

# 架构师

ARCHITECT



## 热点 | Hot

[Apple Watch应用开发总结](#)

[开发者必须关注的微软技术热点](#)

## 推荐文章 | Article

[Java字节码忍者禁术](#)

[高效运维最佳实践（03）](#)

## 专题 | Topic

[深入浅出Mesos](#)

[Netty编解码框架分析](#)

## 观点 | Opinion

[赵海平：开源是个兴趣活儿](#)



# ArchSummit

International Architect Summit

全球架构师峰会 2015

2015.7.17-18 中国·深圳·大梅沙京基海湾大酒店

## 侧重业务场景，引领技术趋势



### HOT

9大热门技术专题

- |         |            |
|---------|------------|
| 研发体系构建  | 电商和零售业的转型  |
| 在线教育    | 移动化机会      |
| 智能硬件    | 大数据背后的价值   |
| 开源与企业发展 | 企业云化的痛点与实践 |
| 互联网金融   |            |

### INVITE

邀请制闭门会议

- |            |
|------------|
| 在线教育机遇与挑战  |
| 开源与企业发展    |
| 企业云化的痛点与实践 |
| 互联网基因的金融   |



议题提交开放，折扣购票启动，详情查阅大会官网

购票热线：010-89880682 会务咨询：arch@cn.infoq.com 大会官网：[www.archsummit.com](http://www.archsummit.com)



北京·2015年5月26日

上海·2015年6月11日



PWorld  
软件架构 & 平台创新大会

# 卷首语

Tim Yang 老师前两天在他的微信公众号 timyang\_net 上更新了一篇名叫《软件的体验障碍与解决之道》的文章。标题有点讳莫如深，所以我一开始没敢进去看；结果 Tim Yang 老师把这篇文章发了朋友圈，号召小伙伴们都去看看捐个茶叶蛋否则过两周才写下一篇（让你们不点赞，等死你们！），于是乖乖的进去仔细拜读了好几遍。

以我浅薄的理解，该文章的前半部分问了一个问题：用户使用某些云同步服务，该服务因为某些莫名其妙的网络问题而造成频繁的同步失败。尝试解决此类问题，将对工程师极不友好，因为将不得不引入非标准技术方案。而放任此类问题，将对这些受影响的“一小撮”用户极不友好。那么，到底这个问题值不值得解决？

文章的后半部分没有直接提供答案，只是说从架构师的角度，自己反对引入非标准技术方案。

这个问题值不值得解决，我是想不出来。不过文中对“工程师体验”和“用户体验”的论述，是个很有意思的话题，因为从某种层面而言（以前的我一直没有意识到），“工程师体验”和“用户体验”是会出现冲突的！而且是经常！

不禁想起上个月 QCon 北京的时候听阿里云的献涛同学分享他们做 Xen 上的内核 hotfix 方案，该方案可以说是用户体验优先的一个典型代表。Xen 在 3 月份爆了一个高危漏洞 XSA-123，大意是你的云服务如果是跑在 Xen 上，那么你的客户机们说不定能够读取到其他客户机们的数据，所以必须修复。目前，内核本身的 hotfix 技术虽然已经相对成熟，但是要想在 hypervisor 上操作却还没有标准的做法，所以从“技术标准化”的角度，只能乖乖的给所有服务器打冷补丁再重启。这无疑会伤害用户体验。

对于用户至上的阿里同学们来说，这能忍吗？不能忍！于是“非标准技术方案”就势在必行了。该方案基于一个内存覆写的原则，一个非常规的实现方式——人工截获 DMA 请求，修改之，以在原本没权限访问的 hypervisor 内存空间实现写入操作。修复思路像极了打电脑游戏用动态修改器作弊的思路，但是细节操作的难度更大，如果没有吃透 Xen 的代码是不敢随便玩的。

这是一次成功的 hack，达成了在没有影响用户体验的前提下给全部集群完成修复的目的。

单独看这次修复，一切都很好。但对于一个庞大的线上系统而言，如果每天都冒出 n 多这种需要用“非标准技术方案”解决的问题，可就大大的不好玩。但凡是有点技术洁癖的工程师，如果他在公司的每一天都在处理这样的需求，恐怕会发疯。

不管用户体验一定是不对的。但，不计代价的“用户体验至上”，大概也是一个误区。“用户体验至上”当作宣传口号还行，如果在具体面对每一个工单需求时把它当作权衡原则，注定会造成大量投入产出比低下的疲于奔命。不过对于这个问题，也有众多业界前辈们提出一些量化的权衡方法，比如有个叫做 T-shirt size estimation 的方法，就是给研发经理用来跟产品经理“拉扯”用的利器。

从工程的角度，有一些事情虽然不利于用户体验，但也要坚持，就好像建筑工地一定要坚持工地安全准则一样重要。其实这也算是“现在”与“未来”之间的权衡吧。

—— 杨赛

# 目录

## 热点 | Hot

Apple Watch 两个月开发的一些收获总结 ..... 6

开发者必须关注的微软技术热点——Build2015 大会综述 ..... 18

Docker 发布新的网络项目，并开始招聘中国区主管 ..... 21

## 专题 | Topic

Netty 系列之 Netty 编解码框架分析 ..... 22

深入浅出 Mesos (一): 为软件定义数据中心而生的操作系统 ..... 46

深入浅出 Mesos (二): Mesos 的体系结构和工作流 ..... 49

微博“异地多活”部署经验谈 ..... 54

## 推荐文章 | Article

Java 字节码忍者禁术 ..... 60

专访 ThoughtWorks 王磊：从单块架构到微服务架构 ..... 74

高效运维最佳实践 (03): Redis 集群技术及 Codis 实践 ..... 79

## 观点 | Opinion

赵海平：开源是个兴趣活儿 ..... 87

性能优化：一个全栈问题 ..... 90

手动测试是否可以退出测试舞台？ ..... 92

封面植物 ..... 93

# Apple Watch 两个月开发的一些收获总结

作者 刘瑞

Apple Watch 即将于 4 月下旬发售，而 Watch App 的开发已成为 iOS 开发的热点。本文作者通过 Watch App 的实际开发经验，将其中的一些注意事项总结分享给大家。以下为正文：

接触 Apple Watch 相关的开发工作已经差不多快三个月时间了，每天都会去逛逛 WatchKit 苹果的开发者论坛，看看最近都有哪些其他开发者 po 出来的问题。我自己也遇到不少问题，其中很多都是我自己摸索着解决掉的。

苹果公布的关于 Apple Watch 的信息很多，用于开发已经足够，但一切感觉都是在抹黑前行，因为无法进行真机测试，包括 Handoff，也包括语音输入，以及发布会上的那个类似 Emoji 的表情都是些什么。

自己来现在的公司实习到今，主要做的工作几乎都和 iOS8 新特性有关，毕竟现在公司这个项目实在是太成熟了，摸熟悉也需要一个过程。包括之前的 Today Widget，到后来的 Handoff，包括因为要适配 iPhone6 做的适配方面的调研等等，都是从去年 WWDC 之后的新事物，转眼就到 2015 年的 WWDC 了，不知道今年会有哪些革新的新事物。

闲话说到这里吧，是时候总结一下这两个月的收获和掉坑了。

目前开发者网站上的这几部分我觉得是开发 Watch 必须学习几遍的东西，还有苹果开发者论坛也是一个不错的交流地方。

[WatchKit Framework Reference](#)

[WatchKit Development Tips : Optimize your WatchKit apps with these tips and best practices.](#)

[Apple Watch Programming Guide](#)

[Developer Forum](#)

## 1. Watch Main App

在 iPhone 上，主程序是大哥，其他的小扩展必须让路，但是在 Watch 上，是不是大哥还要看这个 APP 主要的功能。如果是一个阅读性质的 APP，主程序在手表上作用还真不大，例如阅读新闻等等。如果是这类的应用，想在 Watch 上出彩，或者让用户使用的次数多一些，就要靠良好的 Notification 体验，以及极其方便用户生活的 Glance 了。

## (1) 以 Page-Based 方式启动 Watch App



如上图，现在手上要做的一个交互是，App 启动的时候是六个页面，用户可以左右滑动来切换，这里就需要在 MainInterfaceController 中使用下边这个方法了。

```
[WKInterfaceController reloadRootControllersWithNames:  
_controllersArrays contexts:_contextsArray];
```

在 Watch 上页面之间转换传值，很重要的一个纽带就是这个 context，传递有用的信息和标识，这个方法中，我传递进入六个 controller 的 interface builder identifier，以及事前拼好的六个 context。

因为 Watch App 的打开可以是几种不同方式的，可以写一个统一的方法 [self showController]，在这个方法中去选择启动哪一个具体的 Controller。我在.h 文件中定义了一个枚举来定义不同的启动方式：

```
typedef enum {  
  
    WKOpenForNormal,      //普通打开  
  
    WKOpenForComment,     //打开评论页  
  
    WKOpenForFavorite,    //打开收藏页
```

```
WKOpenForGlance //打开来自 glance 的内容

} WKOpenType;
```

因为用户如果选择了点击 Glance 来查看具体的内容的话，Glance 和 MainApp 是通过 Handoff 来实现通信的，我们可以在入口的控制器中的：

```
- (void)handleUserActivity:(NSDictionary *)userInfo;
```

这个方法中去将 WKOpenType 赋值成 WKOpenForGlance。

当然了，如果是从 **Notification** 来的，我们完全可以通过：

```
- (void)handleActionWithIdentifier:(NSString *)
identifier forRemoteNotification:(NSDictionary *)remoteNotification;
```

这个方法来根据具体的用户点击的动作来区分不同的打开方式。

这里比较难处理的是，如果用户是从 Glance 进来的，退出这个控制器，还是要显示那六个页面的，这里我的解决方法是注册通知。在出来的控制器中的- (void)didDeactivate; 方法中 post 出来通知，来让主控制器重新打开六个 Page 页面。Notification 同 Glance。

## (2) Watch App 与 Host App 联合调试

因为程序中多处用到了下边这个方法，因此主程序和 Watch App 联合调试就显得非常必要了，在 Xcode 的一个新 beta 的 release note 中苹果介绍了一种方法。

```
+ (BOOL)openParentApplication:(NSDictionary *)userInfo reply:
(void(^)(NSDictionary *replyInfo, NSError *error)) reply;
```

首先 run 起来 Apple Watch App 在模拟器中。

在 iphone 模拟器中启动 demo App。

Xcode - Debug - Attach to Process 里找到 host app 线程，Attach 上。

完成以上三个步骤，主程序和手表程序上的端点都可以进行调试。

## (3) 申请数据方面

在开发初期，我是在 extension 中进行数据的申请，这样尝试了一段时间之后发现性能上优化的空间不大，而且写出了很多重复的代码。复用项目中已有的代码是我最好的选择，

尤其是一些第三方用 pod 管理的库，但是考虑到公司的项目已经是非常成熟的了，一些管理的第三方库无法正常的使用，进而又去考虑写一个共用的框架，由于时间问题，项目有点大，抽筋抽骨的不是很合适，所以决定充分发挥 openParent 这个方法，将申请数据这块放在主程序中，顺便将所有需要“问”主程序的东西全部整理到一个类中，这样就可以充分发挥老代码的作用。

数据策略大致如下：首先为了优化 Watch App 的启动速度，采用后台申请数据存起来，Watch 每次去使用就可以了，最后处理一下冷启动的问题，这种情况是当安装了我们的软件，没有在 iPhone 上打开过，直接打开 Watch 上的程序的时候已然有数据，这么做的话除了第一次会启动的稍微慢一点点之外，剩下的启动速度就会快很多。

具体用到的方法是：

```
- (void)application:(UIApplication *)application
performFetchWithCompletionHandler:
(void (^)(UIBackgroundFetchResult result))completionHandler
NS_AVAILABLE_IOS(7_0);
```

我和同事做到这里的时候，就感觉是一个 iPhone 当做了服务器，而 Watch 则是一个终端，有什么需要的数据，我们两个人设计好协议，通过 openparent 这个方法沟通。比如说，软件运行当中如果想要知道一个用户是否登录了，因为没有登录是没有某些功能的，那么这个时候通过 openparent 咨询一下 isLogin 就好，判断一下是否登录。

Demo 中 watch 端代码实现如下：

```
[WKInterfaceController openParentApplication:@{@"type":@@"isLogin"}]

reply:^(NSDictionary *replyInfo, NSError *error) {}
```

Demo 中 iphone 服务端代码实现如下：

```
#pragma mark - WatchKit Data

-(void)application:(UIApplication *)application
handleWatchKitExtensionRequest:(NSDictionary *)userInfo reply:(void (^)(NSDictionary *))reply
{
    NSString *type = userInfo[@"type"];
    NSDictionary *para = userInfo[@"para"];
```

```
NSDictionary *replyInfo;

if ([type isEqualToString:@"isLogin"]) {

    int random = arc4random()%10 + 1;

    NSString *whetherLogin = @"";

    if (random == 1) {

        whetherLogin = @"YES";

    } else

    {

        whetherLogin =@"NO";

    }

    replyInfo = @{@"whetherLogin":whetherLogin};

}

else if ([type isEqualToString:@"isFavorite"])

{

    reply(replyInfo);

}
```

Demo 中有三种协议，分别是是否登录，回复信息，是否收藏，当然都是假的，根据项目需求来进行改变，务必注意的是每一种情况都要回调 `reply(replyInfo);`，否则这个方法实际上会响应失败。

而实际上，项目当中需要在 Watch 上显示很多图片的，这个就需要异步的申请一下，首先要想到的还是 SDWebImage 这个经典框架，这里就可以在 `openParent` 里使用将 `data` 请求到，然后返回给 Watch。

PS：最后的最后，我们发现使用 App Group 来通信数据更加的有效率，因此一部分数据的请求采用了 App Group 来实现。

#### (4) TableView 在 Watch 上的使用

在 SDK 发布的初期,我以为新控件之一 WKInterfaceGroup 可以点击,因为目前来看 watch 上是没有图层的概念的,复杂的 UI 布局是相当困难的,布局方式和之前有很大的区别,包括在故事板中的布局方法。当初为了实现产品给过来的 UI 布局也是脑洞大开啊,比如各种嵌套 Group,为了要实现 demo 中主页的这种感觉,我很自然的想到了,放一个 group,背景放图片,其他控件放在 group 上就好了,解决了无法实现控件在控件之上问题。但是这就需要 group 可以点击,盼星星盼月亮之后, Xcode6.2 正式版出来之后彻底断了我这个念头,没办法,只能通过另一个控件 WKInterfaceTable 来实现了,每一页只有一行不就可以了么,只能这么干了。

WKInterfaceTable 和 UITableView 使用上还是有一些不同的,也比 UITableView 的使用方便了很多。

首先你需要去定义一个 Row 类,这个 Row 类相当于一个 cell,在这个 Row 上去布局,如果你的表格中呈现数据的方式不一样,那就要定义不同的 Row 类。

定义好之后,调用的时候需要使用如下方法:

```
#pragma mark - UI

- (void)setUpUI

{

    [self.newsRowTabel setNumberOfRows:1 withRowType:@"RowForOneNews"];

    for (int i = 0; i < self.newsRowTabel.numberOfRows; i++) {

        JRWKNewsRow *newsRow = [self.newsRowTabel rowControllerAtIndex:i];

        [newsRow.newsCategory setText:[NSString stringWithFormat:@"第 %ld 张", _index+1]];

    }

}
```

RowType 唯一标识了一个 Row 类,这里我设置了只有一行,期间设置 Row 类中每一个属性的 UI 数据。

响应点击事件需要去实现:

```
#pragma mark - Table Row Select

-(void)table:(WKInterfaceTable *)table didSelectRowAtIndex:(NSInteger)rowIndex
```

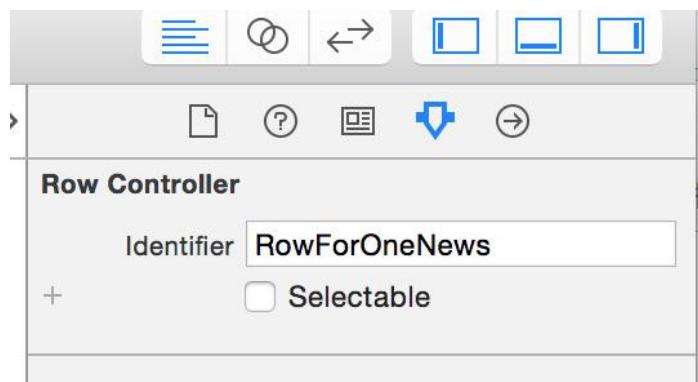
```
{
    NSDictionary *contextDic = @{@"PicName":_picName,@"index":[NSNumber
numberWithInteger:_index]};

    [self presentControllerWithName:WKNEWSDETAILCONTROLLERIDENTIFIER
context:contextDic];

}
```

这里去指定具体要呈现出来的是哪一个 Controller。

如果表格中的一行不能点击的话，在故事板中设定的时候把 selectable 勾选掉就可以了。



## (5) 数据在 Controller 间的传递

API 中的几个关于 Controller 切换的方法当中几乎都有 context 参数，也就是说传递数据由我们决定了。在十二月份刚刚开始写程序的时候，我传递的是一个很大的字典，发现在程序启动的时候非常的慢，后来决定写一个模型管理类，controller 之间只需要传递一个 index 就可以了。在 demo 中保留了完整的类。

## (6) 关于 HandOff

HandOff 在 iOS8 之后出现，着实是为了 Apple Watch 量身打造的好么，实在是太应景了，因此在 Watch 上合理的运用 handoff 是一个顺理成章的事情，而 WKInterfaceController 也带上了相关的一些方法，实际上是要比 iphone 上的简单易用一些的。

另一方面，在 Glance 界面，进入到主 App 上的时候，handoff 也起了决定性的作用，通过 handoff 将具体的信息交给主 App 去处理。

主要有两个 Api，这个是 update 了全局的 Activity，将我们需要传递的信息打包成一个 userinfo 即可。

```
- (void)updateUserActivity:(NSString *)type userInfo:  
(NSDictionary *)userInfo webpageURL:(NSURL *)webpageURL;
```

下面这个我还记得是开发者 watchkit 论坛里有一位开发者问过这个问题，在 watchkit 里怎么没有干掉 Activity 这一个方法。后来苹果的工程师估计是采纳了。但实际的效果来看，这个方法作用不大，例如在公司的项目中，几乎每一个页面都是需要 handoff 的，给它 invalidate 之后，iphone 左下角出现 logo 就会出现异常甚至是不出现的情况。因此如果不是已经很明确的话，轻易的不要用这个方法。

```
- (void)invalidateUserActivity;
```

总之，Handoff 是 Watch 和 iPhone 沟通的绝佳方式之一，苹果也一直很鼓励使用 SDK 新出的一些东西来补充自己的 App 的。不要再幻想（至少是现在）通过 Watch 上的一个按钮能够使得 iPhone 上的 Host App 能够打开并且显示在前台了。

## (7) 其他一些 Tips

(a).dynamic notification 中苹果是希望用户在通知中就把所有的信息都看完的，而不希望用户点击内容本身（实际上也是不能点击的）再进入到 Watch app 内查看这个通知的内容的，恰恰相反的是，glance 的交互理念是相反的，也就是苹果估计用户点击 glance 页面本身（实际上是可以点击的）进入到 Watch app 中进行继续深度阅读的。

(b).关于 WKTextInputMode，一开始选择的是 WKTextInputModeAllowAnimatedEmoji，后来发现这个是动态的大表情，返回的是这个大表情的 data，不太适合我们一一对应到 iphone 上的 emoji 表情，于是后来切换到了 WKTextInputModeAllowEmoji。而 WKTextInputModePlain 只是显示了我们所“推荐的”那些回复文本选项。

```
typedef NS_ENUM(NSInteger, WKTextInputMode) {  
  
    WKTextInputModePlain,  
  
    // text (no emoji) from dictation + suggestions  
  
    WKTextInputModeAllowEmoji,  
  
    // text plus non-animated emoji from dictation + suggestions  
  
    WKTextInputModeAllowAnimatedEmoji,  
  
    // all text, animated emoji (GIF data)  
  
};
```

(c).- (void)becomeCurrentPage; 这个方法主要是在 page based 页面当中，如果第三页在启动的时候你想让它先出来，就要标识好，在 awake 里边获取到之后，调用这个方法，注意的是，这个第三页不是立马就出现在手表的表盘之上的，而是从第一页蹦到第二页，然后再第三页这样转的。

(d).推荐一个很好用的工具，叫做 Bezel,它能够将模拟器中运行的 watch app 映射到真实的手表里，表带的样式也分 38mm 以及 42mm，有很多种，可以更好的查看自己的 App 在真实手表上的样子。更换表带也很方便，直接拖着下边的某一个样式到 Bezel 上就自动换了。举个例子，在开发的时候曾想左右留边，但是放在 Bezel 上就会发现手表自带黑边，于是留下的左右边就是很多余了。

[Bezel 下载地址，页面内包含 N 多种表带](#)



## 2.Notification

从目前来看，手表上出现 push 应该是随着手机一起来的，也就是同时去显示在这两个设备上，除非一切外力因素，比如手表关闭了抬手查看通知等。在之前的 blog 中提到过定义 category 来区分推送通知，如果没有定义 category 的故事板的话，就会在手表上显示一个系统默认的简短的通知。上边说道，苹果还是鼓励在 notification 中将该阅读的内容都阅读完，即使增加按钮也要是一些比较简单的操作，比如说一个日程安排的软件，来了一个 push，一个 done，一个 delete，加上系统的 cancel，就可以了。

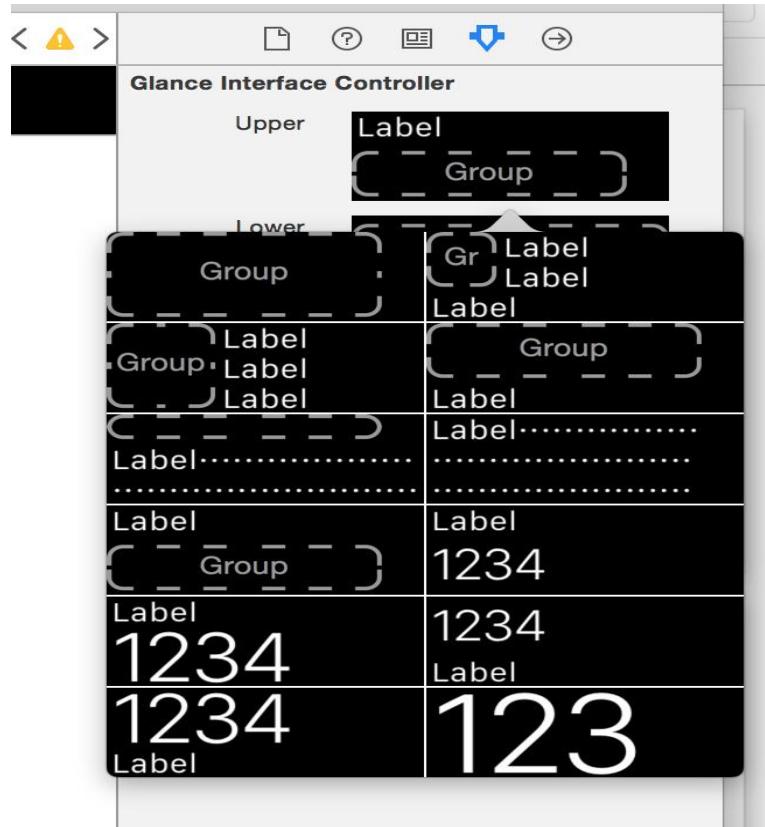
我尝试了在 Dynamic notification 中申请了一个图片资源，发现系统就选择去显示 Static notification,因此在 notification controller 内进行的任务的能力有限，这个在开发的时候要慎重。

