推酷

编程狂人

Programming Madman

NO.43

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/542026abd91b142b7918002f

欢迎下载推酷客户端体验更多阅读乐趣

目录

模板引擎随谈

作者：四火

模板引擎是为了解耦而产生的，从编程范型的角度来说，写模板属于“声明式（Imperative）编程”。JSP大概是最早接触也是最基础的模板引 擎，本来写Servlet嘛，一大堆一大堆的print，实在是没有任何结构性可言，然后JSP出现，先被处理成实质为Servlet的Java文件，编 译以后变成class，接着一样执行。所以本质是编译型的模板引擎，当然模板引擎也有解释型或者二者混合的。通常说来编译型的执行效率要高得多。只要是和 显示相关的编程语言，都会发展出一套或者N套模板引擎，用得多了觉得很多情况下都大同小异。

几年前我在工作中折腾过一段时间的服务端模板引擎，最早遗留系统使用的Velocity，后来我们实现的时候用了FreeMarker， 因为后者功能更强大，IDE支持也更好，对于后者的macro（宏），实在是不知怎么讲，功能上它当然是一个强大的武器，但是没控制好就会让代码写得功能 不清，或者干脆很难看懂。在搞性能调优的时候，到后来不动大刀已经没有什么可以值得改进的地方了。遂眼光瞄到了FreeMarker上面，我们拿 profiler的工具检查出来模板引擎的解释执行耗费了大量的时间，而且其中的模板缓存命中率很低，公司里面有一个团队为此专门改了 FreeMarker 的代码，性能好像有20%的提高。

很多人搞web开始阶段都是自由生长的，或者说野蛮生长，完全没有章法，凭借着搜索引擎加试错大法，因此方法往往都不正统。我也一样。在我知道专门 的模板以前，我已经在粗暴地实现类似的事情了，让一个DIV不可见（display=none），然后里面变化的地方用占位符标识，在Ajax获得数据以 后把占位符替换成真正的文字，然后显示出来——这不就是一最土鳖的模板么？后来开始接触到一些前端模板引擎，Mustache是最早接触的，我不知道 {{ }} 这样的记号是不是从它开始的，然后是Handlebar.js，其实它用

的也是Mustache的引擎。Underscore.js是值得推荐的模板引擎，性能非常出色，而且语法和JSP差不多。AngularJS的模板是我最喜欢的形式（下面我列出了一个官网上面的例子），因为直接融合进HTML里面了，减少了生硬的特殊格式标签，可以给既有DOM对象增加属性，也可以通过directive方式自定义DOM。模板引擎怎么演进而来的，又是怎么从后端移到前端来的，其实都因一个“解耦”，这个过程我在《MVC框架的映射和解耦》以及《Web页面的聚合技术》里面都有部分介绍。

```
<ul>
  <li ng-repeat="phone in phones">
    {{phone.name}}
    <p>{{phone.snippet}}</p>
  </li>
</ul>
```

关于常见几款前端模板的比较，这里有一篇文章。HTML5用新标签的方式收录了模板，这里有一篇文章介绍。另外，这里有一个有趣的帖子，作者在入门Node.js的时候选模板，很多人在讨论Jade，它最有意思的地方是如果打开普通的没有代码辅助的记事本文件，它的编写效率真得高出好多，而且没有烦人的括号、尖括号之类的标记符号，不知道你怎么看。对于性能的横向比较，在JSPerf上面有人做了一个完整的列表，可以打开页面后立即测试。

关于模板引擎的原理解析，推荐一篇文章《高性能JavaScript模板引擎原理解析》，里面提到了“高性能”模板引擎的原理，这也是现在越来越多的JavaScript模板引擎的设计思路，尽量把工作放到预编译阶段去，生成函数以后，原始的模板就不再使用了，后面每次需要渲染的时候调用这个函数传入参数就可以了。

通过一个小小的例子，可以看到模板引擎的工作原理，这里拿Handlerbar.js举例：

```
<table>
    {{#each users}}
    <tr>
        <td>
            {{this.name}}
        </td>
        <td>
            {{this.age}}
        </td>
    </tr>
    {{/each}}
</table>
```

对于这样一段简单的模板，调用语句是：

```
var func = Handlebars.compile(document.getElementById("template").innerHTML);
var result = func({
    users : [
        {
            name : "A",
            age : 10
        },
        {
```

```
            name : "B",
            age : 20
        }
    ]
});
console.log(result);
```

接着动态生成了这样的Function：

```
this.compilerInfo = [4,'>= 1.0.0'];
helpers = this.merge(helpers, Handlebars.helpers); data = data || {};
  var buffer = "", stack1, functionType="function",
escapeExpression=this.escapeExpression, self=this;

function program1(depth0,data) {


  var buffer = "", stack1;
  buffer += "\n <tr>\n        <td>"
    + escapeExpression(((stack1 = depth0.name),typeof stack1 === func-
tionType ? stack1.apply(depth0) : stack1))
    + "</td>\n        <td>"
    + escapeExpression(((stack1 = depth0.age),typeof stack1 === func-
tionType ? stack1.apply(depth0) : stack1))
    + "</td>\n    </tr>\n   ";
  return buffer;
```

```
    }

    buffer += "\n<table>\n  ";
    stack1 = helpers.each.call(depth0, depth0.users,
{hash:{},inverse:self.noop,fn:self.program(1, program1, data),data:data});
    if(stack1 || stack1 === 0) { buffer += stack1; }
    buffer += "\n</table>\n";
    return buffer;
```

其实代码并不难理解，这里的each就是通过内置的工具方法helpers.each来实现的，执行总的来说就是递归调用（第9、11行），如果stack1还是方法就继续调用，否则就直接转码（escapeExpression）显示。最终拼接成字符串输出。

原文链接：http://www.raychase.net/2568

Bootstrap vs Foundation如何选择靠谱前端框架

作者：newghost

现在OurJS开源网站有两套前端模板了，分别基于Foundation5 和Bootstrap2.3 （最近已经提交到Github上)。

经过一段时间使用，对于二者有一些粗略的了解，关于具体的比较细节，可以参考这篇E章 已经写的非常详细了。这里只是从另外一些角度来比较这两个目前最流行的 响应式前端框架;

Bootstrap和Foundation的粗略比较

整体来说Foundation比Bootstrap略显高大上一点，使用的都是比较新的技术，整体观是以Mofile first(移动优先)来考虑的。

Foundation默认不带图标集，它推荐使用开源字体图标来实现ICON，好处是可以通过字体大小来调节图片大小， 而boot-strap自带一个默认的由传统图片实现的图标集;

