NO.35

编程狂人

Programming Madman

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/53d61e48d91b1428e50053dd

欢迎下载推酷客户端体验更多阅读乐趣

版权说明

本刊只用于行业间学习与交流署名文章及插图版权归原作者享有

目录

详解Javascript 中的this指针

作者：Kevin Yang

前言

Javascript是一门基于对象的动态语言，也就是说，所有东西都是对象，一个很典型的例子就是函数也被视为普通的对象。Javascript 可以通过一定的设计模式来实现面向对象的编程，其中this “指针”就是实现面向对象的一个很重要的特性。但是this也是Javascript中一个非常容易理解错，进而用错的特性。特别是对于接触静态语言比较 久了的同志来说更是如此。

示例说明

我们先来看一个最简单的示例：

```
<script type="text/javascript">
    var name = "Kevin Yang";
    function sayHi(){
        alert("你好，我的名字叫" + name);
    }
    sayHi();
</script>
```

这段代码很简单，我们定义了一个全局字符串对象name和函数对象sayHi。运行会弹出一个打招呼的对话框，“你好，我的名字叫Kevin Yang”。

我们把这段代码稍微改一改：

```
<script type="text/javascript">
```

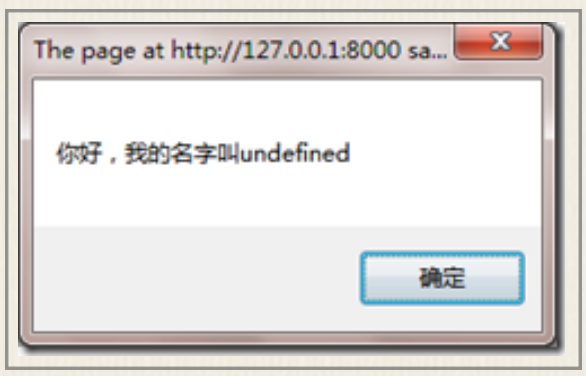
```
    var name = "Kevin Yang";
    function sayHi(){
       alert("你好，我的名字叫" + this.name);
    }
    sayHi();
</script>
```

这段代码和上段代码的区别就在于sayHi函数在使用name的时候加上了this.前缀。运行结果和上面一摸一样。这说明this.name引用的也还是全局的name对象。

开头我们不是说了，函数也是普通的对象，可以将其当作一个普通变量使用。我们再把上面的代码改一改：

```
<script type="text/javascript">
    var name = "Kevin Yang";
    function sayHi(){
       alert("你好，我的名字叫" + this.name);
    }
    var person = {};
    person.sayHello = sayHi;
    person.sayHello();
</script>
```

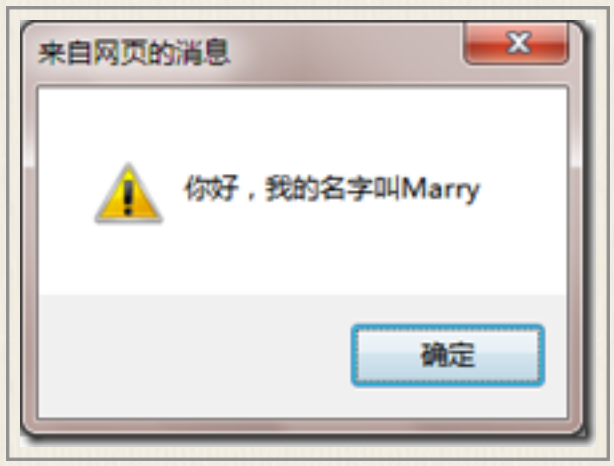
这一次，我们又创建了一个全局对象person，并将sayHi函数对象赋给person对象的sayHello属性。运行结果如下：



这一次打招呼的内容就有点无厘头了，我们发现this.name已经变成undefined了。这说明，在sayHello函数内部执行时已经找不着this.name对象了。如果我们重新定义person对象，在其上面加上一个name属性又会怎么样呢?

var person = {name:"Marry"};

运行代码发现打招呼的“人”变了：



是不是看出点道道了呢?

判别this指针的指导性原则

在Javascript里面，this指针代表的是执行当前代码的对象的所有者。

在上面的示例中我们可以看到，第一次，我们定义了一个全局函数对象sayHi并执行了这个函数，函数内部使用了this关键字，那么执行this这 行代码的对象是sayHi（一切皆对象的体现），sayHi是被定义在全局作用域中。其实在Javascript中所谓的全局对象，无非是定义在 window这个根对象下的一个属性而已。因此，sayHi的所有者是window对象。也就是说，在全局作用域下，你可以通过直接使用name去引用这 个对象，你也可以通过window.name去引用同一个对象。因而this.name就可以翻译为window.name了。

再来看第二个this的示例。我们定义了一个person的对象，并定义了它的sayHello属性，使其指向sayHi全局对象。那么这个时候， 当我们运行person.sayHello的时候，this所在的代码所属对象就是sayHello了（其实准确来说，sayHi和sayHello是只 不过类似两个指针，指向的对象实际上是同

一个），而sayHello对象的所有者就是person了。第一次，person里面没有name属性，因此弹 出的对话框就是this.name引用的就是undefined对象（Javascript中所有只声明而没有定义的变量全都指向undefined对象）；而第二次我们在定义person的时候加了name属性了，那么this.name指向的自然就是我们定义的字符串了。

理解了上面所说的之后，我们将上面最后一段示例改造成面向对象式的代码。

```
<script type="text/javascript">
    var name = "Kevin Yang";
    function sayHi(){
        alert("你好，我的名字叫" + this.name);
    }
    function Person(name){
        this.name = name;
    }
    Person.prototype.sayHello = sayHi;
    var marry = new Person("Marry");
    marry.sayHello();
    var kevin = new Person("Kevin");
    kevin.sayHello();
</script>
```

在上面这段代码中，我们定义了一个Person的“类”（实际上还是一个对象），然后在这个类的原型（类原型相当于C++中的静态成员变量的概念）中定义了sayHello属性，使其指向全局的sayHi对象。运行代码我们可以看到，marry和kevin都成功的向我们打了声“招呼”。

在这段代码中有两点需要思考的，一个是new我们很熟悉，但是在这里new到底做了什么操作呢？另外一个是，这里执行sayHello的时候，this指针为什么能够正确的指向marry和kevin对象呢？

我们来把上面定义“类”和实例化类对象的操作重新“翻译”一下：

```
<script type="text/javascript">
    var name = "Kevin Yang";
    function sayHi(){
        alert("你好，我的名字叫" + this.name);
    }
    function Person(name){
        var this;
        this.name = name;
        return this;
    }
    Person.prototype.sayHello = sayHi;
    var marry = Person("Marry");
    marry.sayHello();
    var kevin = Person("Kevin");
    kevin.sayHello();
</script>
```

当然这段代码并不能正确执行，但是它可以帮助你更好的理解这个过程。

当我们使用new关键字实例化一个“类”对象的时候，Javascript引擎会在这个对象内部定义一个新的对象并将其存入this指针。所有此对 象内部用到this的代码实际上都是指向这个新的对象。如this.name = name，实际上是将参数中的name对象赋值给了这个新创建的对象。函数对象执行完之后

Javascript引擎会将此对象返回给你，于是就有 marry变量得到的对象的name为“Marry”，而kevin变量得到的对象的name属性确实“Kevin”。

显式操纵this指针

在上面的面向对象式编程实例中，我们看到，在使用new操作符的情况下，看起来this的指向和我们前一节中讲到的指导原则并不相符。 this指针并没有指向marry或者kevin的所有者，而是指向marry和kevin变量本身。

实际上，如果你理解什么是指针的话，那么你就会知道，既然是指针，那么当然可以改变其指向的对象。只不过Javascript引擎不允许我们自己写代码来做这样的事情，也就是说，在Javascript中，你不可以直接写this = someObj这样的代码。Javascript引擎通过以下两种方式允许我们显式指定this指针指代的对象：

1. 通过new操作符，Javascript引擎会将this指针返回给被赋值的变量A（对应上面的例子就是marry和kevin变量），这个时候A和 this指针引用的就是同一个对象了，即A == this。过程参见上面的伪代码。

2. 通过Function.apply或者Function.call的原型方法，我们可以将this指针指代的对象以参数的形式传入，这个时候，函数内部使用的this指针就是传入的参数。

注意，对于这种显式指定this指针的情况，上一节提到的指导原则不再适用。

容易误用的情况

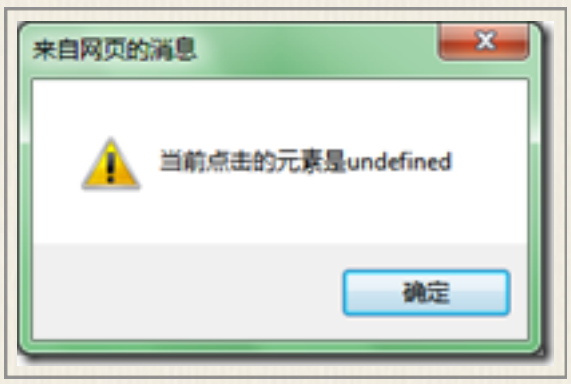
理解了this指针后，我们再来看看一些很容易误用this指针的情况。

示例1——内联式绑定Dom元素的事件处理函数

```
<script type="text/javascript">
    function sayHi(){
        alert("当前点击的元素是" + this.tagName);
    }
</script>
```

```
<input id="btnTest" type="button" value="点击我" onclick="sayHi()">
```

在此例代码中，我们绑定了button的点击事件，期望在弹出的对话框中打印出点击元素的标签名。但运行结果却是：



也就是this指针并不是指向input元素。这是因为当使用内联式绑定Dom元素的事件处理函数时，实际上相当于执行了以下代码：

```
<script type="text/javascript">
    document.getElementById("btnTest").onclick = function(){
        sayHi();
    }
</script>
```

在这种情况下sayHi函数对象的所有权并没有发生转移，还是属于window所有。用上面的指导原则一套我们就很好理解为什么this.tagName是undefined了。

那么如果我们要引用元素本身怎么办呢?

我们知道，onclick函数是属于btnTest元素的，那么在此函数内部，this指针正是指向此Dom对象，于是我们只需要把this作为参数传入sayHi即可。

```
<script type="text/javascript">
    function sayHi(el){
        alert("当前点击的元素是" + el.tagName);
    }
```

```
</script>
<input id="btnTest" type="button" value="点击我" onclick="sayHi(this)">
```

等价代码如下:

```
<script type="text/javascript">
    document.getElementById("btnTest").onclick = function(){
        sayHi(this);
    }
</script>
```

示例2——临时变量导致的this指针丢失

```
<script type="text/javascript">
    var Utility = {
        decode:function(str){
            return unescape(str);
        },
        getCookie:function(key){
            // ... 省略提取cookie字符串的代码
            var value = "i%27m%20a%20cookie";
            return this.decode(value);
        }
    };
    alert(Utility.getCookie("identity"))
</script>
```

我们在写稍微有点规模的Js库的时候，一般都会自己封装一个Utility的类，然后将一些常用的函数作为Utility类的属性，如客户端经常会 用到的getCookie函数和解码函数。如果每个函数都是彼此独立的，那么还好办，

问题是，函数之间有时候会相互引用。例如上面的getCookie函 数，会对从document.cookie中提取到的字符串进行decode之后再返回。如果我们通过Utility.getCookie去调用的话，那 么没有问题，我们知道，getCookie内部的this指针指向的还是Utility对象，而Utility对象时包含decode属性的。代码可以成 功执行。

但是有个人不小心这样使用Utility对象呢?

```
<script type="text/javascript">
    function showUserIdentity(){
        // 保存getCookie函数到一个局部变量，因为下面会经常用到
        var getCookie = Utility.getCookie;
        alert(getCookie("identity"));
    }
    showUserIdentity();
</script>
```

这个时候运行代码会抛出异常“this.decode is not a function”。运用上面我们讲到的指导原则，很好理解，因为此时Utility.getCookie对象被赋给了临时变量getCookie，而临 时变量是属于window对象的——只不过外界不能直接引用，只对Javascript引擎可见——于是在getCookie函数内部的this指针指向 的就是window对象了，而window对象没有定义一个decode的函数对象，因此就会抛出这样的异常来。

这个问题是由于引入了临时变量导致的this指针的转移。解决此问题的办法有几个:

- 不引入临时变量，每次使用均使用Utility.getCookie进行调用
- getCookie函数内部使用Utility.decode显式引用decode对象而不通过this指针隐式引用（如果Utility是一个实例化的对象，也即是通过new生成的，那么此法不可用）
- 使用Funtion.apply或者Function.call函数指定this指针

前面两种都比较好理解，第三种需要提一下。正是因为this指针的指向很容易被转移丢失，因此Javascript提供了两个类似的函数apply和call来允许函数在调用时重新显式的指定this指针。

修正代码如下：

```
<script type="text/javascript">
    function showUserIdentity(){
        // 保存getCookie函数到一个局部变量，因为下面会经常用到
        var getCookie = Utility.getCookie;
        alert(getCookie.call(Utility,"identity"));
        alert(getCookie.apply(Utility,["identity"]));
    }
    showUserIdentity();
</script>
```

call和apply只有语法上的差异，没有功能上的差别。

示例3——函数传参时导致的this指针丢失

我们先来看一段问题代码：

```
<script type="text/javascript">
    var person = {
        name:"Kevin Yang",
        sayHi:function(){
            alert("你好，我是"+this.name);
        }
    }
    setTimeout(person.sayHi,5000);
</script>
```

这段代码期望在访客进入页面5秒钟之后向访客打声招呼。setTimeout函数接收一个函数作为参数，并在指定的触发时刻执行这个函数。可是，当我们等了5秒钟之后，弹出的对话框显示的this.name却是undefined。

其实这个问题和上一个示例中的问题是类似的，都是因为临时变量而导致的问题。当我们执行函数的时候，如果函数带有参数，那么这个时候Javascript引擎会创建一个临时变量，并将传入的参数复制（**注意，Javascript里面都是值传递的，没有引用传递的概念**）给此临时变量。也就是说，整个过程就跟上面我们定义了一个getCookie的临时变量，再将Utility.getCookie赋值给这个临时变量一样。只不过在这个示例中，容易忽视临时变量导致的bug。

函数对象传参

对于函数作为参数传递导致的this指针丢失的问题，目前很多框架都已经有方法解决了。

Proto-
type的解决方案——传参之前使用bind方法将函数封装起来，并返回封装后的对象

```
<script type="text/javascript">
    var person = {
        name:"Kevin Yang",
        sayHi:function(){
            alert("你好，我是"+this.name);
        }
    }
    var boundFunc = person.sayHi.bind(person,person.sayHi);
    setTimeout(boundFunc,5000);
</script>
```

bind方法的实现其实是用到了Javascript又一个高级特性——闭包。我们来看一下源代码：

```
function bind(){
    if (arguments.length < 2 && arguments[0] === undefined)
        return this;
    var __method = this, args = $A(arguments), object = args.shift();
    return function(){
        return __method.apply(object, args.concat($A(arguments)));
    }
}
```

首先将this指针存入函数内部临时变量，然后在返回的函数对象中引用此临时变量从而形成闭包。

微软的Ajax库提供的方案——构建委托对象

```
<script type="text/javascript">
    var person = {
        name:"Kevin Yang",
        sayHi:function(){
            alert("你好，我是"+this.name);
        }
    }
    var boundFunc = Function.createDelegate(person,person.sayHi);
    setTimeout(boundFunc,5000);
</script>
```

其实本质上和prototype的方式是一样的。

著名的Extjs库的解决方案采用的手法和微软是一样的。

原文链接：http://www.cnblogs.com/KevinYang/archive/2009/07/14/1522915.html

Web Components 初探

作者：Letian Zhang

众所周知，Web 页面是由 HTML+CSS+JavaScript 三板斧配合而成的，这体现了一种结构、表现、交互分离的思想。但是随着 Web 应用不断丰富，过度分离的设计也会带来可重用性上的问题。于是各家显神通，各种 UI 组件工具库层出不穷，煞有八仙过海之势。于是 W3C 坐不住了，大手一挥，说道：不如让我们统一一个 Web Components 的标准吧怎么样。

Web Components 的核心思想就是把 UI 元素组件化，即将 HTML、CSS、JS 封装起来，使用的时候就不需要这里贴一段 HTML，那里贴一段样式，最后再贴一段 JS 了。一般来说，它其实是由四个部分的功能组成的：

1. 模板，<template> 标签
2. 自定义元素
3. Shadow DOM（隐匿 DOM）
4. Imports（导入）

我们还是通过一个简单的例子看看这些新玩意儿都是些什么吧。

一段简单的 HTML

假设我们有一个提供 App 介绍的代码片段，为了不让事情变得更复杂，这里只有 HTML 和 CSS，不关 JS 什么事。

```
<div class="app-info">
  <div class="app-bar">
```

```
    <img class="app-icon"
src="http://img.dayanjia.com/di/TOY7/6c2442a7d933c8950f39059ed31373f083020094.png" width="36" height="36"/>
    <div class="app-name">百度手机助手</div>
    <a class="app-downbtn"
href="http://gdown.baidu.com/data/wisegame/de5074e4e28aecec/baidushoujizhushou_16783385.apk">下载</a>
  </div>
  <div class="app-description">
    百度手机助手是Android手机的权威资源平台，拥有最全最好的应用、游戏、壁纸资源，帮助您在海量资源中精准搜索、高速下载、轻松管理，万千汇聚，一触即得。海量资源：免费获取数十万款应用和游戏，更有海量独家正版壁纸，任你挑选。
  </div>
</div>
.app-info {
  padding: 0.2em;
  border-bottom: 1px dotted #ddd;
}
.app-bar {
  display: flex;
  align-items: center;
  font-size: 14px;
}
.app-name {
  flex-grow: 2;
```

```
  margin-left: 1em;
}
.app-downbtn {
  text-decoration: none;
  padding: 0.2em 1.1em;
  margin-right: 1em;
  color: #fff;
  background: #5573eb;
}
.app-description {
  font-size: 12px;
}
```

看上去就是这样的：

模板

HTML 模板这个东西已经存在很久了，模板的实现无非是这么几种。一种是直接写在 DOM 里，但是给它一个 display: none 的样式。使用这种模板，我们可以很方便地用 JavaScript 来操作 DOM 结构，但是如果你在模板里写了一个 img 元素之类，不好意思，即使你看不到，这个图片的网络请求还是要发一下的。此外，与模板相对应的 CSS 也是和页面其他部分平

行的关系，你需要给模板加一个 ID 之类的选择器前缀来指定样式，以保证不和页面中的其他元素冲突。

第二种是使用 <script> 标签，但是给它指定一个非脚本的 type 属性，这样浏览器就不会把它当做 JS 来执行了：

```
<script id="template" type="x-tmpl-mustache">
Hello {{ name }}!
</script>
```

这种方法的好处在于，DOM 元素是不会预先渲染的，因为在被 JS 取得模板数据并插入 DOM 之前，它都是一堆死气沉沉的纯文本。同时这也是它的弊端，因为是纯文本，所以你要手动处理这些复杂的标签，需要格外小心 XSS 之类的问题。

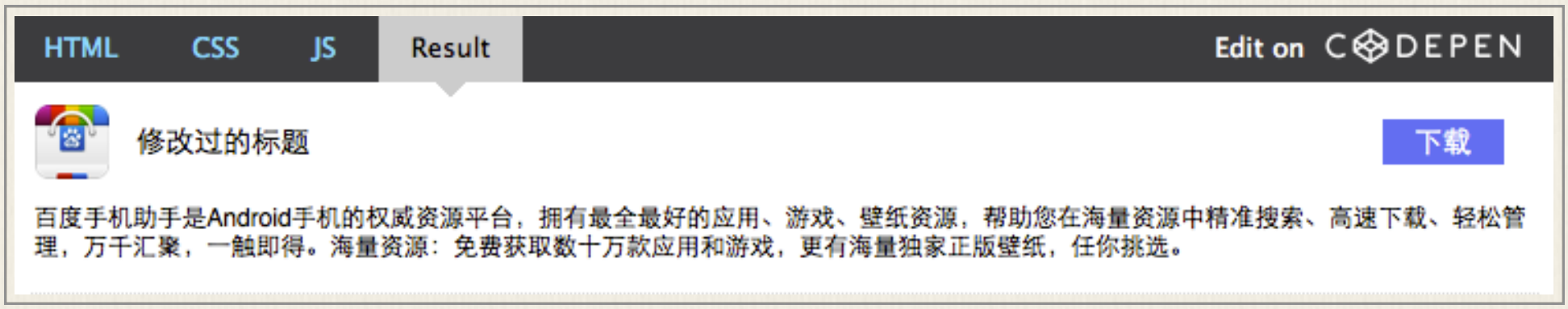
于是新的 <template> 标签就被提出了，它可以看做是结合了上面两种方法的优势。我们将上面的 HTML 模版化后：

```
<template id="appTmpl">
... 和之前一样的内容 ...
</template>
```

使用下面的 JS 就可以访问到模板，并将其插入 DOM 中。

```
var tmpl = document.querySelector('#appTmpl');
// 取到 t 以后，可以像操作 DOM 一样随意修改其中的内容
// 然后需要从模板创建一个深拷贝（Deep Copy），将其插入 DOM
var clone = document.importNode(tmpl.content, true);
// 创建深拷贝还可以使用下面的方法：
// var clone = tmpl.content.cloneNode(true);
document.body.appendChild(clone);
```

最后的效果和之前看到的其实是一样的。



当然了，这个模板的实现其实还是很原始的，并没有像 Mustache、Handlebars 等模板库的占位符替换的功能。

Shadow DOM

这个 Shadow 不太好翻译，反正理解成「隐藏在黑暗中的 DOM」就差不多了。所以说，Shadow DOM 其实是在文档的主 DOM 中生成了一块子 DOM，这个子 DOM 的 CSS 环境是和主文档隔离的。可以说，使用 Shadow DOM，我们就拥有了一个组件封装的原始模型。从外面看，它只是一个 DOM 节点，但是这其实是一个黑盒，里面还可以包含复杂的结构。这种抽象其实在大自然中随处可见，例如当我们谈论太阳系的时候，我们会把地球作为一个节点，但是 当我们深入地球这个节点时，会发现还存在地月系这个结构。

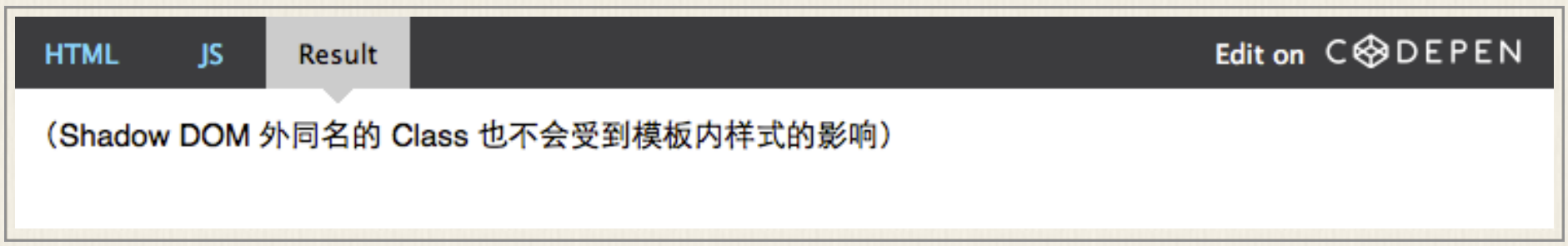
使用 Shadow DOM，我们需要在一个元素上创建一个根（Root），然后将模板内文档添加到这个根上即可。

```
<template id="appTmpl">
  <style>
  /* ... 将 CSS 移动到模板内 ... */
  </style>
  ... 原来的模板内容 ...
</template>

<div class="app"></div>
```

```
var tmpl = document.querySelector('#appTmpl');
var host = document.querySelector('.app');
var root = host.createShadowRoot();
root.appendChild(document.importNode(tmpl.content, true));
```

最终的效果看上去是一样的，但是我们已经将这个 App 信息组件封装了一层 DOM。



自定义元素

现在我们已经能够使用一句 外加一些 JS 来显示这个 App 信息的组件了（如果它够的上被称作是一个「组件」的话）。但是，我们能不能再给力一点，使用一个自己命名的元素呢？答案当然是肯定的。通过自定义元素的功能，就可以实现通过 <app-info></app-info> 这样的方式来调用它了。