开发的时候，Xcode 自动生成的 Payload 很重要，可以定义多个 payload 来进行相应的模拟，搭配不同的 category，不同的 category 故事板。

## 3.Glance

我依然认为 Glance 的地位在 Watch 上是最重要的，至少在第三方独立 app 登上 Watch 前，

Glance 应该是用户使用最频繁的一个功能。因此 Glance 上要呈现的东西不能太少，也不能太多，一定要简明扼要，要呈现出最重要的一些东西。比如说如果自己的 App 不是以天气为主的，放一个天气温度什么的就不是很合适，系统的天气和地图软件还是非常出色的，因此还是在 Glance 只体现自己 App 里边独特的东西最好。



另外，Glance 的 UI 布局是很讲究的，如果可以尽量要按照 Xcode 给的 Upper 和 Lower 的模板进行 UI 布局。不能使用任何可以操作的空间，例如按钮这样的，因为 Glance 就一页（可以滚动也是禁止的），有点像是渲染出来的一张图片似的，因此加个按钮是没有意义的。

同 Notification，Glance controller 中进行任务的能力也比较有限，因为众多的 Glance 会一同呈现出来，用户翻腾着每一个 app 的 Glance，这就要求用户一扫之后就要呈现出来，一个比较好的解决方法就是 Glance 要呈现的数据提前的申请好，用的时候拿出来，具体实现的方法也有很多。比如上边提到的 App Group。

Glance 以及主 App 的通信是依靠 Handoff 来实现的，也就是说用户点击了 Glance 这个页面之后，进入到主 App，要做的事情需要根据传过来的 userinfo 来决定的，主要就是下边这个方法。

```
[self updateUserActivity:XXXXXX userInfo:userInfo webpageURL:nil];
```

在入口 controller 中实现方法，决定启动什么页面，呈现什么内容，可以放在 willActivate 里边。记住的是请求数据这块一定要放在 awake 里边，不要放在 willActivate 里边。

```
- (void)handleUserActivity:(NSDictionary *)userInfo  
  
{  
  
    wkOpenType = JRWKOpenForGlancedemo;  
  
    if (userInfo) {  
  
        NSString *sourceString = [userInfo objectForKey:@"Source"];  
  
        NSString *picName = [userInfo objectForKey:@"PicName"];  
  
        if ([sourceString isEqualToString:@"Glance"]) {  
  
            _glancePicName = picName;  
  
        }  
  
    }  
  
}
```

根据 wkopentype 决定启动页面。

```
switch (wkOpenType) {  
  
    case JRWKOpenForGlancedemo:  
  
        //glance page  
  
        break;  
  
    case JRWKOpenForNotificationdemo:  
  
        //notification page  
  
        break;  
  
    default:  
  
        [self showPageBaseddemoController];  
  
        //默认启动
```

```
        break;  
  
    }
```

Glance 在 demo 中的表现形式，demo 已经整理好，放在了自己的 Github 上。

### [AppleWatchDemo](#)



## 4.总结

其实 WatchKit 的东西真不多，更多的是在一个新的平台遇到的各种问题和 bug 是最让人头疼的。随着真机的即将到来，开发工作也不再是抹黑前行，这些都是利好的消息。不知道什么时候可以有独立的第三方应用的支持，也不知道 WatchKit 会丰满到什么程度，总之我个人还是很看好 Watch 的未来的，毕竟苹果引领的穿戴设备的头。

## 作者信息

刘瑞，中国科学技术大学苏州研究院在读硕士，喜欢科技产品，也喜欢制作开箱、体验视频。大三起开始自学 iOS 开发。

# 开发者必须关注的微软技术热点 ——Build2015 大会综述

作者 崔康

一年一度的[微软 Build 大会](#)在美国旧金山如期举行，当地的天气有点阴冷，不过参会者的热情依然充满了整个 Moscone 会议中心。

Build 大会是微软面向开发者社区举办的重要会议，虽然过去几十年经历了名称、形式等方面的变化，但是依然保留了下来，可见微软对开发者的重视程度。正如 CEO Satya Nadella 在开场致辞中提到的，刚过完 40 岁生日的微软是一家由开发者创建的、并为开发者服务的公司，在 IT 发展的新潮流下，微软致力于成为一家“平台式”公司。对于广大开发者来说，Build2015 大会有哪些技术热点？我们应该如何评价呢，InfoQ 中国进行了系统的梳理，供读者借鉴。

## 云计算

热点关键字：Docker、机器学习支持、数据管理新工具

和前几届 Build 大会不同，关于云计算平台的介绍被安排在头一天的主题演讲中，由此可见其重要地位。目前微软的 Azure 已经在全球部署了 19 个 Region，超过了亚马逊和谷歌的 Region 总数；过去 12 个月发布了超过 500 个新特性；现在每个月还在以 9 万个新客户的速度递增，在这样一种发展态势下，让微软对于云计算平台的投入不断加大，在本次大会上的亮点包括：

- 拥抱 Docker——虽然 Docker 公司总部就在旧金山，但是当 Docker 的 CEO 出现在微软 Build 大会现场时，还是让作者感到有些惊讶。他分享了 Docker 与微软的合作进展，微软的目标是让 Windows 和 Azure 都支持 Docker 相关容器技术，并投入精力在 DockerHub 上发布更多微软系的应用，关于微软与 Docker 的合作，近期将会有专门的新闻报道。
- 全新 Microsoft Azure 数据服务——Azure 上目前每天创建和删除的 SQL 数据库总数达到 16 万个之多，为了提供数据管理的效率，微软在本次大会上推出了新的数据管理工具，包括透明数据加密、全文检索支持和弹性数据池（elastic database pool）。例如，当客户发现某个数据库的负载压力较大时，会考虑将其放到一个独立的 instance 中，当类似的数据库越来越多时，跨数据库的数据管理就会出现困难，而微软推出的新工具会帮助更方便地管理数据。
- 机器学习支持——数据管理和机器学习是相辅相成的，新推出的 Azure Data Lake 可以存储和处理 PB 级的数据，从包括 Hadoop 等来源中导入数据，通过 PowerBI 进行数据分析，支持 R 语言编程，通过 Azure 的机器学习服务建立预测模型。目前国际上三大云计算厂商（微软、谷歌、亚马逊）都已经提供了机器学习服务。

## Visual Studio

**关键字：**多系统支持、开放态度、跨平台化

许多开发者对 Build 大会的关注点集中在 Visual Studio 上，去年底，Visual Studio 2015 推出了免费的社区版，成为 Windows 平台上众多开发者的福利。而这一次 Build 大会上，Visual Studio 得到了很多亮相机会：

- 对 Android/Java、iOS/Object-C 应用的支持，是的，你没有看错，微软在本次大会上正式宣布，Visual Studio 将很快支持开发者只需要修改少量代码，就可以编译上面两种应用，并运行在 Windows 10 系统中。这对移动开发者来说，进入 Windows 平台将不再需要投入大量精力来重建应用，对于微软来说，预计将会有大量的开发者拥抱 Windows 应用，对于消费者来说，可以很快享受到更多的应用。

之前提到的 Docker 支持，Docker CEO 就是在 Visual Studio 中进行了现场演示，目前 VS 支持 Azure 上 Docker 容器技术的开发和部署，并可以与 VS Online 服务联动。主题演讲中提到了 VS 的 Online 服务，可以帮助客户实现从 0 到 1 的全生命周期研发管理流程，实现真正的 Devops。

- 最新推出了支持 Linux 和 Mac 平台的代码编辑器 Visual Studio Code，而且完全免费，意在吸引更多非 Windows 平台的开发者。目前 Visual Studio Code 提供的功能包括编码、高亮显示、智能辅助、Git 集成等等，但与 Visual Studio 依然不是一个量级，两者定位不同。
- 开放态度，就在几个月之前，有关.NET 开源的新闻在国内社区中引起了很大反响。在本次大会上，微软表示，.NET Core 的 Windows 版本进入 RC 状态，而 Linux 和 Mac 相应推出了预览版。

## Windows 10 和 Office

**关键字：**通用应用、平台化

开发者对 Windows 10 和 Office 这两个“传统”产品的期待可能在于看看有什么新玩法。Satya Nadella 对参会者表示，Windows 10 不是新一个版本（release），而是一个新时代（generation）。微软希望到 2018 财年，Windows 10 的活跃设备数超过 10 亿。

- 提到 Windows 10，总是和通用应用（Universal Application）分不开，因为 Windows 10 的定位是全领域系统，从物联网的微小设备到移动手机端，再到 PC 端，都是 Windows 10 的适用范围，那么开发者一旦创建了相应应用，就意味着可以放到支持 Windows 10 的各种设备中运行，有一种像 Java 一样，“一次编写，到处运行”的味道。这种方式将吸引更多的开发者关注 Windows 10 平台。
- Build 大会上对 Office 的展示，表达了其“平台化”的想象空间，包括全新的 Office Graph API、面向 iPad 和 Outlook 的功能扩展插件，以及统一 API 等。包括 Excel 的

SAP 插件、Powerpoint 的股票插件、Outlook 的 Uber 插件，这些丰富的例子，能够让开发者看到 Office 的扩展潜力。

- 现场演示了开发者如何将一个应用部署于不同类型的 Windows 10 设备上，并自动适应不同屏幕尺寸。利用通用 Windows 平台（Universal Windows Platform），开发者可以为每种设备定制独特的功能、整合 Cortana 和 Xbox Live 服务、提供安全交易、创建全息体验（hologram），并最终将应用发布到 Windows 应用商店中。
- 新的浏览器 Microsoft Edge 的发布让开发者松了口气，IE 时代宣告结束。Edge 为开发者提供了更多的应用展示和曝光机会，并提升了应用通过 JavaScript 和 HTML 实现扩展的能力。在未来几年，前端开发者将集中关注这个新生事物。

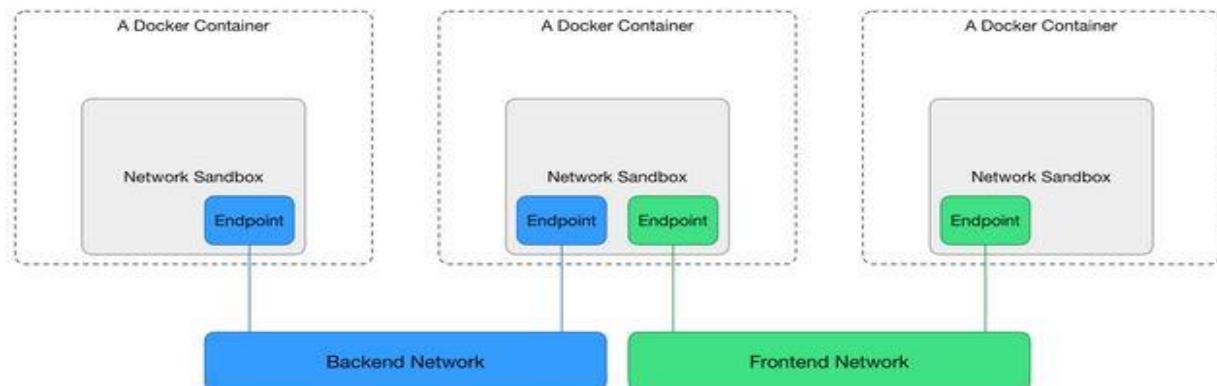
# Docker 发布新的网络项目，并开始招聘中国区主管

作者 郭蕾

5月1日，Docker发布了自家的容器网络管理项目 [libnetwork](#)，libnetwork 使用 Go 语言编写，目标是定义一个容器网络模型（CNM），并为应用程序提供一致的编程接口以及网络抽象。目前 libnetwork 仍在全力开发中，并没有达到使用标准。

3月的时候，Docker公司收购了 [SDN 技术创业公司 SocketPlane](#)，以构建一个健康的容器网络生态系统。于是本周，在网络合作伙伴 Cisco、IBM、Joyent、Microsoft、Rancher、VMware 和 Weave 的帮助下，Docker对外发布了开源项目 libnetwork。Libnetwork一开始的代码只是 libcontainer 和 Docker Engine 中网络部分代码的合并，Docker 官方的愿景是希望 libnetwork 能像 libcontainer 一样，成为一个多平台的容器网络基础包。

受之前的一个 [GitHub issue](#) 启发，libnetwork 引入了容器网络模型（CNM）的概念，CNM 定义了三个新的术语，分别是网络沙箱、Endpoint、Network。网络沙箱指的是在每一个容器中，将会有一个隔离的用于网络配置的环境。Endpoint 是一个网络接口，可用于某一网络上的交流。Network 是一个唯一的且可识别的 Endpoint 组。



从[官方博客](#)中得知，设计这样一个可插拔的网络接口非常困难，因为网络部分的工作涉及 Docker Engine 和 libcontainer。值得高兴的是，现在 Docker 公司正在使用 CNM API 重写 Docker 的 bridge 网络（docker0）。

接下来，Docker 公司将会把 libnetwork 集成到 Docker Engine，并在 Docker CLI 中使用新的网络命令。具体的项目路线图读者可以参考 [GitHub](#)。

另外，从[官方邮件](#)中得知，Docker 公司正准备进军中国。目前他们正在招聘中国区的主管，以运营中国社区，并进一步扩展 Docker 的影响力。新的主管直接汇报给 CFO。具体信息读者可以[参看具体的 JD](#)。

# Netty 系列之 Netty 编解码框架分析

作者李林锋

## 1. 背景

### 1.1. 编解码技术

通常我们也习惯将编码（Encode）称为序列化（serialization），它将对象序列化为字节数组，用于网络传输、数据持久化或者其它用途。

反之，解码（Decode）/反序列化（deserialization）把从网络、磁盘等读取的字节数组还原成原始对象（通常是原始对象的拷贝），以方便后续的业务逻辑操作。

进行远程跨进程服务调用时（例如 RPC 调用），需要使用特定的编解码技术，对需要进行网络传输的对象做编码或者解码，以便完成远程调用。

### 1.2. 常用的编解码框架

#### 1.2.1. Java 序列化

相信大多数 Java 程序员接触到的第一种序列化或者编解码技术就是 Java 默认提供的序列化机制，需要序列化的 Java 对象只需要实现 `java.io.Serializable` 接口并生成序列化 ID，这个类就能够通过 `java.io.ObjectInput` 和 `java.io.ObjectOutput` 序列化和反序列化。

由于使用简单，开发门槛低，Java 序列化得到了广泛的应用，但是由于它自身存在很多缺点，因此大多数的 RPC 框架并没有选择它。Java 序列化的主要缺点如下。

- 1) 无法跨语言：是 Java 序列化最致命的问题。对于跨进程的服务调用，服务提供者可能会使用 C++ 或者其它语言开发，当我们需要和异构语言进程交互时，Java 序列化就难以胜任。由于 Java 序列化技术是 Java 语言内部的私有协议，其它语言并不支持，对于用户来说它完全是黑盒。Java 序列化后的字节数组，别的语言无法进行反序列化，这就严重阻碍了它的应用范围。
- 2) 序列化后的码流太大：例如使用二进制编解码技术对同一个复杂的 POJO 对象进行编码，它的码流仅为 Java 序列化之后的 20% 左右；目前主流的编解码框架，序列化之后的码流都远远小于原生的 Java 序列化。

3) 序列化效率差：在相同的硬件条件下、对同一个 POJO 对象做 100W 次序列化，二进制编码和 Java 原生序列化的性能对比测试如下图所示：Java 原生序列化的耗时是二进制编码的 16.2 倍，效率非常差。

```

Problems @ Javadoc Declaration Search Console Progress
<terminated> PerformTestUserInfo [Java Application] E:\Program Files\Java\jdk1.7.0_45\bin\javaw.exe
The jdk serializable cost time is : 7344 ms
-----
The byte array serializable cost time is : 453 ms

```

图 1-1 二进制编码和 Java 原生序列化性能对比

### 1.2.2. Google 的 Protobuf

Protobuf 全称 Google Protocol Buffers，它由谷歌开源而来，在谷歌内部久经考验。它将数据结构以.proto 文件进行描述，通过代码生成工具可以生成对应数据结构的 POJO 对象和 Protobuf 相关的方法和属性。

它的特点如下：

- 1) 结构化数据存储格式（XML, JSON 等）；
- 2) 高效的编解码性能；
- 3) 语言无关、平台无关、扩展性好；
- 4) 官方支持 Java、C++ 和 Python 三种语言。

首先我们来看下为什么不使用 XML，尽管 XML 的可读性和可扩展性非常好，也非常适合描述数据结构，但是 XML 解析的时间开销和 XML 为了可读性而牺牲的空间开销都非常大，因此不适合做高性能的通信协议。Protobuf 使用二进制编码，在空间和性能上具有更大的优势。

Protobuf 另一个比较吸引人的地方就是它的数据描述文件和代码生成机制，利用数据描述文件对数据结构进行说明的优点如下：

- 1) 文本化的数据结构描述语言，可以实现语言和平台无关，特别适合异构系统间的集成；
- 2) 通过标识字段的顺序，可以实现协议的前向兼容；
- 3) 自动代码生成，不需要手工编写同样数据结构的 C++ 和 Java 版本；
- 4) 方便后续的管理和维护。相比于代码，结构化的文档更容易管理和维护。

### 1.2.3. Apache 的 Thrift

Thrift 源于 Facebook，在 2007 年 Facebook 将 Thrift 作为一个开源项目提交给 Apache 基金会。对于当时的 Facebook 来说，创造 Thrift 是为了解决 Facebook 各系统间大数据量的传输通信以及系统之间语言环境不同需要跨平台的特性，因此 Thrift 可以支持多种程序语言，如 C++、C#、Cocoa、Erlang、Haskell、Java、OCaml、Perl、PHP、Python、Ruby 和 Smalltalk。

在多种不同的语言之间通信，Thrift 可以作为高性能的通信中间件使用，它支持数据（对象）序列化和多种类型的 RPC 服务。Thrift 适用于静态的数据交换，需要先确定好它的数据结构，当数据结构发生变化时，必须重新编辑 IDL 文件，生成代码和编译，这一点跟其他 IDL 工具相比可以视为是 Thrift 的弱项。Thrift 适用于搭建大型数据交换及存储的通用工具，对于大型系统中的内部数据传输，相对于 JSON 和 XML 在性能和传输大小上都有明显的优势。

Thrift 主要由 5 部分组成：

- 1) 语言系统以及 IDL 编译器：负责由用户给定的 IDL 文件生成相应语言的接口代码；
- 2) TProtocol: RPC 的协议层，可以选择多种不同的对象序列化方式，如 JSON 和 Binary；
- 3) TTransport: RPC 的传输层，同样可以选择不同的传输层实现，如 socket、NIO、MemoryBuffer 等；
- 4) TProcessor: 作为协议层和用户提供的服务实现之间的纽带，负责调用服务实现的接口；
- 5) TServer: 聚合 TProtocol、TTransport 和 TProcessor 等对象。

我们重点关注的是编解码框架，与之对应的就是 TProtocol。由于 Thrift 的 RPC 服务调用和编解码框架绑定在一起，所以，通常我们使用 Thrift 的时候会采取 RPC 框架的方式。但是，它的 TProtocol 编解码框架还是可以以类库的方式独立使用的。

与 Protobuf 比较类似的是，Thrift 通过 IDL 描述接口和数据结构定义，它支持 8 种 Java 基本类型、Map、Set 和 List，支持可选和必选定义，功能非常强大。因为可以定义数据结构中字段的顺序，所以它也可以支持协议的前向兼容。

Thrift 支持三种比较典型的编解码方式：

- 1) 通用的二进制编解码；
- 2) 压缩二进制编解码；
- 3) 优化的可选字段压缩编解码。

由于支持二进制压缩编解码，Thrift 的编解码性能表现也相当优异，远远超过 Java 序列化

和 RMI 等。

#### 1.2.4. JBoss Marshalling

JBoss Marshalling 是一个 Java 对象的序列化 API 包，修正了 JDK 自带的序列化包的很多问题，但又保持跟 `java.io.Serializable` 接口的兼容；同时增加了一些可调的参数和附加的特性，并且这些参数和特性可通过工厂类进行配置。

相比于传统的 Java 序列化机制，它的优点如下：

- 1) 可插拔的类解析器，提供更加便捷的类加载定制策略，通过一个接口即可实现定制；
- 2) 可插拔的对象替换技术，不需要通过继承的方式；
- 3) 可插拔的预定义类缓存表，可以减小序列化的字节数组长度，提升常用类型的对象序列化性能；
- 4) 无须实现 `java.io.Serializable` 接口，即可实现 Java 序列化；
- 5) 通过缓存技术提升对象的序列化性能。

相比于前面介绍的两种编解码框架，JBoss Marshalling 更多是在 JBoss 内部使用，应用范围有限。

#### 1.2.5. 其它编解码框架

除了上述介绍的编解码框架和技术之外，比较常用的还有 MessagePack、kryo、hession 和 Json 等。限于篇幅所限，不再一一枚举，感兴趣的朋友可以自行查阅相关资料学习。

## 2. Netty 编解码框架

### 2.1. Netty 为什么要提供编解码框架

作为一个高性能的异步、NIO 通信框架，编解码框架是 Netty 的重要组成部分。尽管站在微内核的角度看，编解码框架并不是 Netty 微内核的组成部分，但是通过 `ChannelHandler` 定制扩展出的编解码框架却是不可或缺的。