Foundation 使用 border-box 盒子模型（box model) 即 它定义width 和 height时，border 和 padding是被包含在内的， IE6即使用这种模型，个人认为这是一把双刃剑，可能会跟有些第三方的前端插件有冲突。

Foundation 的网格流式布局跟 Bootstrap差不多，他们都把网页划分为十二列，针对不同的设备显示不同的列数，如手机只能显视一个列单元，桌面电脑可以显示12个。

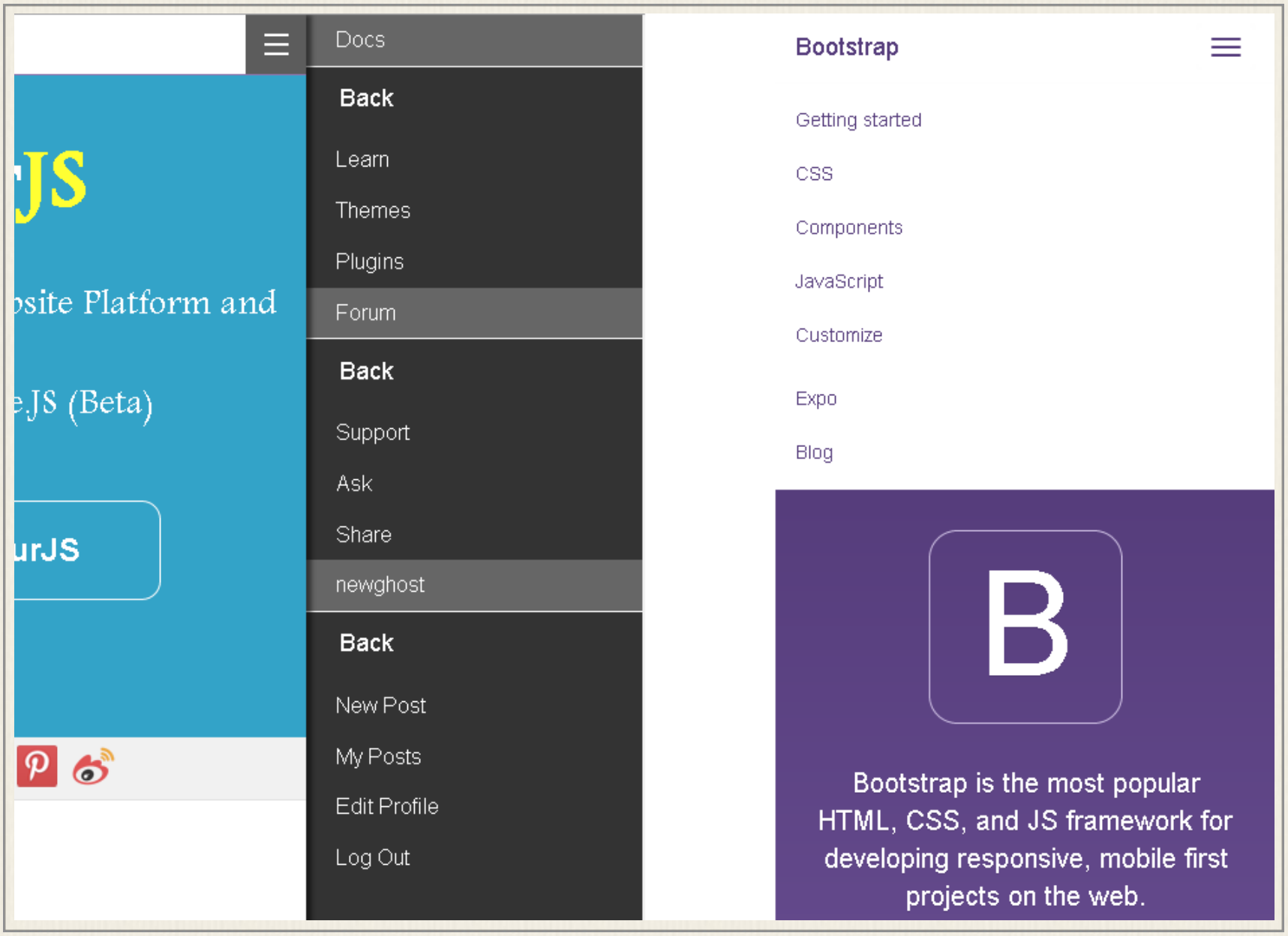
Bootstrap 2.2

* *

```
    <div class="span7"></div>
    <div class="span1"></div>
    <div class="span4"></div>
</div>
```

```
Foundation 5
<div class="row">
    <div class="large-4 columns"></div>
    <div class="large-4 columns"></div>
    <div class="large-4 columns"></div>
</div>
```

Foundation是首先实现Off Canvas布局的，即隐藏的菜单可以从厕面移出；而Bootstrap2的菜单只能从菜单顶部向下展开，个人认为Foundation的体验在手机上更 好一些; 好在Bootstrap 3.0 版本也加入了Off Canvas布局。



Bootstrap和Foundation的浏览器兼容

Bootstrap最大的优点就是浏览器兼容，因为使用的技术较新，Foundation无法支持旧版IE：

Bootstrap 2.3 支持 IE7.0+

Bootstrap 3 支持 IE8.0+

Foundation 支持 IE9.0+

XP 系统最高只能升级到IE8，Win7 默认安装的是IE8， 选择Foundation即意味着放弃整个XP系统和不能连网升级的Win7系统，这也基本上意味着你基本放弃了一部分中国的电脑桌面，对于面向非IT专 业人士的网站来

说，这一点有点致命。对于面向中国用户的网站来说，Bootstrap也许是更更好的选择。

对于IE6则可以放心大胆地不支持了，这是OurJS的浏览器使用情况, 数据来自Google Analytics （谷歌分析）

最近访问者浏览器版本分布情况, 由于OurJS 大多面向程序员，所以IE的比例非常小 < 5%的样子, 因此使用Foundation应该也没什么大问题。

1.	Chrome	26,793	65.78%
2.	Firefox	4,836	11.87%
3.	Safari (in-app)	2,007	4.93%
4.	Internet Explorer	2,000	4.91%
5.	Android Browser	1,986	4.88%
6.	Safari	1,912	4.69%
7.	UC Browser	465	1.14%

