Mac/Linux/Windows系统上应该都可以完成。看似简单，实际上对从没搭建过服务器环境的 初学者可能有点麻烦，不用担心，还有XAMPP这个神器来解决大部分麻烦。当初我图简单省事，就是用XAMPP来搭建基本环境的，不过相关的笔记找不到 了，这里就没办法贴上了。

另外要说明一下，QuincyKit Server端的Web管理界面比较简陋，比如连按时间排序的功能都没有，不过既然是开源的，就不能要求太高，会PHP的完全可以尝试为自己订制更高级的功能。

KSCrash

这里要说的是我只使用了QuincyKit的Server端，Client端我选择了KSCrash，可能有很多人对KSCrash对比较陌生，这也不奇怪，在国内就没见到有人介绍KSCrash。为什么是KSCrash呢，个中原因，且听我慢慢道来：

1）如果我没记错的话，QuincyKit client端生成的crash报告与原生crash报告相比总是缺少最关键的那一行，而KSCrash客户端生成的crash报告会把这关键的一行放在 最后一行，并提供一些额外的信息，非常有利于问题定位。大部分情况下，我只看这最后一行就能定位到问题所在。
2）QuincyKit在2013年初基本就停止更新了，而KSCrash目前仍旧是持续更新的，对于我们来说最重要是Client端的更新，比如考虑未 来支持Swift的可能性。而Server端主要是用来管理crash log，免费开源的QuincyKit足够对付使用。
3）KSCrash客户端生成的crash报告在大部分情况下都不需要dSym符号文件你就可以看到函数名，问题比较明显的话很快就能得到定位。但是默认显示的行号还是不对的，如果需要具体行号，还得利用dSym符号文件解析crash报告才行。（这点似乎QuincyKit客户端也支持的）

最关键是第1点，我想会不会有人因为这放弃使用了QuincyKit。

需要注意的是，KSCrash只是一个Client端，本身是没有Server端的。但这也是它的灵活之处，因为它能对接免费的 QuincyKit，收费的Hockey、Victory等Server端，也能将生成的crash log通过Email的方式发送。这只是KSCrash的特性之一，更多KSCrash的关键特性你可以看下面列举出来的，相信你能看到惊喜：

- 支持设备上解析与离线重新解析
- 生成的报告格式完全兼容iOS原生crash报告格式
- 支持32位和64位模式（实验性质）
- 能处理mach level下的错误，例如堆栈溢出
- 侦测未能捕捉的C++异常的真实原因
- 侦测访问僵尸(zombie or deallocated)对象的行为
- 恢复僵尸对象或内存覆盖情况下NSException的异常信息
- 提取对象异常调用的有效信息（比如发生unrecognized selector sent to instance 0xa26d9a0异常的情况）
- 支持多种Server端，提供方便的API接口
- 显示堆栈内容
- 诊断崩溃原因
- 记录比Apple原生crash报告更多的信息，使用JSON格式储存
- 支持显示用户自定义的额外数据

KSCrash能处理的crash分以下几种：

- Mach kernel 异常
- Fatal signals
- C++ 异常
- Objective-C异常

- 主线程死锁 (实验性质)
- 自定义崩溃（脚本语言）

经过我的测试，使用KSCrash目前主要有以下两个问题：
1) 在部分情况下会造成死锁（deadlock）crash，这通常是应用在做比较耗时的操作时出现的，如果发生这样的问题，你应该尽量优化你的APP，避免耗时操作（可能是我打开了死锁检测的功能，目前这个功能是Unstable Features）
2) 如果发生crash后重启应用时收集crash报告的服务器不可达，则之前的crash报告就会被废弃，即使重新恢复网络，仍然不会重新发送。

如果有人介意这些问题，可以尝试自行修改KSCrash的开源代码达到自己的目的。

需要说明的是，App发布时，还是建议大家使用Flurry、Crashlytics或者友盟，毕竟你面对的可能是数以万计甚至更多的用户。你可以 在你的应用中增加一个开关，测试期打开开关使用KSCrash上报crash log，发布前关闭开关，使用Crashlytics或其它在线crash report工具。万万要记得在自己的check-list里加上一条开关状态核对，以免忘记。

Crash Log自动解析

关于QuincyKit还有一个解析端，或者说是Mac端，因为crash log的解析是必须在Mac上进行的。要解析QuincyKit Server端收集到的crash log，就必须在解析端的Mac电脑上执行一个symbolicate.php脚本。它做的事情实际上就是将QuincyKit上未解析过的crash log下载到本地，批量进行解析，然后再批量上传回去。你可以启动一个定时任务定时去执行这个脚本，就不用每次都手动执行了。

但是有一个问题，如果你想每次都能解析成功，你的Mac上需要提前拥有与crash log对应的.app文件、.app.dSYM文件的索引。这个索引可以通过mdimport命令来实现（mdimport的介绍可参考这里）。不过肯定还是有人觉得手动执行mdimport命令进行索引挺麻烦的，最好的办法还是用脚本将各个流程串通起来自动化实现。