下面我们从几个角度详细谈下这个话题，首先一起看下 Netty 的逻辑架构图：

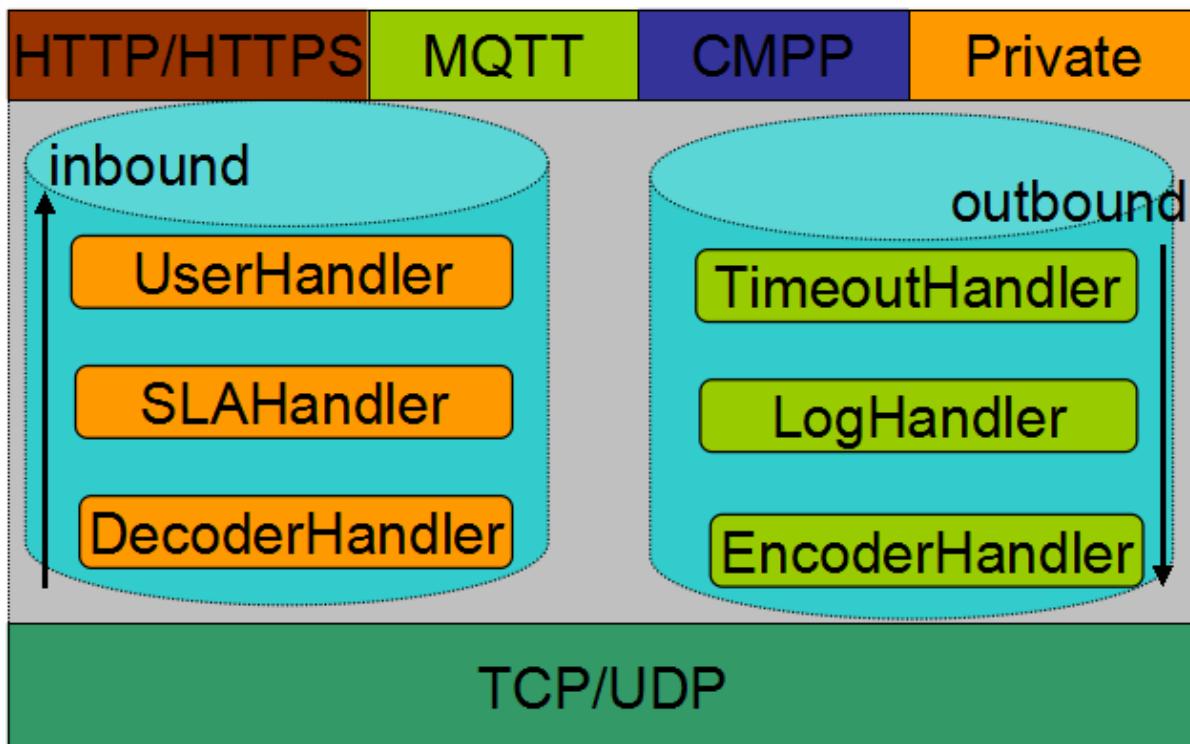


图 2-1 Netty 逻辑架构图

从网络读取的 **inbound** 消息，需要经过解码，将二进制的数据报转换成应用层协议消息或者业务消息，才能够被上层的应用逻辑识别和处理；同理，用户发送到网络的 **outbound** 业务消息，需要经过编码转换成二进制字节数组（对于 Netty 就是 ByteBuf）才能够发送到网络对端。编码和解码功能是 NIO 框架的有机组成部分，无论是由业务定制扩展实现，还是 NIO 框架内置编解码能力，该功能是必不可少的。

为了降低用户的开发难度，Netty 对常用的功能和 API 做了装饰，以屏蔽底层的实现细节。编解码功能的定制，对于熟悉 Netty 底层实现的开发者而言，直接基于 ChannelHandler 扩展开发，难度并不是很大。但是对于大多数初学者或者不愿意去了解底层实现细节的用户，需要提供给他们更简单的类库和 API，而不是 ChannelHandler。

Netty 在这方面做得非常出色，针对编解码功能，它既提供了通用的编解码框架供用户扩展，又提供了常用的编解码类库供用户直接使用。在保证定制扩展性的基础之上，尽量降低用户的开发工作量和开发门槛，提升开发效率。

Netty 预置的编解码功能列表如下：base64、Protobuf、JBoss Marshalling、spdy 等。

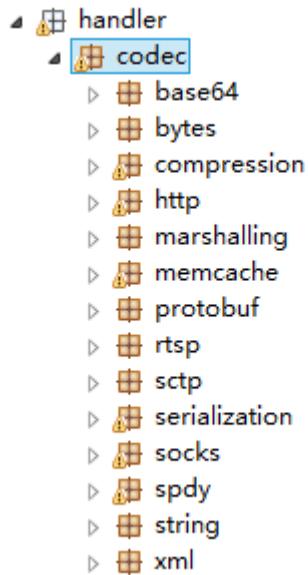


图 2-2 Netty 预置的编解码功能列表

## 2.2. 常用的解码器

### 2.2.1. LineBasedFrameDecoder 解码器

LineBasedFrameDecoder 是回车换行解码器，如果用户发送的消息以回车换行符作为消息结束的标识，则可以直接使用 Netty 的 LineBasedFrameDecoder 对消息进行解码，只需要在初始化 Netty 服务端或者客户端时将 LineBasedFrameDecoder 正确的添加到 ChannelPipeline 中即可，不需要自己重新实现一套换行解码器。

LineBasedFrameDecoder 的工作原理是它依次遍历 ByteBuf 中的可读字节，判断看是否有“\n”或者“\r\n”，如果有，就以此位置为结束位置，从可读索引到结束位置区间的字节就组成了一行。它是以换行符为结束标志的解码器，支持携带结束符或者不携带结束符两种解码方式，同时支持配置单行的最大长度。如果连续读取到最大长度后仍然没有发现换行符，就会抛出异常，同时忽略掉之前读到的异常码流。防止由于数据报没有携带换行符导致接收到 ByteBuf 无限制积压，引起系统内存溢出。

它的使用效果如下：

解码之前：

```
+-----+-----+
```

接收到的数据报

```
"This is a netty example for using the nio framework.\r\n When you"
```

```
+-----+
| 解码之后的 ChannelHandler 接收到的 Object 如下:
+-----+
```

```
+-----+
|       解码之后的文本消息
+-----+
```

```
"This is a netty example for using the nio framework."
```

通常情况下，`LineBasedFrameDecoder` 会和 `StringDecoder` 配合使用，组合成按行切换的文本解码器，对于文本类协议的解析，文本换行解码器非常实用，例如对 HTTP 消息头的解析、FTP 协议消息的解析等。

下面我们简单给出文本换行解码器的使用示例：

```
@Override
protected void initChannel(SocketChannel arg0) throws Exception {
    arg0.pipeline().addLast(new LineBasedFrameDecoder(1024));
    arg0.pipeline().addLast(new StringDecoder());
    arg0.pipeline().addLast(new UserServerHandler());
}
```

初始化 `Channel` 的时候，首先将 `LineBasedFrameDecoder` 添加到 `ChannelPipeline` 中，然后再依次添加字符串解码器 `StringDecoder`，业务 Handler。

### 2.2.2. DelimiterBasedFrameDecoder 解码器

`DelimiterBasedFrameDecoder` 是分隔符解码器，用户可以指定消息结束的分隔符，它可以自动完成以分隔符作为码流结束标识的消息的解码。回车换行解码器实际上是一种特殊的 `DelimiterBasedFrameDecoder` 解码器。

分隔符解码器在实际工作中也有很广泛的应用，笔者所从事的电信行业，很多简单的文本私有协议，都是以特殊的分隔符作为消息结束的标识，特别是对于那些使用长连接的基于文本的私有协议。

分隔符的指定：与大家的习惯不同，分隔符并非以 `char` 或者 `string` 作为构造参数，而是

ByteBuf，下面我们就结合实际例子给出它的用法。

假如消息以“\$\_”作为分隔符，服务端或者客户端初始化 ChannelPipeline 的代码实例如下：

```

@Override

public void initChannel(SocketChannel ch)

throws Exception {

    ByteBuf delimiter = Unpooled.copiedBuffer("$_"

        .getBytes());

    ch.pipeline().addLast(

        new DelimiterBasedFrameDecoder(1024,

            delimiter));

    ch.pipeline().addLast(new StringDecoder());

    ch.pipeline().addLast(new UserServerHandler());

}

```

首先将“\$\_”转换成 ByteBuf 对象，作为参数构造 DelimiterBasedFrameDecoder，将其添加到 ChannelPipeline 中，然后依次添加字符串解码器（通常用于文本解码）和用户 Handler，请注意解码器和 Handler 的添加顺序，如果顺序颠倒，会导致消息解码失败。

DelimiterBasedFrameDecoder 原理分析：解码时，判断当前已经读取的 ByteBuf 中是否包含分隔符 ByteBuf，如果包含，则截取对应的 ByteBuf 返回，源码如下：

```

protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer)
    if (lineBasedDecoder != null) {
        return lineBasedDecoder.decode(ctx, buffer);
    }
    // Try all delimiters and choose the delimiter which yields the
    int minFrameLength = Integer.MAX_VALUE;
    ByteBuf minDelim = null;
    for (ByteBuf delim: delimiters) {
        int frameLength = indexOf(buffer, delim);
        if (frameLength >= 0 && frameLength < minFrameLength) {
            minFrameLength = frameLength;
            minDelim = delim;
        }
    }
}

```

详细分析下 indexOf(buffer, delim) 方法的实现，代码如下：

```

private static int indexOf(ByteBuf haystack, ByteBuf needle) {
    for (int i = haystack.readerIndex(); i < haystack.writerIndex(); i++) {
        int haystackIndex = i;
        int needleIndex;
        for (needleIndex = 0; needleIndex < needle.capacity(); needleIndex++) {
            if (haystack.getByte(haystackIndex) != needle.getByte(needleIndex)) {
                break;
            } else {
                haystackIndex++;
                if (haystackIndex == haystack.writerIndex() &&
                    needleIndex != needle.capacity() - 1) {
                    return -1;
                }
            }
        }
        if (needleIndex == needle.capacity()) {
            // Found the needle from the haystack!
            return i - haystack.readerIndex();
        }
    }
    return -1;
}

```

该算法与 Java String 中的搜索算法类似，对于原字符串使用两个指针来进行搜索，如果搜索成功，则返回索引位置，否则返回-1。

### 2.2.3. FixedLengthFrameDecoder 解码器

FixedLengthFrameDecoder 是固定长度解码器，它能够按照指定的长度对消息进行自动解码，开发者不需要考虑 TCP 的粘包/拆包等问题，非常实用。

对于定长消息，如果消息实际长度小于定长，则往往会进行补位操作，它在一定程度上导致了空间和资源的浪费。但是它的优点也是非常明显的，编解码比较简单，因此在实际项目中仍然有一定的应用场景。

利用 FixedLengthFrameDecoder 解码器，无论一次接收到多少数据报，它都会按照构造函数中设置的固定长度进行解码，如果是半包消息，FixedLengthFrameDecoder 会缓存半包消息并等待下个包到达后进行拼包，直到读取到一个完整的包。

假如单条消息的长度是 20 字节，使用 FixedLengthFrameDecoder 解码器的效果如下：

解码前：

+-----+

接收到的数据报

```
"HELLO NETTY FOR USER DEVELOPER"
```

```
+-----+
```

解码后：

```
+-----+
```

解码后的数据报

```
"HELLO NETTY FOR USER"
```

```
+-----+
```

#### 2.2.4. LengthFieldBasedFrameDecoder 解码器

了解 TCP 通信机制的读者应该都知道 TCP 底层的粘包和拆包，当我们在接收消息的时候，显示不能认为读取到的报文就是个整包消息，特别是对于采用非阻塞 I/O 和长连接通信的程序。

如何区分一个整包消息，通常有如下 4 种做法：

- 1) 固定长度，例如每 120 个字节代表一个整包消息，不足的前面补位。解码器在处理这类定常消息的时候比较简单，每次读到指定长度的字节后再进行解码；
- 2) 通过回车换行符区分消息，例如 HTTP 协议。这类区分消息的方式多用于文本协议；
- 3) 通过特定的分隔符区分整包消息；
- 4) 通过在协议头/消息头中设置长度字段来标识整包消息。

前三种解码器之前的章节已经做了详细介绍，下面让我们来一起学习最后一种通用解码器-`LengthFieldBasedFrameDecoder`。

大多数的协议（私有或者公有），协议头中会携带长度字段，用于标识消息体或者整包消息的长度，例如 SMPP、HTTP 协议等。由于基于长度解码需求的通用性，以及为了降低用户的协议开发难度，Netty 提供了 `LengthFieldBasedFrameDecoder`，自动屏蔽 TCP 底层的拆包和粘包问题，只需要传入正确的参数，即可轻松解决“读半包”问题。

下面我们看看如何通过参数组合的不同来实现不同的“半包”读取策略。第一种常用的方式是消息的第一个字段是长度字段，后面是消息体，消息头中只包含一个长度字段。它的消息结构定义如图所示：

Length	Actual Content
0x000C	"HELLO, WORLD"

图 2-3 解码前的字节缓冲区（14 字节）

使用以下参数组合进行解码：

- 1) lengthFieldOffset = 0;
- 2) lengthFieldLength = 2;
- 3) lengthAdjustment = 0;
- 4) initialBytesToStrip = 0。

解码后的字节缓冲区内容如图所示：

Length	Actual Content
0x000C	"HELLO, WORLD"

图 2-4 解码后的字节缓冲区（14 字节）

通过 `ByteBuf.readableBytes()` 方法我们可以获取当前消息的长度，所以解码后的字节缓冲区可以不携带长度字段，由于长度字段在起始位置并且长度为 2，所以将 `initialBytesToStrip` 设置为 2，参数组合修改为：

- 1) lengthFieldOffset = 0;
- 2) lengthFieldLength = 2;
- 3) lengthAdjustment = 0;
- 4) initialBytesToStrip = 2。

解码后的字节缓冲区内容如图所示：



图 2-5 跳过长度字段解码后的字节缓冲区（12 字节）

解码后的字节缓冲区丢弃了长度字段，仅仅包含消息体，对于大多数的协议，解码之后消息长度没有用处，因此可以丢弃。

在大多数的应用场景中，长度字段仅用来标识消息体的长度，这类协议通常由消息长度字段+消息体组成，如上图所示的几个例子。但是，对于某些协议，长度字段还包含了消息头的长度。在这种应用场景中，往往需要使用 `lengthAdjustment` 进行修正。由于整个消息（包含消息头）的长度往往大于消息体的长度，所以，`lengthAdjustment` 为负数。图 2-6 展示了通过指定 `lengthAdjustment` 字段来包含消息头的长度：

- 1) `lengthFieldOffset = 0;`
- 2) `lengthFieldLength = 2;`
- 3) `lengthAdjustment = -2;`
- 4) `initialBytesToStrip = 0.`

解码之前的码流：



图 2-6 包含长度字段自身的码流

解码之后的码流：



图 2-7 解码后的码流

由于协议种类繁多，并不是所有的协议都将长度字段放在消息头的首位，当标识消息长度的字段位于消息头的中间或者尾部时，需要使用 lengthFieldOffset 字段进行标识，下面的参数组合给出了如何解决消息长度字段不在首位的问题：

- 1) lengthFieldOffset = 2;
- 2) lengthFieldLength = 3;
- 3) lengthAdjustment = 0;
- 4) initialBytesToStrip = 0。

其中 lengthFieldOffset 表示长度字段在消息头中偏移的字节数，lengthFieldLength 表示长度字段自身的长度，解码效果如下：

解码之前：

Header 1	Length	Actual Content
0xCAFE	0x00000C	"HELLO, WORLD"

图 2-8 长度字段偏移的原始码流

解码之后：

Header 1	Length	Actual Content
0xCAFE	0x00000C	"HELLO, WORLD"

图 2-9 长度字段偏移解码后的码流

由于消息头 1 的长度为 2，所以长度字段的偏移量为 2；消息长度字段 Length 为 3，所以 lengthFieldLength 值为 3。由于长度字段仅仅标识消息体的长度，所以 lengthAdjustment 和 initialBytesToStrip 都为 0。

最后一种场景是长度字段夹在两个消息头之间或者长度字段位于消息头的中间，前后都有其它消息头字段，在这种场景下如果想忽略长度字段以及其前面的其它消息头字段，则可以通过 initialBytesToStrip 参数来跳过要忽略的字节长度，它的组合配置示意如下：

- 1) lengthFieldOffset = 1;

- 2) lengthFieldLength = 2;
- 3) lengthAdjustment = 1;
- 4) initialBytesToStrip = 3。

解码之前的码流 (16 字节):



图 2-10 长度字段夹在消息头中间的原始码流 (16 字节)

解码之后的码流 (13 字节):



图 2-11 长度字段夹在消息头中间解码后的码流 (13 字节)

由于 HDR1 的长度为 1，所以长度字段的偏移量 lengthFieldOffset 为 1；长度字段为 2 个字节，所以 lengthFieldLength 为 2。由于长度字段是消息体的长度，解码后如果携带消息头中的字段，则需要使用 lengthAdjustment 进行调整，此处它的值为 1，代表的是 HDR2 的长度，最后由于解码后的缓冲区要忽略长度字段和 HDR1 部分，所以 lengthAdjustment 为 3。解码后的结果为 13 个字节，HDR1 和 Length 字段被忽略。

事实上，通过 4 个参数的不同组合，可以达到不同的解码效果，用户在使用过程中可以根据业务的实际情况进行灵活调整。

由于 TCP 存在粘包和组包问题，所以通常情况下用户需要自己处理半包消息。利用 LengthFieldBasedFrameDecoder 解码器可以自动解决半包问题，它的习惯用法如下：

```
pipeline.addLast("frameDecoder", new LengthFieldBasedFrameDecoder(65536, 0, 2));

pipeline.addLast("UserDecoder", new UserDecoder());
```

在 pipeline 中增加 LengthFieldBasedFrameDecoder 解码器，指定正确的参数组合，它可以帮助将 Netty 的 ByteBuf 解码成整包消息，后面的用户解码器拿到的就是一个完整的数据报，按

照逻辑正常进行解码即可，不再需要额外考虑“读半包”问题，降低了用户的开发难度。

## 2.3. 常用的编码器

Netty 并没有提供与 2.2 章节匹配的编码器，原因如下：

- 1) 2.2 章节介绍的 4 种常用的解码器本质都是解析一个完整的数据报给后端，主要用于解决 TCP 底层粘包和拆包；对于编码，就是将 POJO 对象序列化为 ByteBuf，不需要与 TCP 层面打交道，也就不存在半包编码问题。从应用场景和需要解决的实际问题角度看，双方是非对等的；
- 2) 很难抽象出合适的编码器，对于不同的用户和应用场景，序列化技术不尽相同，在 Netty 底层统一抽象封装也并不合适。

Netty 默认提供了丰富的编解码框架供用户集成使用，本文对较常用的 Java 序列化编码器进行讲解。其它的编码器，实现方式大同小异。

### 2.3.1. ObjectEncoder 编码器

ObjectEncoder 是 Java 序列化编码器，它负责将实现 Serializable 接口的对象序列化为 byte[], 然后写入到 ByteBuf 中用于消息的跨网络传输。

下面我们一起分析下它的实现：

首先，我们发现它继承自 MessageToByteEncoder，它的作用就是将对象编码成 ByteBuf：

```
/**
 * An encoder which serializes a Java object into a {@link ByteBuf}.
 * <p>
 * Please note that the serialized form this encoder produces is not
 * compatible with the standard {@link ObjectInputStream}. Please use
 * {@link ObjectDecoder} or {@link ObjectDecoderInputStream} to ensure the
 * interoperability with this encoder.
 */
@Sharable
public class ObjectEncoder extends MessageToByteEncoder<Serializable> {
```

如果要使用 Java 序列化，对象必须实现 Serializable 接口，因此，它的泛型类型为 Serializable。

MessageToByteEncoder 的子类只需要实现 encode(ChannelHandlerContext ctx, I msg, ByteBuf out) 方法即可，下面我们重点关注 encode 方法的实现：

```

@Override
protected void encode(ChannelHandlerContext ctx, Serializable msg, ByteBuf out)
    throws Exception {
    int startIdx = out.writerIndex();

    ByteBufOutputStream bout = new ByteBufOutputStream(out);
    bout.write(LENGTH_PLACEHOLDER);
    ObjectOutputStream oout = new CompactObjectOutputStream(bout);
    oout.writeObject(msg);
    oout.flush();
    oout.close();

    int endIdx = out.writerIndex();

    out.setInt(startIdx, endIdx - startIdx - 4);
}

```

首先创建 ByteBufOutputStream 和 ObjectOutputStream，用于将 Objec 对象序列化到 ByteBuf 中，值得注意的是在 writeObject 之前需要先将长度字段（4 个字节）预留，用于后续长度字段的更新。

依次写入长度占位符(4 字节)、序列化之后的 Object 对象，之后根据 ByteBuf 的 writeIndex 计算序列化之后的码流长度，最后调用 ByteBuf 的 setInt(int index, int value)更新长度占位符为实际的码流长度。

有个细节需要注意，更新码流长度字段使用了 setInt 方法而不是 writeInt，原因就是 setInt 方法只更新内容，并不修改 readerIndex 和 writerIndex。

### 3. Netty 编解码框架可定制性

尽管 Netty 预置了丰富的编解码类库功能，但是在实际的业务开发过程中，总是需要对编解码功能做一些定制。使用 Netty 的编解码框架，可以非常方便的进行协议定制。本章节将对常用的支持定制的编解码类库进行讲解，以期让读者能够尽快熟悉和掌握编解码框架。

## 3.1. 解码器

### 3.1.1. ByteToMessageDecoder 抽象解码器

使用 NIO 进行网络编程时，往往需要将读取到的字节数组或者字节缓冲区解码为业务可以使用的 POJO 对象。为了方便业务将 ByteBuf 解码成业务 POJO 对象，Netty 提供了 ByteToMessageDecoder 抽象工具解码类。

用户自定义解码器继承 ByteToMessageDecoder，只需要实现 void decode (ChannelHandlerContext ctx, ByteBuf in, List<Object> out)抽象方法即可完成 ByteBuf 到 POJO 对象的解码。

由于 `ByteToMessageDecoder` 并没有考虑 TCP 粘包和拆包等场景，用户自定义解码器需要自己处理“读半包”问题。正因为如此，大多数场景不会直接继承 `ByteToMessageDecoder`，而是继承另外一些更高级的解码器来屏蔽半包的处理。

实际项目中，通常将 `LengthFieldBasedFrameDecoder` 和 `ByteToMessageDecoder` 组合使用，前者负责将网络读取的数据报解码为整包消息，后者负责将整包消息解码为最终的业务对象。