最近访问者IE浏览器版本分布，可以看出IE8的比例还是非常高的，IE6/IE7 非常小，Bootstrap3应该也可以放心大胆的用了。

1.	11.0	707	35.35%
2.	10.0	399	19.95%
3.	8.0	373	18.65%
4.	9.0	296	14.80%
5.	7.0	158	7.90%
6.	6.0	64	3.20%

其实说两套开源框架都在不断的相互学习，很难讲分出优劣，每个人可以根据自己的需要做出选择。

PS: OurJS的中文版选择基于修改过的Bootstrap2.3，英文版则采用了Foundation 5。

原文链接：http://ourjs.com/detail/bootstrap-vs-foundation%E5%A6%82%E4%BD%95%E9%80%89%E6%8B%A9%E9%9D%A0%E8%B0%B1%E5%89%8D%E7%AB%AF%E6%A1%86%E6%9E%B6

Python 代码性能优化技巧

作者：张颖

代码优化能够让程序运行更快，它是在不改变程序运行结果的情况下使得程序的运行效率更高，根据 80/20 原则，实现程序的重构、优化、扩展以及文档相关的事情通常需要消耗 80% 的工作量。优化通常包含两方面的内容：减小代码的体积，提高代码的运行效率。

改进算法，选择合适的数据结构

一个良好的算法能够对性能起到关键作用，因此性能改进的首要点是对算法的改进。在算法的时间复杂度排序上依次是：

O(1) -> O(lg n) -> O(n lg n) -> O(n^2) -> O(n^3) -> O(n^k) -> O(k^n) -> O(n!)

因此如果能够在时间复杂度上对算法进行一定的改进，对性能的提高不言而喻。但对具体算法的改进不属于本文讨论的范围，读者可以自行参考这方面资料。下面的内容将集中讨论数据结构的选择。

- 字典 (dictionary) 与列表 (list)

Python 字典中使用了 hash table，因此查找操作的复杂度为 O(1)，而 list 实际是个数组，在 list 中，查找需要遍历整个 list，其复杂度为 O(n)，因此对成员的查找访问等操作字典要比 list 更快。

清单 1. 代码 dict.py

```
from time import time

t = time()

list = ['a','b','is','python','jason','hello','hill','with','phone','test',
```

```
'dfdf','apple','pddf','ind','basic','none','baecr','var','bana','dd','wrd']
 #list = dict.fromkeys(list,True)
 print list
 filter = []
 for i in range (1000000):
   for find in ['is','hat','new','list','old','.']:
         if find not in list:
               filter.append(find)
 print "total run time:"
 print time()-t
```

上述代码运行大概需要 16.09seconds。如果去掉行 #list = dict.fromkeys (list,True) 的注释，将 list 转换为字典之后再运行，时间大约为 8.375 sec-onds ，效率大概提高了一半。因此在需要多数据成员进行频繁的查找或者访问的时候，使用 dict 而不是 list 是一个较好的选择。

- 集合 (set) 与列表 (list)

set 的 union， intersection，difference 操作要比 list 的迭代要快。因此如果涉及到求 list 交集，并集或者差的问题可以转换为 set 来操作。

清单 2. 求 list 的交集：

```
from time import time
t = time()
lista=[1,2,3,4,5,6,7,8,9,13,34,53,42,44]
listb=[2,4,6,9,23]
intersection=[]
```

```
for i in range (1000000):
  for a in lista:
        for b in listb:
              if a == b:
                    intersection.append(a)


print "total run time:"
print time()-t
```

上述程序的运行时间大概为：

```
total run time:
38.4070000648
```

清单 3. 使用 set 求交集

```
from time import time
t = time()
lista=[1,2,3,4,5,6,7,8,9,13,34,53,42,44]
listb=[2,4,6,9,23]
intersection=[]
for i in range (1000000):
  list(set(lista)&set(listb))
print "total run time:"
print time()-t
```

改为 set 后程序的运行时间缩减为 8.75，提高了 4 倍多，运行时间大大缩短。读者可以自行使用表 1 其他的操作进行测试。

表 1. set 常见用法

语法	操作	说明
set(list1) | set(list2)	union	包含 list1 和 list2 所有数据的新集合
set(list1) & set(list2)	intersection	包含 list1 和 list2 中共同元素的新集合
set(list1) – set(list2)	difference	在 list1 中出现但不在 list2 中出现的元素的集合

对循环的优化

对循环的优化所遵循的原则是尽量减少循环过程中的计算量，有多重循环的尽量将内层的计算提到上一层。 下面通过实例来对比循环优化后所带来的性能的提高。程序清单 4 中，如果不进行循环优化，其大概的运行时间约为 132.375。

清单 4. 为进行循环优化前

```
from time import time
t = time()
lista = [1,2,3,4,5,6,7,8,9,10]
listb =[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.01]
for i in range (1000000):
  for a in range(len(lista)):
        for b in range(len(listb)):
              x=lista[a]+listb[b]
print "total run time:"
```

```
print time()-t
```

现在进行如下优化，将长度计算提到循环外，range 用 xrange 代替，同时将第三层的计算 lista[a] 提到循环的第二层。

清单 5. 循环优化后

```
from time import time
t = time()
lista = [1,2,3,4,5,6,7,8,9,10]
listb =[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.01]
len1=len(lista)
len2=len(listb)
for i in xrange (1000000):
  for a in xrange(len1):
        temp=lista[a]
        for b in xrange(len2):
              x=temp+listb[b]
print "total run time:"
print time()-t
```

上述优化后的程序其运行时间缩短为 102.171999931。在清单 4 中 lista[a] 被计算的次数为 1000000*10*10，而在优化后的代码中被计算的次数为 1000000*10，计算次数大幅度缩短，因此性能有所提升。

充分利用 Lazy if-evaluation 的特性

python 中条件表达式是 lazy evaluation 的，也就是说如果存在条件表达式 if x and y，在 x 为 false 的情况下 y 表达式的值将不再计算。因此可以利用该特性在一定程度上提高程序效率。

清单 6. 利用 Lazy if-evaluation 的特性

```
from time import time
t = time()
abbreviations = ['cf.', 'e.g.', 'ex.', 'etc.', 'fig.', 'i.e.', 'Mr.', 'vs.']
for i in range (1000000):
  for w in ('Mr.', 'Hat', 'is', 'chasing', 'the', 'black', 'cat', '.'):
        if w in abbreviations:
        #if w[-1] == '.' and w in abbreviations:
              pass
print "total run time:"
print time()-t
```