HTML 除了上文的那些模板以外，只需要一个简单的容器。同时，接下来的例子中，我们还可以看到如何使用属性来替换模版中的变量，因此模板中也要做出一些修改。

```
<template id="appTmpl">
  <style>
    /* ... CSS 省略 ... */
  </style>
  <div class="app-info">
    <div class="app-bar">
```

```
      <img class="app-icon" src="" width="36" height="36"/>
      <div class="app-name"></div>
      <a class="app-downbtn" href="">下载</a>
    </div>
    <div class="app-description">
      <content selector=".description"></content>
    </div>
  </div>
</template>

<app-info name="百度手机助手"
downurl="http://gdown.baidu.com/data/wisegame/de5074e4e28aecec/baidushoujizhushou_16783385.apk"
iconurl="http://img.dayanjia.com/di/TOY7/6c2442a7d933c8950f39059ed31373f083020094.png">
    <p class="description">百度手机助手是Android手机的权威资源平台，拥有最全最好的应用、游戏、壁纸资源，帮助您在海量资源中精准搜索、高速下载、轻松管理，万千汇聚，一触即得。海量资源：免费获取数十万款应用和游戏，更有海量独家正版壁纸，任你挑选。</p>
</app-info>
```

可以看到，Shadow DOM 也可以拥有子元素，而这些子元素在模板中将会使用 <content> 标签进行定位并替换。接下来，我们使用 JavaScript 创建这个名叫 app-info 的自定义元素。

```
var tmpl = document.querySelector('#appTmpl');


// 创建新元素的 Prototype
var appInfoProto = Object.create(HTMLElement.prototype);
```

```
// 自定义元素在不同的生命周期有不同的 Callback 可以使用。
// createdCallback 是在创建时调用的，此外还有
// attachedCallback（插入 DOM 时的回调）、
// detachedCallback（从 DOM 中移除时的回调）、
// attributeChangedCallback（属性改变时的回调）
appInfoProto.createdCallback = function() {
  var root = this.createShadowRoot();
  var name = this.getAttribute('name') || '';
  var downUrl = this.getAttribute('downurl') || '';
  var iconurl = this.getAttribute('iconurl') || '';
  tmpl.content.querySelector('.app-name').textContent = name;
  tmpl.content.querySelector('.app-downbtn').href = downUrl;
  tmpl.content.querySelector('.app-icon').src = iconurl;
  // 将模板插入 Shadow DOM
  root.appendChild(document.importNode(tmpl.content, true));
};

// 注册自定义元素
var appInfo = document.registerElement('app-info', {
    prototype: appInfoProto
});
```

最后看到的效果，其实和之前的没什么不同，但是我们很清楚，一个简单的 Web Component 雏形已经诞生了。

HTML JS Result Edit on CODEPEN

百度手机助手是Android手机的权威资源平台，拥有最全最好的应用、游戏、壁纸资源，帮助您在海量资源中精准搜索、高速下载、轻松管理，万千汇聚，一触即得。海量资源：免费获取数十万款应用和游戏，更有海量独家正版壁纸，任你挑选。

通过 Chrome 的开发工具我们可以很清楚地看到 <template> 中的文档片段和我们自定义的 <app-info> 元素中存在的 Shadow DOM。

```
▼<template id="appTmpl">
  ▼#document-fragment
    ▶<style>…</style>
    ▶<div class="app-info">…</div>
  </template>
▼<app-info name="百度手机助手" downurl="http://gdown.baidu.
  ▼#shadow-root
  ▶<style>…</style>
  ▼<div class="app-info">
    ▶<div class="app-bar">…</div>
    ▼<div class="app-description">
       <content selector=".description"></content>
     </div>
   </div>
  ▼<p class="description">
     "百度手机助手是Android手机的权威资源平台，拥有最全最好的应用、游
   </p>
</app-info>
```

导入

Web Components 的最后一部分是导入，这就比较容易理解了，就是提供了一个可复用的途径。我们可以像导入 CSS 一样，导入外部文件中的 HTML 代码。

<link rel="import" href="app-info.html">

小结

Web Components 这个东西还非常新，但是它代表了 Web 前端今后的一个发展方向。包括比较火的 AngularJS 等框架，其中的一些功能也或多或少地在使用 Web Components 的思想，并且推动其标准化（见 the future of AngularJS）。

同时，也是因为它太新了，所以可能还会有非常大的改变，也许过几个月再来看这篇文章，部分内容就已经过时了:D 此外，当前浏览器对 Web Components 的支持也很有限，在 Chrome 35+ 中，本文中的全部例子都可以正常展现，其他浏览器就基本上悲剧了。对于这样一个新生状态，还处于快速变化期的事物，我也仅仅是浅尝辄止，本文更多在于抛砖 引玉，若有疏漏还请读者多多指正。

针对 Web Components 的功能，Google 出了一个叫做 polymer 的项目，用于填补目前浏览器尚不能实现的部分，此外还内建了许多做好的组件。其实这个项目也推出挺久的了，但是一直不温不火，风头赶不上同是出自 Google 的 AngularJS。但是今年 Google IO 大会中，它却被作为 Material Design 的一部分拿出来介绍了，可见其还是很受重视的。下次如果有机会，可以介绍一下它。

原文链接：http://mweb.baidu.com/p/web-components-introduction.html

深度解析LinkedIn大数据平台

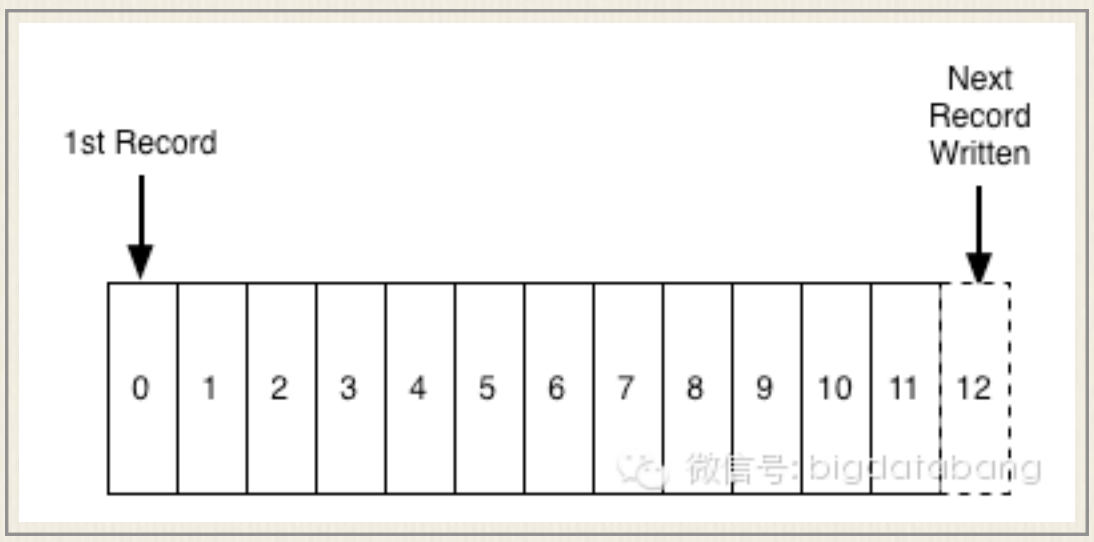
译者：大数据邦

我在六年前的一个令人兴奋的时刻加入到LinkedIn公司。从那个时候开始我们就破解单一的、集中式数据库的限制，并且启动到 特殊的分布式系统套件的转换。这是一件令人兴奋的事情：我们构建、部署，而且直到今天仍然在运行的分布式图形数据库、分布式搜索后端、Hadoop安装以 及第一代和第二代键值数据存储。

从这一切里我们体会到的最有益的事情是我们构建的许多东西的核心里都包含一个简单的理念：日志。有时候也称作预先写入日志或者提交日志或者事务日志，日志几乎在计算机产生的时候就存在，同时它还是许多分布式数据系统和实时应用结构的核心。

不 懂得日志，你就不可能完全懂得数据库，NoSQL存储，键值存储，复制，paxos,Hadoop,版本控制以及几乎所有的软件系统；然而大多数软件工程 师对它们不是很熟悉。我愿意改变这种现状。在这篇博客文章里，我将带你浏览你必须了解的有关日志的所有的东西，包括日志是什么，如何在数据集成、实时处理 和系统构建中使用日志等。

第一部分：日志是什么?



日志是一种简单的不能再简单的存储抽象。它是一个只能增加的，完全按照时间排序的一系列记录。日志看起来如下：

我们可以给日志的末尾添加记录，并且可以从左到右读取日志记录。每一条记录都指定了一个唯一的有一定顺序的日志记录编号。

日 志记录的排序是由“时间”来确定的，这是因为位于左边的日志记录比位于右边的要早些。日志记录编号可以看作是这条日志 记录的“时间戳”。在一开始就把这种排序说成是按时间排序显得有点多余 ， 不过 ， 与任何一个具体的物理时钟相比，时间 属性是非常便于使用的属性。在我们运行多个分布式系统的时候，这个属性就显得非常重要。

对于这篇讨论的目标而言，日志记录的内容和格式不怎么重要。另外提醒一下，在完全耗尽存储空间的情况下，我们不可能 再给日志添加记录。稍后我们将会提到这个问题。

日志并不是完全不同于文件或者数据表的。文件是由一系列字节组成，表是由一系列记录组成，而日志实际上只是按照时间顺序存储记录的 一种数据表或者文件。

此时，你可能奇怪为什么要讨论这么简单的事情呢？ 不同环境下的一个只可增加的有一定顺序的日志记录是怎样与数据系统关联起来的呢？答案是日志有其特定的应用目标：它记录了什么时间发生了什么事情。 而对分布式数据系统许多方面而言， 这才是问题的真正核心。

不 过，在我们进行更加深入的讨论之前，让我先澄清有些让人混淆的概念。每个编程人员都熟悉另一种日志记录-应用使用syslog或者log4j可能写入到本 地文件里的没有结构的错误信息或者追踪信息。为了区分开来，我们把这种情形的日志记录称为“应用日志记录”。应用日志记录是我在这儿所说的日志的一种低级 的变种。最大的区别是：文本日志意味着主要用来方便人们阅读，而我所说明的“日志”或者“数据日志”的建立是方便程序访问。

（实 际上，如果你对它进行深入的思考，那么人们读取某个机器上的日志这种理念有些不顺应时代潮流。当涉及到许多服务和服务器的时候，这种方法很快就变成一个难 于管理的方式，而且为了认识多个机器的行为，日志的目标很快就变成查询和图形化这些行为的输入了-对多个机器的某些行

为而言，文件里的英文形式的文本同这 儿所描述的这种结构化的日志相比几乎就不适合了。）

数据库日志

我 不知道日志概念起源于何处-可能它就像二进制搜索一样：发明者认为它太简单而不能当作一项发明。它早在IBM的系统R出现时候就出现了。数据库里的用法是 在崩溃的时候用它来同步各种数据结构和索引。为了保证操作的原子性和持久性，在对数据库维护的所有各种数据结构做更改之前，数据库把即将修改的信息誊写到 日志里。日志记录了发生了什么，而且其中的每个表或者索引都是一些数据结构或者索引的历史映射。由于日志是即刻永久化的，可以把它当作崩溃发生时用来恢复 其他所有永久性结构的可信赖数据源。

随着时间的推移，日志的用途从实现ACID细节成长为数据库间复制数据的一种方法。利用日志的结果就是发生在数据库上的更改顺序与远端复制数据库上的更改顺序需要保持完全同步。

Oracle,MySQL 和PostgreSQL都包括用于给备用的复制数据库传输日志的日志传输协议。Oracle还把日志产品化为一个通用的数据订阅机制，这样非Oracle 数据订阅用户就可以使用XStreams和GoldenGate订阅数据了，MySQL和PostgreSQL上的类似的实现则成为许多数据结构的关键组 件。正是由于这样的起源，机器可识别的日志的概念大部分都被局限在数据库内部。日志用做数据订阅的机制似乎是偶然出现的，不过要把这种 抽象用于支持所有类型的消息传输、数据流和实时数据处理是不切实际的。

分布式系统日志

日志解决了两个问题：更改动作的排序和数据的分发，这两个问题在分布式数据系统里显得尤为重要。协商出一致的更改动作的顺序（或者说保持各个子系统本身的做法，但可以进行存在副作用的数据拷贝）是分布式系统设计的核心问题之一。

以日志为中心实现分布式系统是受到了一个简单的经验常识的启发，我把这个经验常识称为状态机复制原理：如果两个相同的、确定性的进程从同一状态开始，并且以相同的顺序获得相同的输入，那么这两个进程将会生成相同的输出，并且结束在相同的状态。

这也许有点难以理解，让我们更加深入的探讨，弄懂它的真正含义。

确定性意味着处理过程是与时间无关的，而且任何其他“外部的“输入不会影响到处理结果。例如，如果一个程序的输出会受到线程执行的具体顺序影响，或者受到gettimeofday调用、或者其他一些非重复性事件的影响，那么这样的程序一般最有可能被认为是非确定性的。

进程状态是进程保存在机器上的任何数据，在进程处理结束的时候，这些数据要么保存在内存里，要么保存在磁盘上。

以相同的顺序获得相同输入的地方应当引起注意-这就是引入日志的地方。这儿有一个重要的常识：如果给两段确定性代码相同的日志输入，那么它们就会生成相同的输出。

分布式计算这方面的应用就格外明显。你可以把用多台机器一起执行同一件事情的问题缩减为实现分布式一致性日志为这些进程输入的问题。这儿日志的目的是把所有非确定性的东西排除在输入流之外，来确保每个复制进程能够同步地处理输入。

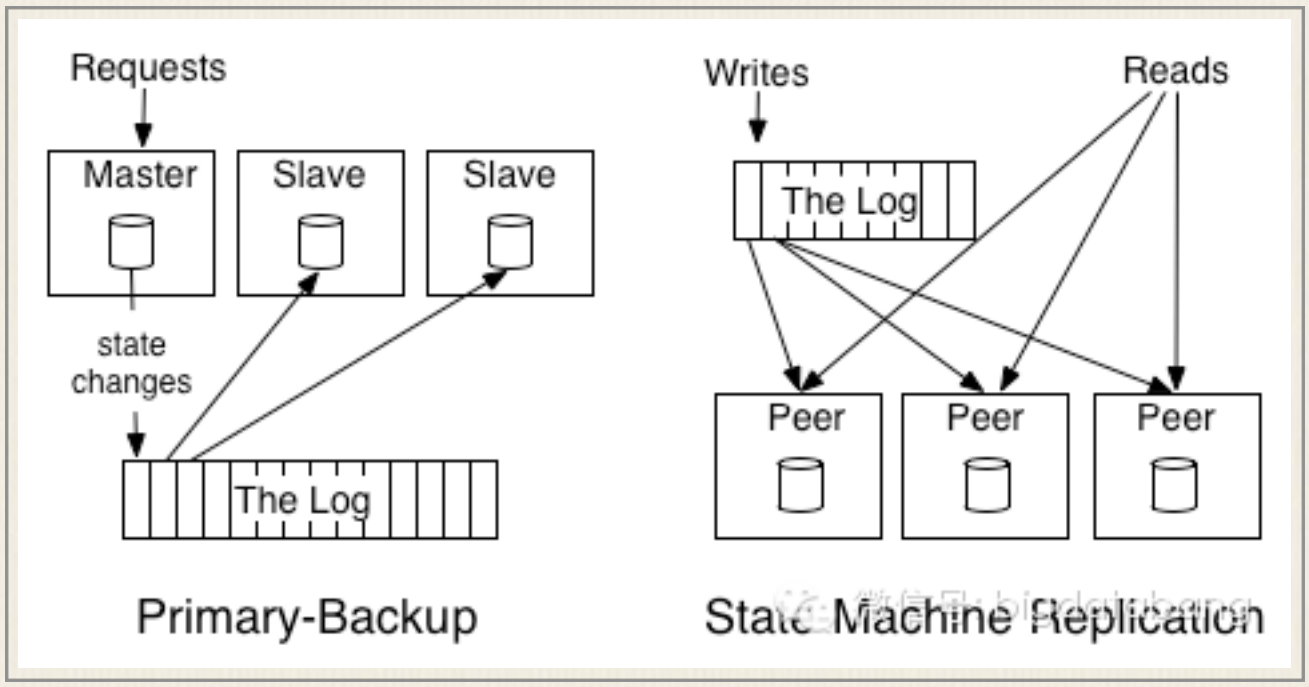
当你理解了这个以后，状态机复制原理就不再复杂或者说不再深奥了：这或多或少的意味着“确定性的处理过程就是确定性的”。不管怎样，我都认为它是分布式系统设计里较常用的工具之一。

这种方式的一个美妙之处就在于索引日志的时间戳就像时钟状态的一个副本——你可以用一个单独的数字描述每一个副本，这就是经过处理的日志的时间戳。时间戳与日志一一对应着整个副本的状态。

由 于写进日志的内容的不同，也就有许多在系统中应用这个原则的不同方式。举个例子，我们记录一个服务的请求，或者服务从请求到响应的状态变化，或者它执行命 令的转换。理论上来说，我们甚至可以为每一个副本记录一系列要执行的机器指令或者调用的方法名和参数。只要两个进程用相同的方式处理这些输入，这些进程就 会保持副本的一致性。

一千个人眼中有一千种日志的用法。数据库工作者通常区分物理日志和逻辑日志。物理日志就是记录每一行被改变的内容。逻辑日志记录的不是改变的行而是那些引起行的内容被改变的SQL语句（insert，update和delete语句）。

分 布式系统通常可以宽泛分为两种方法来处理数据和完成响应。“状态机器模型”通常引用一个主动-主动的模型——也就是我们为之记录请求和响应的对象。对此进 行一个细微的更改，称之为“预备份模型”，就是选出一个副本做为leader，并允许它按照请求到达的时间来进行处理并从处理过程中输出记录其状态改变的 日志。其他的副本按照leader状态改变的顺序而应用那些改变，这样他们之间达到同步，并能够在leader失败的时候接替leader的工作。



为 了理解两种方式的不同，我们来看一个不太严谨的例子。假定有一个算法服务的副本，保持一个独立的数字作为它的状态（初始值为0），并对这个值进行加法和乘 法运算。主动-主动方式应该会输出所进行的变换，比如“+1”，“*2”等。每一个副本都会应用这些变换，从而得到同样的解集。主动-被动方式将会有一个 独立的主体执行这些变换并输出结果日志，比如“1”，“3”，“6”等。这个例子也清楚的展示了为什么说顺序是保证各副本间一致性的关键：一次加法和乘法 的顺序的改变将会导致不同的结果。

分 布式日志可以理解为一致性问题模型的数据结构。因为日志代表了后续追加值的一系列决策。你需要重新审视Paxos算法簇，尽管日志模块是他们最常见的应 用。 在Paxos算法中，它通常通过使用称之为多paxos的协议，这种协议将日志建模为一系列的问题，在日志中每个问题都有对应的部分。在ZAB， RAFT等其它的协议中，日志的作用尤为突出，它直接对维护分布式的、一致性的日志的问题建模。

我怀疑的是，我们就历史 发展的观点是有偏差的，可能是由于过去的几十年中，分布式计算的理论远超过了其实际应用。在现实中，共识的问题是有点太简单了。计算机系统很少需要决定单 个值，他们几乎总是处理成序列的请求。这样的记录，而不是一个简单的单值寄存器，自然是更加抽象。

此外，专注于算法掩盖 了 抽象系统需要的底层的日志。我怀疑，我们最终会把日志中更注重作为一个商品化的基石，不论其是否以同样的方式 实施的，我们经常谈论一个哈希表而不是纠结我们 得到是不是具体某个细节的哈希表，例如线性或者带有什么什么其它变体哈希表。日志将成为一种大众化的接口，为大多数算法和其实现提升提供最好的保证和最佳 的性能。

变更日志101： 表与事件的二相性

让 我们继续聊数据库。数据库中存在着大量变更日志和表之间的二相性。这些日志有点类似借贷清单和银行的流程，数据库表就是当前的盈余表。如果你有大量的变更 日志，你就可以使用这些变更用以创建捕获当前状态的表。这张表将记录每个关键点（日志中一个特别的时间点）的状态信息。这就是为什么日志是非常基本的数据 结构的意义所在：日志可用来创建基本表，也可以用来创建各类衍生表。同时意味着可以存储非关系型的对象。

这 个流程也是可逆的：如果你正在对一张表进行更新，你可以记录这些变更，并把所有更新的日志发布到表的状态信息中。这些变更日志就是你所需要的支持准实时的 克隆。基于此，你就可以清楚的理解表与事件的二相性： 表支持了静态数据而日志捕获变更。日志的魅力就在于它是变更的完整记录，它不仅仅捕获了表的最终版本的内容，它还记录了曾经存在过的其它版本的信息。日志 实质上是表历史状态的一系列备份。

这可能会引起你对源代码的版本管理。源代码管理和数据库之间有密切关 系。版本管理解决了一个大家非常熟悉的问题，那就是什么是分布式数据系统需要解决的— 时时刻刻在变化着的分布式管理。版本管理系统通常以补丁的发布为基础，这实际上可能是一个日志。您可以直接对当前 类似于表中的代码做出“快照”互动。你会注意到， 与其他分布式状态化系统类似，版本控制系统 当你更新时会复制日志，你希望的只是更新补丁并将它们应用到你的当前快照中。

最近，有些人从Datomic –一家销售日志数据库的公司得到了一些想法。这些想法使他们对如何 在他们的系统应用这些想法有了开阔的认识。 当然这些想法不是只针对这个系统，他们会成为 十多年分布式系统和数据库文献的一部分。

这可能似乎有点过于理想化。但是不要悲观！我们会很快把它实现。

接下来的内容

在这篇文章的其余部分，我将试图说明日志除了可用在分布式计算或者抽象分布式计算模型内部之外，还可用在哪些方面。其中包括：

数据集成- 让机构的全部存储和处理系统里的所有数据很容易地得到访问。

实时数据处理-计算生成的数据流。

分布式系统设计- 实际应用的系统是如何通过使用集中式日志来简化设计的。

所有这些用法都是通过把日志用做单独服务来实现的。

在上面任何一种用法里，日志的用途开始都是使用了日志所能提供的某个简单功能：生成永久的、可重现的历史记录。令人意外的是，问题的核心是可以让多少台机器以特定的方式，按照自身的速度重现历史记录的能力。

第二部分:数据集成

请让我首先解释 一下“数据集成”是什么意思，还有为什么我觉得它很重要，之后我们再来看看它和日志有什么关系。

数 据集成就是将数据组织起来，使得在与其有关的服务和系统中可以访问它们。“数据集成”（data integration）这个短语应该不止这么简单，但是我找不到一个更好的解释。而更常见的术语 ETL 通常只是覆盖了数据集成的一个有限子集(译注：ETL，Extraction-Transformation-Loading的缩写，即数据提取、转换和 加载)——相对于关系型数据仓库。但我描述的东西很大程度上可以理解为，将ETL推广至实时系统和处理流程。

你一定不会听到数据集成就兴趣盎然屏住呼吸，并且天花乱坠的想到关于大数据的概念，不过，我相信世俗的问题“让数据可被访问” 是一个组织应该关注的有价值的事情。

对 数据的高效使用遵循一种 马斯洛的需要层次理论 。金字塔的基础部分包括捕获所有相关数据，能够将它们全部放到适当的处理环境（那个环境应该是一个奇妙的实时查询系统，或者仅仅是文本文件和python 脚本）。这些数据需要以统一的方式建模，这样就可以方便读取和数据处理。如果这种以统一的方式捕获数据的基本需求得到满足，那么就可以在基础设施上以若干 种方法处理这些数据——映射化简（MapReduce），实时查询系统，等等。

很明显，有一点值得注意：如果没有可靠 的、完整的数据流，Hadoop集群除了象昂贵的且难于安装的空间取暖器哪样外不会做更多事情了。一旦数据和处理可用，人们就会关心良好数据模型和一致地 易于理解的语法哪些更细致的问题。最后，人们才会关注更加高级的处理-
更好的可视化、报表以及处理和预测算法。

以我的经验，大多数机构在数据金字塔的底部存在巨大的漏洞-它们缺乏可靠的、完整的数据流-而是打算直接跳到高级数据模型技术上。这样做完全是反着来做的。
因此，问题是我们如何构建通过机构内所有数据系统的可靠的数据流。

数据集成：两个并发症

两种趋势使数据集成变得更困难。

事件数据管道

第 一个趋势是增长的事件数据(event data)。事件数据记录的是发生的事情，而不是存在的东西。在web系统中，这就意味着用户活动日志，还有为了可靠的操作以及监控数据中心的机器的目 的，所需要记录的机器级别的事件和统计数字。人们倾向称它们为“日志数据”，因为它们经常被写到应用的日志中，但是这混淆了形式与功能。这种数据位于现代 web的中心：归根结底，Google的资产是由这样一些建立在点击和映像基础之上的相关管道所生成的——那也就是事件。