除了和其它解码器组合形成新的解码器之外，`ByteToMessageDecoder` 也是很多基础解码器的父类，它的继承关系如下图所示：

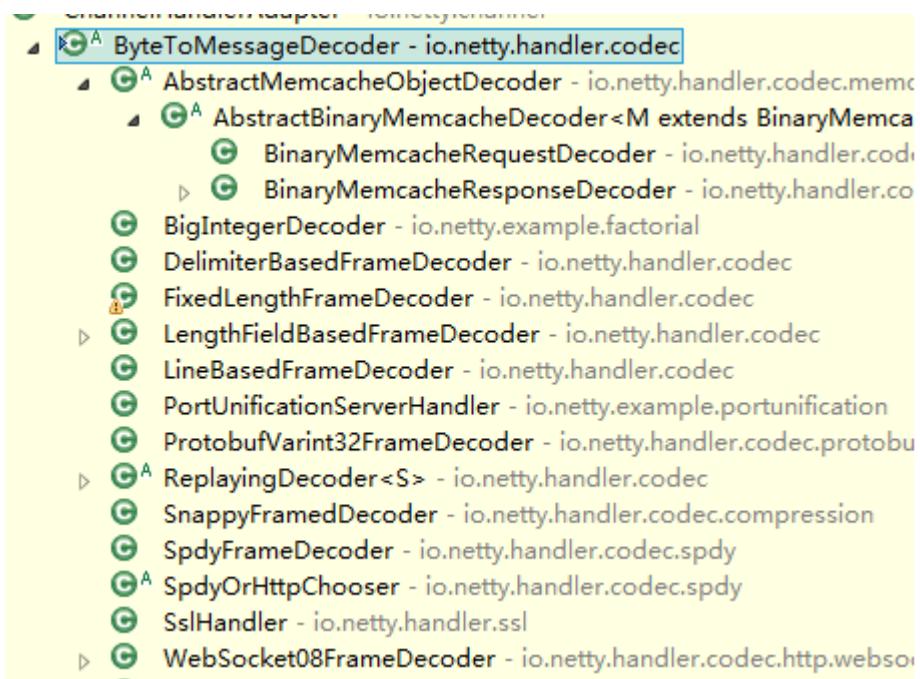


图 3-1 `ByteToMessageDecoder` 继承关系图

### 3.1.2. `MessageToMessageDecoder` 抽象解码器

`MessageToMessageDecoder` 实际上是 Netty 的二次解码器，它的职责是将一个对象二次解码为其它对象。

为什么称它为二次解码器呢？我们知道，从 `SocketChannel` 读取到的 TCP 数据报是 `ByteBuffer`，实际就是字节数组。我们首先需要将 `ByteBuffer` 缓冲区中的数据报读取出来，并将其解码为 Java 对象；然后对 Java 对象根据某些规则做二次解码，将其解码为另一个 POJO 对象。因为 `MessageToMessageDecoder` 在 `ByteToMessageDecoder` 之后，所以称之为二次解码器。

二次解码器在实际的商业项目中非常有用，以 HTTP+XML 协议栈为例，第一次解码往往是将字节数组解码成 `HttpRequest` 对象，然后对 `HttpRequest` 消息中的消息体字符串进行

二次解码，将 XML 格式的字符串解码为 POJO 对象，这就用到了二次解码器。类似这样的场景还有很多，不再一一枚举。

事实上，做一个超级复杂的解码器将多个解码器组合成一个大而全的 MessageToMessageDecoder 解码器似乎也能解决多次解码的问题，但是采用这种方式的代码可维护性会非常差。例如，如果我们打算在 HTTP+XML 协议栈中增加一个打印码流的功能，即首次解码获取 HttpRequest 对象之后打印 XML 格式的码流。如果采用多个解码器组合，在中间插入一个打印消息体的 Handler 即可，不需要修改原有的代码；如果做一个大而全的解码器，就需要在解码的方法中增加打印码流的代码，可扩展性和可维护性都会变差。

用户的解码器只需要实现 void decode(ChannelHandlerContext ctx, I msg, List<Object> out) 抽象方法即可，由于它是将一个 POJO 解码为另一个 POJO，所以一般不会涉及到半包的处理，相对于 ByteToMessageDecoder 更加简单些。它的继承关系图如下所示：



图 3-2 MessageToMessageDecoder 解码器继承关系图

## 3.2. 编码器

### 3.2.1. MessageToByteEncoder 抽象编码器

MessageToByteEncoder 负责将 POJO 对象编码成 ByteBuf，用户的编码器继承 MessageToByteEncoder，实现 void encode(ChannelHandlerContext ctx, I msg, ByteBuf out) 接口接口，示例代码如下：

```

public class IntegerEncoder extends MessageToByteEncoder<Integer> {

    @Override

```

```

public void encode(ChannelHandlerContext ctx, Integer msg, ByteBuf out)
    throws Exception {
    out.writeInt(msg);
}

```

它的实现原理如下：调用 write 操作时，首先判断当前编码器是否支持需要发送的消息，如果不支持则直接透传；如果支持则判断缓冲区的类型，对于直接内存分配 ioBuffer（堆外内存），对于堆内存通过 heapBuffer 方法分配，源码如下：

```

@Override
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)
    throws Exception {
    ByteBuf buf = null;
    try {
        if (acceptOutboundMessage(msg)) {
            @SuppressWarnings("unchecked")
            I cast = (I) msg;
            if (preferDirect) {
                buf = ctx.alloc().ioBuffer();
            } else {
                buf = ctx.alloc().heapBuffer();
            }
        }
    }

```

编码使用的缓冲区分配完成之后，调用 encode 抽象方法进行编码，方法定义如下：它由子类负责具体实现。

```

/**
 * Encode a message into a {@link ByteBuf}. This method will be called for each written message
 * that can be handled
 * by this encoder.
 *
 * @param ctx      the {@link ChannelHandlerContext} which this {@link MessageToByteEncoder}
 * @param msg      the message to encode
 * @param out      the {@link ByteBuf} into which the encoded message will be written
 * @throws Exception is thrown if an error occurs
 */
protected abstract void encode(ChannelHandlerContext ctx, I msg, ByteBuf out) throws Exception;

```

编码完成之后，调用 ReferenceCountUtil 的 release 方法释放编码对象 msg。对编码后的 ByteBuf 进行以下判断：

- 1) 如果缓冲区包含可发送的字节，则调用 ChannelHandlerContext 的 write 方法发送 ByteBuf；

2) 如果缓冲区没有包含可写的字节，则需要释放编码后的 ByteBuf，写入一个空的 ByteBuf 到 ChannelHandlerContext 中。

发送操作完成之后，在方法退出之前释放编码缓冲区 ByteBuf 对象。

### 3.2.2. MessageToMessageEncoder 抽象编码器

将一个 POJO 对象编码成另一个对象，以 HTTP+XML 协议为例，它的一种实现方式是：先将 POJO 对象编码成 XML 字符串，再将字符串编码为 HTTP 请求或者应答消息。对于复杂协议，往往需要经历多次编码，为了便于功能扩展，可以通过多个编码器组合来实现相关功能。

用户的解码器继承 MessageToMessageEncoder 解码器，实现 void encode(ChannelHandlerContext ctx, I msg, List<Object> out) 方法即可。注意，它与 MessageToByteEncoder 的区别是输出是对象列表而不是 ByteBuf，示例代码如下：

```
public class IntegerToStringEncoder extends MessageToMessageEncoder <Integer>

{
    @Override

    public void encode(ChannelHandlerContext ctx, Integer message,
                      List<Object> out)

        throws Exception
    {
        out.add(message.toString());
    }
}
```

MessageToMessageEncoder 编码器的实现原理与之前分析的 MessageToByteEncoder 相似，唯一的差别是它编码后的输出是个中间对象，并非最终可传输的 ByteBuf。

简单看下它的源码实现：创建 RecyclableArrayList 对象，判断当前需要编码的对象是否是编码器可处理的类型，如果不是，则忽略，执行下一个 ChannelHandler 的 write 方法。

具体的编码方法实现由用户子类编码器负责完成，如果编码后的 RecyclableArrayList 为空，说明编码没有成功，释放 RecyclableArrayList 引用。

如果编码成功，则通过遍历 RecyclableArrayList，循环发送编码后的 POJO 对象，代码如下所示：

```

@Override
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)
    throws Exception {
    RecyclableArrayList out = null;
    try {
        if (acceptOutboundMessage(msg)) {
            out = RecyclableArrayList.newInstance();
            @SuppressWarnings("unchecked")
            I cast = (I) msg;
            try {
                encode(ctx, cast, out);
            } finally {
                ReferenceCountUtil.release(cast);
            }

            if (out.isEmpty()) {
                out.recycle();
                out = null;
            }
        }
    }
}

```

### 3.2.3. LengthFieldPrepender 编码器

如果协议中的第一个字段为长度字段，Netty 提供了 LengthFieldPrepender 编码器，它可以计算当前待发送消息的二进制字节长度，将该长度添加到 ByteBuf 的缓冲区头中，如图所示：

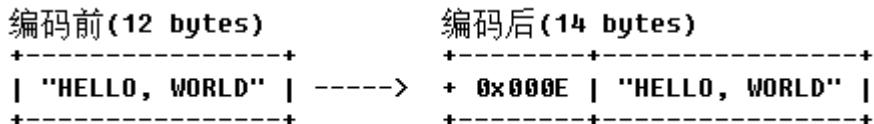


图 3-3 LengthFieldPrepender 编码器

通过 LengthFieldPrepender 可以将待发送消息的长度写入到 ByteBuf 的前 2 个字节，编码后的消息组成为长度字段+原消息的方式。

通过设置 LengthFieldPrepender 为 true，消息长度将包含长度本身占用的字节数，打开 LengthFieldPrepender 后，图 3-3 示例中的编码结果如下图所示：

```

protected void encode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out)
    throws Exception {
    int length = msg.readableBytes() + lengthAdjustment;
    if (lengthIncludesLengthFieldLength) {
        length += lengthFieldLength;
    }

    if (length < 0) {
        throw new IllegalArgumentException(
            "Adjusted frame length (" + length + ") is less than zero");
    }

    switch (lengthFieldLength) {
        case 1:
            if (length >= 256) {
                throw new IllegalArgumentException(
                    "length does not fit into a byte: " + length);
            }
            out.add(ctx.alloc().buffer(1).writeByte((byte) length));
            break;
        case 2:
            if (length >= 65536) {
                throw new IllegalArgumentException(
                    "length does not fit into a short integer: " + length);
            }
    }
}

```

图 3-4 打开 LengthFieldPrepender 开关后编码效果

LengthFieldPrepender 工作原理分析如下：首先对长度字段进行设置，如果需要包含消息长度自身，则在原来长度的基础之上再加上 lengthFieldLength 的长度。

如果调整后的消息长度小于 0，则抛出参数非法异常。对消息长度自身所占的字节数进行判断，以便采用正确的方法将长度字段写入到 ByteBuf 中，共有以下 6 种可能：

- 1) 长度字段所占字节为 1：如果使用 1 个 Byte 字节代表消息长度，则最大长度需要小于 256 个字节。对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的 ByteBuf 并通过 writeByte 将长度值写入到 ByteBuf 中；
- 2) 长度字段所占字节为 2：如果使用 2 个 Byte 字节代表消息长度，则最大长度需要小于 65536 个字节，对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的 ByteBuf 并通过 writeShort 将长度值写入到 ByteBuf 中；
- 3) 长度字段所占字节为 3：如果使用 3 个 Byte 字节代表消息长度，则最大长度需要小于 16777216 个字节，对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的 ByteBuf 并通过 writeMedium 将长度值写入到 ByteBuf 中；
- 4) 长度字段所占字节为 4：创建新的 ByteBuf，并通过 writeInt 将长度值写入到 ByteBuf 中；

- 5) 长度字段所占字节为 8: 创建新的 ByteBuf, 并通过 writeLong 将长度值写入到 ByteBuf 中;
- 6) 其它长度值: 直接抛出 Error。

相关代码如下:

```

protected void encode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out)
    throws Exception {
    int length = msg.readableBytes() + lengthAdjustment;
    if (lengthIncludesLengthFieldLength) {
        length += lengthFieldLength;
    }

    if (length < 0) {
        throw new IllegalArgumentException(
            "Adjusted frame length (" + length + ") is less than zero");
    }

    switch (lengthFieldLength) {
    case 1:
        if (length >= 256) {
            throw new IllegalArgumentException(
                "length does not fit into a byte: " + length);
        }
        out.add(ctx.alloc().buffer(1).writeByte((byte) length));
        break;
    case 2:
        if (length >= 65536) {
            throw new IllegalArgumentException(
                "length does not fit into a short integer: " + length);
        }
    }
}

```

最后将原需要发送的 ByteBuf 复制到 List<Object> out 中, 完成编码:

```

    ...
    case 4:
        out.add(ctx.alloc().buffer(4).writeInt(length));
        break;
    case 8:
        out.add(ctx.alloc().buffer(8).writeLong(length));
        break;
    default:
        throw new Error("should not reach here");
    }
    out.add(msg.retain());
}

```

## 4. 作者简介

**李林锋**, 2007 年毕业于东北大学, 2008 年进入华为公司从事高性能通信软件的设计和开发工作, 有 7 年 NIO 设计和开发经验, 精通 Netty、Mina 等 NIO 框架和平台中间件, 现任华为软件平台架构部架构师, 《Netty 权威指南》作者。

联系方式: 新浪微博 Nettying 微信: Nettying 微信公众号: Netty 之家

对于 Netty 学习中遇到的问题, 或者认为有价值的 Netty 或者 NIO 相关案例, 可以通过上述几种方式联系我。

# 深入浅出 Mesos（一）：为软件定义数据中心而生的操作系统

作者 韩陆

【编者按】Mesos 是 Apache 下的开源分布式资源管理框架，它被称为是分布式系统的内核。Mesos 最初是由加州大学伯克利分校的 AMPLab 开发的，后在 Twitter 得到广泛使用。InfoQ 接下来将会策划系列文章来为读者剖析 Mesos。本文是整个系列的第一篇，简单介绍了 Mesos 的背景、历史以及架构。

注：本文翻译自 [Cloud Architect Musings](#)，InfoQ 中文站在获得作者授权的基础上对文章进行了翻译。

---

我讨厌“软件定义数据中心（SDDC）”这个词，并不是因为我质疑这个概念，而是我发现很多公司都对这个词有误用，他们甚至直接把这个词拿来套用，并急于把自己定位为下一代数据中心的创新者。具体来说，我认为，在商用 x86 硬件上运行软件（应用）并不是什么 SDDC 解决方案，它也不具备虚拟化硬件到资源池的能力。真正的 SDDC 底层基础架构应该可以从运行于其上的应用程序中抽象出来，并根据应用程序不断变化的需求，动态且自动地分配、重新分配应用程序，然后运行于数据中心的不同组件之中。

这就是为什么我一直兴奋地要在后面介绍 Mesos，一个 Apache 开源项目。为什么我对 Mesos 如此兴奋？回想 x86 虚拟化之初对数据中心曾经的承诺：通过增加服务器利用率使其更高效，通过从物理基础架构抽象应用使其更敏捷。虽然收获颇丰，但是以虚拟机为单位，粒度仍不够精细，如果应用程序都过于庞大，那就难以充分实现这一承诺。如今，飞速发展的容器技术、分布式应用程序和微服务技术正悄然改变着我们对数据中心的运行和管理方式。

试想，可否整合数据中心中的所有资源，并将它们放在一个大的虚拟池里，代替单独的物理服务器；然后开放诸如 CPU、内存和 I/O 这些基本资源而不是虚拟机？同样，可否把应用程序拆分成小的、隔离的任务单位，从而根据数据中心应用的需求，从虚拟数据中心池中动态分配任务资源？就像操作系统将 PC 的处理器和 RAM 放入资源池，使其可以为不同的进程协调分配和释放资源。进一步讲，我们可以把 Mesos 作为操作系统内核，然后将数据中心看为 PC。这也是正是我想说的：Mesos 正在改变数据中心，它让真正的 SDDC 成为现实。

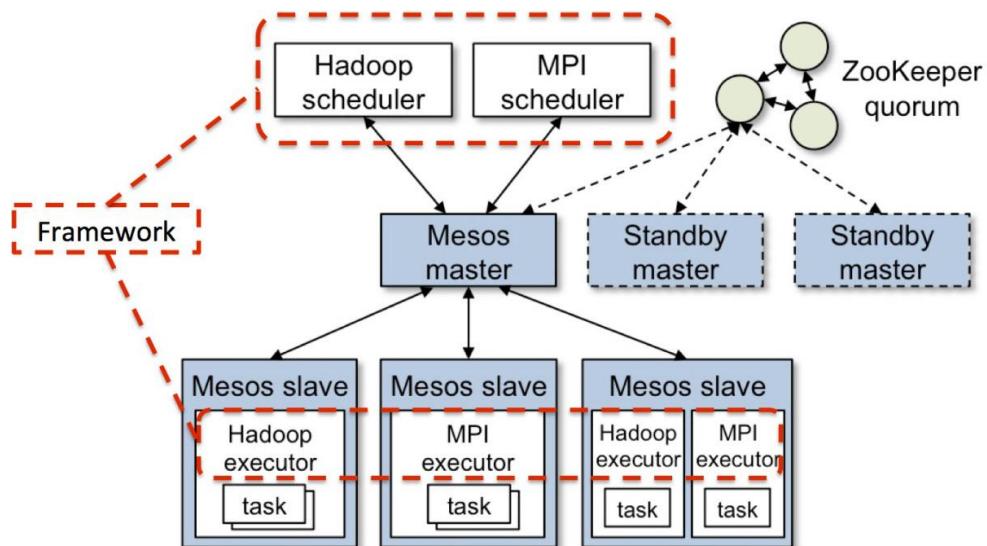


接下来我先介绍下 Mesos 的历史。Mesos 的起源于 Google 的数据中心资源管理系统 Borg。

你可以从 WIRED 杂志的[这篇文章](#)中了解更多关于 Borg 起源的信息及它对 Mesos 影响。

Twitter 从 Google 的 Borg 系统中得到启发，然后就开发一个类似的资源管理系统来帮助他们摆脱可怕的“失败之鲸”（译者注：见上图）。后来他们注意到加州大学伯克利分校 AMPLab 正在开发的名为 Mesos 的项目，这个项目的负责人是 Ben Hindman，Ben 是加州大学伯克利分校的博士研究生。后来 Ben Hindman 加入了 Twitter，负责开发和部署 Mesos。现在 Mesos 管理着 Twitter 超过 30,000 台服务器上的应用部署，“失败之鲸”已成往事。其他公司纷至沓来，也部署了 Mesos，比如 Airbnb（空中食宿网）、eBay（电子港湾）和 Netflix。

Mesos 是如何让 Twitter 和 Airbnb 这样的公司，通过数据中心资源更高效的管理系统，扩展应用的呢？我们从一个相当简单但很优雅的两级调度架构开始说起。



上图修改自 Apache Mesos 网站上的图片，如图所示，Mesos 实现了两级调度架构，它可以管理多种类型的应用程序。第一级调度是 Master 的守护进程，管理 Mesos 集群中所有节点上运行的 Slave 守护进程。集群由物理服务器或虚拟服务器组成，用于运行应用程序的任务，比如 Hadoop 和 MPI 作业。第二级调度由被称作 Framework 的“组件”组成。Framework 包括调度器（Scheduler）和执行器（Executor）进程，其中每个节点上都会运行执行器。Mesos 能和不同类型的 Framework 通信，每种 Framework 由相应的应用集群管理。上图中只展示了 Hadoop 和 MPI 两种类型，其它类型的应用程序也有相应的 Framework。

Mesos Master 协调全部的 Slave，并确定每个节点的可用资源，聚合计算跨节点的所有可用资源的报告，然后向注册到 Master 的 Framework（作为 Master 的客户端）发出资源邀约。Framework 可以根据应用程序的需求，选择接受或拒绝来自 master 的资源邀约。一旦接受邀约，Master 即协调 Framework 和 Slave，调度参与节点上任务，并在容器中执行，以使多种类型的任务，比如 Hadoop 和 Cassandra，可以在同一个节点上同时运行。

我将在接下来的文章中，详细介绍 Mesos 的体系结构和工作流。我认为，Mesos 使用的两级调度架构以及算法、隔离技术让在同一个节点上运行多种不同类型的应用成为了现实，这才是数据中心的未来。正如我之前所述，这是到目前为止我所见过的，履行 SDDC 承诺最好的现成技术。

我希望这篇介绍让你受用并吊起你了解 Mesos 的胃口。接下来，我将带你深入技术细节，教你一些上手方法，还会告诉你如何加入社区。

查看英文原文：[APACHE MESOS: THE TRUE OS FOR THE SOFTWARE DEFINED DATA CENTER?](#)

# 深入浅出 Mesos (二): Mesos 的体系结构和工作流

作者 韩陆

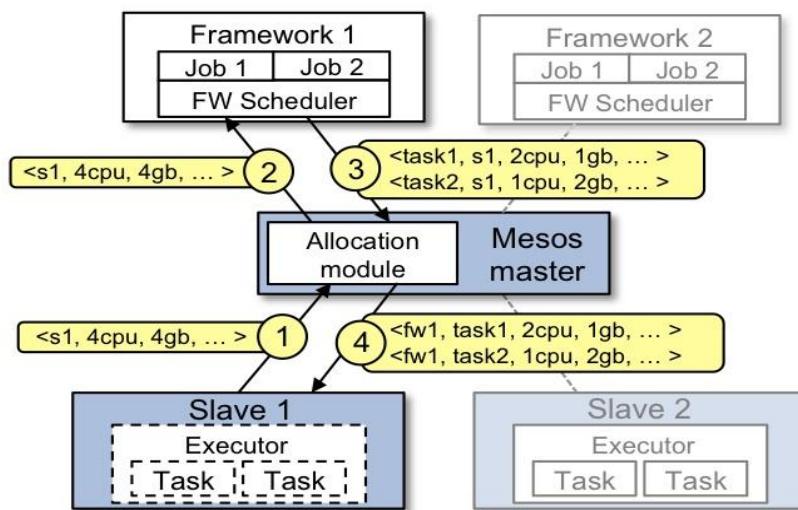
【编者按】Mesos 是 Apache 下的开源分布式资源管理框架，它被称为是分布式系统的内核。Mesos 最初是由加州大学伯克利分校的 AMPLab 开发的，后在 Twitter 得到广泛使用。InfoQ 接下来将会策划系列文章来为读者剖析 Mesos。本文是整个系列的第一篇，简单介绍了 Mesos 的背景、历史以及架构。