在未进行优化之前程序的运行时间大概为 8.84，如果使用注释行代替第一个 if，运行的时间大概为 6.17。

字符串的优化

python 中的字符串对象是不可改变的，因此对任何字符串的操作如拼接，修改等都将产生一个新的字符串对象，而不是基于原字符串，因此这种持续的 copy 会在一定程度上影响 python 的性能。对字符串的优化也是改善性能的一个重要的方面，特别是在处理文本较多的情况下。字符串的优化主要集中在以下几个方面：

1. 在字符串连接的使用尽量使用 join() 而不是 +：在代码清单 7 中使用 + 进行字符串连接大概需要 0.125 s，而使用 join 缩短为 0.016s。因此在字符的操作上 join 比 + 要快，因此要尽量使用 join 而不是 +。

清单 7. 使用 join 而不是 + 连接字符串

```
from time import time


t = time()
s = ""
list = ['a','b','b','d','e','f','g','h','i','j','k','l','m','n']
for i in range (10000):
        for substr in list:
                s+= substr
print "total run time:"
print time()-t
```

同时要避免：

```
s = ""
for x in list:
  s += func(x)
```

而是要使用：

```
slist = [func(elt) for elt in somelist]
s = "".join(slist)
```

2. 当对字符串可以使用正则表达式或者内置函数来处理的时候，选择内置函数。如 str.isalpha()，str.isdigit()，str.startswith((‘x’, ‘yz’))，str.endswith((‘x’, ‘yz’))

3. 对字符进行格式化比直接串联读取要快，因此要使用 out = "<html>%s%s%s%s</html>" % (head, prologue, query, tail)
而避免
out = "<html>" + head + prologue + query + tail + "</html>"

使用列表解析（list comprehension）和生成器表达式（generator expression）

列表解析要比在循环中重新构建一个新的 list 更为高效，因此我们可以利用这一特性来提高运行的效率。

```
from time import time
t = time()
list = ['a','b','is','python','jason','hello','hill','with','phone','test',
'dfdf','apple','pddf','ind','basic','none','baecr','var','bana','dd','wrd']
total=[]
for i in range (1000000):
  for w in list:
        total.append(w)
print "total run time:"
print time()-t
```

使用列表解析:

```
for i in range (1000000):
```

```
a = [w for w in list]
```

上述代码直接运行大概需要 17s，而改为使用列表解析后 ，运行时间缩短为 9.29s。将近提高了一半。生成器表达式则是在 2.4 中引入的新内容，语法和列表解析类似，但是在大数据量处理时，生成器表达式的优势较为明显，它并不创建一个列表，只是返回一个生成器，因此效率较高。在上 述例子上中代码 a = [w for w in list] 修改为 a = (w for w in list)，运行时间进一步减少，缩短约为 2.98s。

其他优化技巧

1.如果需要交换两个变量的值使用 a,b=b,a 而不是借助中间变量 t=a;a=b;b=t；

```
>>> from timeit import Timer
>>> Timer("t=a;a=b;b=t","a=1;b=2").timeit()
0.25154118749729365
>>> Timer("a,b=b,a","a=1;b=2").timeit()
0.17156677734181258
>>>
```

2.在循环的时候使用 xrange 而不是 range；使用 xrange 可以节省大量的系统内存，因为 xrange() 在序列中每次调用只产生一个整数元素。而 range() 將直接返回完整的元素列表，用于循环时会有不必要的开销。在 python3 中 xrange 不再存在，里面 range 提供一个可以遍历任意长度的范围的 iterator。

3.使用局部变量，避免”global” 关键字。python 访问局部变量会比全局变量要快得多，因 此可以利用这一特性提升性能。

4.if done is not None 比语句 if done != None 更快，读者可以自行验证；

5.在耗时较多的循环中，可以把函数的调用改为内联的方式；

6.使用级联比较 “x < y < z” 而不是 “x < y and y < z”；

7.while 1 要比 while True 更快（当然后者的可读性更好）；

8.build in 函数通常较快，add(a,b) 要优于 a+b。

定位程序性能瓶颈

对代码优化的前提是需要了解性能瓶颈在什么地方，程序运行的主要时间是消耗在哪里，对于比较复杂的代码可以借助一些工具来定位，python 内置了丰富的性能分析工具，如 profile,cProfile 与 hotshot 等。其中 Profiler 是 python 自带的一组程序，能够描述程序运行时候的性能，并提供各种统计帮助用户定位程序的性能瓶颈。Python 标准模块提供三种 profilers:cProfile,profile 以及 hotshot。

profile 的使用非常简单，只需要在使用之前进行 import 即可。具体实例如下：

清单 8. 使用 profile 进行性能分析

```
import profile
def profileTest():
    Total =1;
    for i in range(10):
        Total=Total*(i+1)
        print Total
    return Total
if __name__ == "__main__":
    profile.run("profileTest()")
```

程序的运行结果如下：

图 1. 性能分析结果

其中输出每列的具体解释如下：

- ncalls：表示函数调用的次数；
- tottime ： 表示指定函数的总的运行时间，除掉函数中调用子函数的运行时间；
- percall：（第一个 percall）等于 tottime/ncalls；
- cumtime ： 表示该函数及其所有子函数的调用运行的时间，即函数开始调用到返回的时间；
- percall：（第二个 percall）即函数运行一次的平均时间，等于 cumtime/ncalls；
- filename:lineno(function)：每个函数调用的具体信息；

如果需要将输出以日志的形式保存，只需要在调用的时候加入另外一个参数。如 profile.run(“profileTest()”,”testprof”)。

对于 profile 的剖析数据，如果以二进制文件的时候保存结果的时候，可以通过 pstats 模块进行文本报表分析，它支持多种形式的报表输出，是文本界面下一个较为实用的工具。使用非常简单：

```
import pstats
p = pstats.Stats('testprof')
p.sort_stats("name").print_stats()
```

其中 sort_stats() 方法能够对剖分数据进行排序， 可以接受多个排序字段，如 sort_stats(‘name’, ‘file’) 将首先按照函数名称进行排序，然后再按照文件名进行排序。常见的排序字段有 calls(被调用的次数)，time（函数内部运行时间），cumulative（运行的总时间）等。此外 pstats

也提供了命令行交互工具，执行 python – m pstats 后可以通过 help 了解更多使用方式。

对于大型应用程序，如果能够将性能分析的结果以图形的方式呈现，将会非常实用和直观，常见的可视化工具有 Gprof2Dot，visualpytune，K-CacheGrind 等，读者可以自行查阅相关官网，本文不做详细讨论。