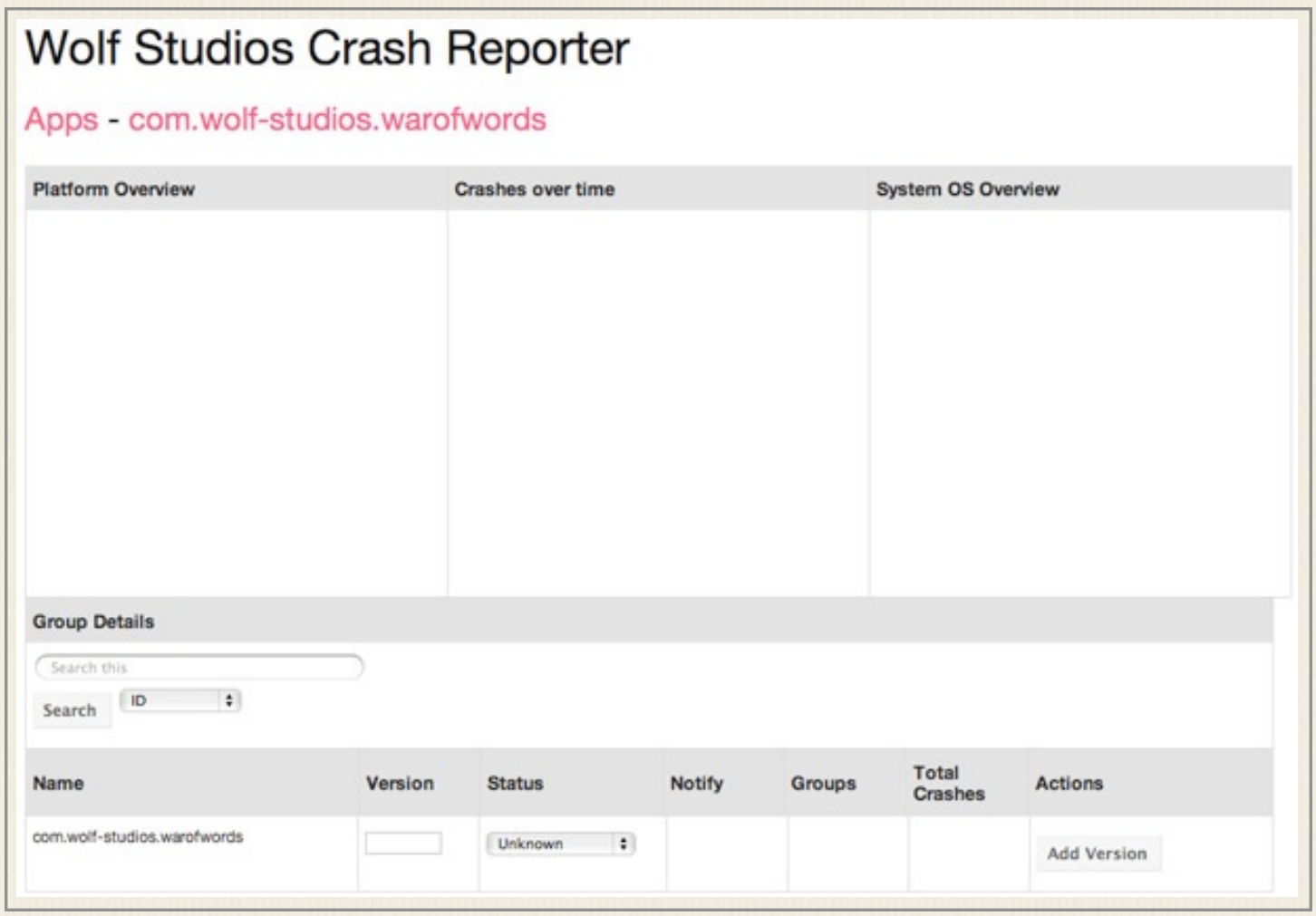
我能想到的一个自动化流程是：
(1) 自动构建版本，生成ipa文件和dSYM.zip文件
(2) 解析端通过脚本拿到ipa文件和dSYM.zip文件，然后copy到指定文件夹，解压，执行mdimport
(3) 在解析端的Mac电脑上开启一个定时任务，定时执行symbolicate.php

经历这3步，就可以保证你在QuincyKit web网页上永远看到的是解析后的crash log。

做到了这一切，如果你经历过手动执行symbolicate-crash命令来解析crash log的阶段，就知道一个是天上，一个是地下了。

后记

原谅我没有贴图，因为我都是在公司里搭的系统，家里没有。在网上就找到一张QuincyKit的web管理界面，供参考



搜索资料时还有意外发现，《Doutzen: Local symbolication for QuincyKit》一文的作者做了一个Mac应用来完成解析端的工作，将配置简