注：本文翻译自 [Cloud Architect Musings](#)，InfoQ 中文站在获得作者授权的基础上对文章进行了翻译。

在本系列的[第一篇文章](#)中，我简单介绍了 Apache Mesos 的背景、架构，以及它在数据中心资源管理中的价值。本篇文章将深入剖析 Mesos 的技术细节和组件间的流程，以便大家更好地理解为什么 Mesos 是数据中心操作系统内核的重要候选者。文中所述的大部分技术细节都来自 [Ben Hindman](#) 团队 2010 年在加州大学伯克利分校时发表的[白皮书](#)。顺便说一句，Hindman 已经离开 Twitter 去了 [Mesosphere](#)，着手建设并商业化以 Mesos 为核心的[数据中心操作系统](#)。在此，我将重点放在提炼白皮书的主要观点上，然后给出一些我对相关技术所产生的价值的思考。

## Mesos 流程

接着上一篇文章说。并结合前述的加州大学伯克利分校的白皮书以及 [Apache Mesos 网站](#)，开始我们的讲述：



我们来研究下上图的事件流程。上一篇谈到，Slave 是运行在物理或虚拟服务器上的 Mesos 守护进程，是 Mesos 集群的一部分。Framework 由调度器（Scheduler）应用程序和任务执行器（Executor）组成，被注册到 Mesos 以使用 Mesos 集群中的资源。

Slave 1 向 Master 汇报其空闲资源：4 个 CPU、4GB 内存。然后，Master 触发分配策略模块，得到的反馈是 Framework 1 要请求全部可用资源。

Master 向 Framework 1 发送资源邀约，描述了 Slave 1 上的可用资源。

Framework 的调度器（Scheduler）响应 Master，需要在 Slave 上运行两个任务，第一个任务分配<2 CPUs, 1 GB RAM>资源，第二个任务分配<1 CPUs, 2 GB RAM>资源。

最后，Master 向 Slave 下发任务，分配适当的资源给 Framework 的任务执行器（Executor），接下来由执行器启动这两个任务（如图中虚线框所示）。此时，还有 1 个 CPU 和 1GB 的 RAM 尚未分配，因此分配模块可以将这些资源供给 Framework 2。

## 资源分配

为了实现在同一组 Slave 节点集合上运行多任务这一目标，Mesos 使用了隔离模块，该模块使用了一些应用和进程隔离机制来运行这些任务。不足为奇的是，虽然可以使用虚拟机隔离实现隔离模块，但是 Mesos 当前模块支持的是容器隔离。Mesos 早在 2009 年就用上了 Linux 的容器技术，如 cgroups 和 Solaris Zone，时至今日这些仍然是默认的。然而，Mesos 社区增加了 Docker 作为运行任务的隔离机制。不管使用哪种隔离模块，为运行特定应用程序的任务，都需要将执行器全部打包，并在已经为该任务分配资源的 Slave 服务器上启动。当任务执行完毕后，容器会被“销毁”，资源会被释放，以便可以执行其他任务。

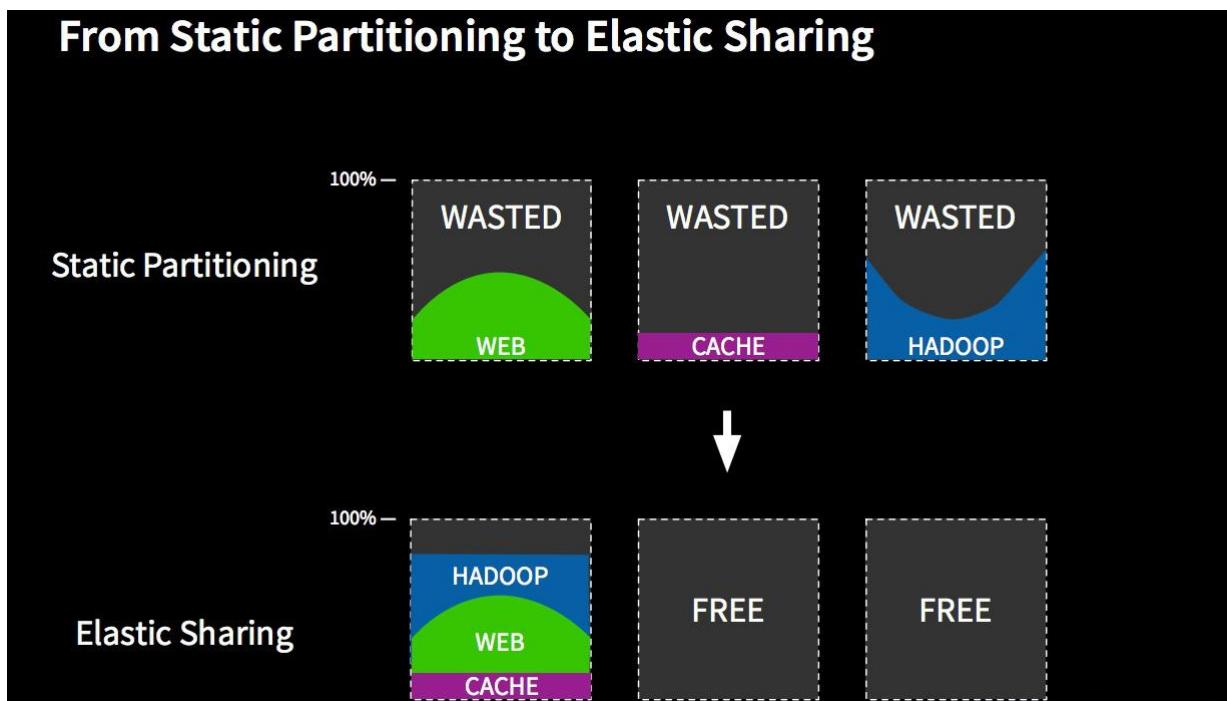
我们来更深入地研究一下资源邀约和分配策略，因为这对 Mesos 管理跨多个 Framework 和应用的资源，是不可或缺的。我们前面提到资源邀约的概念，即由 Master 向注册其上的 Framework 发送资源邀约。每次资源邀约包含一份 Slave 节点上可用的 CPU、RAM 等资源的列表。Master 提供这些资源给它的 Framework，是基于分配策略的。分配策略对所有的 Framework 普遍适用，同时适用于特定的 Framework。Framework 可以拒绝资源邀约，如果它不满足要求，若此，资源邀约随即可以发给其他 Framework。由 Mesos 管理的应用程序通常运行短周期的任务，因此这样可以快速释放资源，缓解 Framework 的资源饥饿；Slave 定期向 Master 报告其可用资源，以便 Master 能够不断产生新的资源邀约。另外，还可以使用诸如此类的技术，每个 Framework 过滤不满足要求的资源邀约、Master 主动废除给定周期内一直没有被接受的邀约。

分配策略有助于 Mesos Master 判断是否应该把当前可用资源提供给特定的 Framework，以及应该提供多少资源。关于 Mesos 中使用资源分配以及可插拔的分配模块，实现非常细粒度的资源共享，会单独写一篇文章。言归正传，Mesos 实现了公平共享和严格优先级（这两个概念我会在资源分配那篇讲）分配模块，确保大部分用例的最佳资源共享。已经实现的新分配模块可以处理大部分之外的用例。

## 集大成者

现在来回答谈及 Mesos 时，“那又怎样”的问题。对于我来说，令人兴奋的是 Mesos 集四大好于一身（概述如下），正如我在前一篇文章中所述，我预测 Mesos 将为下一代数据中心的操作系统内核。

效率 - 这是最显而易见的好处，也是 Mesos 社区和 Mesosphere 经常津津乐道的。



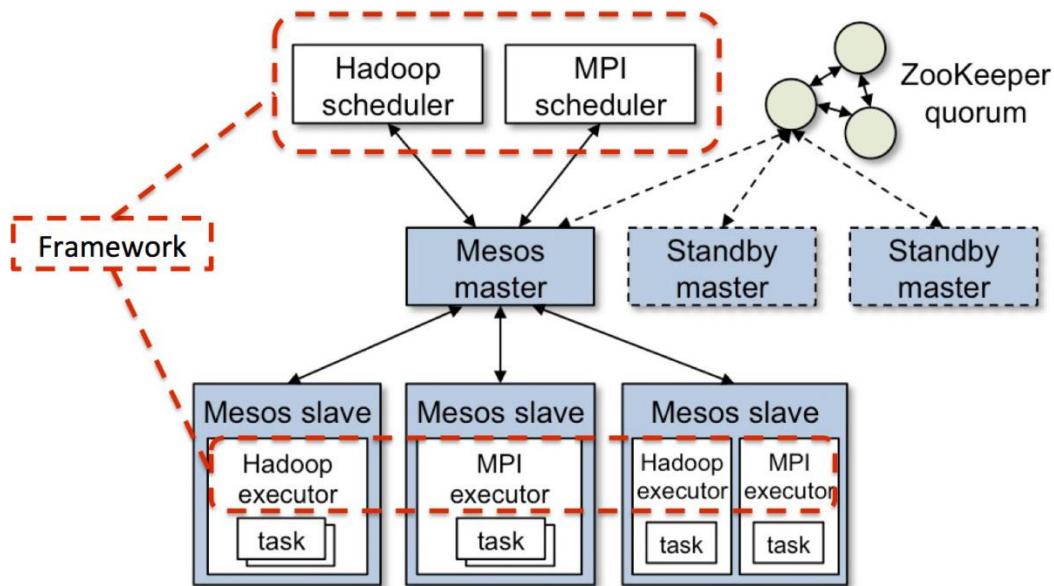
上图来自 Mesosphere 网站，描绘出 Mesos 为效率带来的好处。如今，在大多数数据中心中，服务器的静态分区是常态，即使使用最新的应用程序，如 Hadoop。这时常令人担忧的是，当不同的应用程序使用相同的节点时，调度相互冲突，可用资源互相争抢。静态分区本质上是低效的，因为经常会面临，其中一个分区已经资源耗尽，而另一个分区的资源却没有得到充分利用，而且没有什么简单的方法能跨分区集群重新分配资源。使用 Mesos 资源管理器仲裁不同的调度器，我们将进入动态分区/弹性共享的模式，所有应用程序都可以使用节点的公共池，安全地、最大化地利用资源。一个经常被引用的例子是 Slave 节点通常运行 Hadoop 作业，在 Slave 空闲阶段，动态分配给他们运行批处理作业，反之亦然。值得一提的是，这其中的某些环节可以通过虚拟化技术，如 VMware vSphere 的[分布式资源调度 \(DRS\)](#) 来完成。然而，Mesos 具有更精细的粒度，因为 Mesos 在应用层而不是机器层分配资源，通过容器而不是整个虚拟机 (VM) 分配任务。前者能够为每个应用程序的特殊需求做考量，应用程序的调度器知道最有效地利用资源；后者能够更好地“装箱”，运行一个任务，没有必要实例化一整个虚拟机，其所需的进程和二进制文件足矣。

- 敏捷。与效率和利用率密切相关，这实际上是我认为最重要的好处。往往，效率解决的是“如何花最少的钱最大化数据中心的资源”，而敏捷解决的是“如何快速用上手

头的资源。”正如我和我的同事 [Tyler Britten](#) 经常指出，IT 的存在是帮助企业赚钱和省钱的；那么如何通过技术帮助我们迅速创收，是我们要达到的重要指标。这意味着要确保关键应用程序不能耗尽所需资源，因为我们无法为应用提供足够的基础设施，特别是在数据中心的其他地方都的资源是收费情况下。

- 可扩展性。为可扩展而设计，这是我真心欣赏 Mesos 架构的地方。这一重要属性使数据可以指数级增长、分布式应用可以水平扩展。我们的发展已经远远超出了使用巨大的整体调度器或者限定群集节点数量为 64 的时代，足矣承载新形式的应用扩张。

Mesos 可扩展设计的关键之处是采用两级调度架构。使用 Framework 代理任务的实际调度，Master 可以用非常轻量级的代码实现，更易于扩展集群发展的规模。因为 Master 不必知道所支持的每种类型的应用程序背后复杂的调度逻辑。此外，由于 Master 不必为每个任务做调度，因此不会成为容量的性能瓶颈，而这在为每个任务或者虚拟机做调度的整体调度器中经常发生。



- 模块化。对我来说，预测任何开源技术的健康发展，很大程度上取决于围绕该项目的生态系统。我认为 Mesos 项目前景很好，因为其设计具有包容性，可以将功能插件化，比如分配策略、隔离机制和 Framework。将容器技术，比如 Docker 和 Rocket 插件化的好处是显而易见。但是我想在此强调的是围绕 Framework 建设的生态系统。将任务调度委托给 Framework 应用程序，以及采用插件架构，通过 Mesos 这样的设计，社区创造了能够让 Mesos 问鼎数据中心资源管理的生态系统。因为每接入一种新的 Framework，Master 无需为此编码，Slave 模块可以复用，使得在 Mesos 所支持的宽泛领域中，业务迅速增长。相反，开发者可以专注于他们的应用和 Framework 的选择。当前而且还在不断地增长着的 Mesos Framework 列表参见[此处](#)以及下图：



## 总结

在接下来的文章中，我将更深入到资源分配模块，并解释如何在 Mesos 栈的各级上实现容错。同时，我很期待读者的反馈，特别是关于如果我打标的地方，如果你发现哪里不对，请反馈给我。我也会在 [Twitter](#) 响应你的反馈，请关注 @hui\_kenneth。

下一篇是关于 Mesos 的持久性存储和容错的。

查看英文原文：[DIGGING DEEPER INTO APACHE MESOS](#)

# 微博“异地多活”部署经验谈

作者 刘道儒

【编者按】近日，InfoQ 专访了阿里巴巴的研究员林昊（花名毕玄），了解了他们的[数据中心异地多活项目的来龙去脉](#)。本文在微博上引起了很多[讨论](#)。新浪微博的高级技术经理刘道儒（[@liudaoru](#)）也总结了微博平台的一些经验。

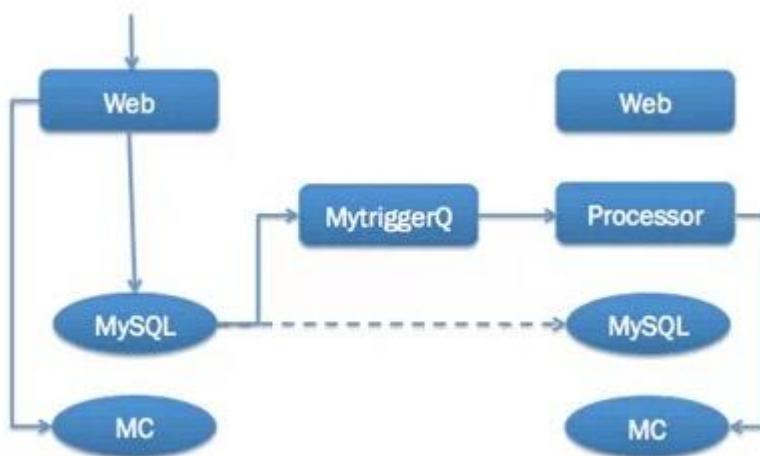
## 正文

异地多活的好处阿里巴巴的同学已经充分阐述，微博的初始出发点包括异地灾备、提升南方电信用户访问速度、提升海外用户访问速度、降低部署成本（北京机房机架费太贵了）等。通过实践，我们发现优势还包括异地容灾、动态加速、流量均衡、在线压测等，而挑战包括增加研发复杂度、增加存储成本等。

## 微博外部历程

先说说微博外部的历程，整个过程可谓是一波多折。微博的主要机房都集中在北京，只有很小一部分业务在广州部署，2010年10月，因微博高速发展，所以准备扩大广州机房服务器规模，并对微博做异地双活部署。

第一版跨机房消息同步方案采取的是基于自研的 MytriggerQ（借助 MySQL 从库的触发器将 INSERT、UPDATE、DELETE 等事件转为消息）的方案，这个方案的好处是，跨机房的消息同步是通过 MySQL 的主从完成的，方案成熟度高。而缺点则是，微博同一个业务会有好几张表，而每张表的信息又不全，这样每发一条微博会有多条消息先后到达，这样导致有较多时序问题，缓存容易花。



第一套方案未能成功，但也让我们认识到跨机房消息同步的核心问题，并促使我们全面下线 MytriggerQ 的消息同步方案，而改用基于业务写消息到 MCQ（[MemcacheQ](#)，新浪自研的一套消息队列，类 MC 协议）的解决方案。

2011 年底在微博平台化完成后，开始启用基于 MCQ 的跨机房消息同步方案，并开发出跨机房消息同步组件 WMB（Weibo Message Broker）。经过与微博 PC 端等部门同学的共同努力，终于在 2012 年 5 月完成 [Weibo.com](#) 在广州机房的上线，实现了“异地双活”。

由于广州机房总体的机器规模较小，为了提升微博核心系统容灾能力，2013 年年中我们又将北京的机房进行拆分，至此微博平台实现了异地三节点的部署模式。依托于此模式，微博具备了在线容量评估、分级上线、快速流量均衡等能力，应对极端峰值能力和应对故障能力大大提升，之后历次元旦、春晚峰值均顺利应对，日常上线导致的故障也大大减少。上线后，根据微博运营情况及成本的需要，也曾数次调整各个机房的服务器规模，但是整套技术上已经基本成熟。

## 异地多活面临的挑战

根据微博的实践，一般做异地多活都会遇到如下的问题：

**机房之间的延时：**微博北京的两个核心机房间延时在 1ms 左右，但北京机房到广州机房则有近 40ms 的延时。对比一下，微博核心 Feed 接口的总平均耗时也就在 120ms 左右。微博 Feed 会依赖几十个服务上百个资源，如果都跨机房请求，性能将会惨不忍睹；

**专线问题：**为了做广州机房外部，微博租了两条北京到广州的专线，成本巨大。同时单条专线的稳定性也很难保障，基本上每个月都会有或大或小的问题；

**数据同步问题：**MySQL 如何做数据同步？HBase 如何做数据同步？还有各种自研的组件，这些统统要做多机房数据同步。几十毫秒的延时，加上路途遥远导致的较弱网络质量（我们的专线每个月都会有或大或小的问题），数据同步是非常大的挑战；

**依赖服务部署问题：**如同阿里巴巴目前只做了交易单元的“异地双活”，微博部署时也面临核心服务过多依赖小服务的问题。将小服务全部部署，改造成本、维护成本过大，不部署则会遇到之前提到的机房之间延时导致整体性能无法接受的问题；

**配套体系问题：**只是服务部署没有流量引入就不能称为“双活”，而要引入流量就要求配套的服务和流程都能支持异地部署，包括预览、发布、测试、监控、降级等都要进行相应改造。

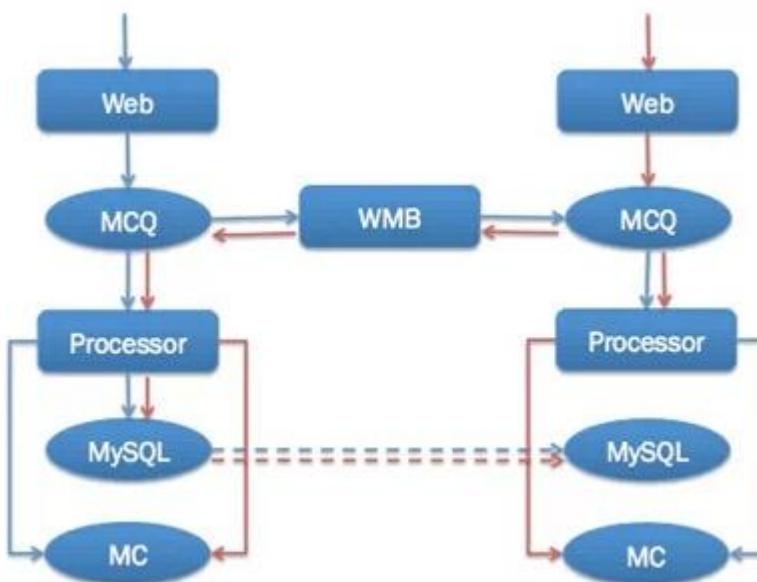
## 微博异地多活解决方案

由于几十毫秒的延时，跨机房服务调用性能很差，异地多活部署的主体服务必须要做数据的冗余存储，并辅以缓存等构成一套独立而相对完整的服务。数据同步有很多层面，

包括消息层面、缓存层面、数据库层面，每一个层面都可以做数据同步。由于基于 MytriggerQ 的方案的失败，微博后来采取的是基于 MCQ 的 WMB 消息同步方案，并通过消息对缓存更新，加上微博缓存高可用架构，可以做到即便数据库同步有问题，从用户体验看服务还是正常的。

这套方案中，每个机房的缓存是完全独立的，由每个机房的 Processor（专门负责消息处理的程序，类 Storm）根据收到的消息进行缓存更新。由于消息不会重复分发，而且信息完备，所以 MytriggerQ 方案存在的缓存更新脏数据问题就解决了。而当缓存不存在时，会穿透到 MySQL 从库，然后进行回种。可能出现的问题是，缓存穿透，但是 MySQL 从库如果此时出现延迟，这样就会把脏数据种到缓存中。我们的解决方案是做一个延时 10 分钟的消息队列，然后由一个处理程序来根据这个消息做数据的重新载入。一般从库延时时间不超过 10 分钟，而 10 分钟内的脏数据在微博的业务场景下也是可以接受的。

微博的异地多活方案如下图（三个节点类似，消息同步都是通过 WMB）：



跟阿里巴巴遇到的问题类似，我们也遇到了数据库同步的问题。由于微博对数据库不是强依赖，加上数据库双写的维护成本过大，我们选择的方案是数据库通过主从同步的方式进行。这套方案可能的缺点是如果主从同步慢，并且缓存穿透，这时可能会出现脏数据。这种同步方式已运行了三年，整体上非常稳定，没有发生因为数据同步而导致的服务故障。从 2013 年开始，微博启用 HBase 做在线业务的存储解决方案，由于 HBase 本身不支持多机房部署，加上早期 HBase 的业务比较小，且有单独接口可以回调北京机房，所以没有做异地部署。到今年由于 HBase 支撑的对象库服务已经成为微博非常核心的基础服务，我们也在规划 HBase 的异地部署方案，主要的思路跟 MySQL 的方案类似，同步也在考虑基于 MCQ 同步的双机房 HBase 独立部署方案。

阿里巴巴选择了单元化的解决方案，这套方案的优势是将用户分区，然后所有这个用户相关的数据都在一个单元里。通过这套方案，可以较好的控制成本。但缺点是除了主维

度（阿里巴巴是买家维度），其他所有的数据还是要做跨机房同步，整体的复杂度基本没降低。另外就是数据分离后由于拆成了至少两份数据，数据查询、扩展、问题处理等成本均会增加较多。总的来讲，个人认为这套方案更适用于 WhatsApp、Instagram 等国外业务相对简单的应用，而不适用于国内功能繁杂、依赖众多的应用。