Python 性能优化工具

Python 性能优化除了改进算法，选用合适的数据结构之外，还有几种关键的技术，比如将关键 python 代码部分重写成 C 扩展模块，或者选用在性能上更为优化的解释器等，这些在本文中统称为优化工具。python 有很多自带的优化工具，如 Psyco，Pypy，Cython，Pyrex 等，这些优化工具各有千秋，本节选择几种进行介绍。

Psyco

psyco 是一个 just-in-time 的编译器，它能够在不改变源代码的情况下提高一定的性能，Psyco 将操作编译成有点优化的机器码，其操作分成三个不同的级别，有”运行时”、”编译时”和”虚拟时”变量。并根据需要提高和降低变量的级别。运行时变量只是 常规 Python 解释器处理的原始字节码和对象结构。一旦 Psyco 将操作编译成机器码，那么编译时变量就会在机器寄存器和可直接访问的内存位置中表示。同时 python 能高速缓存已编译的机器码以备今后重用，这样能节省一点时间。但 Psyco 也有其缺点，其本身运行所占内存较大。目前 psyco 已经不在 python2.7 中支持，而且不再提供维护和更新了，对其感兴趣的可以参考 http://psyco.sourceforge.net/

Pypy

PyPy 表示 “用 Python 实现的 Python”，但实际上它是使用一个称为 RPython 的 Python 子集实现的，能够将 Python 代码转成 C， .NET， Java 等语言和平台的代码。PyPy 集成了一种即时 (JIT) 编译器。和许多编译器，解释器不同，它不关心 Python 代码的词法分析和语法树。 因为它是用 Python 语言写的，所以它直接利用 Python 语言的 Code Object.。 Code Object 是 Python 字节码的表示，也就是说， PyPy 直接分析 Python 代码所对应的字节码 ,，这些字节码即不是以字符形式也不是以某种二进制格式保存在文件中， 而在 Python 运行环境中。目前版本是 1.8. 支持不同的平台

安装，windows 上安装 Pypy 需要先下载 https://bitbucket.org/pypy/pypy/downloads/pypy-1.8-win32.zip， 然后解压到相关的目录，并将解压后的路径添加到环境变量 path 中即可。在命令行运行 pypy，如果出现如下错误：”没有找到 MSVCR100.dll, 因此这个应用程序未能启动，重新安装应用程序可能会修复此问题”，则还需要在微软的官网上下载 VS 2010 runtime libraries 解决该问题。具体地址为 http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=5555

安装成功后在命令行里运行 pypy，输出结果如下：

```
C:\Documents and Settings\Administrator>pypy
Python 2.7.2 (0e28b379d8b3, Feb 09 2012, 18:31:47)
[PyPy 1.8.0 with MSC v.1500 32 bit] on win32
Type "help", "copyright", "credits" or "license" for more information.
And now for something completely different: ``PyPy is vast, and con-
tains
multitudes"
>>>>
```

以清单 5 的循环为例子，使用 python 和 pypy 分别运行，得到的运行结果分别如下：

```
C:\Documents and Settings\Administrator\ 桌面 \doc\python>pypy
loop.py
total run time:
8.42199993134

C:\Documents and Settings\Administrator\ 桌面 \doc\python>python
loop.py
total run time:
```

106.391000032

可见使用 pypy 来编译和运行程序，其效率大大的提高。

Cython

Cython 是用 python 实现的一种语言，可以用来写 python 扩展，用它写出来的库都可以通过 import 来载入，性能上比 python 的快。cython 里可以载入 python 扩展 (比如 import math)，也可以载入 c 的库的头文件 (比如 :cdef extern from “math.h”)，另外也可以用它来写 python 代码。将关键部分重写成 C 扩展模块

Linux Cpython 的安装：

第一步：下载

[root@v5254085f259 cpython]# wget -N http://cython.org/release/Cython-0.15.1.zip

--2012-04-16 22:08:35-- http://cython.org/release/Cython-0.15.1.zip

Resolving cython.org... 128.208.160.197

Connecting to cython.org|128.208.160.197|:80... connected.

HTTP request sent, awaiting response... 200 OK

Length: 2200299 (2.1M) [application/zip]

Saving to: `Cython-0.15.1.zip'

100%[======================================>] 2,200,299 1.96M/s in 1.1s

2012-04-16 22:08:37 (1.96 MB/s) - `Cython-0.15.1.zip' saved [2200299/2200299]

第二步：解压

[root@v5254085f259 cpython]# unzip -o Cython-0.15.1.zip

第三步：安装

python setup.py install

安装完成后直接输入 cython，如果出现如下内容则表明安装成功。

```
[root@v5254085f259 Cython-0.15.1]# cython
Cython (http://cython.org) is a compiler for code written in the
Cython language.  Cython is based on Pyrex by Greg Ewing.

Usage: cython [options] sourcefile.{pyx,py} ...

Options:
  -V, --version                  Display version number of cython compiler
  -l, --create-listing           Write error messages to a listing file
  -I, --include-dir <directory>  Search for include files in named directory
                                 (multiple include directories are allowed).
  -o, --output-file <filename>   Specify name of generated C file
  -t, --timestamps               Only compile newer source files
  -f, --force                    Compile all source files (overrides implied -t)
  -q, --quiet                    Don't print module names in recursive mode
  -v, --verbose                  Be verbose, print file names on multiple compil
ation
  -p, --embed-positions          If specified, the positions in Cython files of
each
  function definition is embedded in its docstring.
  --cleanup <level>
  Release interned objects on python exit, for memory debugging.
```

Level indicates aggressiveness, default 0 releases nothing.

-w, --working <directory>

Sets the working directory for Cython (the directory modules are searched from)

--gdb Output debug information for cygdb

-D, --no-docstrings

Strip docstrings from the compiled module.

-a, --annotate

Produce a colorized HTML version of the source.

--line-directives

Produce #line directives pointing to the .pyx source

--cplus

Output a C++ rather than C file.

--embed[=<method_name>]

Generate a main() function that embeds the Python interpreter.

-2 Compile based on Python-2 syntax and code seman tics.

-3 Compile based on Python-3 syntax and code seman tics.