这些东西并不是仅限于网络公司，只是网络公司已经完全数字化，所以它们更容易用设备记录。财务数据一直是面向事件的。RFID(无线射频识别)将这种跟踪能力赋予物理对象。我认为这种趋势仍将继续，伴随着这个过程的是传统商务活动的数字化。

这种类型的事件数据记录下发生的事情，而且往往比传统数据库应用要大好几个数量级。这对于处理提出了重大挑战。

专门的数据系统的爆发

第二个趋势来自于专门的数据系统的爆发，通常这些数据系统在最近的五年中开始变得流行，并且可以免费获得。专门的数据系统是为OLAP, 搜索, 简单 在线 存储, 批处理, 图像分析, 等 等 而存在的。

更多的不同类型数据的组合，以及将这些数据存放到更多的系统中的愿望，导致了一个巨大的数据集成问题。

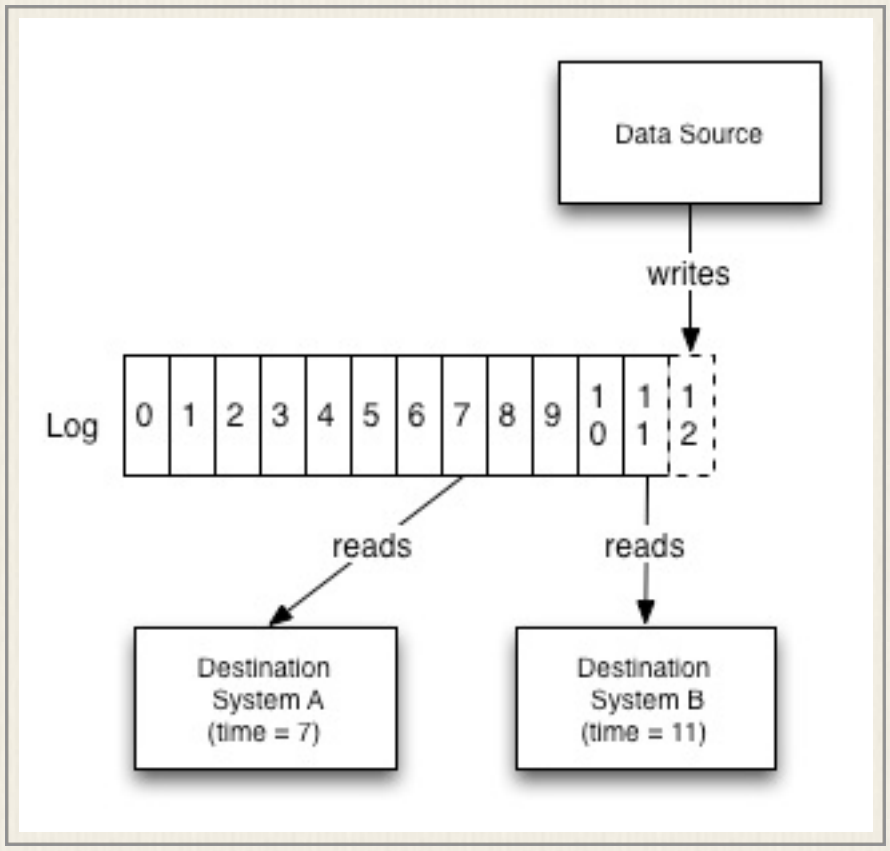
日志结构数据流

为了处理系统之间的数据流，日志是最自然的数据结构。其中的秘诀很简单：

将所有组织的数据提取出来，并将它们放到一个中心日志，以便实时查阅。

每 个逻辑数据源都可以建模为它自己的日志。一个数据源可以是一个应用程序的事件日志（如点击量或者页面浏览量），或者是一个接受修改的数据库表。每个订阅消 息的系统都尽可能快的从日志读取信息，将每条新的记录保存到自己的存储，并且提升其在日志中的地位。订阅方可以是任意一种数据系统 —— 一个缓存，Hadoop，另一个网站中的另一个数据库，一个搜索系统，等等。

例如，日志针对每个更改给出了逻辑时钟的概念，这样所有的订阅方都可以被测量。推导不同的订阅系统的状态也因此变得相对简单的多，因为每个系统都有一个读取动作的“时间点”。



为 了让这个显得更具体，我们考虑一个简单的案例，有一个数据库和一组缓存服务器集群。日志提供了一种同步更新所有这些系统，并推导出每一个系统的接触时间点 的方法。我们假设写了一条日志X，然后需要从缓存做一次读取。如果我们想保证看到的不是陈旧的数据，我们只需保证没有从任何尚未复制X的缓存中读取即可。

日 志也起到缓存的作用，使数据生产与数据消费相同步。由于许多原因这个功能很重要，特别是在多个订阅方消费数据的速度各不相同的时候。这意味着一个订阅数据 系统可以宕机，或者下线维护，之后重新上线以后再赶上来：订阅方按照自己控制的节拍来消费数据。批处理系统，如Hadoop或者是一个数据仓库，或许只是 每小时或者每天消费一次数据，而实时查询系统可能需要及时到秒。由于无论是原始数据源还是日志，都没有各种目标数据系统的相关知识，因此消费方系统可以被 添加和删除，而无需传输管道的变化。

“每个工作数据管道设计得就像是一个日志；每个损坏的数据管道以其自己的方式损坏。”—Count Leo Tolstoy (由作者翻译)

特 别重要的是：目标系统只知道日志，不知道数据源系统的任何细节。消费方系统自身无需考虑数据到底是来自于一个RDBMS（关系型数据库管理系统 Relational Database Management System），一种新型的键值存

储，或者它不是由任何形式的实时查询系统所生成的。这似乎是一个小问题，但实际上是至关重要的。

这 里我使用术语“日志”取代了“消息系统”或者“发布-订阅”，因为它在语义上更明确，并且对支持数据复制的实际实现这样的需求，有着更接近的描述。我发现 “发布订阅”并不比间接寻址的消息具有更多的含义——如果你比较任何两个发布-订阅的消息传递系统的话，你会发现他们承诺的是完全不同的东西，而且大多数 模型在这一领域都不是有用的。你可以认为日志是一种消息系统，它具有持久性保证和强大的订阅语义。在分布式系统中，这个通信模型有时有个(有些可怕的)名 字叫做原子广播。

值得强调的是，日志仍然只是基础设施。这并不是管理数据流这个故事的结束：故事的其余部分围绕着元数据，模式，兼容性，以及处理数据结构的所有细节及其演化。除非有一种可靠的，一般的方法来处理数据流运作，语义在其中总是次要的细节。

在 LinkedIn（SNS社交网站）

在LinkedIn从集中式关系数据库向分布式系统集合转化的过程中，我看到这个数据集成问题迅速演变。

现在主要的数据系统包括：

搜索

社交图谱

Voldemort (键值存储)(译注：一种分布式数据库)

Espresso (文档存储)

推举引擎

OLAP查询引擎(译注：OLAP联机分析技术)

Hadoop

Terradata

Ingraphs (监控图表和指标服务)

这些都是专门的分布式系统，在其专业领域提供先进的功能。

这 种使用日志作为数据流的思想，甚至在我到这里之前就已经与LinkedIn相伴了。我们开发的一个最早的基础设施之一，是一种称为databus 的服务，它在我们早期的Oracle表上提供了一种日志缓存抽象，可伸缩订阅数据库修改,这样我们就可以很好支持我们的社交网络和搜索索引。

我 会给出一些历史并交代一下上下文。我首次参与到这些大约是在2008年左右，在我们转移键值存储之后。我的下一个项目是让一个工作中的Hadoop配置演 进，并给其增加一些我们的推荐流程。由于缺乏这方面的经验，我们自然而然的安排了数周计划在数据的导入导出方面，剩下的时间则用来实现奇妙的预测算法。这 样我们就开始了长途跋涉。

我们本来计划是仅仅将数据从现存的Oracle数据仓库中剖离。但是我们首先发现将数据从 Oracle中迅速取出是一种黑暗艺术。更糟的是，数据仓库的处理过程与我们为Hadoop而计划的批处理生产过程不适合——其大部分处理都是不可逆转 的，并且与即将生成的报告具体相关。最终我们采取的办法是，避免使用数据仓库，直接访问源数据库和日志文件。最后，我们为了加载数据到键值存储 并生成结果，实现了另外一种管道。

这种普通的数据复制最终成为原始开发项目的主要内容之一。糟糕的是，在任何时间任意管道都有一个问题，Hadoop系统很大程度上是无用的——在错误的数据基础上运行奇特的算法，只会产生更多的错误数据。

虽 然我们已经以一种通用的方式创建事物，但是每个数据源都需要自定义配置安装。这也被证明是巨量错误与失败的根源。我们在Hadoop上实现的网站功能已经 开始流行起来，同时我们发现我们有一长串感兴趣的工程师。每个用户都有他们想要集成的一系列系统，他们想要的一系列新数据源。

古希腊时代的 ETL（提取转换加载Extract Transform and Load）。并没有太多变化

有些东西在我面前开始渐渐清晰起来。

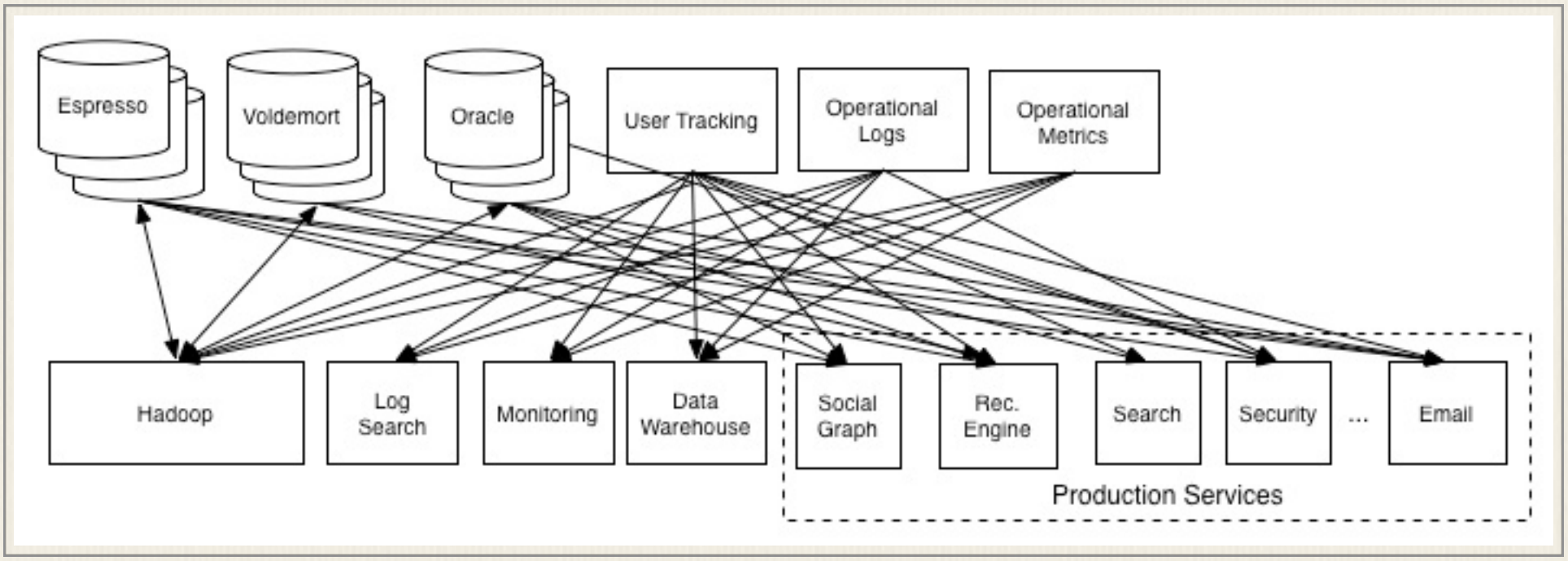
首先，我们已建成的通道虽然有一些杂乱，但实质上它们是很有价值的。在采用诸如Hadoop的新的处理系统生成可用数据的过程，它开启了大

量的可能性。 基于这些数据过去很难实现的计算，如今变为可能。 许多新的产品和分析技术都来源于把分片的数据放在一起，这些数据过被锁定在特定的系统中。

第二， 众所周知，可靠的数据加载需要数据通道的深度支持。如果我们可以捕获所有我们需要的结构，我就就可以使得Hadoop数据全自动的加载，这样就不需要额外 的操作来增加新的数据源或者处理模式变更–数据就会自动的出现在HDFS，Hive表就会自动的生成对应于新数据源的恰当的列。

第三，我们的数据覆盖率仍然非常低。如果你查看存储于Hadoop中的可用的Linked 数据的全部百分比，它仍然是不完整的。花费大量的努力去使得各个新的数据源运转起来，使得数据覆盖度完整不是一件容易的事情。

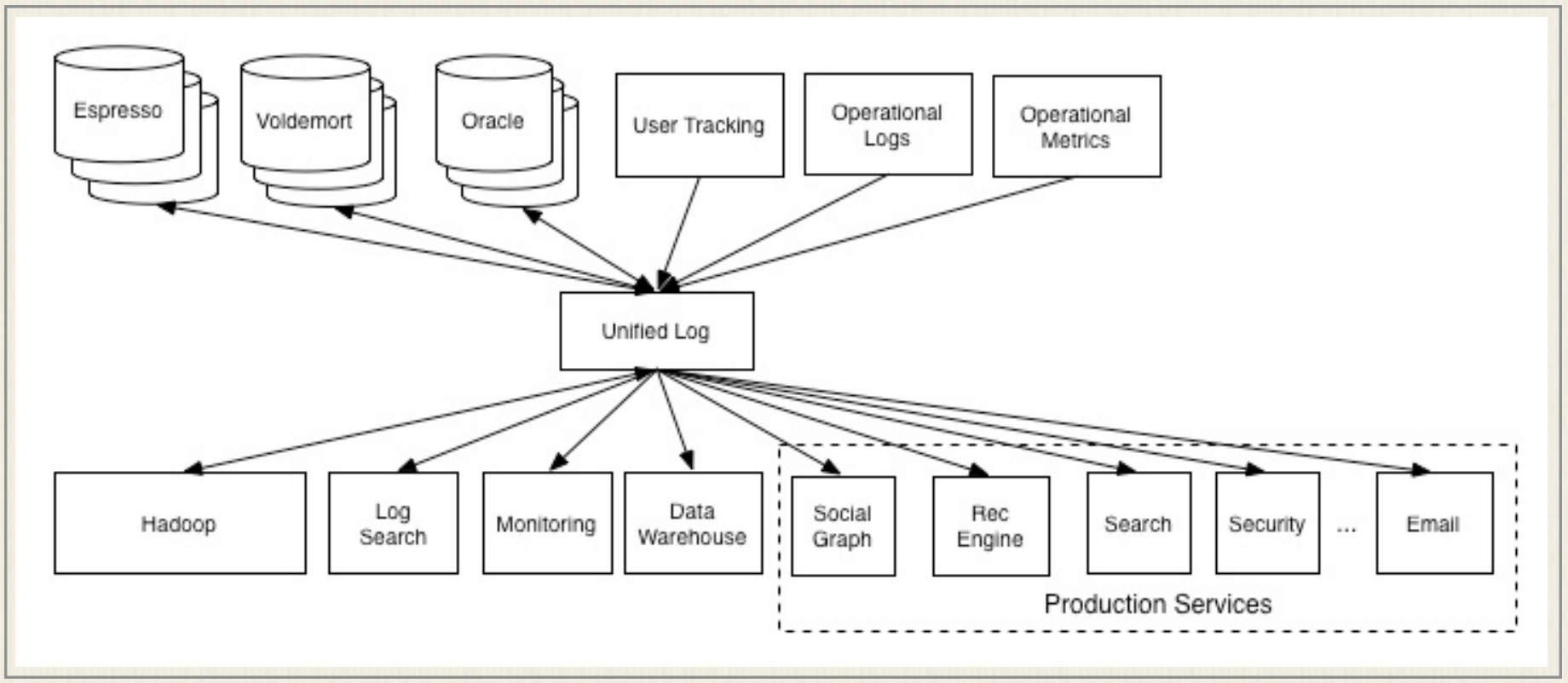
我们正在推行的，为每个数据源和目标增建客户化数据加载，这种方式很显然是不可行的。我们有大量的数据系统和数据仓库。把这些系统和仓库联系起来，就会导致任意一对系统会产生如下所示的客户化通道。



需要注意的是：数据是双向流动的：例如许多系统诸如数据库和Hadoop既是数据转化的来源又是数据转化的目的地。这就意味着我们我们不必为每个系统建立两个通道：一个用于数据输入，一个用于数据输出。

这显然需要一大群人，而且也不具有可操作性。随着我们接近完全连接，最终我们将有差不多O（N2）条管道。

替代的，我们需要像这样通用的东西：



我们需要尽可能的将每个消费者与数据源隔离。理想情形下，它们应该只与一个单独的数据仓库集成，并由此让他们能访问到所有东西。

这个思想是增加一个新的数据系统——或者它是一个数据源或者它是一个数据目的地——让集成工作只需连接到一个单独的管道，而无需连接到每个数据消费方。

这种经历使得我关注创建Kafka来关联我们在消息系统所见的与数据库和分布式系统内核所发布的日志。我们希望一些实体作为中心的通道，首先用于所有的活动数据，逐步的扩展到其他用途，例如Hadoop外的数据实施，数据监控等。

在相当长的时间内，Kafka是独一无二的底层产品，它既不是数据库，也不是日志文件收集系统，更不是传 统的消息系统。但是最近Amazon提供了非常类似Kafka的服务，称之为Kinesis。相似度包括了分片处理的方式，数据的保持，甚至包括在 Kafka API中，有点特别的高端和低端消费者分类。

我很开心看到这些，这表明了你已经创建了很好的底层协议，AWS已经把它作为服务提供。他们对此的期待与我所 描述的吻合：通道联通了所有的分布式系统，诸如DynamoDB，RedShift，S3等，它同时作为使用EC2进行分布式流处理的基础。

与ETL和数据仓库的关系

我们再来聊聊数据仓库。数据仓库是清洗和归一数据结构用于支撑数据分析的仓库。这是一个伟大的理念。对不 熟悉数据仓库概念的人来说，数据仓库方法论包括了：周期性的从数据源抽取数据，把它们转化为可理解的形式，然后把它导入中心数据仓库。对于数据集中分析和 处理，拥有高度集中的位置存放全部数据的原始副本是非常宝贵的资产。在高层级上，也许你抽取和加载数据的顺序略微调整，这个方法论不会有太多变化,无论你 使用传统的数据仓库Oracle还是Teradata或者Hadoop。

数据仓库是极其重要的资产，它包含了原始的和规整的数据，但是实现此目标的机制有点过时了。

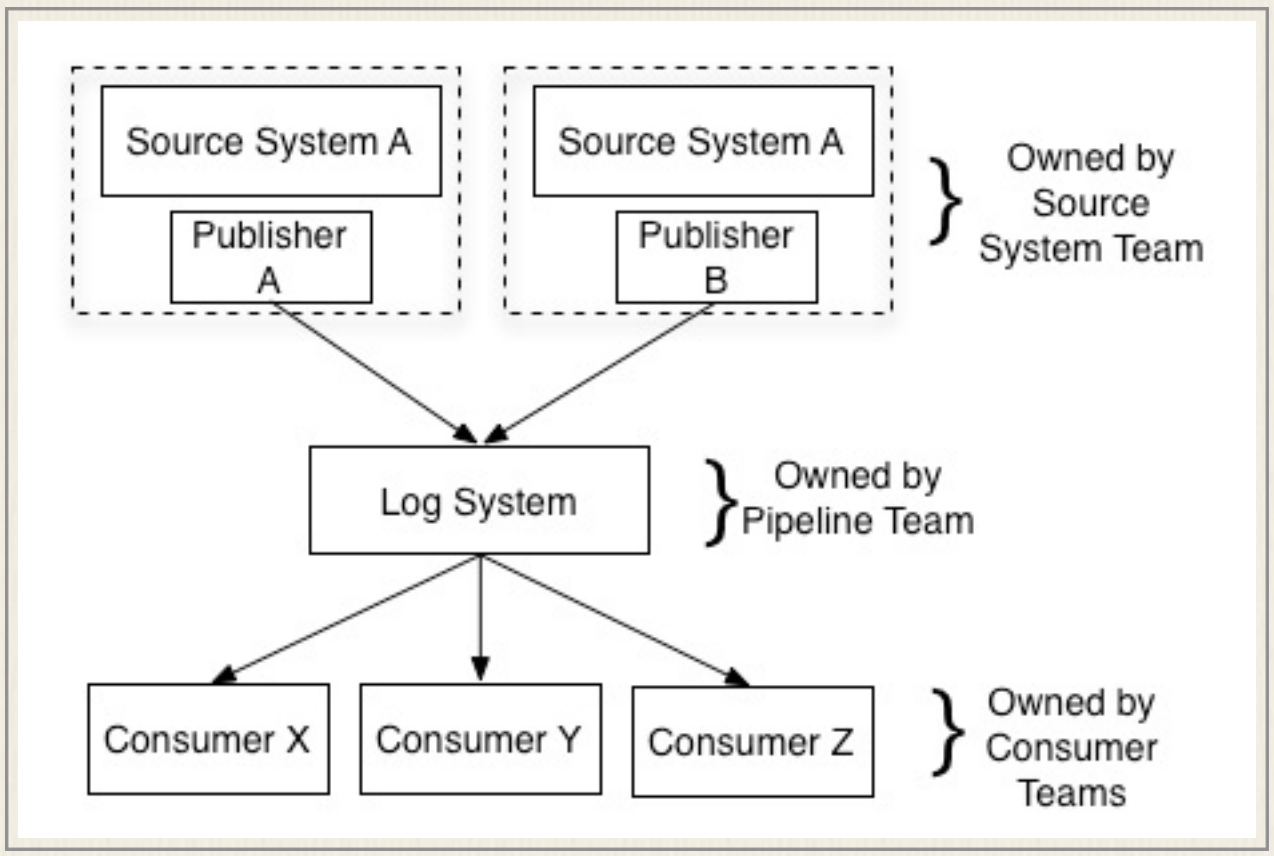
对以数据为中心的组织关键问题是把原始的归一数据联结到数据仓库。数据仓库是批处理的基础查询：它们适用于各类报表和临时性分析，特别是当查询包含 了简单的计数、聚合和过滤。但是如果一个批处理系统仅仅包含了原始的完整的数据的数据仓库，这就意味着这些数据对于实时数据处理、搜索索引和系统监控等实 时的查询是不可用的。

依我之见，ETL包括两件事：首先，它是抽取和数据清洗过程–特别是释放被锁在组织的各类系统中的数据， 移除系统专有的无用物。第二，依照数据仓库的查询重构数据。例如使其符合关系数据库类型系统，强制使用星号、雪花型模式，或者分解为高性能的柱状格式等。 合并这两者是有困难的。这些规整的数据集应当可以在实时或低时延处理中可用，也可以在其它实施存储系统索引。

在我看来，正是因为这个原因有了额外好处：使得数据仓库ETL更具了组织级的规模。数据仓库的精典问题是 数据仓库负责收集和清洗组织中各个组所生成的全部数据。各组织的动机是不同的，数据的生产者并不知晓在数据仓库中数据的使用情况，最终产生的数据很难抽 取，或者需要花费规

模化的转化才可以转化为可用的形式。当然， 中心团队不可能恰到好处的掌握规模，使得这规模刚好与组织中其它团队相匹配，因此数据的覆盖率常常差别很大，数据流是脆弱的同时变更是缓慢的。

较好的方法是有一个中心通道，日志和用于增加数据的定义良好的API。与通道集成的且提供良好的结构化的 数据文件的职责依赖于数据的生产者所生成的数据文件。这意味着在设计和实施其它系统时应当考虑数据的输出以及输出的数据如何转化为结构良好的形式并传递给 中心通道。增加新的存储系统倒是不必因为数据仓库团队有一个中心结点需要集成而关注数据仓库团队。数据仓库团队仅需处理简单的问题，例如从中心日志中加载 结构化的数据，向其它周边系统实施个性化的数据转化等。



如图所示：当考虑在传统的数据仓库之外增加额外的数据系统时，组织结构的可扩展性显得尤为重要。例如，可以考虑为组织的完整的数据集提供搜索功能。 或者提供二级的数据流监控实时数据趋势和告警。无论是这两者中的哪一个，传统的数据仓库架构甚至于Hadoop聚簇都不再适用。更糟的是，ETL的流程通 道的目的就是支持数据加载，然而ETL似乎无法输出到其它的各个系统，也无法通过引导程序，使得这些外围的系统的各个架构成为适用于数据仓库的重要资产。 这就不难解释为什么组织很难轻松的使用