单化，并实现了定时自动解析。不过根据评论，应该是不兼容Lion系统，估计更不会兼容最新的10.9/10.10了，好在代码是开源的，会Mac开发的稍作修改就可以拿来为己所用了。

还有人用Rails写了一个类QuincyKit的Server端，提供在线服务网站holdbug.com，但是代码好像不开源，网站也打不开了，估计是停止支持了，链接见 http://ju.outofmemory.cn/entry/37054

原文链接：http://wonderffee.github.io/blog/2014/07/19/quincykit-and-kscrash/

冯森林：手机淘宝中的那些Web技术

作者：郭蕾

Native APP与Web APP的技术融合已经逐渐成为一种趋势，使用标准的Web技术来开发应用中的某些功能，不仅可以降低开发成本，同时还可以方便的进行功能迭代更新。但是如 何保证Web APP的流畅性也一直是业内讨论的热点。InfoQ此次专访了手机淘宝客户端高级技术专家冯森林来谈谈手机淘宝在Web技术方面的一些实践经验，另外作为 ArchSummit深圳2014大会《移动互联网，一浪高过一浪》专题的讲师，冯森林将会分享手机淘宝的客户端架构探索之路。

InfoQ：**淘宝手机客户端是否使用了HTML5技术？能简单介绍下吗？**

冯森林：手机淘宝大量使用了Web技术，但对于HTML5，由于兼容性的要求，我们相对比较保守，使用到的 特性并不多。主要是一些与触屏体验相关的HTML5特性，大部分运用在基础JS库中，业务开发直接使用的场景不多。优点在于可以更好的支持一些优化的体 验，充分发挥新技术优势和移动端独有的能力。缺点也很明显，兼容性上需要考虑较多的适配问题。

InfoQ：**我们知道Web页面与原生的页面在性能上还是有很大差距的，手机淘宝是如何处理Web页面的体验瓶颈的?**

冯森林：通过使用缓存技术，可以在再次加载页面（及用到的资源）时避免缓慢和不可靠的网络请求，从本地缓存 加载基本可以做到即时完成，接近于原生应用的加载体验。为了解决首次加载，我们在缓存机制的基础上引入了预缓存机制（采用与AppCache一致的协 议），提前将需要的页面及资源缓存到本地，即使在用户首次打开时，也相当于打开已经缓存过的页面。

InfoQ：**在Web页面中不可避免的会调用一些Native 的功能，手机淘宝是如何实现的?**

冯森林：我们采取了类似于PhoneGap的实现（但更轻量级），在Android和iOS两个平台上分别 实现了JsBridge，在JavaScript的命名空间内创建可映射到native对象的代理。并在业界通行的实现基础上，我们还加入了一些安全增强 和保护机制，封堵常见的JS注入漏洞。

InfoQ：看来手机淘宝在HTML页面方面做了大量的技术投入，能分享下你们的经验吗?

冯森林：其实，无论是缓存还是网络方面的优化，都是在传统Web开发领域内已经相对成熟的实践。在App 内，由于我们所能掌控的部分大幅度的下移，能影响一部分以往受制于浏览器实现的技术层次，所以可以在这两方面做的更多更深入。但是兼容Web的既有标准和 实践是我们做这些优化的基本前提，比如使用AppCache协议支持预缓存，这样有助于前端技术和实践的跨平台兼容和复用。

InfoQ：淘宝自己实现的WebView 缓存机制模块同时兼容iOS 和 Android吗？ 能否具体讲一下大概的实现思路？图片缓存有特殊处理吗?

冯森林：是的，在两个平台上，我们都采用了相似的实现。实现思路上完全参照标准的HTTP Cache-Control协议，在一些特定的场景下使用ETag。图片与API采用一致的缓存实现，唯服务端的缓存策略配置不同而已。

InfoQ：手机淘宝开发Web页面时有没有用到过一些开源框架？如果有自研框架，是否考虑开源?

冯森林：手机淘宝中使用到的大部分是前端业界成熟的开源框架，如JQuery、Mustache，也有一些我们自己构建的框架，如已经开源的Kissy。基本上，除了对接私有设计的框架之外，我们都优先考虑使用成熟的开源项目，并且将我们的补充反馈给开源社区。

InfoQ：淘宝对安全性要求很高，请问在Web与Native交互的过程中，是否进行过一些加密和其它的处理?

冯森林：Web与Native的交互，本身还是受到OS安全性保护的。无论在Android还是iOS下， 这都是App内部的通信通道，因此在非越狱/ROOT的设备上不存在已知的安全风险。由于越狱和ROOT在国内环境中的普遍性，所以在安全问题上，我们整 体性的策略是将整个客户端视同安全藩篱较低

的终端，从云端通信及行为分析上去强化安全保护。比如我们已经在Web容器中加入的『人机识别』模块，可有效识 别各种利用Web页面和接口进行的自动『刷XX』行为。

原文链接：http://www.infoq.com/cn/news/2014/07/mobile-taobao-web-technology