--fast-fail Abort the compilation on the first error

--warning-error, -Werror Make all warnings into errors

--warning-extra, -Wextra Enable extra warnings

-X, --directive <name>=<value>

[,<name=value,...] Overrides a compiler directive

其他平台上的安装可以参考文档：http://docs.cython.org/src/quickstart/install.html

Cython 代码与 python 不同，必须先编译，编译一般需要经过两个阶段，将 pyx 文件编译为 .c 文件，再将 .c 文件编译为 .so 文件。编译有多种方法：

- 通过命令行编译：假设有如下测试代码，使用命令行编译为 .c 文件。 def sum(int a,int b):
- print a+b
-
- [root@v5254085f259 test]# cython sum.pyx
- [root@v5254085f259 test]# ls
- total 76
- 4 drwxr-xr-x 2 root root 4096 Apr 17 02:45 .
- 4 drwxr-xr-x 4 root root 4096 Apr 16 22:20 ..
- 4 -rw-r--r-- 1 root root   35 Apr 17 02:45 1
- 60 -rw-r--r-- 1 root root 55169 Apr 17 02:45 sum.c
- 4 -rw-r--r-- 1 root root   35 Apr 17 02:45 sum.pyx

在 linux 上利用 gcc 编译为 .so 文件：

[root@v5254085f259 test]# gcc -shared -pthread -fPIC -fwrapv -O2

- -Wall -fno-strict-aliasing -I/usr/include/python2.4 -o sum.so sum.c
- [root@v5254085f259 test]# ls
- total 96
- 4 drwxr-xr-x 2 root root 4096 Apr 17 02:47 .
- 4 drwxr-xr-x 4 root root 4096 Apr 16 22:20 ..
- 4 -rw-r--r-- 1 root root   35 Apr 17 02:45 1
- 60 -rw-r--r-- 1 root root 55169 Apr 17 02:45 sum.c
- 4 -rw-r--r-- 1 root root   35 Apr 17 02:45 sum.pyx

- 20 -rwxr-xr-x 1 root root 20307 Apr 17 02:47 sum.so
- 使用 distutils 编译建立一个 setup.py 的脚本： from distutils.core import setup
- from distutils.extension import Extension
- from Cython.Distutils import build_ext
-
- ext_modules = [Extension("sum", ["sum.pyx"])]
-
- setup(
- name = 'sum app',
- cmdclass = {'build_ext': build_ext},
- ext_modules = ext_modules
-)
-
-
- [root@v5254085f259 test]# python setup.py build_ext --inplace
- running build_ext
- cythoning sum.pyx to sum.c
- building 'sum' extension
- gcc -pthread -fno-strict-aliasing -fPIC -g -O2 -DNDEBUG -g -fwrapv -O3
- -Wall -Wstrict-prototypes -fPIC -I/opt/ActivePython-2.7/include/python2.7
- -c sum.c -o build/temp.linux-x86_64-2.7/sum.o
- gcc -pthread -shared build/temp.linux-x86_64-2.7/sum.o

- -o /root/cpython/test/sum.so

编译完成之后可以导入到 python 中使用：

```
[root@v5254085f259 test]# python
ActivePython 2.7.2.5 (ActiveState Software Inc.) based on
Python 2.7.2 (default, Jun 24 2011, 11:24:26)
[GCC 4.0.2 20051125 (Red Hat 4.0.2-8)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyximport; pyximport.install()
>>> import sum
>>> sum.sum(1,3)
```

下面来进行一个简单的性能比较：

清单 9. Cython 测试代码

```
from time import time
def test(int n):
        cdef int a =0
        cdef int i
        for i in xrange(n):
                a+= i
        return a

t = time()
test(10000000)
print "total run time:"
print time()-t
```

测试结果：

```
[GCC 4.0.2 20051125 (Red Hat 4.0.2-8)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyximport; pyximport.install()
>>> import ctest
total run time:
0.00714015960693
```

清单 10. Python 测试代码

```
from time import time
def test(n):
        a =0;
        for i in xrange(n):
                a+= i
        return a

t = time()
test(10000000)
print "total run time:"
print time()-t

[root@v5254085f259 test]# python test.py
total run time:
0.971596002579
```

从上述对比可以看到使用 Cython 的速度提高了将近 100 多倍。

总结

本文初步探讨了 python 常见的性能优化技巧以及如何借助工具来定位和分析程序的性能瓶颈，并提供了相关可以进行性能优化的工具或语言，希望能够更相关人员一些参考。

原文链接：http://www.ibm.com/developerworks/cn/linux/l-cn-python-optim/

CoreOS 实战：CoreOS 及管理工具介绍

作者：桂阳

1. 概述

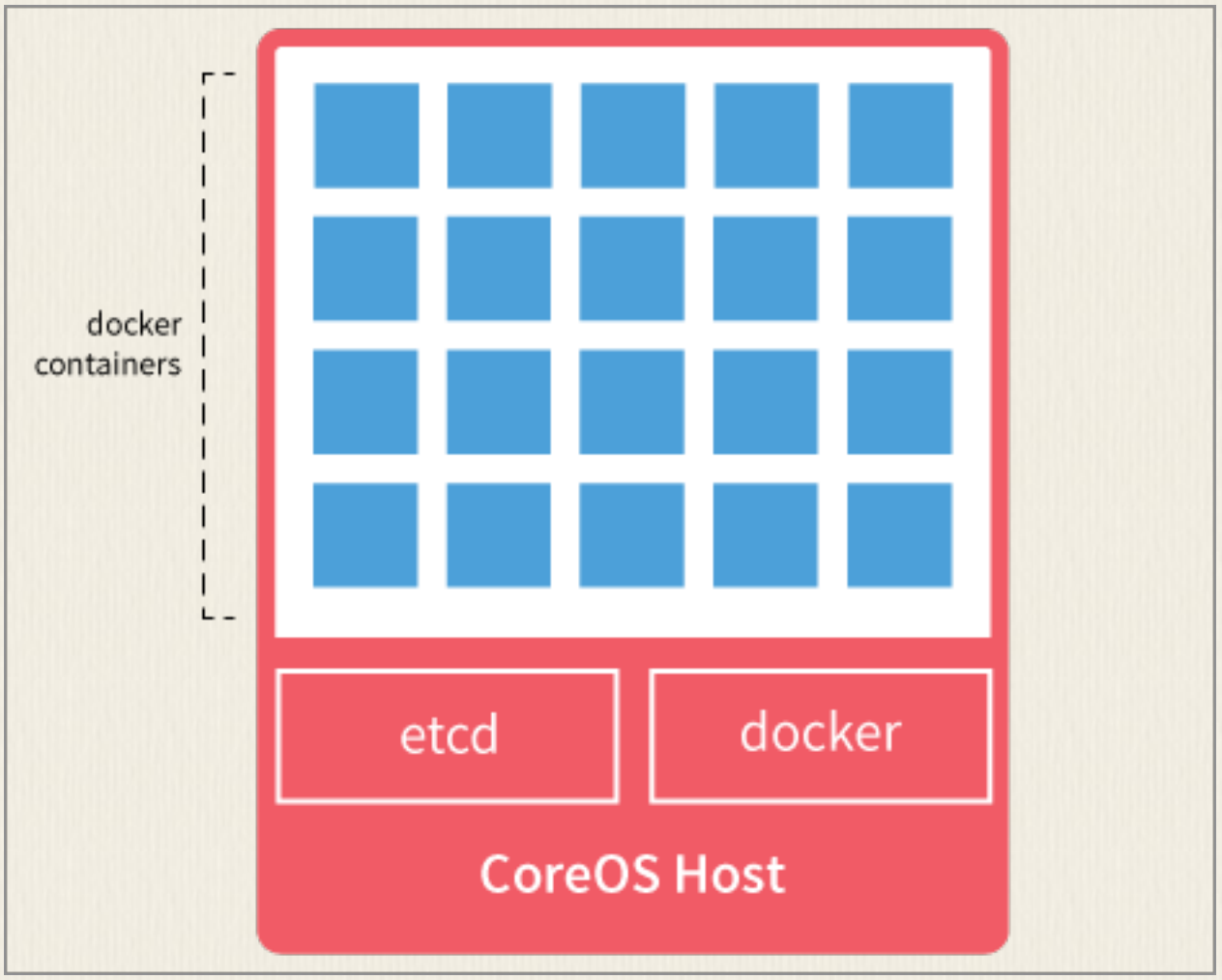
随着 Docker 的走红，CoreOS 作为一个基于 Docker 的轻量级容器化 Linux 发行版日益得到大家的重视，目前所有的主流云服务商都提供了对 CoreOS 的支持。CoreOS 是新时代下的Linux 发行版，它有哪些独特的魅力了？本篇作为《CoreOS实战》的第一部分，将向大家简要介绍 CoreOS 以及CoreOS 相关的管理工具，试图向您揭开CoreOS背后神秘的面纱。