它的全部数据。反之，如果组织已建立起了一套标准的、结构良好的数据，那么任何新的系统要使用这些数据仅仅需要与通 道进行简单的集成就可以实现。

这种架构引出了数据清理和转化在哪个阶段进行的不同观点：

- 由数据的生产者在把数据增加到公司全局日志之前。
- 在日志的实时转化阶段进行，这将会产生一个新的转化日志。
- 在向目标系统加载数据时，做为加载过程的一部分进行。

理想的模形是：由数据的生产者在把数据发布到日志之前对数据进行清理。这样可以确保数据的权威 性，不需要维护其它的遗留物例如为数据产生的特殊处理代码或者维护这些数据的其它的存储系统。这些细节应当由产生数据的团队来处理，因为他们最了解他们自 己的数据。这个阶段所使用的任何逻辑都应该是无损的和可逆的。

任何可以实时完成的增值转化类型都应当基于原始日志进行后期处理。这一过程包括了事件数据的会话流程，或者增加大众感兴趣的衍生字段。原始的日志仍然是可用的，但是这种实时处理产生的衍生日志包含了参数数据。

最终，只有针对目标系统的聚合需要做了加载流程的一部分。它包括了把数据转化成特定的星型或者雪花状模式，从而用于数据仓库的分析和报表。因为在这个阶段，大部分自然的映射到传统的ETL流程中，而现在它是在一个更加干净和规整的数据流集在进行的，它将会更加的简单。

日志文件和事件

我们再来聊聊这种架构的优势：它支持解耦和事件驱动的系统。

在网络行业取得活动数据的典型方法是把它记为文本形式的日志，这些文本文件是可分解进入数据仓库或者Hadoop，用于聚合和查询处理的。由此产生的问题与所有批处理的ETL的问题是相同的：它耦合了数据流进入数据仓库系统的能力和流程的调度。

在LinkedIn中，我们已经以中心日志的方式构建了事件数据处理。我们正在使用Kafka做 为中心的、多订阅者事件日志。我们已经定义了数百种事件类型，每种类型都会捕获用于特定类型动作的独特的属性。这将会覆盖包括页面视图、表达式、搜索以及 服务调用、应用异常等方方面面。

为了进一步理解这一优势：设想一个简单的事务–在日志页面显示已发布的日志。这个日志页面应当只包括显示日志所需要的逻辑。然而，在相当多的动态站点中，日志页面常常变的添加了很多与显示日志无关的逻辑。例如，我们将对如下的系统进行集成：

- 需要把数据传送到Hadoop和数据仓库中用于离线数据处理。
- 需要对视图进行统计，确保视图订阅者不会攻击一些内容片段。
- 需要聚合这些视图，视图将用于作业发布者的分析页面显示。
- 需要记录视图以确保我们为作业推荐的使用者提供了恰当的印象覆盖，我们不想一次次的重复同样的事情。
- 推荐系统需要记录日志用于正确的跟踪作业的普及度。
- 等等。

不久，简单的作业显示变得相当的复杂。我们增加了作业显示的其它终端——移动终端应用等——这些 逻辑必须继续存在，复杂度不断的增加。更糟的是我们需要与之做接口交互的系统现在是错综复杂的–在为显示日作业而工作的工程师们需要知晓多个其它系统和它 们的特征，才可以确保它们被正确的集成了。这仅仅是问题的简单版本，真实的的应用系统只会更加的复杂。

“事件驱动”的模式提供了一种简化这类问题的机制。作业显示页面现在只显示作业并记录与正在显示 的作业，作业订阅者相关的其它属性，和其它与作业显示相关的其它有价值的属性。每个与此相关的其它系统诸如推荐系统、安全系统、作业推送分析系统和数据仓 库，所有这些只是订阅种子文件，并进行它们的操作。显示代码并不需要关注其它的系统，也不需要因为增加了数据的消费者而相应的进行变更。

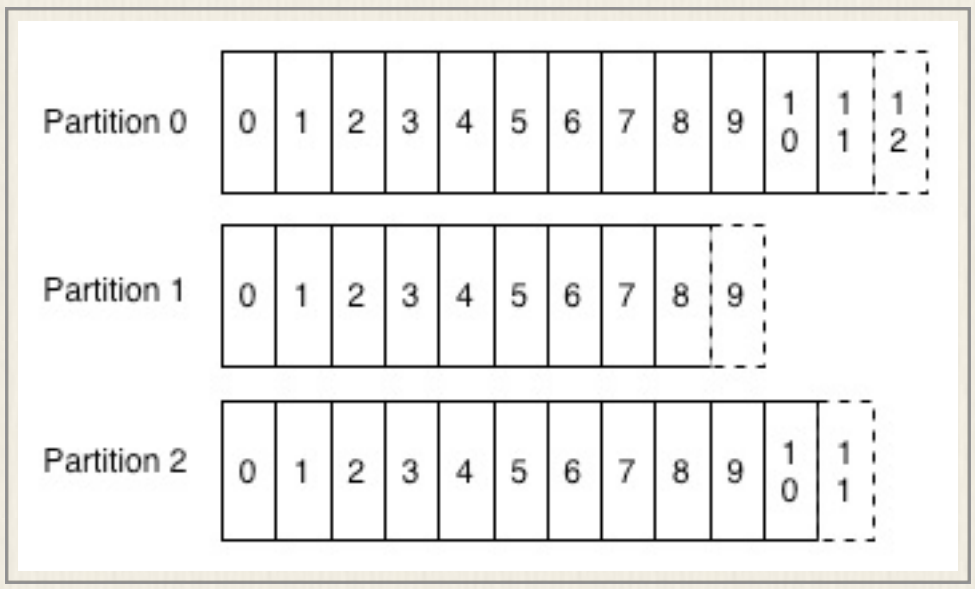
构建可伸缩的日志

当然，把发布者与订阅者分离不再是什么新鲜事了。但是如果你想要确保提交日志的行为就像多个订阅 者实时的分类日志那样记录网站发生的每件事时，可扩展性就会成为你所面临的首要挑战。如果我们不能创建快速、高性价比和可扩展性灵活的日志以满足实际的可 扩展需求，把日志做为统一的集成机制不再是美好的想像，

人们普遍认为分布式日志是缓慢的、重量经的概念（并且通常会把它仅仅与“原数据”类型的使用联系 起来，对于这类使用Zookeeper可以适用）。但是深入实现并重点关注分类记录大规模的数据流，这种需求是不切实际的。在LinkedIn，我们现在 每天通过Kafka运行着超过600亿个不同的消息写入点（如果统计镜相与数据中心之间的写入，那么这个数字会是数千亿）。

我们在Kafk中使用了一些小技巧来支持这种可扩展性：

- 日志分片
- 通过批处理读出和写入优化吞吐力
- 规避无用的数据复制。



为了确保水平可扩展性，我们把日志进行切片：

每个切片都是一篇有序的日志，但是各片之间没有全局的次序（这个有别于你可能包含在消息中的挂钟 时间）。把消息分配到特定的日志片段这是由写入者控制的，大部分使用者会通过用户ID等键值来进行分片。分片可以把日志追加到不存在协作的片段之间，也可 以使系统的吞吐量与Kafka聚簇大小成线性比例关系。

每个分片都是通过可配置数量的复制品复制的，每个复制品都有分片的一份完全一致的拷贝。无论何时，它们中的任一个都可以做为主分片，如果主分片出错了，任何一个复制品都可以接管并做为主分片。

缺少跨分片的全局顺序是这个机制的局限性，但是我们不认为它是最主要的。事实上，与日志的交互主 要来源于成百上千个不同的流程，以致于对于它们的行为排一个总体的顺序是没什么意义的。相反，我们可以确保的是我们提供的每个分片都是按顺序保留的。 Kafka保证了追加到由单一发送者送出的特定分片会按照发送的顺序依次处理。

日志，就像文件系统一样，是容易优化成线性可读可写的样式的。日志可以把小的读入和写出组合成大 的、高吞吐量的操作。Kafka一直至立于实现这一优化目标。批处理可以发生在由客户端向服务器端发送数据、写入磁盘;在服务器各端之间复制；数据传递给 消费者和确认提交数据等诸多环节。

最终，Kafka使用简单的二进制形式维护内存日志，磁盘日志和网络数据传送。这使得我们可以使用包括“0数据复制传送”在内的大量的优化机制。

这些优化的积累效应是你常常进行的写出和读入数据的操作可以在磁盘和网络上得到支持，甚至于维护内存以外的大量数据集。

这些详细记述并不意味着这是关于Kafka的主要内容，那么我就不需要了解细节了。你可阅读到更多的关于LinkedIn的方法在这个链接，和Kafka的设计总述在这个链接。

第三部分：日志和实时流处理

到此为止，我只是描述从端到端数据复制的理想机制。但是在存储系统中搬运字节不是所要讲述内容的全部。最终我们发现日志是流的另一种说法，日志是流处理的核心。

但是，等等，什么是流处理呢?

如果你是90年代晚期或者21世纪初数据库文化或者数据基础架构产品的爱好者，那么你就可能会把流处理与建创SQL引擎或者创建“箱子和箭头”接口用于事件驱动的处理等联系起来。

如果你关注开源数据库系统的大量出现，你就可能把流处理和一些开源数据库系统关联起来，这些系统 包括了：Storm，Akka，S4和Samza。但是大部分人会把这些系统作为异步消息处理系统，这些系统与支持群集的远程过程调用层的应用没什么差别 （而事实上在开源数据库系统领域某些方面确实如此）。

这些视图都有一些局限性。流处理与SQL是无关的。它也局限于实时流处理。不存在内在的原因限制你不能处理昨天的或者一个月之前的流数据，且使用多种不同的语言表达计算。

我把流处理视为更广泛的概念：持续数据流处理的基础架构。我认为计算模型可以像MapReduce或者分布式处理架构一样普遍，但是有能力处理低时延的结果。

处理模型的实时驱动是数据收集方法。成批收集的数据是分批处理的。数据是不断收集的，它也是按顺序不断处理的。

美国的统计调查就是成批收集数据的良好典范。统计调查周期性的开展，通过挨门挨户的走 访，使用蛮力发现和统计美国的公民信息。1790年统计调查刚刚开始时这种方式是奏效的。那时的数据收集是批处理的，它包括了骑着马悠闲的行进，把信息写 在纸上，然后把成批的记录传送到人们统计数据的中心站点。现在，在描述这个统计过程时，人们立即会想到为什么我们不保留出生和死亡的记录，这样就可以产生 人口统计信息这些信息或是持续的或者是其它维度的。

这是一个极端的例子，但是大量的数据传送处理仍然依赖于周期性的转储，批量转化和集成。处理大容量转储的唯一方法就是批量的处理。但是随着这些批处理被持续的供给所取代，人们自然而然的开始不间断的处理以平滑的处理所需资源并且消除延迟。

例如LinkedIn几乎没有批量数据收集。大部分的数据或者是活动数据或者是数据库变 更，这两者都是不间断发生的。事实上，你可以想到的任何商业，正如：Jack Bauer告诉我们的，低层的机制都是实时发生的不间断的流程事件。数据是成批收集的，它总是会依赖于一些人为的步骤，或者缺少数字化或者是一些自动化的 非数字化流程处理的遗留信息。当传送和处理这些数据的机制是邮件或者人工的处理时，这一过程是非常缓慢的。首轮自动化总是保持着最初的处理形式，它常常会 持续相当长的时间。

每天运行的批量处理作业常常是模拟了一种一天的窗口大小的不间断计算。当然，低层的数据也经常变化。在LinkedIn，这些是司空见贯的，并且使得它们在Hadoop运转的机制是有技巧的，所以我们实施了一整套管理增量的Hadoop工作流的架构。

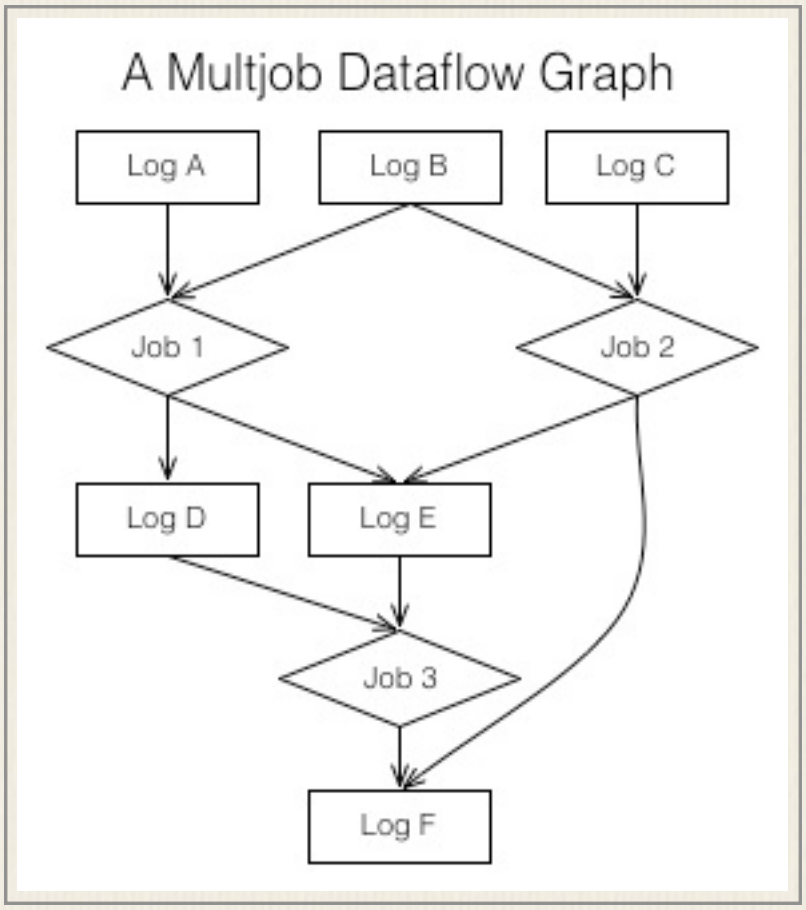
由此看来，对于流处理可以有不同的观点。流处理包括了在底层数据处理的时间概念，它不需要数据的静态快照，它可以产生用户可控频率的输出，而不用等待数据集的全部到达。从这个角度上讲，流处理就是广义上的批处理，随着实时数据的流行，会儿更加普遍。

这就是为什么从传统的视角看来流处理是利基应用。我个人认为最大的原因是缺少实时数据收集使得不间断的处理成为了学术性的概念。

我想缺少实时数据收集就像是商用流处理系统注定的命运。他们的客户仍然需要处理面向文件 的、每日批量处理ETL和数据集成。公司建设流处理系统关注的是提供附着在实时数据流的处理引擎，但是最终当时极少数人真正使用了实时数据流。事实上，在 我在LinkedIn工作的初期，有一家公司试图把一个非常棒的流处理系统销售给我们，但是因为当时我们的全部数据都按小时收集在的文件里，当时我们提出 的最好的应用就是在每小时的最后把这些文件输入到流处理系统中。他们注意到这是一个普遍性的问题。这些异常证明了如下规则：流处理系统要满足的重要商业目 标之一是：财务， 它是实时数据流已具备的基准，并且流处理已经成为了瓶颈。

甚至于在一个健康的批处理系统中，流处理作为一种基础架构的实际应用能力是相当广泛的。它跨越了实时数据请求——应答服务和离线批量处理之间的鸿沟。现在的互联网公司，大约25%的代码可以划分到这个类型中。

最终这些日志解决了流处理中绝大部分关键的技术问题。在我看来，它所解决的最大的问题是 它使得多订阅者可以获得实时数据。对这些技术细节感兴趣的朋友，我们可以用开源的Samza,它是基于这些理念建设的一个流处理系统。这些应用的更多技术 细节我们在此文档中有详细的描述。



流处理最有趣的角度是它与流处理系统内部无关，但是与之密切相关的是如何扩展了我们谈到的早期数据集成的数据获取的理念。我们主要讨论了基础数据的 获取或日志——事件和各类系统执行中产生的数据等。但是流处理允许我们包括了计算其它数据的数据。这些衍生的数据在消费者看来与他们计算的原始数据没什么 差别。这些衍生的数据可以按任意的复杂度进行压缩。

让我们再深入一步。我们的目标是：流处理作业可以读取任意的日志并把日志写入到日志或者 其它的系统中。他们用于输入输出的日志把这些处

理关联到一组处理过程中。事实上，使用这种样式的集中日志，你可以把组织全部的数据抓取、转化和工作流看成 是一系列的日志和写入它们的处理过程。

流处理器根本不需要理想的框架：它可能是读写日志的任何处理器或者处理器集合，但是额外的基础设施和辅助可以提供帮助管理处理代码。

日志集成的目标是双重的：

首先，它确保每个数据集都有多个订阅者和有序的。让我们回顾一下状态复制原则来记住顺序的重要性。为了使这个更加具体，设想一下从数据库中更新数据流–如果在处理过程中我们把对同一记录的两次更新重新排序，可能会产生错误的输出。 TCP之类的链接仅仅局限于单一的点对点链接，这一顺序的持久性要优于TCP之类的链接，它可以在流程处理失败和重连时仍然存在。

第二，日志提供了流程的缓冲。这是非常基础的。如果处理流程是非同步的，那么上行生成流 数据的作业比下行消费流数据的作业运行的更快。这将会导致处理流程阻塞，或者缓冲数据，或者丢弃数据。丢弃数据并不是可行的方法，阻塞将会导致整个流程图 立即停止。 日志实际上是一个非常大的缓冲，它允许流程重启或者停止但不会影响流程图其它部分的处理速度。如果要把数据流扩展到更大规模的组织，如果处理作业是由多个 不同的团队提供的，这种隔离性是极其重的。我们不能容忍一个错误的作业引发后台的压力，这种压力会使得整个处理流程停止。

Storm和Sama这两者都是按非同步方式设计的，可以使用Kafka或者其它类似的系统作为它们的日志。

有状态的实时流处理

一些实时流处理在转化时是无状态的记录。在流处理中大部分的应用会是相当复杂的统计、聚 合、不同窗口之间的关联。例如有时人们想扩大包含用户操作信息的事件流（一系列的单击动作）–实际上关联了用户的单击动作流与用户的账户信息数据库。不变 的是这类流程最终会需要由处理器维护的一些状态信息。例如数据统计时，你需要统计到目前为止需要维护的计数器。如果处理器本身失败了，如何正确的维护这些 状态信息呢?

最简单的替换方案是把这些状态信息保存在内存中。但是如果流程崩溃，它就会丢失中间状态。如果状态是按窗口维护的，流程就会回退到日志中窗口开始的时间点上。但是，如果统计是按小时进行的，那么这种方式就会变得不可行。

另一个替换方案是简单的存储所有的状态信息到远程的存储系统，通过网络与这些存储关联起来。这种机制的问题是没有本地数据和大量的网络间通信。

我们如何支持处理过程可以像表一样分区的数据呢?

回顾一下关于表和日志二相性的讨论。这一机制提供了工具把数据流转化为与处理过程协同定位的表，同时也提供了这些表的容错处理的机制。

流处理器可以把它的状态保存在本地的表或索引——bdb，或者leveld-b，甚至于类 似于Lucene 或fastbit一样不常见的索引。这些内容存储在它的输入流中（或许是使用任意的转化）。生成的变更日志记录了本地的索引，它允许存储事件崩溃、重启等 的状态信息。流处理提供了通用的机制用于在本地输入流数据的随机索引中保存共同分片的状态。

当流程运行失败时，它会从变更日志中恢复它的索引。每次备份时，日志把本地状态转化成一系列的增量记录。

这种状态管理的方法有一个优势是把处理器的状态也做为日志进行维护。我们可以把这些日志 看成与数据库表相对应的变更日志。事实上，这些处理器同时维护着像共同分片表一样的表。因为这些状态它本身就是日志，其它的处理器可以订阅它。如果流程处 理的目标是更新结点的最后状态，这种状态又是流程的输出，那么这种方法就显得尤为重要。

为了数据集成，与来自数据库的日志关联，日志和数据库表的二象性就更加清晰了。变更日志可以从数据库中抽取出来，日志可以由不同的流处理器（流处理器用于关联不同的事件流）按不同的方式进行索引。

我们可以列举在Samza中有状态流处理管理的更多细节和大量实用的例子。

日志压缩

当然，我们不能奢望保存全部变更的完整日志。除非想要使用无限空间，日志不可能完全清 除。为了澄清它，我们再来聊聊Kafka的实现。在Kafka中，清理有两种选择，这取决于数据是否包括关键更新和事件数据。对于事件数据，Kafka支 持仅维护一个窗口的数据。通常，配置需要一些时间，窗口可以按时间或空间定义。虽然对于关键数据而言，完整日志的重要特征是你可以重现源系统的状态信息， 或者在其它的系统重现。

随着时间的推移，保持完整的日志会使用越来越多的空间，重现所耗费的时间越来越长。因些 在Kafka中，我们支持不同类型的保留。我们移除了废弃的记录（这些记录的主键最近更新过）而不是简单的丢弃旧日志。我们仍然保证日志包含了源系统的完 整备份，但是现在我们不再重现原系统的全部状态，而是仅仅重现最近的状态。我们把这一特征称为日志压缩。

第四部分：系统建设

我们最后要讨论的是在线数据系统设计中日志的角色。

在分布式数据库数据流中日志的角色和在大型组织机构数据完整中日志的角色是相似的。在这两个应用场景中，日志是对于数据源是可靠的，一致的和可恢复的。组织如果不是一个复杂的分布式数据系统呢，它究竟是什么?

分类计价吗?