数据同步问题解决之后，紧接着就要解决依赖服务部署的问题。由于微博平台对外提供的都是 Restful 风格的 API 接口，所以独立业务的接口可以直接通过专线引流回北京机房。但是对于微博 Feed 接口的依赖服务，直接引流回北京机房会将平均处理时间从百毫秒的量级直接升至几秒的量级，这对服务是无法接受的。所以，在 2012 年我们对微博 Feed 依赖的主要服务也做了异地多活部署，整体的处理时间终于降了下来。当然这不是最优的解决方案，但在当时微博业务体系还相对简单的情况下，很好地解决了问题，确保了 2012 年 5 月的广州机房部署任务的达成。

而配套体系的问题，技术上不是很复杂，但是操作时却很容易出问题。比如，微博刚开始做异地多活部署时，测试同学没有在上线时对广州机房做预览测试，曾经导致过一些线上问题。配套体系需要覆盖整个业务研发周期，包括方案设计阶段的是否要做多机房部署、部署阶段的数据同步、发布预览、发布工具支持、监控覆盖支持、降级工具支持、流量迁移工具支持等方方面面，并需开发、测试、运维都参与进来，将关键点纳入到流程当中。

关于为应对故障而进行数据冗余的问题，阿里巴巴的同学也做了充分的阐述，在此也补充一下我们的一些经验。微博核心池容量冗余分两个层面来做，前端 Web 层冗余同用户规模成正比，并预留日常峰值 50% 左右的冗余度，而后端缓存等资源由于相对成本较低，每个机房均按照整体两倍的规模进行冗余。这样如果某一个机房不可用，首先我们后端的资源是足够的。接着我们首先会只将核心接口进行迁移，这个操作分钟级即可完成，同时由于冗余是按照整体的 50%，所以即使所有的核心接口流量全部迁移过来也能支撑住。接下来，我们会把其他服务池的前端机也改为部署核心池前端机，这样在一小时内即可实现整体流量的承接。同时，如果故障机房是负责数据落地的机房，DBA 会将从库升为主库，运维调整队列机开关配置，承接数据落地功能。而在整个过程中，由于我们核心缓存可以脱离数据库支撑一个小时左右，所以服务整体会保持平稳。

## 异地多活最好的姿势

以上介绍了一下微博在异地多活方面的实践和心得，也对比了一下阿里巴巴的解决方案。就像没有完美的通用架构一样，异地多活的最佳方案也要因业务情形而定。如果业务请求量比较小，则根本没有必要做异地多活，数据库冷备足够了。不管哪种方案，异地多活的资源成本、开发成本相比与单机房部署模式，都会大大增加。

以下是方案选型时需要考虑的一些维度：

- 能否整业务迁移：如果机器资源不足，建议优先将一些体系独立的服务整体迁移，这样可以为核心服务节省出大量的机架资源。如果这样之后，机架资源仍然不足，再做异地多活部署。

- 服务关联是否复杂：如果服务关联比较简单，则单元化、基于跨机房消息同步的解决方案都可以采用。不管哪种方式，关联的服务也都要做异地多活部署，以确保各个机房对关联业务的请求都落在本机房内。
- 是否方便对用户分区：比如很多游戏类、邮箱类服务，由于用户可以很方便地分区，就非常适合单元化，而 SNS 类的产品因为关系公用等问题不太适合单元化。
- 谨慎挑选第二机房：尽量挑选离主机房较近（网络延时在 10ms 以内）且专线质量好的机房做第二中心。这样大多数的小服务依赖问题都可以简化掉，可以集中精力处理核心业务的异地多活问题。同时，专线的成本占比也比较小。以北京为例，做异地多活建议选择天津、内蒙古、山西等地的机房。
- 控制部署规模：在数据层自身支持跨机房服务之前，不建议部署超过两个的机房。因为异地两个机房，异地容灾的目的已经达成，且服务器规模足够大，各种配套的设施也会比较健全，运维成本也相对可控。当扩展到三个点之后，新机房基础设施磨合、运维决策的成本等都会大幅增加。
- 消息同步服务化：建议扩展各自的消息服务，从中间件或者服务层面直接支持跨机房消息同步，将消息体大小控制在 10k 以下，跨机房消息同步的性能和成本都比较可控。机房间的数据一致性只通过消息同步服务解决，机房内部解决缓存等与消息的一致性问题。跨机房消息同步的核心点在于消息不能丢，微博由于使用的是 MCQ，通过本地写远程读的方式，可以很方便的实现高效稳定的跨机房消息同步。

## 异地多活的新方向

时间到了 2015 年，新技术层出不穷，之前很多成本很高的事情目前都有了很好的解决方案。接下来我们将在近五年异地多活部署探索的基础上，继续将微博的异地多活部署体系化。

升级跨机房消息同步组件为跨机房消息同步服务。面向业务隔离跨机房消息同步的复杂性，业务只需要对消息进行处理即可，消息的跨机房分发、一致性等由跨机房同步服务保障。且可以作为所有业务跨机房消息同步的专用通道，各个业务均可以复用，类似于快递公司的角色。

推进 Web 层的异地部署。由于远距离专线成本巨大且稳定性很难保障，我们已暂时放弃远程异地部署，而改为业务逻辑近距离隔离部署，将 Web 层进行远程异地部署。同时，计划不再依赖昂贵且不稳定的专线，而借助于通过算法寻找较优路径的方法通过公网进行数据传输。这样我们就可以将 Web 层部署到离用户更近的机房，提升用户的访问性能。依据我们去年做微博 Feed 全链路的经验，中间链路占掉了 90% 以上的用户访问时间，将 Web 层部署的离用户更近，将能大大提升用户访问性能和体验。

借助微服务解决中小服务依赖问题。将对资源等的操作包装为微服务，并将中小业务迁移到微服务架构。这样只需要对几个微服务进行异地多活部署改造，众多的中小业务就不再需要关心异地部署问题，从而可以低成本完成众多中小服务的异地多活部署改造。

利用 Docker 提升前端快速扩容能力。借助 Docker 的动态扩容能力，当流量过大时分钟级从其他服务池摘下一批机器，并完成本服务池的扩容。之后可以将各种资源也纳入到 Docker 动态部署的范围，进一步扩大动态调度的机器源范围。

以上是对微博异地多活部署的一些总结和思考，希望能够对大家有所启发，也希望看到更多的同学分享一下各自公司的异地多活架构方案。

## 关于作者

**刘道儒**，毕业于北京信息工程学院，现任微博平台研发高级技术经理、Feed 项目技术负责人，负责 Feed 体系后端研发，先后在 TRS、搜狗、新浪微博从事社区&社交业务研发，专注于大规模分布式系统的研发、高可用等领域。他曾代表平台在 2012 年主导了微博广州机房部署项目以及北京双机房部署项目。

# Java 字节码忍者禁术

作者 Ben Evans , 译者 邵思华

Java 语言本身是由 Java 语言规格说明 (JLS) 所定义的，而 Java 虚拟机的可执行字节码则是一个完全独立的标准，即 Java 虚拟机规格说明（通常也被称为 VM Spec）所定义的。

JVM 字节码是通过 javac 对 Java 源代码文件进行编译后生成的，生成的字节码与原本的 Java 语言存在着很大的不同。比方说，在 Java 语言中为人熟知的一些高级特性，在编译过程中会被移除，在字节码中完全不见踪影。

这方面最明显的一个例子莫过于 Java 中的各种循环关键字了 (for、while 等等)，这些关键字在编译过程中会被消除，并替换为字节码中的分支指令。这就意味着在字节码中，每个方法内部的流程控制只包含 if 语句与 jump 指令（用于循环）。

在阅读本文前，我假设读者对于字节码已经有了基本的了解。如果你需要了解一些基本的背景知识，请参考《Java 程序员修炼之道》(Well-Grounded Java Developer) 一书（作者为 Evans 与 Verburg，由 Manning 于 2012 年出版），或是来自于 RebelLabs 的这篇[报告](#)（下载 PDF 需要注册）。

让我们来看一下这个示例，它对于还不熟悉的 JVM 字节码的新手来说很可能会感到困惑。该示例使用了 javap 工具，它本质上是一个 Java 字节码的反汇编工具，在下载的 JDK 或 JRE 中可以找到它。在这个示例中，我们将讨论一个简单的类，它实现了 Callable 接口：

```
public class ExampleCallable implements Callable {  
  
    public Double call() {  
  
        return 3.1415;  
  
    }  
  
}
```

我们可以通过对 javap 工具进行最简单形式的使用，对这个类进行反汇编后得到以下结果：

```
$ javap kathik/java/bytocode_examples/ExampleCallable.class  
  
Compiled from "ExampleCallable.java"  
  
public class kathik.java.bytocode_examples.ExampleCallable
```

```
implements java.util.concurrent.Callable {  
  
    public kathik.java bytecode_examples.ExampleCallable();  
  
    public java.lang.Double call();  
  
    public java.lang.Object call() throws java.lang.Exception;  
  
}
```

这个反汇编后的结果看上去似乎是错误的，毕竟我们只写一个 `call` 方法，而不是两个。而且即使我们尝试手工创建这两个方法，`javac` 也会提示，代码中有两个具有相同名称和参数的方法，它们仅有返回类型的不同，因此这段代码是无法编译的。然而，这个类确确实实是由上面那个真实的、有效的 Java 源文件所生成的。

这个示例能够清晰地表明在使用 Java 中广为人知的一种限制：不可对返回类型进行重载，其实这只是 Java 语言的一种限制，而不是 JVM 字符码本身的强制要求。`javac` 确实会在代码中插入一些不存在于原始的类文件中的内容，如果你为此感到担忧，那大可放心，因为这种事每时每刻都在发生！每一位 Java 程序员最先学到的一个知识点就是：“如果你不提供一个构造函数，那么编译器会为你自动添加一个简单的构造函数”。在 `javap` 的输出中，你也能看到其中有一个构造函数存在，而它并不存在于我们的代码中。

这些额外的方法从某种程度上表明，语言规格说明的需求比 VM 规格说明中的细节更为严格。如果我们能够直接编写字节码，就可以实现许多“不可能”实现的功能，而这种字节码虽然是合法的，却没有任何一个 Java 编译器能够生成它们。

举例来说，我们可以创建出完全不含构造函数的类。Java 语言规格说明中要求每个类至少要包含一个构造函数，而如果我们在代码中没有加入构造函数，`javac` 会自动加入一个简单的 `void` 构造函数。但是，如果我们能够直接编写字节码，我们完全可以忽略构造函数。这种类是无法实例化的，即使通过反射也不行。

我们的最后一个例子已经接近成功了，但还是差一口气。在字节码中，我们可以编写一个方法，它将试图调用一个其它类中定义的私有方法。这段字节码是有效的，但如果任何程序打算加载它，它将无法正确地进行链接。这是因为在类型加载器中（`classloader`）的校验器会检测出这个方法调用的访问控制限制，并且拒绝这个非法访问。

## 介绍 ASM

如果我们打算在创建的代码中实现这些超越 Java 语言的行为，那就需要完全手动创建这样的一个类文件。由于这个类文件的格式是两进制的，因此可以选择使用某种类库，它能够让我们对某个抽象的数据结构进行操作，随后将其转换为字节码，并通过流方式将其写入磁盘。

具备这种功能的类库有多个选择，但在本文中我们将关注于 ASM。这是一个非常常见的类库，在 Java 8 分发包中有一个以内部 API 的形式提供的版本（其内容稍有不同）。对于用户代码来说，我们选择使用通用的开源类库，而不是 JDK 中提供的版本，毕竟我们不应当依赖于内部 API 来实现所需的功能。

ASM 的核心功能在于，它提供了一种 API，虽然它看上去有些神秘莫测（有时也会显得有些粗糙），但能够以一种直接的方式反映出字节码的数据结构。

我们看到的 Java 运行时是由多年之前的各种设计决策所产生的结果，而在后续各个版本的类文件格式中，我们能够清晰地看到各种新增的内容。

ASM 致力于尽量使构建的类文件接近于真实形态，因此它的基础 API 会分解为一系列相对简单的方法片段（而这些片段正是用于建模的二进制所关注的）。

如果程序员打算完全手动编写类文件，就必需理解类文件的整体结构，而这种结构是会随时改变的。幸运的是，ASM 能够处理多个不同 Java 版本中的类文件格式之间的细微差别，而 Java 平台本身对于可兼容性的高要求也侧面帮助了我们。

一个类文件依次包含以下内容：

- 某个特殊的数字（在传统的 Unix 平台上，Java 中的特殊数字是这个历史悠久的、人见人爱的 0xCAFEBAE）；
- 正在使用中的类文件格式版本号；
- 常量；
- 访问控制标记（例如类的访问范围是 public、protected 还是 package 等）；
- 该类的类型名称；
- 该类的超类；
- 该类所实现的接口；
- 该类拥有的字段（处于超类中的字段上方）；
- 该类拥有的方法（处于超类中的方法上方）；
- 属性（类级别的注解）。

可以用下面这个方法帮助你记忆 JVM 类文件中的主要部分：



My	Very	Cute	Animal	Turns	Savage	In	Full	Moon	Areas
M	V	C	A	T	S	I	F	M	A
Magic	Version	Constant	Access	This	Super	Interfaces	Fields	Methods	Attributes

ASM 中提供了两个 API，其中最简单的那个依赖于访问者模式。在常见的形式中，ASM 只包含最简单的字段以及 ClassWrite 类（当已经熟悉了 ASM 的使用和直接操作字节码的方式之后，许多开发者会发现 CheckClassAdapter 是一个很实用的起点，作为一个 ClassVisitor，它对代码进行检查的方式，与 Java 的类加载子系统中的校验器的工作方式非常想像。）

让我们看几个简单的类生成的例子，它们都是按照常规的模式创建的：

- 启动一个 ClassVisitor（在我们的示例中就是一个 ClassWriter）；
- 写入头信息；
- 生成必要的方法和构造函数；
- 将 ClassVisitor 转换为字节数组，并写入输出。

## 示例

```
public class Simple implements ClassGenerator {

    // Helpful constants

    private static final String GEN_CLASS_NAME = "GetterSetter";

    private static final String GEN_CLASS_STR = PKG_STR + GEN_CLASS_NAME;
```

```
@Override

public byte[] generateClass() {

    ClassWriter cw = new ClassWriter(0);

    CheckClassAdapter cv = new CheckClassAdapter(cw);

    // Visit the class header

    cv.visit(V1_7, ACC_PUBLIC, GEN_CLASS_STR, null, J_L_O, new String[0]);

    generateGetterSetter(cv);

    generateCtor(cv);

    cv.visitEnd();

    return cw.toByteArray();

}

private void generateGetterSetter(ClassVisitor cv) {

    // Create the private field myInt of type int. Effectively:

    // private int myInt;

    cv.visitField(ACC_PRIVATE, "myInt", "I", null, 1).visitEnd();

    // Create a public getter method

    // public int getMyInt();

    MethodVisitor getterVisitor =

        cv.visitMethod(ACC_PUBLIC, "getMyInt", "()I", null, null);
```

```
// Get ready to start writing out the bytecode for the method

getterVisitor.visitCode();

// Write ALOAD_0 bytecode (push the this reference onto stack)

getterVisitor.visitVarInsn(ALOAD, 0);

// Write the GETFIELD instruction, which uses the instance on

// the stack (& consumes it) and puts the current value of the

// field onto the top of the stack

getterVisitor.visitFieldInsn(GETFIELD, GEN_CLASS_STR, "myInt", "I");

// Write IRETURN instruction - this returns an int to caller.

// To be valid bytecode, stack must have only one thing on it

// (which must be an int) when the method returns

getterVisitor.visitInsn(IRETURN);

// Indicate the maximum stack depth and local variables this

// method requires

getterVisitor.visitMaxs(1, 1);

// Mark that we've reached the end of writing out the method

getterVisitor.visitEnd();

// Create a setter

// public void setMyInt(int i);

MethodVisitor setterVisitor =

cv.visitMethod(ACC_PUBLIC, "setMyInt", "(I)V", null, null);
```

```
setterVisitor.visitCode();

// Load this onto the stack

setterVisitor.visitVarInsn(ALOAD, 0);

// Load the method parameter (which is an int) onto the stack

setterVisitor.visitVarInsn(ILOAD, 1);

// Write the PUTFIELD instruction, which takes the top two

// entries on the execution stack (the object instance and

// the int that was passed as a parameter) and set the field

// myInt to be the value of the int on top of the stack.

// Consumes the top two entries from the stack

setterVisitor.visitFieldInsn(PUTFIELD, GEN_CLASS_STR, "myInt", "I");

setterVisitor.visitInsn(RETURN);

setterVisitor.visitMaxs(2, 2);

setterVisitor.visitEnd();

}

private void generateCtor(ClassVisitor cv) {

// Constructor bodies are methods with special name

MethodVisitor mv =

cv.visitMethod(ACC_PUBLIC, INSTCTOR, VOID_SIG, null, null);

mv.visitCode();

mv.visitVarInsn(ALOAD, 0);
```

```
// Invoke the superclass constructor (we are basically  
  
// mimicing the behaviour of the default constructor  
  
// inserted by javac)  
  
// Invoking the superclass constructor consumes the entry on the top  
// of the stack.  
  
mv.visitMethodInsn(INVOKESTATIC, J_L_O, INST_CTOR, VOID_SIG);  
  
// The void return instruction  
  
mv.visitInsn(RETURN);  
  
mv.visitMaxs(2, 2);  
  
mv.visitEnd();  
  
}  
  
  
@Override  
  
public String getGenClassName() {  
  
    return GEN_CLASS_NAME;  
  
}  
  
}
```

这段代码使用了一个简单的接口，用一个单一的方法生成类的字节，一个辅助方法以返回生成的类名，以及一些实用的常量：

```
interface ClassGenerator {  
  
    public byte[] generateClass();  
  
  
    public String getGenClassName();
```

```
// Helpful constants

public static final String PKG_STR = "kathik/java/bytocode_examples/";

public static final String INST_CTOR = "";

public static final String CL_INST_CTOR = "";

public static final String J_L_O = "java/lang/Object";

public static final String VOID_SIG = "()V;

}
```

为了驾驭生成的类，我们需要使用一个 `harness` 类，它叫做 `Main`。`Main` 类提供了一个简单的类加载器，并且提供了一种反射式的方式对生成类中的方法进行回调。为了简便起见，我们将生成的类定入 `Maven` 的目标文件夹的正确位置，让 `IDE` 中的 `classpath` 能够顺利地找到它：

```
public class Main {

    public static void main(String[] args) {

        Main m = new Main();

        ClassGenerator cg = new Simple();

        byte[] b = cg.generateClass();

        try {

            Files.write(Paths.get("target/classes/" + PKG_STR +

                cg.getGenClassName() + ".class"), b, StandardOpenOption.CREATE);

        } catch (IOException ex) {

            Logger.getLogger(Simple.class.getName()).log(Level.SEVERE, null, ex);

        }

        m.callReflexive(cg.getGenClassName(), "getMyInt");

    }

}
```

下面的类提供了一种方法，能够对受保护的 `defineClass()` 进行访问，这样一来我们就能够将一个字节数组转换为某个类对象，以便在反射中使用。

```
private static class SimpleClassLoader extends ClassLoader {

    public Class simpleDefineClass(byte[] clazzBytes) {

        return defineClass(null, clazzBytes, 0, clazzBytes.length);
    }
}

private void callReflexive(String typeName, String methodName) {
    byte[] buffy = null;

    try {
        buffy = Files.readAllBytes(Paths.get("target/classes/" + PKG_STR +
            typeName + ".class"));

        if (buffy != null) {
            SimpleClassLoader myCl = new SimpleClassLoader();

            Class newClz = myCl.simpleDefineClass(buffy);

            Object o = newClz.newInstance();

            Method m = newClz.getMethod(methodName, new Class[0]);

            if (o != null && m != null) {
                Object res = m.invoke(o, new Object[0]);

                System.out.println("Result: " + res);
            }
        }
    } catch (IOException | InstantiationException | IllegalAccessException |

```

```
NoSuchMethodException | SecurityException |  
  
IllegalArgumentException | InvocationTargetException ex) {  
  
    Logger.getLogger(Simple.class.getName()).log(Level.SEVERE, null, ex);  
  
}  
  
}
```

有了这个类以后，我们只要通过细微的改动，就可以方便地测试各种不同的类生成器，以此对字节码生成器的各个方面进行探索。

实现无构造函数的类的方式也很相似。举例来说，以下这种方式可以在生成的类中仅包含一个静态字段，以及它的 getter 和 setter（生成器不会调用 generateCtor()方法）：

```
private void generateStaticGetterSetter(ClassVisitor cv) {  
  
    // Generate the static field  
  
    cv.visitField(ACC_PRIVATE | ACC_STATIC, "myStaticInt", "I", null,  
    1).visitEnd();  
  
  
  
  
    MethodVisitor getterVisitor = cv.visitMethod(ACC_PUBLIC | ACC_STATIC,  
    "getMyInt", "()I", null, null);  
  
    getterVisitor.visitCode();  
  
    getterVisitor.visitFieldInsn(GETSTATIC, GEN_CLASS_STR, "myStaticInt", "I");  
  
  
  
  
    getterVisitor.visitInsn(IRETURN);  
  
    getterVisitor.visitMaxs(1, 1);  
  
    getterVisitor.visitEnd();
```

```
MethodVisitor setterVisitor = cv.visitMethod(ACC_PUBLIC | ACC_STATIC,
"setMyInt",

        "(I)V", null, null);

setterVisitor.visitCode();

setterVisitor.visitVarInsn(ILOAD, 0);

setterVisitor.visitFieldInsn(PUTSTATIC, GEN_CLASS_STR, "myStaticInt", "I");

}

setterVisitor.visitInsn(RETURN); setterVisitor.visitMaxs(2, 2); setterVisitor.visitEnd();
```

请留意一下该方法在生成时使用了 ACC\_STATIC 标记，此外还请注意方法的参数是位于本地变量列表中的最前面的（这里使用的 ILOAD 0 模式暗示了这一点——而在生成实例方法时，此处应该改为 ILOAD 1，这是因为实例方法中的“this”引用存储在本地变量表中的偏移量为 0）。

通过使用 javap，我们就能够确认在生成的类中确实不包括任何构造函数：

```
$ javap -c kathik/java/bytocode_examples/StaticOnly.class

public class kathik.StaticOnly {

    public static int getMyInt(); Code:

        0: getstatic    #11           // Field myStaticInt:I

        3: ireturn

    public static void setMyInt(int); Code:

        0: iload_0

        1: putstatic    #11           // Field myStaticInt:I

        4: return

}
```