2. CoreOS之禅

云计算新星 Docker 正在以火箭般的速度发展，与它相关的生态圈也渐入佳境，CoreOS 就是其中之一。CoreOS 是一个全新的、面向数据中心设计的 Linux 操作系统，在2014年7月发布了首个稳定版本，目前已经完成了800万美元的A轮融资。 CoreOS 专门针对大型数据中心而设计，旨在以轻量的系统架构和灵活的应用程序部署能力简化数据中心的维护成本和复杂度。现在CoreOS 已经推出了付费产品。通过付费，用户可以使用可视化工具管理自己的 CoreOS 集群。

与其他历史悠久、使用广泛的 Linux 操作系统相比，CoreOS 拥有下面几个优点。

首先，CoreOS 没有提供包管理工具，而是通过容器化 (containerized) 的运算环境向应用程序提供运算资源。应用程序之间共享系统内核和资源，但是彼此之间又互不可见。这样就意味着应用程序将不会再被直接安装到操作系统中，而 是通过 Docker 运行在容器中。这种方式使得操作系统、应用程序及运行环境之间的耦合度大大降低。相对于传统的部署方式而言，在 CoreOS 集群中部署应用程序更加灵活便捷，应用程序运行环境之间的干扰更少，而且操作系统自身的维护也更加容易。



其次， CoreOS 采用双系统分区 (dual root partition) 设计。两个分区分别被设置成主动模式和被动模式并在系统运行期间各司其职。主动分区负责系统运行，被动分区负责系统升级。一旦新版本的操作系统被发布，一 个完整的系统文件将被下载至被动分区，并在系统下一次重启时从新版本分区启动，原来的被动分区将切换为主动分区，而之前的主动分区则被切换为被动分区，两 个分区扮演的角色将相互对调。同时在系统运行期间系统分区被设置成只读状态，这样也确保了 CoreOS 的安全性。CoreOS 的升级过程在默认条件下将自动完成，并且通过 cgroup 对升级过程中使用到的网络和磁盘资源进行限制，将系统升级所带来的影响降至最低。

另外，CoreOS 使用 Systemd 取代 SysV 作为系统和服务的管理工具。与 SysV 相比，Systemd 不但可以更好的追踪系统进程，而且也具备优秀的并行化处理能力，加之按需启动等特点，并结合 Docker 的快速启动能力，在 CoreOS 集群中大规模部署 Docker Containers 与使用其他操作系统相比在性能上的优势将更加明显。Systemd 的另一个特点是引入了 “target” 的概念，每个 target 应用于一个特定的服务，并且可以通过继承一个已有