如果换个角度，你可以看到把整个组织系统和数据流看做是单一的分布式数据系统。你可以把 所有的子查询系统（诸如Redis，SOLR，Hive表等）看成是数据的特定索引。你可以把Storm或Samza一样的流处理系统看成是发展良好的触 发器和视图具体化机制。我已经注意到，传统的数据库管理人员非常喜欢这样的视图，因为它最终解释了这些不同的数据系统到底是做什么用的–它们只是不同的索 引类型而已。

不可否认这类数据库系统现在大量的出现，但是事实上，这种复杂性一直都存在。即使是在关 系数据库系统的鼎盛时期，组织中有大量的关系数据库系统。或许自大型机时代开始，所有的数据都存储在相同的位置，真正

的集成是根本不存在的。存在多种外在 需求，需要把数据分解成多个系统，这些外在需求包括：规模、地理因素、安全性，性能隔离是最常见的因素。这些需求都可以由一个优质的系统实现：例如，组织 可以使用单一的Hadoop聚簇，它包括了全部的数据，可以服务于大型的和多样性的客户。

因此在向分布式系统变迁的过程中，已经存在一种处理数据的简便的方法：把大量的不同系统的小的实例聚合成为大的聚簇。许多的系统还不足以支持这一方法：因为它们不够安全，或者性能隔离性得不到保证，或者规模不符合要求。不过这些问题都是可以解决的。

依我之见，不同系统大量出现的原因是建设分布式数据库系统很困难。通过削减到单一的查询或者用例，每个系统都可以把规模控制到易于实现的程度。但是运行这些系统产生的复杂度依然很高。

未来这类问题可能的发展趋势有三种：

第一种可能是保持现状：孤立的系统还会或长或短的持续一段时间。这是因为建设分布式系统的困难很难克服，或者因为孤立系统的独特性和便捷性很难达到。基于这些原因，数据集成的核心问题仍然是如何恰当的使用数据。因此，集成数据的外部日志非常的重要。

第二种可能是重构：具备通用性的单一 的系统逐步融合多个功能形成超极系统。这个超级系统表面看起来类似关系数据库系统，但是在组织中你使用时最大的不同是你只需要一个大的系统而不是无数个小 系统。在这个世界里，除了在系统内已解决的这个问题不存在什么真正的数据集成问题。我想这是因为建设这样的系统的实际困难。

虽然另一种可能的结果对于工程师来说是很有吸引力的。新一代数据库系统的特征之一是它们是完全开源的。开源提供了一种可能性：数据基础架构不必打包成服务集或者面向应用的系统接口。在Java栈中，你可以看到在一定程度上，这种状况已经发生了。

Zookeeper用于处理多个系统之间的协调，或许会从诸如Helix 或者Curator等高级别的抽象中得到一些帮助。

- Mesos和YARN用于处理流程可视化和资源管理。
- Lucene和LevelDB等嵌入式类库做为索引。
- Netty，Jetty和Finagle，rest.li等封装成高级别的用于处理远程通信。
- Avro，Protocol Buffers，Thrift和umpteen zillion等其它类库用于处理序列化。
- Kafka和Bookeeper提供支持日志。

如果你把这些堆放在一起，换个角度看，它有点像是简化版的分布式数据库系统工程。你可以 把这些拼装在一起，创建大量的可能的系统。显而易见，现在探讨的不是最终用户所关心的API或者如何实现，而是在不断多样化和模块化的过程中如何设计实现 单一系统的途径。因为随着可靠的、灵活的模块的出现，实施分布式系统的时间周期由年缩减为周，聚合形成大型整体系统的压力逐步消失。

日志文件在系统结构中的地位

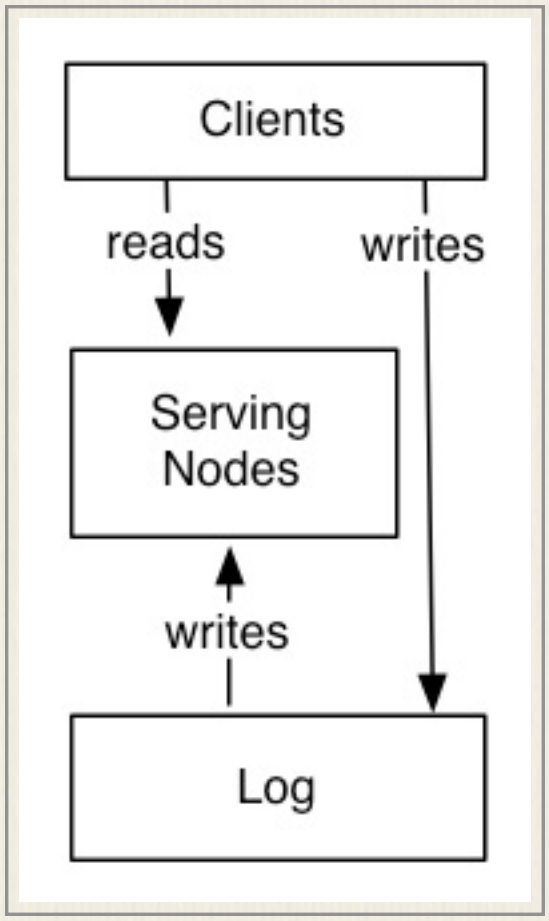
那些提供外部日志的系统如今已允许个人电脑抛弃他们自身复杂的日志系统转而使用共享日志。在我看来，日志可以做到以下事情：

- 通过对节点的并发更新的排序处理数据的一致性（无论在及时还是最终情况下）
- 提供节点之间的数据复制
- 提供”commit“语法（只有当写入器确保数据不会丢失时才会写入）
- 位系统提供外部的数据订阅资源
- 提供存储失败的复制操作和引导新的复制操作的能力
- 处理节点间的数据平衡

这实际上是一个数据分发系统最重要的部分，剩下的大部分内容与终端调用的API和索引策略相关。这正是不同系统间的差异所在，例如：一个全

文本查询语句需要查询所有的分区，而一个主键查询只需要查询负责键数据的单个节点就可以了。

下面我们来看下该系统是如何工作的。系统被分为两个逻辑区域：日志和服务层。日志按顺序 捕获状态变化，服务节点存储索引提供查询服务需要的所有信息（键—值的存储可能以B-tree或SSTable的方式进行，而搜索系统可能存在与之相反的 索引）。写入器可以直接访问日志，尽管需要通过服务层代理。在写入日志的时候会产生逻辑时间戳（即log中的索引），如果系统是分段式的，那么就会产生与 段数目相同数量的日志文件和服务节点，这里的数量和机器数量可能会有较大差距。



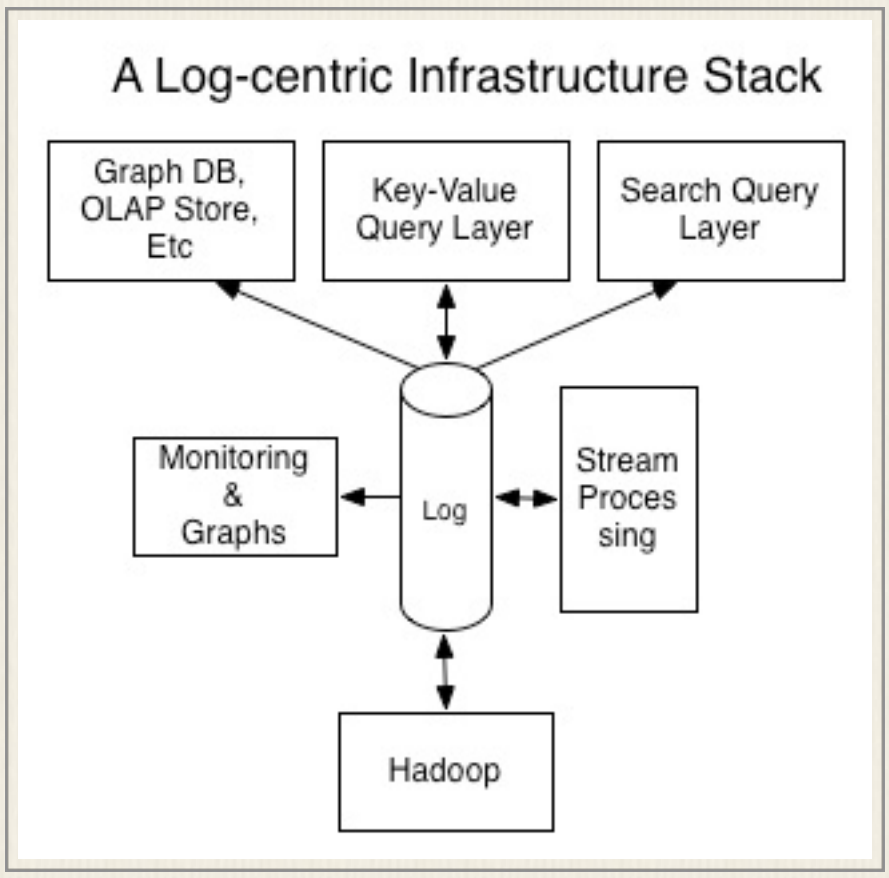
服务节点订阅日志信息并将写入器按照日志存储的顺序尽快应用到它的本地索引上。

客户端只要在查询语句中提供对应的写入器的时间戳，它就可以从任何节点中获取”读写“语义。服务节点收到该查询语句后会将其中的时间戳与自身的索引比较，如果必要，服务节点会延迟请求直到对应时间的索引建立完毕，以免提供旧数据。

服务节点或许根本无需知道”控制“或”投标选择（leader election）“的概念，对很多简单的操作，服务节点可以爱完全脱离领导的情况下提供服务，日志即是信息的来源。

分发系统所需要做的其中一个比较复杂的工作，就是修复失败节点并移除几点之间的隔离。保留修复的数据并结合上各区域内的数据快照是一种较为典型的做法，它与保留完整的数据备份并从垃圾箱内回收日志的做法几乎等价。这就使得服务层简单了很多，日志系统也更有针对性。

有了这个日志系统，你可以订阅到API，这个API提供了把ETL提供给其它系统的数据内容。事实上，许多系统都可以共享相同的日志同时提供不同的索引，如下所示：



这样一个以日志为中心的系统是如何做到既数据流的提供者又同时加载其它系统的数据的呢？因为流处理器既可以消费多个输入的数据流，随后又可以通过其它系统对数据做索引为它们提供服务。

这个系统的视图可以清晰的分解到日志和查询API，因为它允许你从系统的可用性和一致性角度分解查询的特征。这可以帮助我们对系统进行分解，并理解那些并没按这种方式设计实施的系统。

虽然Kafka和Bookeeper都是一致性日志，但这不是必须的，也没什么意义。你可以轻松的把Dynamo之类的数据构分解为一致性的AP日志和键值对服务层。这样的日志使用起来灵活，因为它重传了旧消息，像Dynamo一样，这样的处理取决于消息的订阅者。

在很多人看来，在日志中另外保存一份数据的完整复本是一种浪费。事实 上，虽然有很多因素使得这件事并不困难。首先，日志可以是一种有效的存储机制。我们在Kafka生产环境的服务器上存储了5 TB的数据。同时有许多的服务系统需要更多的内存来提供有效的数据服务，例如文本搜索，它通常是在内存中的。服务系统同样也需样硬盘的优化。例如，我们的 实时数据系统或者在内存外提供服务或者使用固态硬盘。相反，日志系统只需要线性的读写，因此，它很乐于使用TB量级的硬盘。最终，如上图所示，由多个系统 提供的数据，日志的成本分摊到多个索引上，这种聚合使得外部日志的成本降到了最低点。

LinkedIn就是使用了这种方式实现它的多个实时查询系统的。这些 系统提供了一个数据库（使用数据总线做为日志摘要，或者从Kafka去掉专用的日志），这些系统在顶层数据流上还提供了特殊的分片、索引和查询功能。这也 是我们实施搜索、社交网络和OLAP查询系统的方式。事实上这种方式是相当普遍的：为多个用于实时服务的服务系统提供单一的数据（这些来自Hadoop的 数据或是实时的或是衍生的）。这种方式已被证实是相当简洁的。这些系统根本不需要外部可写入的API，Kafka和数据库被用做系统的记录和变更流，通过 日志你可以查询系统。持有特定分片的结点在本地完成写操作。这些结点盲目的把日志提供的数据转录到它们自己的存储空间中。通过回放上行流日志可以恢复转录 失败的结点。

这些系统的程度则取决于日志的多样性。一个完全可靠的系统可以用日志来对数据分片、存储结点、均衡负载，以及用于数据一致性和数据复制等多方面。在这一过程中，服务层实际上只不过是一种缓存机制，这种缓存机制允许直接写入日志的流处理。

结束语

如果你对于本文中所谈到的关于日志的大部内容，如下内容是您可以参考的其它资料。对于同一事务人们会用不同的术语，这会让人有一些困惑，

从数据库系统到分布式系统，从各类企业级应用软件到广阔的开源世界。无论如何，在大方向上还是有一些共同之处。

学术论文、系统、评论和博客

关于状态机和主备份复现的概述。

- PacificA是实施微软基于日志的分布式存储系统的通用架构。
- Spanner-并不是每个人都支持把逻辑时间用于他们的日志，Google最新的数据库就尝试使用物理时间，并通过把时间戳直接做为区间来直接建时钟迁移的不确定性。
- Datanomic:解构数据库，它是Rich Hickey 在它的首个数据库产品中的的重要陈述之一，Rich Hickey是Clojure的创建者。
- 在消息传递系统中回卷恢复协议的调查。我发现这个有助于引入容错处理和数据库以外的应用系统日志恢复。
- Reactive Manifesto-事实上我并不清楚反应编程的确切涵义，但是我想它和“事件驱动”指的是同一件事。这个链接并没有太多的讯息，但由久富盛史的Martin Odersky讲授的课程是很有吸引力的。

Paxos!

1）Leslie Lamport有一个有趣的历史：在80年代算法是如何发现的，但是直到1998年才发表了，因为评审组不喜欢论文中的希腊寓言，而作者又不愿修改。

2）甚至于论文发布以后，它还是不被人们理解。Lamport再次尝试，这次它包含了一些并不有趣的小细节，这些细节是关于如何使用这些新式的自动化的计算机的。它仍然没有得到广泛的认可。

3）Fred Schneider和Butler Lampson 分别给出了更多细节关于在实时系统中如何应用Paxos.

4）一些Google的工程师总结了他们在Chubby中实施Paxos的经验。

5）我发现所有关于Paxos的论文理解起来很痛苦，但是值得我们费大 力气弄懂。你不必忍受这样的痛苦了，因为日志结构的文件系统的大师John Ousterhout的这个视频让这一切变得相当的容易。这些一致性算法用展开的通信图表述的更好，而不是在论文中通过静态的描述来说明。颇为讽刺的是， 这个视频录制的初衷是告诉人们Paxos很难理解。

6）使用Paxos来构造规模一致的数据存储。

Paxos有很多的竞争者。如下诸项可以更进一步的映射到日志的实施，更适合于实用性的实施。

1）由Barbara Liskov 提出的视图戳复现是直接进行日志复现建模的较早的算法。

2）Zab是Zookeeper所使用的算法。

3）RAFT是易于理解的一致性算法之一。由John Ousterhout讲授的这个视频非常的棒。

你可以的看到在不同的实时分布式数据库中动作日志角色：

1）PNUTS是探索在大规模的传统的分布式数据库系统中实施以日志为中心设计理念的系统。

2）Hbase和Bigtable都是在目前的数据库系统中使用日志的样例。

3）LinkedIn自己的分布式数据库Espresso和PNUTs一样，使用日志来复现，但有一个小的差异是它使用自己底层的表做为日志的来源。

流处理：这个话题要总结的内容过于宽泛，但还是有几件我所关注的要提一下：

1）TelegraphCQ

2）Aurora

3）NiagaraCQ

4）离散流：这篇论文讨论了Spark的流式系统。

5）MillWheel 它是Google的流处理系统之一。

6）Naiad：一个实时数据流系统

7）在数据流系统中建模和相关事件：它可能是研究这一领域的最佳概述之一。

8）分布处式流处理的高可用性算法。

企业级软件存在着同样的问题，只是名称不同，或者规模较小，或者是XML格式的。哈哈，开个玩笑。

事件驱动——据我所知：它就是企业级应用的工程师们常说的“状态机的复 现”。有趣的是同样的理念会用在如此迥异的场景中。事件驱动关注的是小的、内存中的使用场景。这种机制在应用开发中看起来是把发生在日志事件中的“流处 理”和应用关联起来。因此变得不那么琐碎：当处理的规模大到需要数据分片时，我关注的是流处理作为独立的首要的基础设施。

变更数据捕获–在数据库之外会有些对于数据的舍入处理，这些处理绝大多数都是日志友好的数据扩展。

企业级应用集成，当你有一些现成的类似客户类系管理CRM和供应链管理SCM的软件时，它似乎可以解决数据集成的问题。

复杂事件处理（CEP），没有人知道它的确切涵义或者它与流处理有什么不同。这些差异看起来集中在无序流和事件过滤、发现或者聚合上，但是依我之见，差别并不明显。我认为每个系统都有自己的优势。

企业服务总线（ESB）——我认为企业服务总线的概念类似于我所描述的数据集成。在企业级软件社区中这个理念取得了一定程度的成功，对于从事网络和分布式基础架构的工程师们这个概念还是很陌生的。

一些相关的开源软件：

- Kafka是把日志作为服务的一个项目，它是后边所列各项的基础。
- Bookeeper 和Hedwig 另外的两个开源的“把日志作为服务”的项目。它们更关注的是数据库系统内部构件而不是事件数据。
- Databus是提供类似日志的数据库表的覆盖层的系统。

- Akka 是用于Scala 的动作者架构。它有一个事件驱动的插件，它提供持久化和记录。
- Samza是我们在LinkedIn中用到的流处理框架，它用到了本文论述的诸多理念，同时与Kafka集成来作为底层的日志。
- Storm是广泛使用的可以很好的与Kafka集成的流处理框架之一。
- Spark Streaming一个流处理框架，它是Spark的一部分。
- Summingbird是在Storm或Hadoop 之上的一层，它提供了便洁的计算摘要。

原译文链接：http://mp.weixin.qq.com/s?__biz=MjM5MTQ4NzgwNA==&mid=200446096&idx=1&sn=042354b5323598b5e3aa5ebc0e270ed8

http://mp.weixin.qq.com/s?__biz=MjM5MTQ4NzgwNA==&mid=200446096&idx=2&sn=96bf23f106d42beb7e11ba7f73256895

原文链接：http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

设计一种简化的 protocol buffer 协议

作者：云风

我们一直使用 google protocol buffer 协议做客户端服务器通讯，为此，我还编写了 pbc 库。

经过近三年的使用，我发现其实我们用不着那么复杂的协议，里面很多东西都可以简化。而另一方面，我们总需要再其上再封装一层 RPC 协议。当我们做这层 RPC 协议封装的时候，这个封装层的复杂度足以比全新设计一套更合乎我们使用的全部协议更复杂了。

由于我们几乎一直在 lua 下使用它，所以可以按需定制，但也不局限于 lua 使用。这两天，我便构思了下面的东西：

我们只需要提供 boolean integer (32bit signed int) string id (64bit unsigned int) 四种基础类型，和 array 以及 struct 两种用户定义的复合类型即可。

为什么没有 float ? 因为在我们这几年的项目中，使用 float 的地方少之又少，即使有，也完全可以用 string 传输。

为什么没有 enum ? 因为在业务层完全可以自己做 int 到 enum 的互转，没必要把复杂度放在通讯协议中。

为什么不需要 union ? 因为按 protocol buffer 的做法，结构中的每个域都可以用一个数字 tag 来标识，而不是用数据布局来指示。不需要传递的域不需要打包到传输包中。

为什么不需要 default value ? 我们的项目中，依赖 default value 的地方少之又少，反而从我维护 pbc 的大量反馈看，最容易被误用的用法就是通讯协议依赖一个字段有没有最终被打包。所以干脆让（不打包）等价于 default value 就好了。明确这个（在 google protocol buffer 中是错误的）用法。

我设计的这个新协议，命名为 ejoyproto ， 它的协议描述成人可读的文本大约是这样的：

```
.person {
    .address {
        email 0 : string
        phone 1 : string
    }
    name 0 : string
    age 1 : integer
    marital 2 : boolean
    children 3 : *person  # 这是一个 preson 类型的数组
    address 4 : address
}
```

所有涉及命名的地方，都遵循 C 语言规则：以英文字母数字和下划线构成，但不能以数字开头，大小写敏感。

自定义类型用 . 开头。自定义类型可以嵌套。自定义类型的名字不可以是 integer, string 和 boolean 。

每个类型由若干字段构成，每个字段有一个在当前类型中唯一名字和一个唯一 tag 。tag 是一个非负整数，范围是 [0,32767] 。不要求连续，但建议用比较小的数字。

每个字段必须有一个类型，如果希望它是一个数组，在类型前标注 * 。

协议定义次序没有要求，但建议引用一个类型时，类型的定义放在前面。

符号 # 以后是注释。

换行和 tab 没有特别要求，只要是空白符即可。

同时，协议文件里可以描述 RPC 协议，范例如下：

```
foobar 1 {
    request person
    response {
        ok 0 : boolean
    }
}
```

这里定义了一条叫做 foobar 的 RPC 协议，赋予它一个唯一的 tag 1 。（在网络传输的时候，可以用 1 代替 foobar ）

每条协议都是由两个类型 request 和 response 构成，其中，response 是可选的。

这两个类型都必须是结构，而不能是基本类型或数组。这里可以在 request 或 response 后直接写上引用的类型名，或就地定义一个匿名类型。

这样，一组协议描述数据就可以用 ejoyproto 本身描述了：

```
.type {
    .field {
        name 0 : string
        type 1 : string
        id 2 : integer
        array 3 : boolean
    }
    name 0 : string
    fields 1 : *field
}
```

```
.protocol {
    name 0 : string
    id 1 : integer
    request 2 : string
    response 3 : string
}

.group {
    type 0 : *type
    protocol 1 : *protocol
}
```

最终提供的 api 会类似这样：

```
local tag, bytes = encode("foobar.request",
   { name = "alice", age = 13, marital = false })
```

可用来打包一个 foobar 请求，返回 foobar 的 tag 以及打包的数据。然后再将它们组合起来构成最终的通讯包（可能还需要置入 session size 等信息）。

Wire Protocol

所有的数字编码都是以小端方式(little-endian) 编码。

打包的单位是一个结构体（用户定义的类型） 每个包分两个部分：1. 字段 2. 数据块

对于数据块，用于描述字段中的大数据。它是由一个 dword 长度 + 按 4 字节对齐的字节串构成。也就是说，每个数据块的长度都一定是 4 的倍数。 对齐时，用 0 填补多余的位置。

字段必须以 tag 的升序排列, 可以不连续; 数据块的次序必须和字段中的引用次序一致。

对于每个字段，由两个 word 构成。

第一个 word 是 tag 。记录的是当前字段的 tag 相较上一个的差值 -1（对于第一个字段，和 -1 比较）。如果被打包的字段的 tag 是连续的，那么这个位置通常是 0；如果不连续，则记录的跳开的数字差。

第二个 word ，是这个字段的值。如果值为 0 ，表示数据放在数据区；否则这个值减 1 就是这个字段的值（只有是整数和布尔值才有效）。

注：在解码的时候，遇到不能识别的 tag ，解码器应选择跳过（不必确定字段的类型）。这对协议新旧版本兼容有好处。

数据类型在协议描述数据中提供，不在通讯中传输。根据 tag 可以查询到这个字段的类型。如果是对数据块的引用，且数据类型为：

- integer : 数据块长度一定为 4 ，数据内容就是一个 32bit signed integer 。
- id : 数据块长度一定为 8 ，数据内容就是 8 个字节的 id 。
- string : 数据块的长度就是string 的长度， 内容就是字符串。
- usertype : 那么数据块里就是整个结构。
- array : 那么数据块就是顺序排列的数据。对于 integer array ，每 4 个字节是一个整数；对于 boolean array ，每个字节可表示 8 个布尔量，从低位向高位排列；对于 string 和 struct ，都是顺序嵌入数据块（长度+内容）。

下面有两个范例：

person { name = "Alice" , age = 13, marital = false } :

03 00 01 00 (fn = 3, dn = 1)

00 00 00 00 (id = 0, ref = 0)

00 00 0E 00 (id = 1, value = 13)

00 00 01 00 (id = 2, value = false)

05 00 00 00 (sizeof "Alice")

41 6C 69 63 65 00 00 00 ("Alice" align by 4)

person

{

name = "Bob",

age = 40,

marital = true,

children = {

{ name = "Alice" , age = 13, marital = false },

}

}

04 00 02 00 (fn = 4, dn = 2)

00 00 00 00 (id = 0, ref = 0)

00 00 29 00 (id = 1, value = 40)

00 00 02 00 (id = 2, value = true)

00 00 00 00 (id = 3, ref = 1)

03 00 00 00 (sizeof "Bob")

42 6F 62 00 ("Bob" align by 4)

03 00 01 00 (fn = 3, dn = 1)

00 00 00 00 (id = 0, ref = 0)

00 00 0E 00 (id = 1, value = 13)

00 00 01 00 (id = 2, value = false)

05 00 00 00 (sizeof "Alice")

41 6C 69 63 65 00 00 00 ("Alice" align by 4)

0 压缩

这样打包出来的数据的长度必定 4 的倍数，里面会有大量的 0 。我们可以借鉴 Cap'n proto 的压缩方案：

首先，如果数据长度不是 8 的倍数，就补 4 个 0 。

按 8 个字节一组做压缩，用一个字节 (8bit) 表示各个字节是否为 0 ，然后把非 0 的字节紧凑排列，例如：

unpacked (hex): 08 00 00 00 03 00 02 00 19 00 00 00 aa 01 00 00

packed (hex): 51 08 03 02 31 19 aa 01

当 8 个字节全不为 0 时，这个标识字节为 FF ，这时后面跟一个字节表示有几组 (1~256) 连续非 0 组。 所以，在最坏情况下，如果有 2K 的全不为 0 的数据块，需要增加两个字节 FF FF 的数据头（表示后续有 256 组 8 字节的非 0 块）。

原文链接：http://blog.codingnow.com/2014/07/ejoyproto.html

四层和七层负载均衡的区别

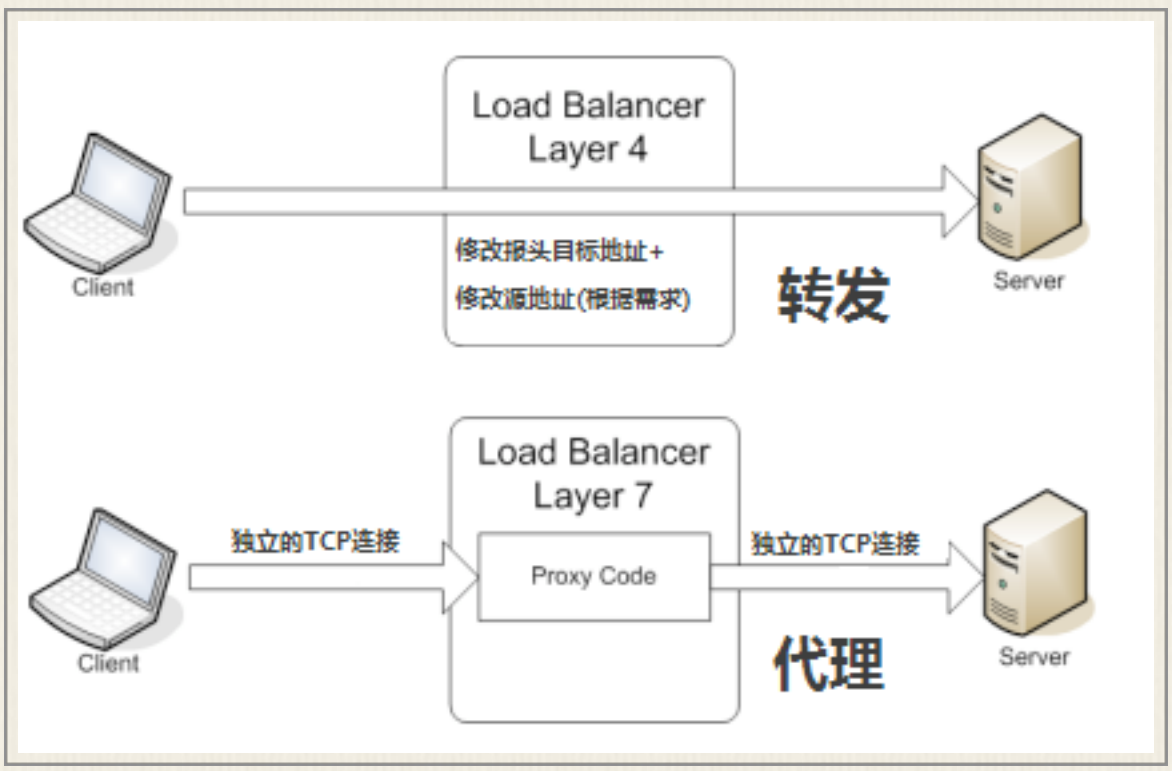
作者：virtualadc

负载均衡设备也常被称为"四到七层交换机"，那么四层和七层两者到底区别在哪里?