## 使用生成的类

目前为止，我们是使用反射的方式调用我们通过 ASM 所生成的类的。这有助于保持这个示例的自包含性，但在很多情况下，我们希望能够将这些代码生成在常规的 Java 文件中。要实现这一点非常简单。以下示例将生成的类保存在 Maven 的目标目录下，写法很简单：

```
$ cd target/classes

$ jar cvf gen-asm.jar kathik/java/bytocode_examples/GetterSetter.class
kathik/java/bytocode_examples/StaticOnly.class

$ mv gen-asm.jar ../../lib/gen-asm.jar
```

这样一来我们就得到了一个 JAR 文件，可以作为依赖项在其它代码中使用。比方说，我们可以这样使用这个 GetterSetter 类：

```
import kathik.java.bytocode_examples.GetterSetter;

public class UseGenCodeExamples {

    public static void main(String[] args) {

        UseGenCodeExamples ugcx = new UseGenCodeExamples();

        ugcx.run();
    }

    private void run() {

        GetterSetter gs = new GetterSetter();

        gs.setMyInt(42);

        System.out.println(gs.getMyInt());
    }
}
```

这段代码在 IDE 中是无法通过编译的（因为 GetterSetter 类没有配置在 classpath 中）。但

如果我们直接使用命令行，并且在 classpath 中指向正确的依赖，就可以正确地运行了：

```
$ cd ../../src/main/java/  
  
$ javac -cp ../../lib/gen-asm.jar  
kathik/java/bytocode_examples/withgen/UseGenCodeExamples.java  
  
$ java -cp ../../lib/gen-asm.jar  
kathik.java.bytocode_examples.withgen.UseGenCodeExamples
```

42

## 结论

在本文中，我们通过使用 ASM 类库中所提供的简单 API，学习了完全手动生成类文件的基础知识。我们也为读者展示了 Java 语言和字节码有哪些不同的要求，并且了解到 Java 中的某些规则其实只是语言本身的规范，而不是运行时所强制的要求。我们还看到，一个正确编写的手工类文件可以直接在语言中使用，与通过 javac 生成的文件没有区别。这一点也是 Java 与其它非 Java 语言，例如 Groovy 或 Scala 进行互操作的基础。

这方面的应用还有许多高级技巧，通过本文的学习，读者应该已经掌握了基本的知识，并且能够进一步深入研究 JVM 的运行时，以及如何对它进行各种操作的技术。

## 关于作者

**Ben Evans** 是 Java/JVM 性能分析初创公司 jClarity 的 CEO。在业余时间他是伦敦 Java 社区的领导者之一并且是 Java 社区进程执行委员会的一员。之前的项目经验包括谷歌 IPO 的性能测试，金融交易系统，为 90 年代一些最大的电影编写备受好评的网站，以及其他。

查看英文原文：[Secrets of the Bytecode Ninjas](#)

# 专访 ThoughtWorks 王磊： 从单块架构到微服务架构

作者 姚琪琳

微服务架构是近年社区讨论比较多的话题，为此，InfoQ 在 [QCon 北京 2015 大会](#) 上策划了“[微服务架构”专题](#)。

ThoughtWorks 的首席咨询师[王磊](#)是国内较早倡导和实践微服务的先行者。他将在 QCon 北京的“微服务架构”专题中分享[《使用微服务架构改造企业核心业务系统的实践》](#)。

王磊是开源软件的爱好者和贡献者，社区活动的参与者，[《Ruby Gems 开发实战》\(Practical RubyGems\)](#)一书的译者，GDCR 西安的组织者。他于 2012 年加入 ThoughtWorks，为国内外诸多客户提供项目交付和咨询服务；在加入 ThoughtWorks 之前，曾就职过多家知名外企，具有丰富的敏捷项目实战经验。目前致力于微服务架构、高可用的 Web 应用以及 DevOps 的研究与实践。

InfoQ 在会前对他进行了专访。

**InfoQ：**首先，您能谈谈微服务这个概念提出的背景吗？

**王磊：**我觉得主要基于三个方面，互联网行业的快速发展，敏捷、精益方法论的深入人心以及传统 IT 系统面临互联网的挑战。过去的十年中，互联网对我们的生活产生了翻天覆地的变化。网上购物、网上订餐、网上支付，想到的，想不到的活动都可以在网上进行，越来越多的公司开始依赖互联网技术打造其核心的竞争优势。在这种情况下，如何快速响应用户的需求，如何用有效的技术服务于用户、并为用户持续提供价值，逐渐成为决定企业是否具有市场竞争力的重要因素之一。

纵观 IT 行业过去这些年中敏捷、精益、持续交付等价值观、方法论的提出以及实践，归根到底也是在围绕着如何帮助企业应变市场变化、提高对市场的响应力以及可持续发展能力。Lean Startup 帮助组织分析并建立最小 MVP，通过迭代持续改进；敏捷方法论帮助组织消除浪费，通过反馈不断找到正确的方向；持续交付帮助组织构建更快、更可靠、可频繁发布的部署和交付机制；云、虚拟化和基础设施自动化(Infrastructure As Code)的使用则极大的简化环境的创建、配置、安装以及部署；DevOPS 文化的推行更是打破了传统开发与运维之间的壁垒，帮助组织形成从开发、测试到部署、运维这样一个全功能化的高效能团队。经过这些方法论、价值观，或者说流程改进实践的推行和尝试后，在宏观上已经基本上形成了一套可遵循、可参考、可实施的体系。这时候，逐渐完善并改进各个细节的需求就会更加强烈。所谓细节，就是类似如何找到灵活性高、扩展性好的架构方式、如何用更合适的技术解决业务难题等。譬如说，在持续交付流水线已经趋于可靠、稳定的前提下，如何在技术上、架构上进行优化，进一步缩短产品构建的周期、进一步缩短自动化测试的周期等。

互联网时代的产品通常有几类特点：创新成本低、需求变化快，用户群体庞大，它和几年前我们熟悉的传统企业的 IT 系统——我们一般称为单块架构应用(逻辑分层、功能集中、代码和数据中心化，并且运行在同一进程的应用程序)——有着本质的不同。这类单块架构系统随着需求变化快，用户访问量增加，面临着越来越多的挑战。譬如说，随着功能的增多，代码量逐渐增多，产品的维护成本、人员的培养成本、缺陷修复成本、技术架构演进的成本、系统扩展成本等都在增加。因此，在面临这种挑战下，如何找到一种更有效的、更灵活、更适应当前时代需求的应用架构方式就成了大家关注的焦点。

所以说，微服务的诞生并不是偶然，而是多重因素推动下的一个必然结果。

### InfoQ：它能解决我们哪些痛点呢？

**王磊：**基本上，从我个人的观点，我认为微服务的出现解决了单块架构面临的以下一些挑战。

#### 痛点 1：技术架构/平台升级难

传统的单块架构系统倾向采用统一的技术平台或方案来解决所有问题。而微服务的异构性，可以针对不同的业务特征选择不同的技术方案，有针对性的解决具体的业务问题。

对于单块架构的系统，初始的技术选型严重限制将来采用不同语言或框架的能力。如果想尝试新的编程语言或者框架，没有完备的功能测试集，很难平滑的完成替换，而且系统规模越大，风险越高。基于微服务架构，使我们更容易在遗留系统上尝试新的技术或解决方案。譬如说，可以先挑选风险最小的服务作为尝试，快速得到反馈后再决定是否试用于其他服务。这也意味着，即便对一项新技术的尝试失败，也可以抛弃这个方案，并不会对整个产品带来风险。

#### 痛点 2：测试、部署成本高

单块架构系统运行在一个进程中，因此系统中任何程序的改变，都需要对整个系统重新测试并部署。而对于微服务架构而言，不同服务之间的打包、测试或者部署等，与其它服务都是完全独立的。对某个服务所做的改动，只需要关注该服务本身。从这个角度来说，使用微服务后，代码修改、测试、打包以及部署的成本和风险都比单块架构系统降低很多。

#### 痛点 3：可伸缩性差

单块架构系统由于单进程的局限性，水平扩展时只能基于整个系统进行扩展，无法针对某一个功能模块按需扩展。而服务架构则可以完美地解决伸缩性的扩展问题。系统可以根据需要，实施细粒度的自由扩展。

#### 痛点 4：构建全功能团队难

康威定律指出：一个组织的设计成果，其结构往往对应于这个组织中的沟通结构。传统的开发模式在分工时往往以技术为单位，比如 UI 团队、服务端团队和数据库团队，这样的分工可能会导致任何功能上的改变都需要跨团队沟通和协调。而微服务则倡导围绕服务来分工，团队需要具备服务设计、开发、测试到部署所需的所有技能。

### 痛点 5：异常破坏性大

微服务架构同时也能提升错误的隔离性。例如，如果某个服务的内存泄露，只会影响自己，其他服务能够继续正常地工作。与之形成对比的是，单块架构中如果有一个不合格的组件发生异常，有可能会拖垮整个系统。

**InfoQ：**对于一个已有的单块架构，如何才能恰当地拆分成多个微服务？在一个微服务架构中，如何才能避免出现 **monster service**？因为如果业务稍微复杂的话，很难如愿地划分出高内聚低耦合的微服务，很多服务之间会有千丝万缕的联系。把它们放在一起容易产生 **monster service**，拆分出来相互之间的调用又过于频繁。比如在一个订单系统中，**Order Service** 可能会调用 **Product Service**、**Inventory Service** 和 **Customer Service**，而 **Inventory Service** 可能也会调用 **Product Service**。如果系统越发复杂，各个服务之间的联系更加紧密的时候，微服务的粒度应该是什么样的？

**王磊：**首先，我认为微服务的‘微’并不是一个真正可衡量、看得见、摸得着的‘微’。这个‘微’所表达的，是一种设计思想和指导方针，是需要团队或者组织共同努力找到的一个平衡点。譬如说，有人觉得用代码行数来作为‘微’的衡量标准比较合适，也有人觉得微服务就应该逻辑简单，应该花费个 2 周的时间就能重写。我们知道对于不同的语言，完成同样功能的代码行数完全不一样，因此数字显然无法成为衡量微服务是否够微的因素。而 2 周时间，对个人而言，和其经验、业务背景、技术的熟悉程度都有关系，所以也无法成为衡量的标准。因此，微服务到底有多微，是个仁者见仁，智者见智的问题，最重要的是，团队觉得合适就好。

另外，微服务的实践，对运维和部署流水线要求非常高。微服务的粒度越细，就意味着需要部署的业务单元就越多，业务单元越多，就需要更稳定的自动化机制，能够创建运行环境，安装依赖，部署应用等；同时，由于生产环境上可运行的业务单元增多，登录服务器节点，查看日志，发现问题已经变得不现实。因此如何建立高效的监控、报警机制也是一个值得探索的话题。除此之外，单块架构的集成测试、功能测试维护起来都不容易，在分布式环境下、多个业务单元相互协同工作的微服务架构下，如何有效做集成测试以及测试策略会发生怎么样的改变，也是需要组织或者团队仔细思考的问题。当然，随着微服务数量的增多，最复杂的还是如何管理这些服务以及可视化服务之间的依赖，传统的 SOA、WebService 定义了类似服务注册、服务查找等机制，这是一种解决方式，但会形成‘中心’点，因为服务之间需要先从中心点找到需要协作的服务，再开始互相通信。我们目前采用的方式是通过对两两服务间进行测试，并上传测试成功后的接口数据，然后采用其他工具根据这些数据画出整体的服务依赖图。目前感觉这种方式现阶段还比较适合。所以，微服务的粒度也需要团队内部根据实际的运维能力出发，切忌为了微服务而微服务。

对于已有的单块架构而言，拆分微服务其实有很多的考虑因素，例如

- 业务的独立性
- 业务单元可能的扩展性
- 组织的运维能力
- 组织的持续交付能力
- 微服务管理的成本

另外，对于微服务的划分，可以基于业务模型，例如您所说的 Product、Customer、Inventory 等，也可以基于具体的业务行为，例如 Checkout、发送 Email 等，还可以考虑使用 Domain Driven Design 提倡的 BoundedContext 来做划分。

至于微服务的粒度，按照 Gartner 的企业系统分层，系统其实是分成三类的：数据系统，核心系统和创新系统。对于数据系统，考虑到需求不会那么频繁的变化，而且数据量也不一定很大，但考虑到需要企业长期维护，因此微服务的易维护性能发挥很大的作用，但也许可扩展性不是重要的考虑因素。对于核心系统，可能和数据系统类似，但更多需要考虑的是不同系统之间的集成，因此，如何有效集成不同系统，尽量不应该依赖于单一数据库作为集成点，可能是粒度划分的重要因素。对于创新性要求较高的系统，一般要求变化频繁，更新频繁，因此更能体现其灵活性、易维护性、独立性的价值。我们目前的助力客户的核心业务系统，其实是核心系统和创新系统的结合体。

**InfoQ：**相比单块架构，微服务是把进程内的调用转换成了服务之间的 HTTP 调用，是否会由此产生性能问题？

**王磊：**同单块架构相比，微服务更强调分布式的、跨进程的调用。由于分布式系统设计本身的复杂性，需要考虑到网络、带宽、延迟，以及容错性、数据一致性等，因此，如果只考虑接口调用之间的成本，分布式调用的成本肯定会远远高于进程内的调用成本。但是如果我们把范围放大了看，从整个产品或者应用的角度出发，因为微服务以及分布式带来的优势是易扩展性、灵活性以及独立性，因此，我们需要根据实际业务、根据组织的运维能力，找到一个实施微服务的平衡点。

**InfoQ：**如果某个服务的 API 发生改变的话，如何确保其消费者仍然能够得到正确的数据？能否简要介绍一下你们是如何编写契约测试的？

**王磊：**对，基于契约测试的机制就是为了解决这个问题的。

目前，我们是采用 [PACT](#) 作为契约测试工具。PACT 是采用 Ruby 实现的一个契约测试工具，个人感觉从设计的思路、实现方式、易用性等方面来看，它非常棒，另外文档也很全面。实际上，PACT 也是随着 ThoughtWorks 在助力客户（澳洲最大的房产互联网门户）使用微服务改造后台业务系统的过程中所诞生的，它的灵感是来自微服务架构下如何解决服务调用者和服务提供者之间集成测试难的问题。关于如何编写 PACT 测试，官方的文档已经很全面了，所以，有兴趣的朋友们可以了解一下。

另外，ThoughtWorks 还开源了一个叫 [PACTO](#) 的契约测试工具，也是用 Ruby 实现的，不过我在项目中并没有使用过，感兴趣的朋友也可以了解一下。

### InfoQ：微服务架构是否适用于互联网应用？

**王磊：**我觉得是非常适合的。因为从微服务诞生的背景、时机，到微服务本身的特点和优势来看，微服务作为一种应用架构模式，是非常适合互联网的快速变化的。一方面，微服务能满足互联网快速变化的需求，另外一方面，微服务也能降低技术选型、包括技术替换的风险，除此之外，微服务是真正意义上的按需扩展，我们可以根据对系统功能进行合适粒度的划分，来决定什么样的功能需要扩展，以及怎么样的水平扩展。

我们知道，互联网不仅市场变化快，而且技术方案变化也快。譬如，短短几年时间，光前端 JavaScript 的框架，就出现了好几十个，从早一点的 Backbone、Ember 到 AngularJS、ReactJS 等等，类似的后端的框架、工具，数据存储系统等也是层出不穷。

其实，业界著名的使用微服务架构例子，大部分都是来自互联网行业。譬如 Amazon、Netflix、Gilt 以及 REA 等等。不过，他们也是在经过了长时间对持续交付、DevOps 以及运维方面的实践和经验积累之后，逐渐地、持续地采用微服务架构的。

InfoQ：感谢您接受我们的采访，期待您在 QCon 上的精彩分享。

# 高效运维最佳实践（03）： Redis 集群技术及 Codis 实践

作者 萧田国

## 专栏介绍

“高效运维最佳实践”是 InfoQ 在 2015 年推出的精品专栏，由触控科技运维总监萧田国撰写，InfoQ 总编辑崔康策划。

## 前言

诚如开篇文章所言，高效运维包括管理的专业化和技术的专业化。前两篇我们主要在说些管理相关的内容，本篇说一下技术专业化。希望读者朋友们能适应这个转换，谢谢。

互联网早在几年前就已进入 Web 2.0 时代，对后台支撑能力的要求，提高了几十倍甚至几百倍。在这个演化过程中，缓存系统扮演了举足轻重的角色。

运维进化到今天，已经不是重复造轮子的时代。所以，我们在架构优化和自动化运维中，可以尽可能地选用优秀的开源产品，而不是自己完全从头再来（各种技术 geek 除外）。

本文主要讨论 Redis 集群相关技术及新发展，关于 Redis 运维等内容，以后另开主题讨论。

本文重点推荐 Codis——豌豆荚开源的 Redis 分布式中间件（该项目于 4 个月前在 GitHub 开源，目前 star 已超过 2100）。其和 Twemproxy 相比，有诸多激动人心的新特性，并支持从 Twemproxy 无缝迁移至 Codis。

本文主要目录如下，对 Redis 比较了解的朋友，可跳过前两部分，直接欣赏 Codis 相关内容。

- 1. Redis 常见集群技术
  - 1.1 客户端分片
  - 1.2 代理分片
  - 1.3 Redis Cluster
- 2. Twemproxy 及不足之处
- 3. Codis 实践
  - 3.1 体系架构
  - 3.2 性能对比测试
  - 3.3 使用技巧、注意事项

好吧我们正式开始。

## 1. Redis 常见集群技术

长期以来，Redis 本身仅支持单实例，内存一般最多 10~20GB。这无法支撑大型线上业务系统的需求。而且也造成资源的利用率过低——毕竟现在服务器内存动辄 100~200GB。

为解决单机承载能力不足的问题，各大互联网企业纷纷出手，“自助式”地实现了集群机制。在这些非官方集群解决方案中，物理上把数据“分片”（sharding）存储在多个 Redis 实例，一般情况下，每一“片”是一个 Redis 实例。

包括官方近期推出的 Redis Cluster，Redis 集群有三种实现机制，分别介绍如下，希望对大家选型有所帮助。

### 1.1 客户端分片

这种方案将分片工作放在业务程序端，程序代码根据预先设置的路由规则，直接对多个 Redis 实例进行分布式访问。这样的好处是，不依赖于第三方分布式中间件，实现方法和代码都自己掌控，可随时调整，不用担心踩到坑。

这实际上是一种静态分片技术。Redis 实例的增减，都得手工调整分片程序。基于此分片机制的开源产品，现在仍不多见。

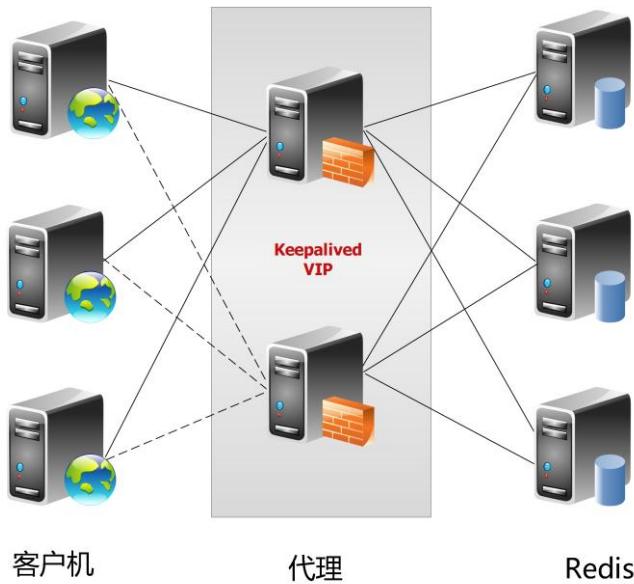
这种分片机制的性能比代理式更好（少了一个中间分发环节）。但缺点是升级麻烦，对研发人员的个人依赖性强——需要有较强的程序开发能力做后盾。如果主力程序员离职，可能新的负责人，会选择重写一遍。

所以，这种方式下，可运维性较差。出现故障，定位和解决都得研发和运维配合着解决，故障时间变长。

这种方案，难以进行标准化运维，不太适合中小公司（除非有足够的 DevOPS）。

### 1.2 代理分片

这种方案，将分片工作交给专门的代理程序来做。代理程序接收到来自业务程序的数据请求，根据路由规则，将这些请求分发给正确的 Redis 实例并返回给业务程序。



这种机制下，一般会选用第三方代理程序（而不是自己研发），因为后端有多个 Redis 实例，所以这类程序又称为分布式中间件。

这样的好处是，业务程序不用关心后端 Redis 实例，运维起来也方便。虽然会因此带来些性能损耗，但对于 Redis 这种内存读写型应用，相对而言是能容忍的。

这是我们推荐的集群实现方案。像基于该机制的开源产品 Twemproxy，便是其中代表之一，应用非常广泛。

### 1.3 Redis Cluster

在这种机制下，没有中心节点（和代理模式的重要不同之处）。所以，一切开心和不开心的事情，都将基于此而展开。

Redis Cluster 将所有 Key 映射到 16384 个 Slot 中，集群中每个 Redis 实例负责一部分，业务程序通过集成的 Redis Cluster 客户端进行操作。客户端可以向任一实例发出请求，如果所需数据不在该实例中，则该实例引导客户端自动去对应实例读写数据。

Redis Cluster 的成员管理（节点名称、IP、端口、状态、角色）等，都通过节点之间两两通讯，定期交换并更新。

由此可见，这是一种非常“重”的方案。已经不是 Redis 单实例的“简单、可依赖”了。可能这也是延期多年之后，才近期发布的原因之一。

这令人想起一段历史。因为 Memcache 不支持持久化，所以有人写了一个 Membase，后来改名叫 Couchbase，说是支持 Auto Rebalance，好几年了，至今都没多少家公司在使用。

这是个令人忧心忡忡的方案。为解决仲裁等集群管理的问题，Oracle RAC 还会使用存储设备的一块空间。而 Redis Cluster，是一种完全的去中心化……

本方案目前不推荐使用，从了解的情况来看，线上业务的实际应用也并不多见。

## 2. Twemproxy 及不足之处

Twemproxy 是一种代理分片机制，由 Twitter 开源。Twemproxy 作为代理，可接受来自多个程序的访问，按照路由规则，转发给后台的各个 Redis 服务器，再原路返回。

这个方案顺理成章地解决了单个 Redis 实例承载能力的问题。当然，Twemproxy 本身也是单点，需要用 Keepalived 做高可用方案。

我想很多人都应该感谢 Twemproxy，这么些年来，应用范围最广、稳定性最高、最经考验的分布式中间件，应该就是它了。只是，他还有诸多不方便之处。

Twemproxy 最大的痛点在于，无法平滑地扩容/缩容。

这样导致运维同学非常痛苦：业务量突增，需增加 Redis 服务器；业务量萎缩，需要减少 Redis 服务器。但对 Twemproxy 而言，基本上都很难操作(那是一种锥心的、纠结的痛……)。