第一，技术原理上的区别。

所谓四层负载均衡，也就是主要通过报文中的目标地址和端口，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

以常见的TCP为例，负载均衡设备在接收到第一个来自客户端的SYN 请求时，即通过上述方式选择一个最佳的服务器，并对报文中目标IP地址进行修改(改为后端服务器IP），直接转发给该服务器。TCP的连接建立，即三次握 手是客户端和服务器直接建立的，负载均衡设备只是起到一个类似路由器的转发动作。在某些部署情况下，为保证服务器回包可以正确返回给负载均衡设备，在转发 报文的同时可能还会对报文原来的源地址进行修改。



所谓七层负载均衡，也称为“内容交换”，也就是主要通过报文中的真正有意义的应用层内容，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

以常见的TCP为例，负载均衡设备如果要根据真正的应用层内容再选择服务器，只能先代理最终 的服务器和客户端建立连接(三次握手)后，才可能接受到客户端发送的真正应用层内容的报文，然后再根据该报文中的特定字段，再加上负载均衡设备设置的服务 器选择方式，决定最终选择的内部服务器。负载均衡设备在这种情况下，更类似于一个代理服务器。负载均衡和前端的客户端以及后端的服务器会分别建立TCP连 接。所以从这个技术原理上来看，七层负载均衡明显的对负载均衡设备的要求更高，处理七层的能力也必然会低于四层模式的部署方式。那么，为什么还需要七层负 载均衡呢?

第二，应用场景的需求。

七层应用负载的好处，是使得整个网络更"智能化", 参考我们之前的另外一篇专门针对HTTP应用的优化的介绍， 就可以基本上了解这种方式的优势所在。例如访问一个网站的用户流量，可以通过七层的方式，将对图片类的请求转发到特定的图片服务器并可以使用缓存技术；将 对文字类的请求可以转发到特定的文字服务器并可以使用压缩技术。当然这只是七层应用的一个小案例，从技术原理上，这种方式可以对客户端的请求和服务器的响 应进行任意意义上的修改，极大的提升了应用系统在网络层的灵活性。很多在后台，(例如Nginx或者Apache)上部署的功能可以前移到负载均衡设备 上，例如客户请求中的Head-
er重写，服务器响应中的关键字过滤或者内容插入等功能。

另外一个常常被提到功能就是安全性。网络中最常见的SYN Flood攻击，即黑客控制众多源客户端，使用虚假IP地址对同一目标发送SYN攻击，通常这种攻击会大量发送SYN报文，耗尽服务器上的相关资源，以达到Denial of Service(DoS)的目的。从技术原理上也可以看出，四层模式下这些SYN攻击都会被转发到后端的服务器上；而七层模式下这些SYN攻击自然在负载均衡设备上就截止，不会影响后台服务器的正常运营。另外负载均衡设备可以在七层层面设定多种策略，过滤特定报文，例如SQL Injec-
tion等应用层面的特定攻击手段，从应用层面进一步提高系统整体安全。

现在的7层负载均衡，主要还是着重于应用广泛的HTTP协议，所以其应用范围主要是众多的网站或者内部信息平台等基于B/S开发的系统。 4层负载均衡则对应其他TCP应用，例如基于C/S开发的ERP等系统。

第三，七层应用需要考虑的问题。

1：是否真的必要，七层应用的确可以提高流量智能化，同时必不可免的带来设备配置复杂，负载均衡压力增高以及故障排查上的复杂性等问题。在设计系统时需要考虑四层七层同时应用的混杂情况。

2：是否真的可以提高安全性。例如SYN Flood攻击，七层模式的确将这些流量从服务器屏蔽，但负载均衡设备本身要有强大的抗DDoS能力，否则即使服务器正常而作为中枢调度的负载均衡设备故障也会导致整个应用的崩溃。

3：是否有足够的灵活度。七层应用的优势是可以让整个应用的流量智能化，但是负载均衡设备需 要提供完善的七层功能，满足客户根据不同情况的基于应用的调度。最简单的一个考核就是能否取代后台Nginx或者Apache等服务器上的调度功能。能够 提供一个七层应用开发接口的负载均衡设备，可以让客户根据需求任意设定功能，才真正有可能提供强大的灵活性和智能性。

原文链接：http://virtualadc.blog.51cto.com/3027116/591396

iOS开发如何提高

作者：唐巧

许多人在博客和微信上咨询我iOS开发如何提高，经过一番思考之后，我能想到如下一些提高的办法，我个人也是通过这些方法来提高的。

阅读博客

在现在这个碎片化阅读流行的年代，博客的风头早已被微博盖过。而我却坚持写作博客，并且大量地阅读同行的iOS开发博客。博客的文章长度通常在 3000字左右，许多iOS开发知识都至少需要这样的篇幅才能完整地讲解清楚。并且博客相对于书籍来说，并没有较长的出版发行时间，所以阅读博客对于获取 最新的iOS开发知识有着非常良好的效果。

我自己精心整理了国内40多位iOS开发博主的博客地址列表：https://github.com/tangqiaoboy/iOSBlogCN，希望大家都能培养起阅读博客的习惯。

国外也有很多优秀的iOS开发博客，他们整体质量比中文的博客更高，以下是一些推荐的博客地址列表：

objc.io

http://www.objc.io/

Ray Wenderlich

http://www.raywenderlich.com

iOS Developer Tips

http://iosdevelopertips.com/

iOS Dev Weekly

http://iosdevweekly.com/

NSHipster

http://nshipster.com/

Bartosz Ciechanowski

http://ciechanowski.me

Big Nerd Ranch Blog

http://blog.bignerdranch.com

Nils Hayat

http://nilsou.com/

另外，使用博客RSS聚合工具（例如Feedly：http://www.feedly.com/）可以获得更好的博客阅读体验。手机上也有很多优秀的博客阅读工具（我使用的是Newsify）。合理地使用这些工具也可以将你在地铁上、睡觉前等碎片时间充分利用上。

读书

博客的内容通常只能详细讲解一个知识点，而书籍则能成体系地介绍整个知识树。相比国外，中国的书籍售价相当便宜，所以这其实是一个非常划算的提高的方式。建议大家每年至少坚持读完1本高质量的iOS开发书籍。

去年出版的《iOS 7 Programming Pushing the Limits》 以及《Objective-C高级编程：iOS与OS X多线程和内存管理》 都算是不错的进阶方面的读物。顺便打个广告，我自己也在写一本iOS进阶方面的图书，年底前应该能上市。

看WWDC视频

由于iOS开发在快速发展，每年苹果都会给我们带来很多新的知识。而对于这些知识，第一手的资料就是WWDC的视频。

通常情况下，一个iOS开发的新知识首先会在WWDC上被苹果公开，然后3个月左右，会有国内外的博客介绍这些知识，再过半年左右，会有国外的图书介绍这些知识。所以如果想尽早地了解这些知识，那么只有通过WWDC的视频。

现在每年的WWDC视频都会在会议过程中逐步放出，重要的视频会带有英文字幕。坚持阅读这些视频不但可以获得最新的iOS开发知识，还可以提高英文听力水平。

看苹果的官方文档

苹果的官方文档相当详尽，对于不熟悉的API，阅读官方文档也是最直接有效地方式。

苹果的文档比较海量，适合选一些重点来阅读，比如人机交互指南就是必读的，而其它的内容可以遇到的时候作为重点资源来查阅。

看开源项目的代码

大家一定有这样的感受，很多时候用文字讲解半天，还不如写几行代码来得直观。阅读优秀的开源项目代码，不但可以学习到iOS开发本身的基本知识，还能学习到设计模式等软件架构上的知识。

如果读者能够参与到开源项目的开发中，则能进一步提高自己的能力。

多写代码，多思考

知识的积累离不开实践和总结，我认为iOS代码量如果没有超过10万行，是不能称得上熟悉iOS开发的。某些在校的学生，仅仅做了几个C++的大作业，就在求职简历里面写上“精通C++”，则真是让人哭笑不得。

在多写代码的同时，我们也要注意不要”重复造轮子”，尽量保证每次写的代码都能具有复用性。在代码结构因为业务需求需要变更时，及时重构，在不要留下技术债的同时，我们也要多思考如何设计应用架构，能够保证满足灵活多变的产品需求。

在多次重构和思考的过程中，我们就会慢慢积累出一类问题的“最佳实践”方式，成为自己宝贵的经验。

多和同行交流

有些时候遇到一些难解的技术问题，和同行的几句交流就可能让你茅塞顿开。。另外常见的技术问题通常都有人以前遇到过，简单指导几句就能让你一下子找到正确的解决方向。

国内开发者之间的交流，可以通过论坛，微博，QQ群等方式来进行。另外各大公司有时候会办技术沙龙，这也是一个认识同行的好机会。

需要特别提醒的是，和国内开发者之前交流要注意讨论质量，有一些论坛和QQ群讨论质量相当低下，提的问题都是能通过简单Google获得的，这种社区一定要远离，以提高自己的沟通效率。

除了在国内的技术社区交流，建议读者可以去国外的stackoverflow：http://www.stackoverflow.com上提问或回答问题。

分享

值得尝试的分享方式有：发起一个开源项目、写技术博客、在技术会议上做报告。这几种方式都比较有挑战，但是如果能大胆尝试，肯定会有巨大的收获。

原文链接：http://blog.devtang.com/blog/2014/07/27/ios-levelup-tips/

Android几种推送方案的比较

作者：乂乂

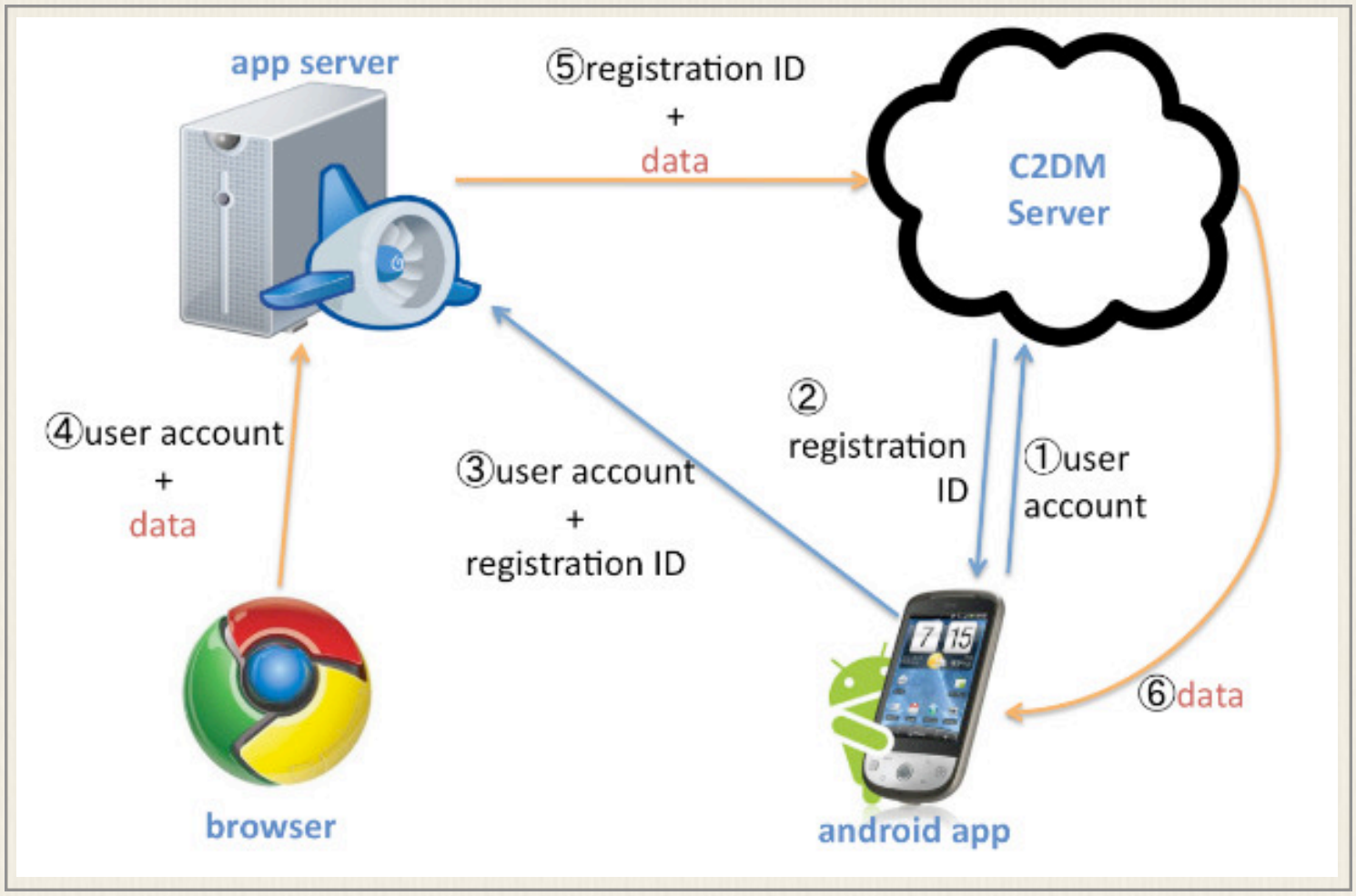
当开发需要和服务器交互的应用程序时，基本上都需要获取服务器端的数据。要获取服务器上不定时更新的信息一般来说有两种方法，第一种是客户端使 用Pull的方式，隔一段时间就去服务器上获取信息，看是否有更新的信息出现。第二种就是服务器使用Push的方式，当服务器端有新信息了，则把最新的信 息Push到客户端上。

虽然Pull和Push两种方式都能实现获取服务器端更新信息的功能，但是明显来说Push方式更好。因为Pull方式更费客户端的网络流量，同時也耗费电量。

在开发Android和iPhone应用程序时，我们往往需要从服务器不定的向手机客户端即时推送各种通知消息，iPhone上已经有了比较简单的和完美的推送通知解决方案，可是Android平台上实现起来却相对比较麻烦。

在Android手机平台上，Google提供了C2DM（Cloudto Device Messaging）服务。Android Cloud to Device Messaging (C2DM)是一个用来帮助开发者从服务器向Android应用程序发送数据的服务。该服务提供了一个简单的、轻量级的机制，允许服务器可以通知移动应用 程序直接与服务器进行通信，以便于从服务器获取应用程序更新和用户数据。C2DM服务负责处理诸如消息排队等事务并向运行于目标设备上的应用程序分发这些 消息。

C2DM操作过程图:



但是这个服务存在一些问题：

1）C2DM内置于Android的2.2系统上，无法兼容老的1.6到2.1系统;

2）C2DM需要依赖于Google官方提供的C2DM服务器，由于国内的网络环境，这个服务经常不可用，如果想要很好的使用，App Server必须也在国外，这个恐怕不是每个开发者都能够实现的。

如果C2DM无法满足你的要求，那么就需要自己来实现Android手机客户端与App Server之间的通信协议，保证在App Server向指定的Android设备发送消息时，Android设备能够及时的收到。下面介绍几种常见的方案：

1）轮询(Pull)：应用程序应当阶段性的与服务器进行连接并查询是否有新的消息到达，你必须自己实现与服务器之间的通信，例如消息排队等。而且你还要考虑轮询的频率，如果太慢可能导致某些消息的延迟，如果太快，则会大量消耗网络带宽和电池。

2）SMS(Push)： 在Android平台上，你可以通过拦截SMS消息并且解析消息内容来了解服务器的意图。这是一个不错的想法，市场上有一些程

序就是采用这个方案的。这个 方案的好处是，可以实现完全的实时操作。但是问题是这个方案的成本相对比较高，你很难找到免费的短消息发送网关来部署这个方案。

3） 持久连接(Push)：这个方案可以解决由轮询带来的性能问题，但是还是会消耗手机的电池。Apple的推送服务之所以工作的很好，是因为每一台手机仅仅 保持一个与服务器之间的连接，事实上C2DM也是这么工作的。不过这个方案也存在不足，就是我们很难在手机上实现一个可靠的服务。Android操作系统 允许在低内存情况下杀死系统服务，所以你的通知服务很可能被操作系统Kill掉了。

三个方案都有不足，不过可以通过良好的设计来弥补，以便于让该方案可以有效的工作。毕竟，要知道GMail，GTalk以及GoogleVoice是都可以实现实时更新的。

采用MQTT协议实现Android推送

MQTT是一个轻量级的消息发布/ 订阅协议，它是实现基于手机客户端的消息推送服务器的理想解决方案。

wmqtt.jar 是IBM提供的MQTT协议的实现。你可以从如下站点下载它。你可以将该jar包加入你自己的Android应用程序中。

Really Small Message Broker (RSMB)，它是一个简单的MQTT代理，同样由IBM提供。缺省打开1883端口，应用程序当中，它负责接收来自服务器的消息并将其转发给指定的移动设备。

采用XMPP协议实现Android推送

事实上Google 官方的C2DM服务器底层也是采用XMPP协议进行的封装。

XMPP(可扩展通讯和表示协议)是基于可扩展标记语言（XML）的协议，它用于即时消息（IM）以及在线探测。这个协议可能最终允许因特网用户向因特网上的其他任何人发送即时消息。

androidpn是一个基于XMPP协议的java开源Android push notification实现。它包含了完整的客户端和服务器端。阅读源代码时我发现，该服务器端基本是在另外一个开源工程openfire基础上修改实现的。

androidpn 客户端需要用到一个基于java的开源XMPP协议包asmack，这个包同样也是基于openfire下的另外一个开源项目smack，不过我们不需要 自己编译，可以直接把androidpn客户端里面的asmack.jar拿来使用。客户端利用asmack中提供的XMPPConnection类与服 务器建立持久连接，并通过该连接进行用户注册和登录认证，同样也是通过这条连接，接收服务器发送的通知。

androidpn 服务器端也是java语言实现的，基于openfire开源工程，不过它的Web部分采用的是spring框架，这一点与openfire是不同的。Androidpn服务器包含两个部分，一个是侦听在5222端口上的XMPP服务，负责与客户端的XMPPConnection类进行通信，作用是用户 注册和身份认证，并发送推送通知消息。另外一部分是Web服务器，采用一个轻量级的HTTP服务器，负责接收用户的Web请求。服务器架构如下：

最 上层包含四个组成部分，分别是SessionManager，Auth Manager，PresenceManager以及Notification Manager。SessionManager负责管理客户端与服务器之间的会话，Auth Manager负责客户端用户认证管理，Presence Manager负责管理客户端用户的登录状态，NotificationManager负责实现服务器向客户端推送消息功能。

这个解决方案的最大优势就是简单，不需要象C2DM那样依赖操作系统版本，也不会担心某一天Google服务器不可用。利用XMPP协议我们还可以进一步的对协议进行扩展，实现更为完善的功能。

采用这个方案，我们目前只能发送文字消息，不过对于推送来说一般足够了，因为我们不能指望通过推送得到所有的数据，一般情况下，利用推送只是告诉手机端服务器发生了某些改变，当客户端收到通知以后，应该主动到服务器获取最新的数据。

原文链接：http://zwkufo.blog.163.com/blog/static/258825120130910555222/

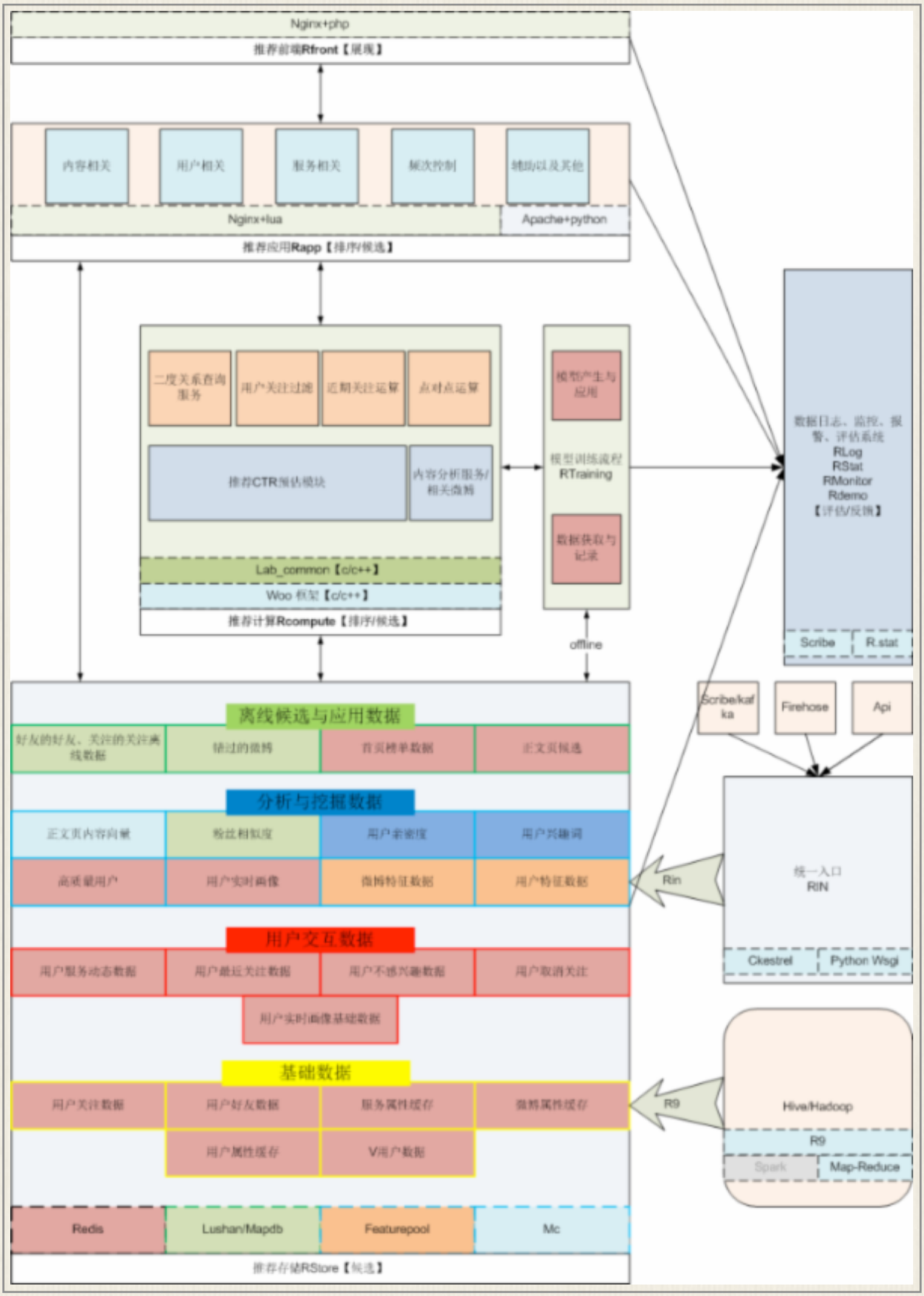
微博推荐引擎体系结构简述

作者：未知

这里有个“道术孰优”的问题，何为“道”？何为“术”？举个例子的话，《孙子兵法》是道，而《三十六计》则为术。“道”所述，是宏观的、原理性的、长久不变的基本原理，而“术”则是在遵循基本原理基础上的具体手段和措施，具有易变性。技术也是如此，算法本省的细节是“术”，算法体现的基本思想则是 “道”，知“道”而学“术”，两者虽不可偏废，但是若要选择优先级的话，无疑我会选择先“道”后“术”——张俊林《这就是搜索引擎》

上一篇文章介绍了推荐产品，给大家有一个初步的认识：微博推荐的目标和使命、推荐产品有哪些以及推荐的分类角度。本文将会给大家描述当前微博推荐的体系结构。

任何不拿出干货的技术文档都是耍流氓，首先上体系结构图，如图所示，在整体体系结构上，微博推荐可以被划分为4层：前端展现层、应用层、计算层以 及数据层，其中我们把数据日志、统计、监控以及评估也都分在数据层。接下来我会逐一介绍他们的目的，作用、技术与发展。更为细致的描述应该会在以后的博客 中体现。


Nginx+php
推荐前端Rfront【展现】
内容相关
用户相关
服务相关
展次控制
辅助以及其他
Nginx+lua
Apache+python
推荐应用Rapp【排序/候选】
二度关系查询服务
用户关注过滤
近期关注运算
点对点运算
推荐CTR预估模块
内容分析服务/相关微博
Lab_common【c/c++】
Woo 框架【c/c++】
推荐计算Rcompute【排序/候选】
模型产生与应用
模型训练流程RTraining
数据获取与记录
offline
数据日志、监控、报警、评估系统
RLog
RStat
RMonitor
Rdemo
【评估/反馈】
Scribe
R.stat
Scribe/kafka
Firehose
Api
离线候选与应用数据
好友的好友、关注的关注离线数据
转过的微博
首页榜单数据
正文页候选
分析与挖掘数据
正文页内容向量
粉丝相似度
用户密密度
用户兴趣词
高质量用户
用户实时画像
微博特征数据
用户特征数据
Rin
统一入口
RIN
Ckestrel
Python Wsgi
用户交互数据
用户服务动态数据
用户最近关注数据
用户不感兴趣数据
用户取消关注
用户实时画像基础数据
基础数据
用户关注数据
用户好友数据
服务属性缓存
微博属性缓存
用户属性缓存
V用户数据
R9
Hive/Hadoop
R9
Spark
Map-Reduce
Redis
Lushan/Mapdb
Featurepool
Mc
推荐存储RStore【候选】

1、推荐前端RFront

RFront主要目的是展现以及渲染微博内容，由于当前微博推荐在web以及客户端都有相对应的产品，展现差异较大，但数据和方法却是通用的。那么，需要有这么易于维护、拥抱变化的一层高效地响应产品需求。当然在微博推荐实际业务中，由于产品形态的多样以及策略的负责性，RFront做了不少工作 以及相关的应用技术很多，在接下来的文章中会有相关的同学着重介绍这一块工作。

2、推荐应用RApp

RApp主要目的是为前端提供候选以及起到部分排序功能。该层利用通用框架【nginx + lua 以及apache + python两个版本】提供应用接口服务。这些应用包括：内容、用户、服务、feed推荐频次以及辅助功能。

在这里有一个工具叫做通用推荐框架（CRF, common recommon framework），它主要的作用是：融合推荐资源、规范推荐应用接口以及统一工作流。早期版本使用的是apache+mod_python的形式，后 来在RApp的定位上，认为它是一个数据通路，同时需要获取各种协议的数据内容，因而将其扩展到nginx + lua的版本。

CRF是一个二次开发框架，无论是早期的apache+mod_python版本还是nginx+lua版本，其核心思想是相似的，它们的目标都是为了较为快速进行推荐策略开发，快速的使用既有推荐存储数据。主要的体现在：

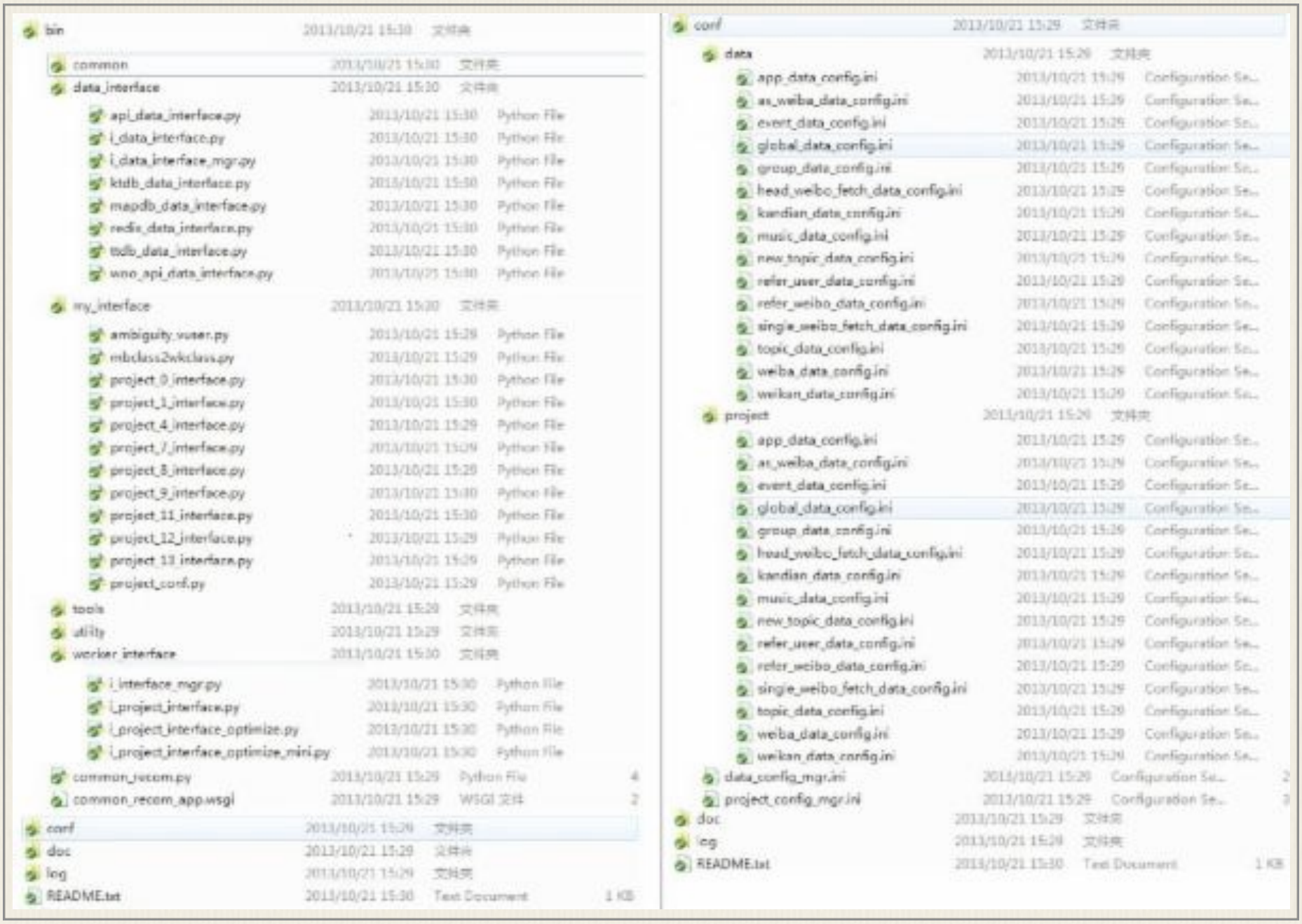
- 通过透明化不同协议的存储数据获取方法，方面获取推荐资源。比如，获取mc、redis以及openapi的数据方式是一致的，get以及mget是一个标准的获取方法。

- 通过建立project的概念，以继承的方式让二次开发者建立自己的项目，同时将project中work_core暴露出来，完成主体业务。当然其中也有一个小的技巧，比如global_data以及并行化获取数据等等。

- 通过暴露出来的common_recom的核心接口来定义推荐的统一接口，我们整理和抽象了所有推荐的接口数据形式，在一定程度上进行了规范和定义，这样方面了进行接口管理以及对应。比如obj【多用来表示访问者】，tobj【多用来表示访问对象】，from【来源】，appkey【分配

key】，num【数量】，type【类型，用来区分同一个项目中不同接口】，pid【区分项目】等等。

以下图是其三层目录结构【apache+mod_python】：



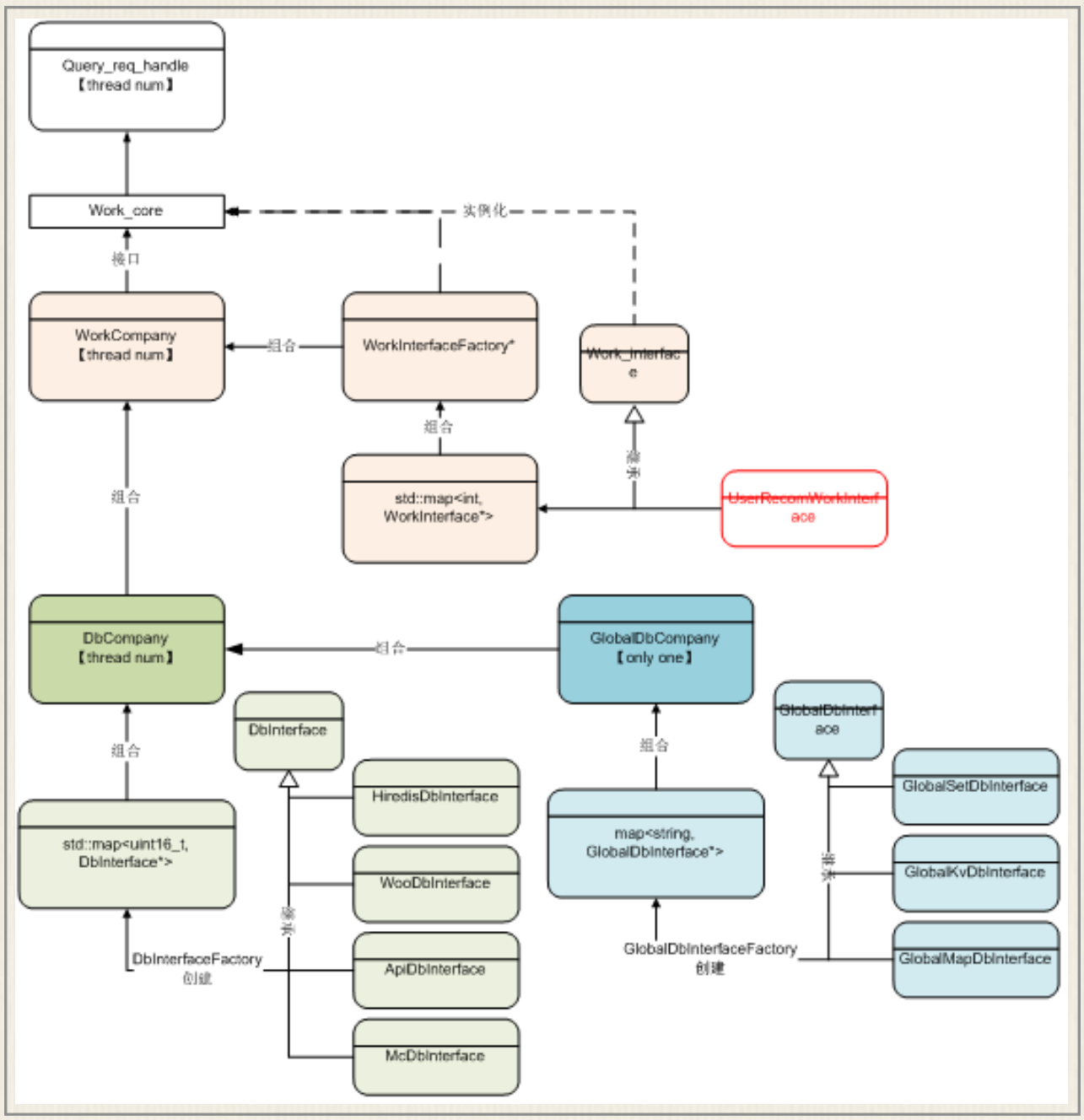
其中我们通过data_interface来实现数据透明化，通过work_interface来管理项目，my_interface二次开发者开 发属于自己的project。同时需要配置文件【data以及project】进行数据和项目的对应。这样会带来一个好处，通过这个框架将所有的资源整合 起来了，在推荐早期项目中起到了比较明显的作用：快速响应算法和产品策略、项目迭代比较迅速同时让业务人员在一定程度上关注业务，后来也推动了后来推荐存 储架构的诞生。通过不断的优化，这个逐渐被lua以及c/c++版本的通用框架取代。

3、推荐计算层RCompute

推荐计算层的职责是做高效运算，比如二度关系运算、用户之间桥梁关系运算、CTR预估等等CPU密集型的功能模块。同时在RApp以及 RFront有一些离线数据采用远程访问以前比较低效的，因此我们需要一个更加高效的通用框架来满足上述要求，lab_common_so孕育而生。

Lab_common_so是基于woo框架的【感谢曾经的同事支持，这是一个轻型的通讯框架，通讯协议以及日志系统比较完善】一个业务流框架， 同时兼有了CRF的特点——整合融入数据资源、规范业务流程、统一接口形态，还具有了数据本地化功能。其实在在线demo c/c++开发框架中有很多，在这里主要介绍一些自身特点：

- 通过c++的一些特性，让二次开发者在工作流上尽可能减少关注其他架构以及通讯包括存储客户端的事情，下图是主要的类结构图【UML的一套傻傻的不会，大家凑合看吧。】



- 通过global_data以及data的类实现获取数据的透明化
- 在后期的改造中，将work独立出来，有开发方自己进行编写具体实现和具体类，进行so化支持线上更新

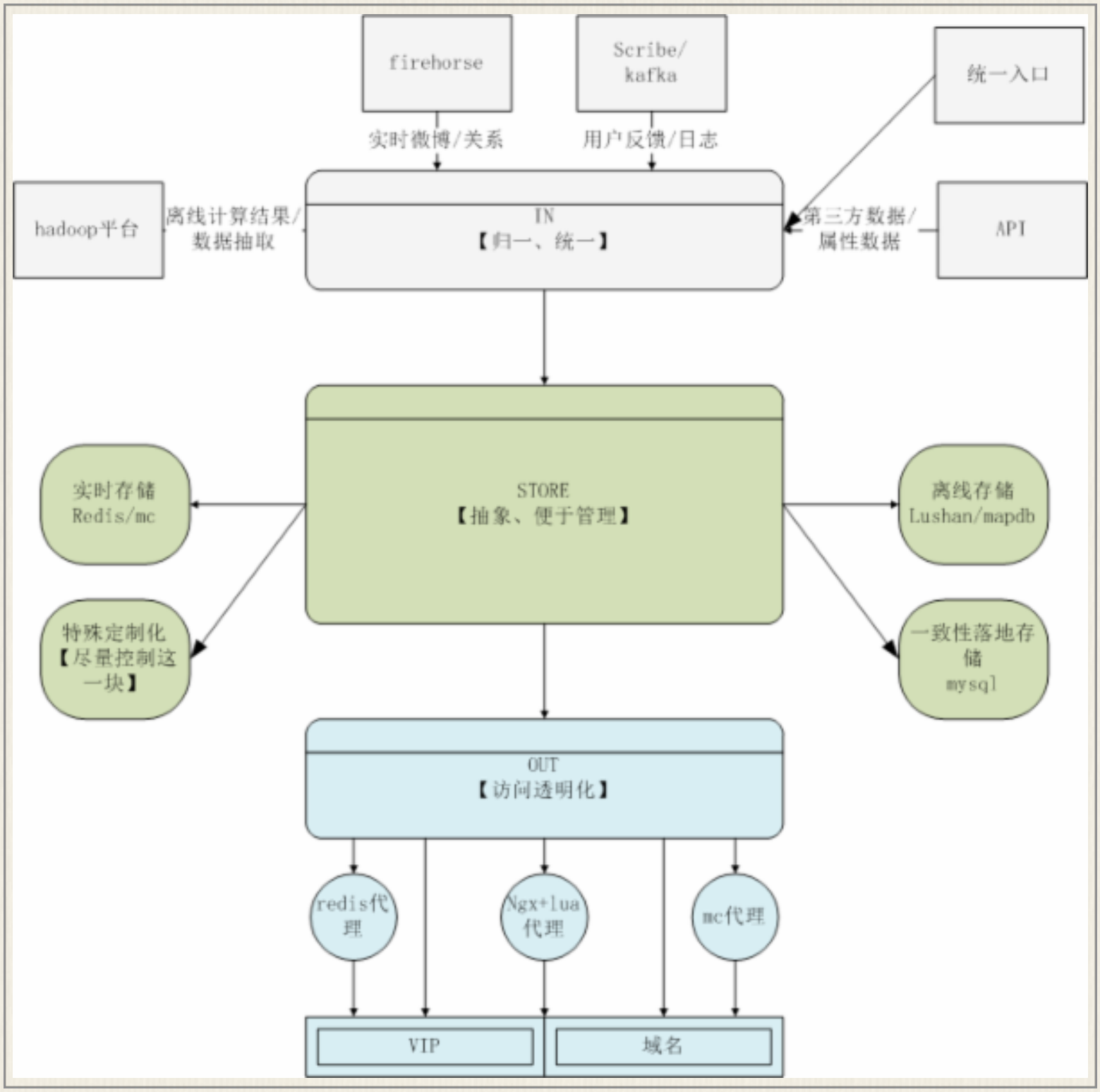
在计算层中有一个模块是RTaining，主要用来进行模型更新以及线上学习的，前期主要用在点击反馈或者实时模型上。

4、数据RStore

推荐数据对于推荐而言是基础，决定了性能也决定了效果。因此在数据存储上花的精力较多。微博推荐存储的数据拥有以下特点：

- 种类繁杂，比如需要存储挖掘的静态数据，还有用户实时行为的动态数据，有为提升性能而存储的属性缓存也有离线运算的直接结果数据
- 存储选型上也不能一概而论，动态数据以及静态数据的要求是不一样，统一都用高效的内存存储浪费资源，不宜扩展，都用离线的静态存储又达不到性能要求
- 多个外部门的数据，沟通和写作方式各式各样
- 某些数据量较大，同时在某些数据的实时性要求上较高。

针对上述特点，推荐抽象解决IN/OUT/STORE三部分。下图详细的描述了推荐数据的结构图：



- IN：输入的关键是解决统一和，因此通过建立基于ckestral的统一入口来解决数据规范的事情，同时在日志以及实时数据上也有相应的解决方案，统一入口为RIN。

- OUT：输出的关键是需要透明化，对于业务方而言不需要关系如何进行的服务器分配以及存储类型，而能够方面快速稳定地获取到数据就好。

- STORE：这一块主要解决数据类型繁杂的工作，推荐的经验是离线使用lushan以及mapdb，在线使用redis/mc，如果有特殊需求，比如批量高效压缩存储特征数据，也会使用一些自己定制的数据存储形式。

在这里面，需要特别强调的是离线静态数据存储，在性能以及成本上找到一定的平衡点是推荐比较有特色的地方，lushan以及mapdb会在之后做专题介绍。同时在推荐发展的过程中,hadoop的引入在一定程度上解决了很多候选以及排序的问题，因此一些map-reduce的结果数据如何放置于线上，推荐系统的r9项目很好的解决了这个问题。

5、数据值辅助工具：日志、监控、评估以及报警

推荐日志系统主要为了解决离线数据分析以及在线监控使用。当前推荐系统针对离线的分析来自于两块，一个是来源于自己的日志系统，另外一个是来自于公司的hadoop集群【几乎涵盖了非业务方的全局数据】，也在考虑将这两部分进行整合归并。

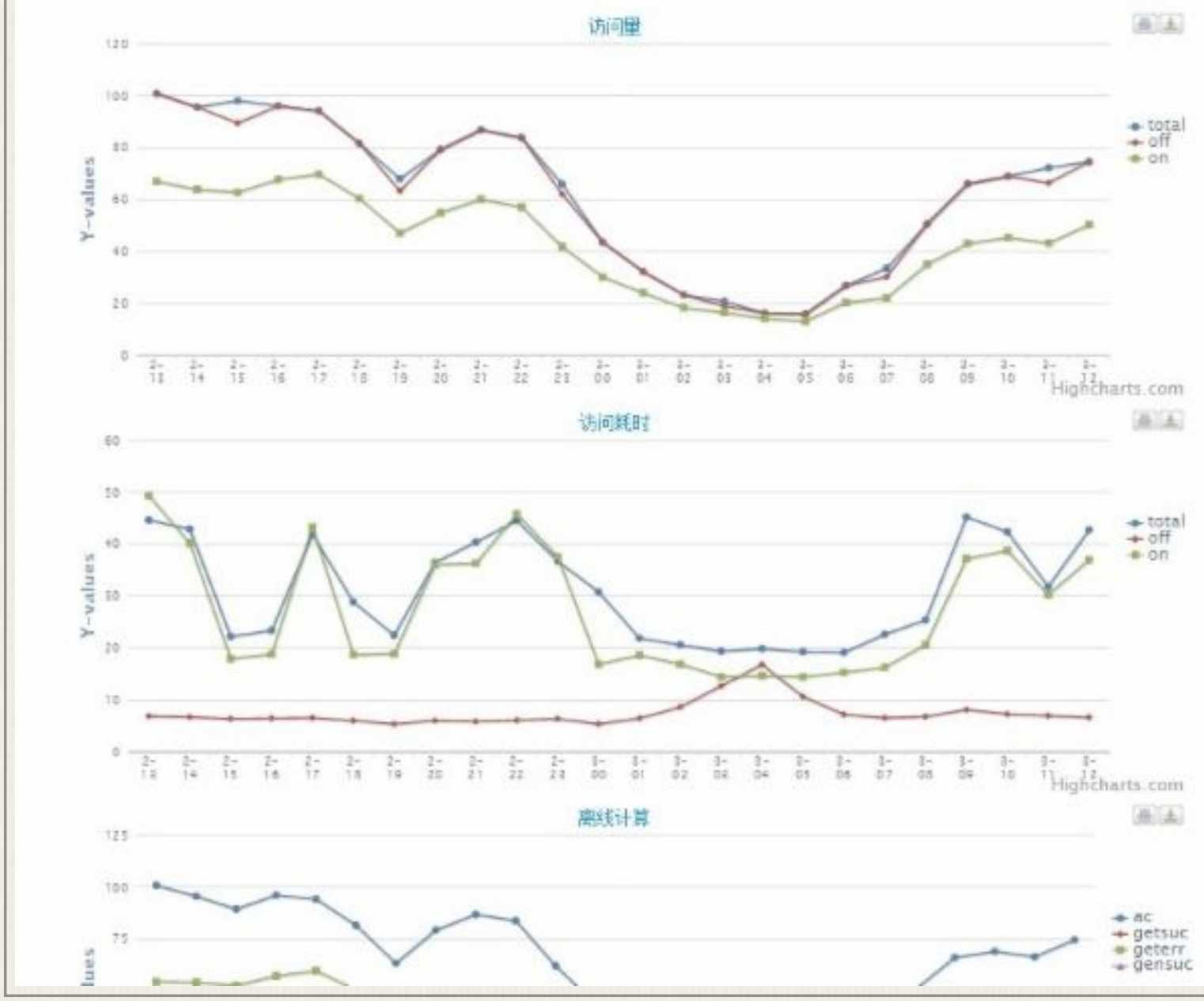
推荐监控系统是去年建设起来的，主要分为三部分：

- 性能监控，主要查看应用、计算以及存储的性能服务指标，同时与报警体系联动，及早发现问题、解决问题。目标是实时以及准实时，现在是分钟级别的。
- 效果监控，主要跟踪效果的优化和改进，当前重点业务线的效果均在监控系统中
- 对比测试监控，主要针对线上的A/B测试，进行分维度数据展示和对比

以下是我们监控的一些信息：

曝光以及性能监控：

趋势推荐好友计算服务



效果监控:

推荐评估系统，当前主要以demo实验、下线评估以及线上评估三个体系构成，其中线上评估体系主要跟着监控体系来的。而下线以及demo主要是一套前端的展示框架，几乎涵盖了推荐所有的接口以及对比测试数据。

推荐报警体系，微博本身是有的，不过由于性能指标以及效果指标存在一定上的定制化需求，这一块正在花精力解决。

好了，这一篇就到这里，推荐体系结构的介绍基本上告一段落了，其中涉及到的比较细节的工作将会在专题中进行介绍。也请大家多多捧场。

还是老规矩：

Simple is Beautiful！ 设计复杂的系统不是难事，难的是用简单的东西满足复杂业务，大家共勉！

原文链接：http://wbrecom.sinaapp.com/?p=48

.NET技术+25台服务器怎样支撑世界第54大网站

作者：Tod Hoff

意料之中，也是意料之外，Stack Overflow仍然重度使用着微软的产品。他们认为既然微软的基础设施可以满足需求，又足够便宜，那么没有什么理由去做根本上的改变。而在需要的地方，他们同样使用了Linux。究其根本，一切都是为了性能。