或者说，Twemproxy 更加像服务器端静态 sharding。有时为了规避业务量突增导致的扩容需求，甚至被迫新开一个基于 Twemproxy 的 Redis 集群。

Twemproxy 另一个痛点是，运维不友好，甚至没有控制面板。

Codis 刚好击中 Twemproxy 的这两大痛点，并且提供诸多其他令人激赏的特性。

## 3. Codis 实践

Codis 由豌豆荚于 2014 年 11 月开源，基于 Go 和 C 开发，是近期涌现的、国人开发的优秀开源软件之一。现已广泛用于豌豆荚的各种 Redis 业务场景（已得到豌豆荚@刘奇同学的确认，呵呵）。

从 3 个月的各种压力测试来看，稳定性符合高效运维的要求。性能更是改善很多，最初比 Twemproxy 慢 20%；现在比 Twemproxy 快近 100%（条件：多实例，一般 Value 长度）。

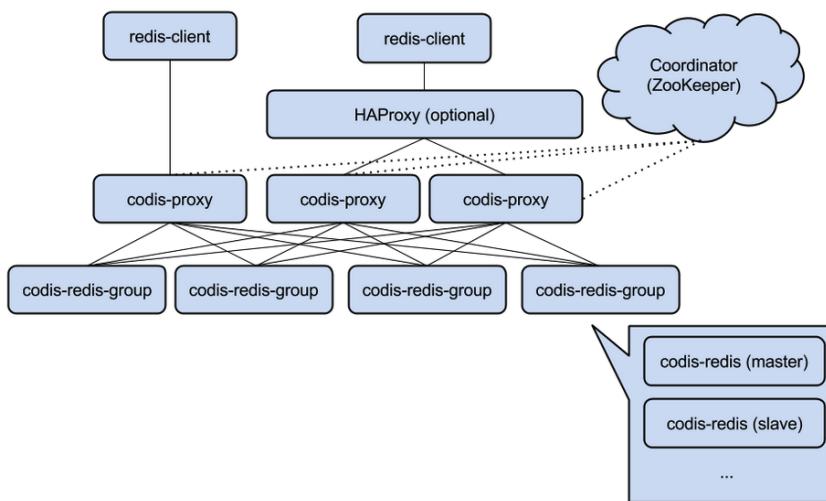
### 3.1 体系架构

Codis 引入了 Group 的概念，每个 Group 包括 1 个 Redis Master 及至少 1 个 Redis Slave，这是和 Twemproxy 的区别之一。这样做好处是，如果当前 Master 有问题，则运维人员

可通过 Dashboard“自助式”切换到 Slave，而不需要小心翼翼地修改程序配置文件。

为支持数据热迁移（Auto Rebalance），出品方修改了 Redis Server 源码，并称之为 Codis Server。

Codis 采用预先分片（Pre-Sharding）机制，事先规定好了，分成 1024 个 slots（也就是说，最多能支持后端 1024 个 Codis Server），这些路由信息保存在 ZooKeeper 中。



ZooKeeper 还维护 Codis Server Group 信息，并提供分布式锁等服务。

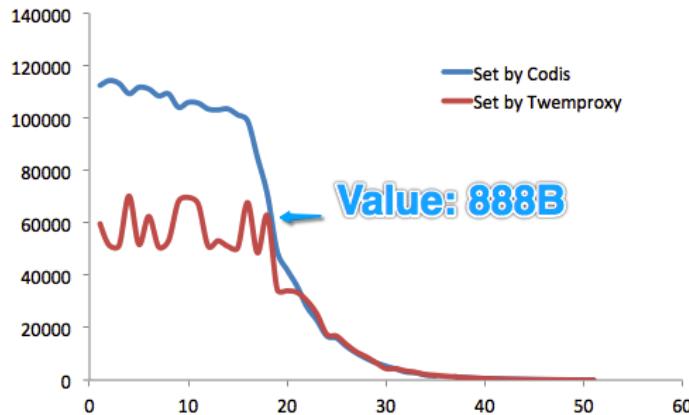
### 3.2 性能对比测试

Codis 目前仍被精益求精地改进中。其性能，从最初的比 Twemproxy 慢 20%（虽然这对于内存型应用而言，并不明显），到现在远远超过 Twemproxy 性能（一定条件下）。

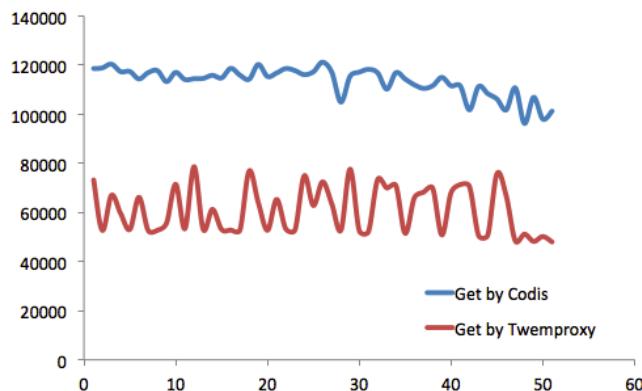
我们进行了长达 3 个月的测试。测试基于 redis-benchmark，分别针对 Codis 和 Twemproxy，测试 Value 长度从 16B~10MB 时的性能和稳定性，并进行多轮测试。

一共有 4 台物理服务器参与测试，其中一台分别部署 codis 和 twemproxy，另外三台分别部署 codis server 和 redis server，以形成两个集群。

从测试结果来看，就 Set 操作而言，在 Value 长度<888B 时，Codis 性能优越优于 Twemproxy（这在一般业务的 Value 长度范围之内）。



就 Get 操作而言，Codis 性能一直优于 Twemproxy。



### 3.3 使用技巧、注意事项

Codis 还有很多好玩的东东，从实际使用来看，有些地方也值得注意。

#### 1) 无缝迁移 Twemproxy

出品方贴心地准备了 Codis-port 工具。通过它，可以实时地同步 Twemproxy 底下的 Redis 数据到你的 Codis 集群。同步完成后，只需修改一下程序配置文件，将 Twemproxy 的地址改成 Codis 的地址即可。是的，只需要做这么多。

#### 2) 支持 Java 程序的 HA

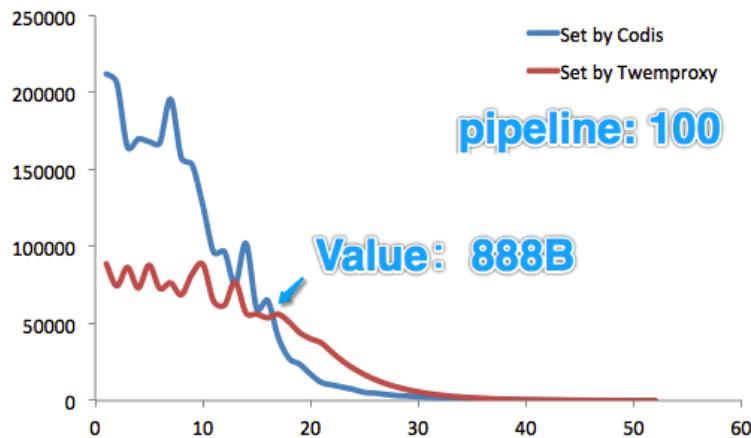
Codis 提供一个 Java 客户端，并称之为 Jodis（名字很酷，是吧？）。这样，如果单个 Codis Proxy 宕掉，Jodis 自动发现，并自动规避之，使得业务不受影响（真的很酷！）。

#### 3) 支持 Pipeline

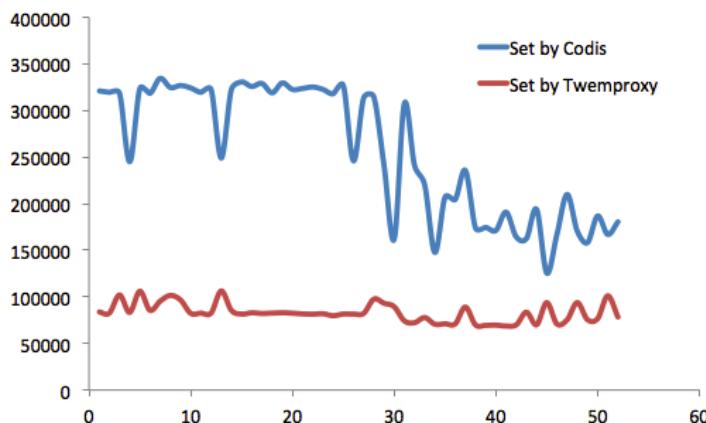
Pipeline 使得客户端可以发出一批请求，并一次性获得这批请求的返回结果。这提升了

Codis 的想象空间。

从实际测试来看，在 Value 长度小于 888B 字节时，Set 性能迅猛提升；



Get 性能亦复如是。



#### 4) Codis 不负责主从同步

也就是说，Codis 仅负责维护当前 Redis Server 列表，由运维人员自己去保证主从数据的一致性。

这是我最赞赏的地方之一。这样的好处是，没把 Codis 搞得那么重。也是我们敢于放手在线上环境中上线的原因之一。

#### 5) 对 Codis 的后续期待？

好吧，粗浅地说两个。希望 Codis 不要变得太重。另外，加 pipeline 参数后，Value 长度如果较大，性能反而比 Twemproxy 要低一些，希望能有改善(我们多轮压测结果都如此)。

因篇幅有限，源码分析不在此展开。另外 Codis 源码、体系结构及 FAQ，参见如下链接：

<https://github.com/wandoulabs/codis>

PS：线上文档的可读性，也是相当值得称赞的地方。一句话：很走心，赞！

最后，Redis 初学者请参考这个链接：<http://www.gamecbg.com/bc/db/redis/13852.html>，文字浅显易懂，而且比较全面。

本文得到 Codis 开发团队刘奇和黄东旭同学的大力协助，并得到 Tim Yang 老师等朋友们对内容把控方面的指导。本文共同作者为赵文华同学，他主要负责 Codis 及 Twemproxy 的对比测试。在此一并谢过。

## 关于作者

**萧田国**，男，硕士毕业于北京科技大学，ACMUG 核心成员，目前为触控科技运维负责人。拥有十多年运维及团队管理经验。先后就职于联想集团（Oracle 数据库主管）、搜狐畅游（数据库主管）、智明星通及世纪互联等。从 1999 年开始，折腾各种数据库如 Oracle/MySQL/MS SQL Server/NoSQL 等，兼任数据库培训讲师若干年。

曾经的云计算行业从业者，现在喜欢琢磨云计算及评测、云端数据库，及新技术在运维中的应用。主张管理学科和运维体系的融合、人性化运维管理，打造高效、专业运维团队。

# 赵海平：开源是个兴趣活儿

作者 郭蕾

3月底，当我得知可以采访赵海平老师的消息后，我非常兴奋。因为之前就参与报道过关于赵海平老师回国的新闻，而当时也只是从侧面简单的了解到这位大牛背后的一些故事。此次有幸在阿里巴巴园区和赵老师面对面畅聊了两个多小时，听他分享了他的个人发展历程以及他对开源的一些认识。而整个聊天过程中，不管是谈到哪一个点，赵老师说的最多的却是兴趣和爱好，而非我想象中的那些高大上的道理，这也是最触动我的地方。

之前 readwrite 的报道称 Facebook 可能已经成为全球最大的开源公司，它将开源的模式推向极致。Facebook 向用户开放了他们的软件、硬件和技术解决方案。而这在一定程度上，也成为 Facebook 吸引开发者的方式之一。近日连线上也有一篇报道 Facebook 开源成果的新闻，读者可以参考阅读。赵海平老师在 2007 年就加入了 Facebook，参与过 Facebook 多个核心系统的建设，对 Facebook 的文化、价值观非常了解。带着早就准备好的几个开源方面的问题，我开始了对赵海平老师的采访。

**InfoQ：**前两天还看到连线上一篇报道 Facebook 开源的文章，Facebook 非常喜欢对外分享他们的成果，这几年，也开源了不少的内部项目。从你的角度来看，Facebook 为什么要做开源？有什么战略上的意义？

**赵海平：**并不是说美国所有的公司都愿意开源，比如说 Google 就不愿意开源，这个我认为其实每一个公司在选择什么开源，什么不开源，这是他自己的一个考量。Google 认为如果开源了某个核心技术，就会形成竞争上的压力，所以他可以选择不开源。开源并非时髦，开源之前首先应该考虑是否会造成商业上的影响。

如果商业上允许开源，我个人认为，开源非常好。开源之后，更多的人就会参与到该软件的设计中，社区会给与你很多的启发，而这些启发对于开源软件的发展非常有益。

同样在开源之后，程序里面本身的问题也就随之暴露了出来，有更多的人使用你的项目，帮你测试，帮你修复，项目质量可以不断提升。其次，开源会给公司带来正面的评价，外界会欣赏公司的开放心态，更容易了解你的领先技术。

需要注意的是，在开源过程中，公司需要投入很多的人力和物力。比如需要有人去跟进用户的反馈，需要有人去管理开源项目。要把一个开源项目运营好，并非易事。Facebook 有专门管理开源的人，他非常有经验，知道在开源过程中会遇到哪些问题，也知道如何与社区交流。

总体来看，我认为开源是利大于弊，只要在商业上允许的情况下，开源都是一件好的事情。

**InfoQ：**你刚才说到，Facebook 有专门负责开源的人，能详细介绍下这个角色吗？

**赵海平：**他的职位是开源经理，负责管理整个公司的开源项目。比如在项目开源之前，他会检查是否有泄露公司核心机密的代码，并帮助解决。他会处理 90% 的外部反馈，这样可以大幅度减少核心工程师的工作量。总体来说，这个人在开源方面有非常丰富的经验，懂技术，并且运营也很专业。

**InfoQ：**Facebook 开源自己的内部项目后，内部也会使用该开源版本的项目吗？

**赵海平：**是的，内部的版本可能会有更多新的特性，但肯定是一个版本。不然的话，你就丧失掉了开源的意义，而且你也希望是一个版本，这样外面有新的功能或者 Patch，你都可以很快合并进来。

**InfoQ：**您对中国目前的开源现状有何看法？

**赵海平：**我觉得国内的开源还没有形成气候，现在大家都是在各做各的。在美国，开源就像旗帜一样的，旗帜一立起来，大家就往上拼。最后拼起来之后，就像一个金字塔一样，非常壮观。咱们可能还没有拼起来，但是我觉得慢慢来，不着急，我们可以学习国外的开源建设经验，在他们的基础上，结合我们的国情，稳步前进。

**InfoQ：**员工在参与开源项目与本职工作之间应该如何平衡？

**赵海平：**这个说起来，我并没有一个特别好的答案，这个问题都可以上升到公司的高度去探讨，并且也值得探讨，这应该是一个公司和员工之间的默契。

说心里话，如果能够把员工搞技术的积极性给调动起来，对于公司来说，绝对是一件好事。如果说员工愿意在开源的环境当中做一个技术的佼佼者，那他在公司也一定会很优秀的，优秀的人一定是处处优秀，也许公司真的应该给工程师较多的时间去做这个事情。

你刚才也说了，很多中国员工的日常工作特别的多，这也正是我想来阿里巴巴体会的事情，我想知道大家都在干什么，为什么有这么多的工作，这个也是需要慢慢的去体会思考的一件事情。过段时间我可以再和大家分享下我的体会。

在美国有一大批的人，他上班的目的也许跟中国员工不太一样。他们可能会认为上班是为了生活，而咱们觉得要生活必须得上班，这个主次的关系也许可能会让他们觉得开源是很重要的一件事情，而上班是次要的，这也是我的一个推测，和文化、国情有关。

**InfoQ：**所以公司其实也应该适当去鼓励工程师参与开源项目？

**赵海平：**阿里巴巴开源了很多项目，这个意识在国内很超前，但并不是每一个中国公司都这么想。有的公司可能会认为，你这个员工做的事情和 KPI 根本没关系，所以他们会禁止这样的工作。其实这些事情都是息息相关的，工程师在参与开源的过程，就是一个学习的过程。通过开源，他的业务能力和技术水平都得到了提升，怎么会对公

司没有好处呢？既使是花掉了公司一点时间，没有做公司本身做的工作，但回过头来也是有意义的。

**InfoQ：**一个网友的问题，您是从生物学转计算机的，跨越不同的学科，仍能做到顶尖，是不是付出了很多？

**赵海平：**每一个行业做到最顶尖的人，都会付出很多。当然凡是做到最顶尖的人，那都是爱好那个行业的人，所以他才肯比别人花更多的时间。而如果你能比别人投入更多的时间，那你肯定会比别人更擅长。当你更擅长的时候，你就会更喜欢这东西。所以爱好和付出是一个正反馈的过程。

# 性能优化：一个全栈问题

作者 谢丽

Ronald Bradford 是一名有 26 年行业经验的 IT 专家，他撰写了 [Effective MySQL 系列](#) 书籍并已被翻译为中文。近日，他在[个人博客](#)上发表了一篇探讨性能优化的[文章](#)。他认为，性能优化是一个全栈问题，提高 Web 系统的性能需要了解整个技术栈的运转和交互。他列举了一些快速提升系统性能的常用技巧，包括：

- 使用 [CDN](#) 资源；
- 压缩内容；
- 减少请求次数（Web、缓存、数据库）；
- 异步管理；
- 优化 SQL 语句；
- 数据库服务器使用固态硬盘；
- 更新软件版本；
- 增加服务器；
- 正确配置软件。

Ronald 指出，性能优化需要制定详细的优化计划，确定每一部分的优化目标，并进行相应的测试验证。如果系统任何一部分未能达到预期的优化效果，那么就无法实现整个系统的性能提升。比如，一个系统的 MySQL 数据库饱和了，InnoDB 并发事务数达到上限。在旧版本中，这是一个不可配置的参数。因此，升级是一种简单直接的优化方法。但同时，工程团队升级了 PHP 应用程序框架（如 [Slim](#) 和 [Twig](#)），导致应用程序响应时间增加。结果，虽然系统负载增加了，但整体性能却未能得到提升。

因此，Ronald 指出，为了实现系统整体性能的优化，需要重点从以下几个方面考虑问题：

- 了解 CPU 饱和度；
- 检测和缓解网络延迟；
- 了解虚拟云实例的虚拟模式选项；
- 了解网络协议栈，利用好不同的主机操作系统；
- 模拟生产负载并不容易；

- 性能分析，分析，再分析；
- 工具有误导性，需要了解不同监控工具的工作原理；
- 全栈优化是一个迭代过程；
- 了解如何优化技术栈的每一部分；
- 不是每项优化都能达到预期效果；
- 了解什么时候停止优化及优先优化哪一部分。

# 手动测试是否可以退出测试舞台？

作者 李小兵

来自测试管理平台 [Zephyr](#) 的负责客户服务的副总裁 [Sanjay Zalavadia](#) 对手动测试和自动化测试进行了[对比](#)，Sanjay 首先指出在一些实际场景中，自动化测试需要手动测试作为补充。同时，Sanjay 还指出了 QA 团队应该使用手动测试而非自动化测试的一些场景。现对这些场景进行归纳和总结，以供读者学习和参考，具体内容如下。

## 1. 当项目需要灵活性时

虽然手动测试并非总是和自动化测试一样精确，但是手动测试过程使得测试人员能够更加灵活地进行测试。因为自动化测试对于重复的场景具有一定的优势，即相同的代码和脚本能够被多次复用。但是如果测试人员突然有了一个测试想法，手动测试人员就能够很快测试新想法；而对于自动化测试，就需要创建测试用例、使用自动化测试工具编写代码，然后运行测试，这将花费许多时间。在这种情况下，通过人工操作进行操作会更加简单和快速。此外，手动测试人员还能够快速查看测试结果。

## 2. 当是短期的敏捷项目时

自动化测试需要事先的投入和计划，需要投入较高的成本，这些工作在短期的项目中是没有必要的。自动化测试比较短期的项目，则使得项目的前期成本可能比较高，手动测试更加合适对于这种场景。

## 3. 当可用性需要测试时

一个应用的成功或失败很大程度上取决于它的可用性，然而，仅仅依靠自动化测试是很难确保应用没有缺陷存在的。计算机没有原创的思想，它们运行事先靠程序设定好的操作，它给不出像 APP 用户给出的那些反馈。

最后，Sanjay 还指出了自动化测试具有很多优点，但是手动测试在一些场景中则显得更加有优势，只有 QA 团队找出最适合自己的实际场景的测试方法，他们才能够简化操作、提高效率以及改进整个产品的质量。

# 封面植物

国家一级重点保护野生植物—冷杉

名称：百山祖冷杉

拉丁名：*Abies beshanzuensis*

别名：冷杉

英文名：*Beshanzufir*

科目：松科冷杉属。

地域：分布于浙江庆元县百山祖



第四纪冰川期遗留下来的植物，有“植物活化石”和“植物大熊猫”之称，对研究古气候、古地质变迁、古生物、古植被等具有重要意义。目前，野生百山祖冷杉全球仅存3株，均生长在浙江省庆元县的百山祖国家级自然保护区核心区。1988年，被世界受危胁植物委员会评为最濒危的12种植物之一，其中一株衰弱，一株生长不良。

冷杉属[植物](#)发生于晚白垩世，至第三纪中新世及第四纪种类增多，分布区扩大，经冰期与间冰期保留下，繁衍至今。在秦岭以南及东南的平原和西南低山地区的晚更新世沉积物中发现了冷杉花粉。冷杉具有较强的耐阴性，适应温凉和寒冷的气候，[土壤](#)以山地棕壤、暗棕壤为主。常在高[纬度](#)地区至低纬度的亚高山至高山地带的阴坡、半阴坡及谷地形成纯林，或与性喜冷湿的云杉、落叶松、铁杉和某些松树及[阔叶树](#)组成针叶混交林或针阔混交林。



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 | .....



# AWSome Day

## 让您大有作为

合作伙伴



大连站、北京站、深圳站、上海站  
[点击查看详情](#)



促进软件开发领域知识与创新的传播

# 架构师

ARCHITECT



架构师 2015 年 5 月刊  
每月 8 号出版

本期主编：郭蕾

流程编辑：丁晓昀

发行人：霍泰稳

读者反馈/投稿：editors@cn.infoq.com

InfoQ 中文站新浪微博：<http://weibo.com/infoqchina>

商务合作：sales@cn.infoq.com 15810407783

InfoQ 

2015年05月



本期主编：郭蕾，InfoQ 技术编辑，文艺范儿程序员，并发编程网站长。在 CRM 行业厮混 3 年多，喜欢技术写作和社区运营，信奉见城彻先生的那句话：偏执、冒险、狂妄的人终是英雄。我不是英雄，但我会努力成为英雄。