另一个值得关注的地方是，Stack Overflow仍然使用着纵向扩展策略，没有使用云。他们使用了384GB的内存和2TB的SSD来支撑SQL Servers，如果使用AWS的话，花费可想而知。没有使用云的另一个原因是Stack Overflow认为云会一定程度上的降低性能，同时也会给优化和排查系统问题增加难度。此外，他们的架构也并不需要横向扩展。峰值期间是横向扩展的杀手 级应用场景，然而他们有着丰富的系统调整经验去应对。该公司仍然坚持着Jeff Atwood的名言——硬件永远比程序员便宜。

Marco Ceccon曾提到，在谈及系统时，有一件事情必须首先弄明白——需要解决问题的类型。首先，从简单方面着手，StackExchange究竟是用来做什么的——首先是一些主题，然后围绕这些主题建立社区，最后就形成了这个令人敬佩的问答网站。

其次则是规模相关。StackExchange在飞速增长，需要处理大量的数据传输，那么这些都是如何完成的，特别是只使用了25台服务器，下面一起追根揭底：

状态

- StackExchange拥有110个站点，以每个月3到4个的速度增长。
- 400万用户

- 800万问题
- 4000万答案
- 世界排名54位
- 每年增长100%
- 月PV 5.6亿万
- 大多数工作日期间峰值为2600到3000请求每秒，作为一个编程相关网站，一般情况下工作日的请求都会高于周末
- 25台服务器
- SSD中储存了2TB的SQL数据
- 每个web server都配置了2个320G的SSD，使用RAID 1
- 每个ElasticSearch 主机都配备了300GB的机械硬盘，同时也使用了SSD
- Stack Overflow的读写比是40:60
- DB Server的平均CPU利用率是10%
- 11个web server，使用IIS
- 2个负载均衡器，1个活跃，使用HAProxy
- 4个活跃的数据库节点，使用MS SQL
- 3台实现了tag engine的应用程序服务器，所有搜索都通过tag
- 3台服务器通过ElasticSearch做搜索
- 2台使用了Redis的服务器支撑分布式缓存和消息
- 2台Networks（Nexus 5596 + Fabric Extenders）
- 2 Cisco 5525-X ASAs
- 2 Cisco 3945 Routers
- 主要服务Stack Exchange API的2个只读SQL Servers
- VM用于部署、域控制器、监控、运维数据库等场合

平台

- ElasticSearch
- Redis
- HAProxy
- MS SQL
- Opserver
- TeamCity
- Jil——Fast .NET JSON Serializer，建立在Sigil之上
- Dapper——微型的ORM

UI

- UI拥有一个信息收件箱，用于新徽章获得、用户发送信息、重大事件发生时的信息收取，使用WebSockets实现，并通过Redis支撑。
- 搜索箱通过 ElasticSearch 实现，使用了一个REST接口。
- 因为用户提出问题的频率很高，因此很难显示最新问题，每秒都会有新的问题产生，从而这里需要开发一个关注用户行为模式的算法，只给用户显示感兴趣的问题。它使用了基于Tag的复杂查询，这也是开发独立Tag Engine的原因。
- 服务器端模板用于生成页面。

服务器

- 25台服务器并没有满载，CPU使用率并不高，单计算SO（Stack Overflow）只需要5台服务器。
- 数据库服务器资源利用率在10%左右，除下执行备份时。

- 为什么会这么低？因为数据库服务器足足拥有384GB内存，同时web server的CPU利用率也只有10%-15%。
- 纵向扩展还没有遇到瓶颈。通常情况下，如此流量使用横向扩展大约需要100到300台服务器。
- 简单的系统。基于.Net，只用了9个项目，其他系统可能需要100个。之所以使用这么少系统是为了追求极限的编译速度，这点需要从系统开始时就进行规划，每台服务器的编译时间大约是10秒。
- 11万行代码，对比流量来说非常少。
- 使用这种极简的方式主要基于几个原因。首先，不需要太多测试，因为Meta.stackoverflow本来就是一个问题和bug讨论社区。其次，Meta.stackoverflow还是一个软件的测试网站，如果用户发现问题的话，往往会提出并给予解决方案。
- 纽约数据中心使用的是Windows 2012，已经向2012 R2升级（Oregon已经完成了升级），Linux系统使用的是Centos 6.4。

SSD

- 默认使用的是Intel 330（Web层等）
- Intel 520用于中间层写入，比如Elastic Search
- 数据层使用Intel 710和S3700
- 系统同时使用了RAID 1和RAID 10（任何4+以上的磁盘都使用RAID 10）。不畏惧故障发生，即使生产环境中使用了上千块2.5英寸SSD，还没碰到过一块失败的情景。每个模型都使用了1个以上的备件，多个磁盘发生故障的情景不在考虑之中。
- ElasticSearch在SSD上表现的异常出色，因为SO writes/re-indexes的操作非常频繁。
- SSD改变了搜索的使用方式。因为锁的问题，Luncene.net并不能支撑SO的并发负载，因此他们转向了ElasticSearch。在全SSD环境下，并不需要围绕Binary Reader建立锁。

高可用性

- 异地备份——主数据中心位于纽约，备份数据中心在Oregon。
- Redis有两个从节点，SQL有2个备份，Tag Engine有3个节点，elastic有3个节点，冗余一切，并在两个数据中心同时存在。
- Nginx是用于SSL，终止SSL时转换使用HAProxy。
- 并不是主从所有，一些临时的数据只会放到缓存中
- 所有HTTP流量发送只占总流量的77%，还存在Oregon数据中心的备份及一些其他的VPN流量。这些流量主要由SQL和Redis备份产生。

数据库

- MS SQL Server
- Stack Exchange为每个网站都设置了数据库，因此Stack Overflow有一个、Server Fault有一个，以此类推。
- 在纽约的主数据中心，每个集群通常都使用1主和1只读备份的配置，同时还会在Oregon数据中心也设置一个备份。如果是运行的是Oregon集群，那么两个在纽约数据中心的备份都会是只读和同步的。
- 为其他内容准备的数据库。这里还存在一个“网络范围”的数据库，用于储存登陆凭证和聚合数据（大部分是stackexchange.com用户文件或者API）。
- Careers Stack Overflow、stackexchange.com和Area 51等都拥有自己独立的数据库模式。
- 模式的变化需要同时提供给所有站点的数据库，它们需要向下兼容，举个例子，如果需要重命名一个列，那么将非常麻烦，这里需要进行多个操作：增加一个新列，添加作用在两个列上的代码，给新列写数据，改变代码让新列有效，移除旧列。
- 并不需要分片，所有事情通过索引来解决，而且数据体积也没那么大。如果有filtered indexes 需求，那么为什么不更高效的进行？常见模式只

在DeletionDate = Null上做索引，其他则通过为枚举指定类型。每项votes都设置了1个表，比如一张表给post votes，1张表给comment votes。大部分的页面都可以实时渲染，只为匿名用户缓存，因此，不存在缓存更新，只有重查询。

- Scores是非规范化的，因此需要经常查询。它只包含IDs和dates，post votes表格当下大约有56454478行，使用索引，大部分的查询都可以在数毫秒内完成。

- Tag Engine是完全独立的，这就意味着核心功能并不依赖任何外部应用程序。它是一个巨大的内存结构数组结构，专为SO用例优化，并为重负载组合进行预计算。Tag Engine是个简单的windows服务，冗余的运行在多个主机上。CPU使用率基本上保持在2-5%，3个主机专门用于冗余，不负责任何负载。如果所有主机同时发生故障，网络服务器将把Tag Engine加载到内存中持续运行。

- 关于Dapper无编译器校验查询与传统ORM的对比。使用编译器有很多好处，但在运行时仍然会存在fundamental disconnect问题。同时更重要的是，由于生成nasty SQL，通常情况还需要去寻找原始代码，而Query Hint和parameterization控制等能力的缺乏更让查询优化变得复杂。

编码

- 流程
- 大部分程序员都是远程工作，自己选择编码地点
- 编译非常快
- 然后运行少量的测试
- 一旦编译成功，代码即转移至开发交付准备服务器
- 通过功能开关隐藏新功能
- 在相同硬件上作为其他站点测试运行

- 然后转移至Meta.stackoverflow测试，每天有上千个程序员在使用，一个很好的测试环境
- 如果通过则上线，在更广大的社区进行测试
- 大量使用静态类和方法，为了更简单及更好的性能
- 编码过程非常简单，因为复杂的部分被打包到库里，这些库被开源和维护。.Net 项目数量很低，因为使用了社区共享的部分代码。
- 开发者同时使用2到3个显示器，多个屏幕可以显著提高生产效率。

缓存

- 缓存一切
- 5个等级的缓存
- 1级是网络级缓存，缓存在浏览器、CDN以及代理服务器中。
- 2级由.Net框架 HttpRuntime.Cache完成，在每台服务器的内存中。
- 3级Redis ，分布式内存键值存储，在多个支撑同一个站点的服务器上共享缓存项。
- 4级SQL Server Cache，整个数据库，所有数据都被放到内存中。
- 5级SSD。通常只在SQL Server预热后才生效。
- 举个例子，每个帮助页面都进行了缓存，访问一个页面的代码非常简单:
- 使用了静态的方法和类。从OOP角度来看确实很糟，但是非常快并有利于简洁编码。
- 缓存由Redis和Dapper支撑，一个微型ORM

- 为了解决垃圾收集问题，模板中1个类只使用1个副本，被建立和保存在缓存中。监测一切，包括GC操。据统计显示，间接层增加GC压力达到了某个程度时会显著的降低性能。
- CDN Hit 。鉴于查询字符串基于文件内容进行哈希，只在有新建立时才会被再次取出。每天3000万到5000万Hit，带宽大约为300GB到600GB。
- CDN不是用来应对CPU或I/O负载，而是帮助用户更快的获得答案

部署

- 每天5次部署，不去建立过大的应用。主要因为
- 可以直接的监视性能
- 尽可能最小化建立，可以工作才是重点
- 产品建立后再通过强大的脚本拷贝到各个网页层，每个服务器的步骤是：
  - 通过POST通知HAProxy下架某台服务器
  - 延迟IIS结束现有请求（大约5秒）
  - 停止网站（通过同一个PSSession结束所有下游）
  - Robocopy文件
  - 开启网站
  - 通过另一个POST做HAProxy Re-enable
- 几乎所有部署都是通过puppet或DSC ，升级通常只是大幅度调整RAID阵列并通过PXE boot安装，这样做非常快速。

协作

- 团队
- SRE （System Reliability Engineering）：5人
- Core Dev（Q&A site）6-7人
- Core Dev Mobile：6人
- Careers团队专门负责SO Careers产品开发：7人
- Devops和开发者结合的非常紧密
- 团队间变化很大
- 大部分员工远程工作
- 办公室主要用于销售，Denver和London除外
- 一切平等，些许偏向纽约工作者，因为面对面有助于工作交流，但是在线工作影响也并不大
- 对比可以在同一个办公室办公，他们更偏向热爱产品及有才华的工程师，他们可以很好的衡量利弊
- 许多人因为家庭而选择远程工作，纽约是不错，但是生活并不宽松
- 办公室设立在曼哈顿，那是个人才的诞生地。数据中心不能太偏，因为经常会涉及升级
- 打造一个强大团队，偏爱极客。早期的微软就聚集了大量极客，因此他们征服了整个世界
- Stack Overflow 社区也是个招聘的地点，他们在那寻找热爱编码、乐于助人及热爱交流的人才。

编制预算

- 预算是项目的基础。钱只花在为新项目建立基础设施上，如此低利用率的 web server还是3年前数据中心建立时购入。

测试

- 快速迭代和遗弃
- 许多测试都是发布队伍完成的。开发拥有一个同样的SQL服务器，并且运行在相同的Web层，因此性能测试并不会糟糕。
- 非常少的测试。Stack Overflow并没有进行太多的单元测试，因为他们使用了大量的静态代码，还有一个非常活跃的社区。
- 基础设施改变。鉴于所有东西都有双份，所以每个旧配置都有备份，并使用了一个快速故障恢复机制。比如，keepalived可以在负载均衡器中快速回退。
- 对比定期维护，他们更愿意依赖冗余系统。SQL备份用一个专门的服务器进行测试，只为了可以重存储。计划做每两个月一次的全数据中心故障恢复，或者使用完全只读的第二数据中心。
- 每次新功能发布都做单元测试、集成测试盒UI测试，这就意味着可以预知输入的产品功能测试后就会推送到孵化网站，即meta.stackexchange（原meta.stackoverflow）。

监视/日志

- 当下正在考虑使用http://logstash.net/做日志管理，目前使用了一个专门的服务将syslog UDP传输到SQL数据库中。网页中为计时添加header，这样就可以通过HAProxy来捕获并且融合到syslog传输中。
- Opserver和Realog用于显示测量结果。Realog是一个日志展示系统，由Kyle Brandt和Matt Jibson使用Go建立。
- 日志通过HAProxy负载均衡器借助syslog完成，而不是IIS，因为其功能比IIS更丰富。

关于云

- 还是老生常谈，硬件永远比开发者和有效率的代码便宜。基于木桶效应，速度肯定受限于某个短板，现有的云服务基本上都存在容量和性能限制。

- 如果从开始就使用云来建设SO说不定也会达到现在的水准。但毫无疑问的是，如果达到同样的性能，使用云的成本将远远高于自建数据中心。

性能至上

- StackOverflow是个重度的性能控，主页加载的时间永远控制在50毫秒内，当下的响应时间是28毫秒。

- 程序员热衷于降低页面加载时间以及提高用户体验。

- 每个独立的网络提交都予以计时和记录，这种计量可以弄清楚提升性能需要修改的地方。

- 如此低资源利用率的主要原因就是高效的代码。web server的CPU平均利用率在5%到15%之间，内存使用为15.5 GB，网络传输在20 Mb/s到40 Mb/
s。SQL服务器的CPU使用率在5%到10%之间，内存使用是365GB，网络传输为100 Mb/s到200 Mb/
s。这可以带来3个好处：给升级留下很大的空间；在严重错误发生时可以保持服务可用；在需要时可以快速回档。

学到的知识

1. 为什么使用MS产品的同时还使用Redis？什么好用用什么，不要做无必要的系统之争，比如C#在Windows机器上运行最好，我们使用IIS；Redis在*nix机器上可以得到充分发挥，我们使用*nix。

2. Overkill即策略。平常的利用率并不能代表什么，当某些特定的事情发生时，比如备份、重建等完全可以将资源使用拉满。

3. 坚固的SSD。所有数据库都建立在SSD之上，这样可以获得0延时。

4. 了解你的读写负载。

5. 高效的代码意味着更少的主机。只有新项目上线时才会因为特殊需求增加硬件，通常情况下是添加内存，但在此之外，高效的代码就意味着0硬件添加。所以经常只讨论两个问题：为存储增加新的SSD；为新项目增加硬件。

6. 不要害怕定制化。SO在Tag上使用复杂查询，因此专门开发了所需的Tag Engine。

7. 只做必须做的事情。之所以不需要测试是因为有一个活跃的社区支撑，比如，开发者不用担心出现“Square Wheel”效应，如果开发者可以制作一个更更轻量级的组件，那就替代吧。

8. 注重硬件知识，比如IL。一些代码使用IL而不是C#。聚焦SQL查询计划。使用web server的内存转储究竟做了些什么。探索，比如为什么一个s-plit会产生2GB的垃圾。

9. 切勿官僚作风。总有一些新的工具是你需要的，比如，一个编辑器，新版本的Visual Studio，降低提升过程中的一切阻力。

10. 垃圾回收驱动编程。SO在减少垃圾回收成本上做了很多努力，跳过类似TDD的实践，避免抽象层，使用静态方法。虽然极端，但是确实打造出非常高效的代码。

11. 高效代码的价值远远超出你想象，它可以让硬件跑的更快，降低资源使用，切记让代码更容易被程序员理解。

译文链接：http://www.csdn.net/article/2014-07-22/2820774-stackoverflow-update-560m-pageviews-a-month-25-servers

原文链接：http://highscalability.com/blog/2014/7/21/stackoverflow-update-560m-pageviews-a-month-25-servers-and-i.html

如何看待陈皓在微博上对闭源和开源软件的评论?

作者：陈皓

好吧，我来我答我制造的话题。

首先，我先表达一下开源软件的伟大，并向开源的人们致于我最真诚的敬意。但，即使这样，我们也要很客观的承认大多数开源软件是存在大量问题的。 就像我这个人主观上并不喜欢微软和IBM，但是我必需要承认，没有微软和IBM，计算机这个行业不会有今天这样的爆炸。（插曲：我94年上大学选的专业是 计算机科学，当时很多人都觉得计算机专业是一个ZB的专业，因为几乎没有哪个企业在用电脑，但我幸运的是95年的时候微软出了Win95，Sun出了 Java，然后互联网极度膨胀，才导致我选的专业最终变得很火）不可否认，IBM和微软里有相当NB让人五体投地的人。

至于我说的这两句话，我知道是有争议的，尤其是我用了“抄”，在此请原谅我用词不当，像从事开源的朋友道歉。这个“抄”其实是“模仿”的意思。

要是没有开源，我们中国人的软件公司能做出云平台吗？能做出手机吗？能大数据吗？真心感谢开源！

简单地说一下开源的历史

简 单的回顾一下历史，而开源源自Unix最初的发展史，然后，就被商业化了，于是N多的Unix变种就出来了，那些Unix老牌黑客们一下就被成了像罗宾汉 一样的成了丛林草莽，以至于Microsoft用次等的技术占领了市场，而RMS也开始了他的GNU项目，但是GNU并没有获得那些Unix老牌黑客的青 睐，因为他们觉得RMS就像当年马克思满世界鼓吹共产主义一样鼓吹他人的GNU，最终Linus出来把这些Unix老牌黑客召集了起来，让Unix的开源

精神重生。这段历史起源于Ken/Dennis，再次向他们致敬！详细的历史大家可以看看我7年前写的：Unix传奇(上篇)、Unix传奇(下篇)

我个人以为开源软件自Unix 以来，最杀手级的组合是LAMP，今天，Apache 基金会，Linux基金会……让开源的力量越来越大，很多商业公司都参与开源，比如IBM、Yahoo、SUN、Intel、Google……，

但是，我们可以看出，商业公司支持开源有一个很主要的原因是为了阻击竞争对手，理由很简单——用众包这种不花钱不花人的模式来牵制竞争对手实在是一个“低投入，大收益”的事。比 如IBM支持Linux和Java，目的主要是阻击微软。Google的Android和Chrome目的也是苹果和微软。而对于这些商公司的很多核心技 术是不会开放的，包括Google，连Google Reader都宁可自废都不愿意捐给开源社区维护，更别说Google的那三篇论文的东西了，以及Google的搜索引擎的技术。

这里，我是想说，如果开源是像Unix那样，有几个在那些顶尖的牛人以最纯粹的目的来搞开源的话，才可能还能搞得非常地好。

关于我观点中的逻辑

我观点中的逻辑其实很简单：

- 如果有人掌握了一个很核心技术，这个技术足以改变世界，你觉得会有多少人会开源？基本不会有人的。
- 你看看这个世界上的引领软件潮流的技术基本上都是商业公司做出来的。因为技术研发要花时间，花精力，更重要的是要花钱。如果你花了2-3年的时间，花费成百上千万的财力，你会开源吗？
- 大量开源软件都是受不了这些商业公司对技术的垄断以及非常高的价格。所以，基于这个动机，结果很自然就出现了“模仿”。

了解这些因果关系后，我相信你大约知道为什么闭源的东西要牛一些。

关于各种软件的对比

很多人对我的这个观点例了一些例子，但这些例子面太窄了，他们企图以点代面。我在这里帮大家补充一些吧，这样会更客观一些（眼界不妨放大一些）：

注意：千万不要用“用户量”来定义“技术含量”，如果你觉得：“有技术含量”===“有绝对的用户量”，那么，你就会得出“QQ空间甚至hao123可能是这世上最有技术含量的软件或网站”这样荒谬的结论。

1) VMWare 和 Xen/KVM

2) Google的三篇论文 和 hadoop

3) AWS 和 OpenStack

4) Google Reader和一干开源的reader

5) Websphere/Weblogic和Jboss, tomcat （注：互联网的大多数应用都比较简单）

6) 输入法，你是用sogou的还是google的，还是微软的？还是fcitx？

7) iOS和Android，你觉得哪个质量做得更高一些呢？

8) Windows/MacOS 和 Linux (对此需要分开：桌面、服务器、嵌入式)（再注：苹果公司开放了Mac OS的内核Dar-win的源代码，但没有包括GUI）

9) 多媒体方面的软件，比如：3D动画、音频、图像……Photoshop, Maya, 3DMax...开源的：Blender，GIMP，Inkscape……

10) 闭源的游戏和开源的游戏，你更喜欢玩哪个的呢？

11) 各种办公用的OA软件，MS Office，还有SAS和People-Soft的软件。开源的……

12) 用于出版行业排版的软件，Adobe的inDesign, PageMaker, MS的publisher，开源的Tex

13) 安全方面的软件，大家见得最多的就是杀毒软件了。（花絮：OpenSSL的代码大家看过吗？的确写得很ugly）

14) 软件测试相关的：PurifyPlus, VTune, CodeAnalyst, JProfiler……开源的：valgrind, gprof..

15) 企业内部的IT管理软件，大家可以看一下三个公司：IBM，BMC，CA，大家可以看看他们做了些什么样的ITIL的软件（关于ITIL请大家自行Google吧）。

16) 行业软件：集成电路设计的软件、石油勘探的软件、航空软件、汽车里的嵌入式的软件、医疗设备用的软件、金融行业的软件、建筑设计方面的（AutoCAD）

17) 聊天软件：QQ，Skype，YY，FaceTime，YIM

18) 编程IDE：关于IDE的比较，请移步参看Wikipedia：Comparison of integrated development environments （注：IntelliJ IDEA是半开源的）

19）网页制作：Dreamwaver, Fireworks, Flash...

.....

我还可以一直把例子举下去，因为还有很多地方的软件很少人见过的软件，比如，NASA的、CERN的、DreamWorks，等等。

希望你的视野比我更宽一点，别只看自己编程用的那些东西，多看看这个世界高精尖的地方。

其它

我有这样的观点主要是因为我的成长史主要是在商业公司，我能看得到这些商业公司中有很多比开源软件很NB的东西。我为什么喜欢进这些顶尖的商业公司，因为只有进到这些公司我才能看有权限看到这些令人惊叹的软件是怎么做出来的。

这也是我没有花精力贡献开源的原因。

这 和我不写书的原因是一样的。我不写书的原因是因为我看过Effective C++，TCP/IP详解，Unix编程艺术，等等这类经典的书，我觉得我根本就没有资格写书，如果我有他们的两三成的功力，我都会考虑出书，但是我真的 不行。（另一方面再看看书店里那些95%以上的垃圾的书，真是令人恶心）

同理，我没有做开源的原因也是一样，因为我看过很多商业公司里的那些令人惊叹的东西，我觉得我还没有资格去干个开源软件，最多跟随几个我力能及而且觉得还不错的开源软件。不过，我更愿意把我的时间和精力花在向这些商业公司学习之上，以获得更有技术含量的知识。

更新：以 前也做过些开源，比如：我在读过TCP/IP的书后就去写了一个类似Wireshark的东西，也开过源，但后来出现了Wireshark，觉得自己写完全写过不Wireshark，差距太大。之后又出来了两次这样的事。所以，后来，我就不转向更多的学习了，而不仓促地去做开源了。原因就是——因为我觉得开源不是为了开源而开源，也不是为了装逼而开源，更不是为了吸社会的血把项目分包给社会，开源就是为了给社会做贡献，我对给社会做真正的贡献的能力还达不到。

人生苦短，我又没有那个聪明的DNA，这世界上的垃圾已经有很多了，我就不必再为垃圾添砖加瓦了。我还是把精力放在多看和多学上吧。因为我连一个C++ STL或JDK中的一个容器类都没有信心能写好。

当然，我并不是说干开源的人不行。只不过，我没有信心贡献罢了。说得好听点，我标准比较高，说得难听点，我能力差。你对我的这两种理解都对。我的技术的确水，我在我的博客上的个人简介也说了我不是牛人。

谢谢大家看我的这个冗长的答案。

原文链接：http://www.zhihu.com/question/24616693/answer/28430044?utm_source=tuicool

、