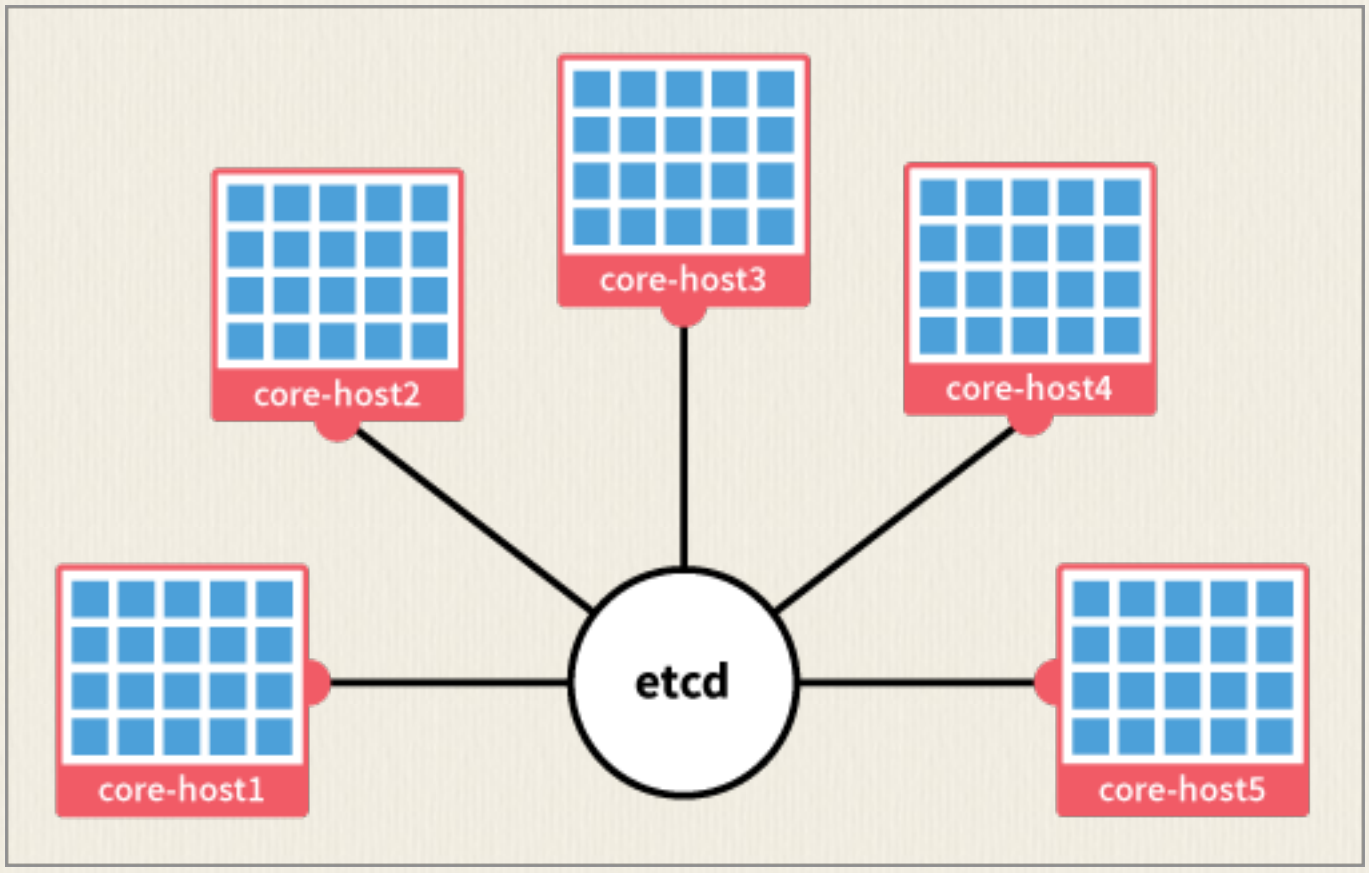
的 target 扩展额外的功能，这样使得操作系统对系统上运行的服务拥有更好的控制力。

通过对系统结构的重新设计，CoreOS 剔除了任何不必要的软件和服务。在一定程度上减轻了维护一个服务器集群的复杂度，帮助用户从繁琐的系统及软件维护工作中解脱出来。虽然CoreOS 最初源自于Google ChromeOS，但是从一开始就决定了 CoreOS 更加适合应用于一个集群环境而不是一个传统的服务器操作系统。

3. CoreOS相关工具

除了操作系统之外，CoreOS 团队和其他团队还提供了若干工具帮助用户管理 CoreOS 集群以及部署 Docker containers。

3.1. etcd



在CoreOS 集群中处于骨架地位的是 etcd。 etcd 是一个分布式 key/value 存储服务，CoreOS 集群中的程序和服务可以通过 etcd 共享信息或做服务发现 。etcd 基于非常著名的 raft 一致性算法：通过选举形式在服务器

之中选举 Lead 来同步数据，并以此确保集群之内信息始终一致和可用。etcd 以默认的形式安装于每个 CoreOS 系统之中。在默认的配置下，etcd 使用系统中的两个端口：4001和7001，其中4001提供给外部应用程序以HTTP+Json的形式读写数据，而7001则用作在每个 etcd 之间进行数据同步。用户更可以通过配置 CA Cert让 etcd 以 HTTPS 的方式读写及同步数据，进一步确保数据信息的安全性。

3.2. fleet

fleet 是一个通过 Systemd对CoreOS 集群中进行控制和管理的工具。fleet 与 Systemd 之间通过 D-Bus API 进行交互，每个 fleet agent 之间通过 etcd 服务来注册和同步数据。fleet 提供的功能非常丰富，包括查看集群中服务器的状态、启动或终止 Docker container、读取日志内容等。更为重要的是 fleet 可以确保集群中的服务一直处于可用状态。当出现某个通过 fleet 创建的服务在集群中不可用时，如由于某台主机因为硬件或网络故障从集群中脱离时，原本运行在这台服务器中的一系列服务将通过fleet 被重新分配到其他可用服务器中。虽然当前 fleet 还处于非常早期的状态，但是其管理 CoreOS 集群的能力是非常有效的，并且仍然有很大的扩展空间，目前已提供简单的 API 接口供用户集成。

3.3. Kubernetes

Kuberenetes 是由 Google 开源的一个适用于集群的 Docker containers 管理工具。用户可以将一组 containers 以 “POD” 形式通过 Kubernetes 部署到集群之中。与 fleet 更加侧重 CoreOS 集群的管理不同，Kubernetes 生来就是一个 Containers 的管理工具。Kubernetes 以 “POD” 为单位管理一系列彼此联系的 Containers，这些 Containers 被部署在同一台物理主机中、拥有同样地网络地址并共享存储配额。

3.4. flannel (rudder)

flannel (rudder) 是 CoreOS 团队针对 Kubernetes 设计的一个覆盖网络 (overlay network) 工具，其目的在于帮助每一个使用 Kuberentes 的 CoreOS 主机拥有一个完整的子网。Kubernetes 会为每一个 POD 分配一个独立的 IP 地址，这样便于同一个 POD 中的 Containers 彼此连接，而之前的 CoreOS 并不具备这种能力。为了解决这一问题，flannel 通过在集群中创建一个覆盖网格网络 (overlay mesh network) 为主机设定一个子网。

原文链接：http://www.infoq.com/cn/articles/what-is-coreos?utm_source=tuicool

图解SSL/TLS协议

作者：阮一峰

本周，CloudFlare宣布，开始提供Keyless服务，即你把网站放到它们的CDN上，不用提供自己的私钥，也能使用SSL加密链接。

我看了CloudFlare的说明，突然意识到这是绝好的例子，可以用来说明SSL/TLS协议的运行机制。它配有插图，很容易看懂。

下面，我就用这些图片作为例子，配合我半年前写的《SSL/TLS协议运行机制的概述》，来解释SSL协议。

一、SSL协议的握手过程

开始加密通信之前，客户端和服务器首先必须建立连接和交换参数，这个过程叫做握手（handshake）。

假定客户端叫做爱丽丝，服务器叫做鲍勃，整个握手过程可以用下图说明。

握手阶段分成五步。

第一步，爱丽丝给出协议版本号、一个客户端生成的随机数（Client random），以及客户端支持的加密方法。

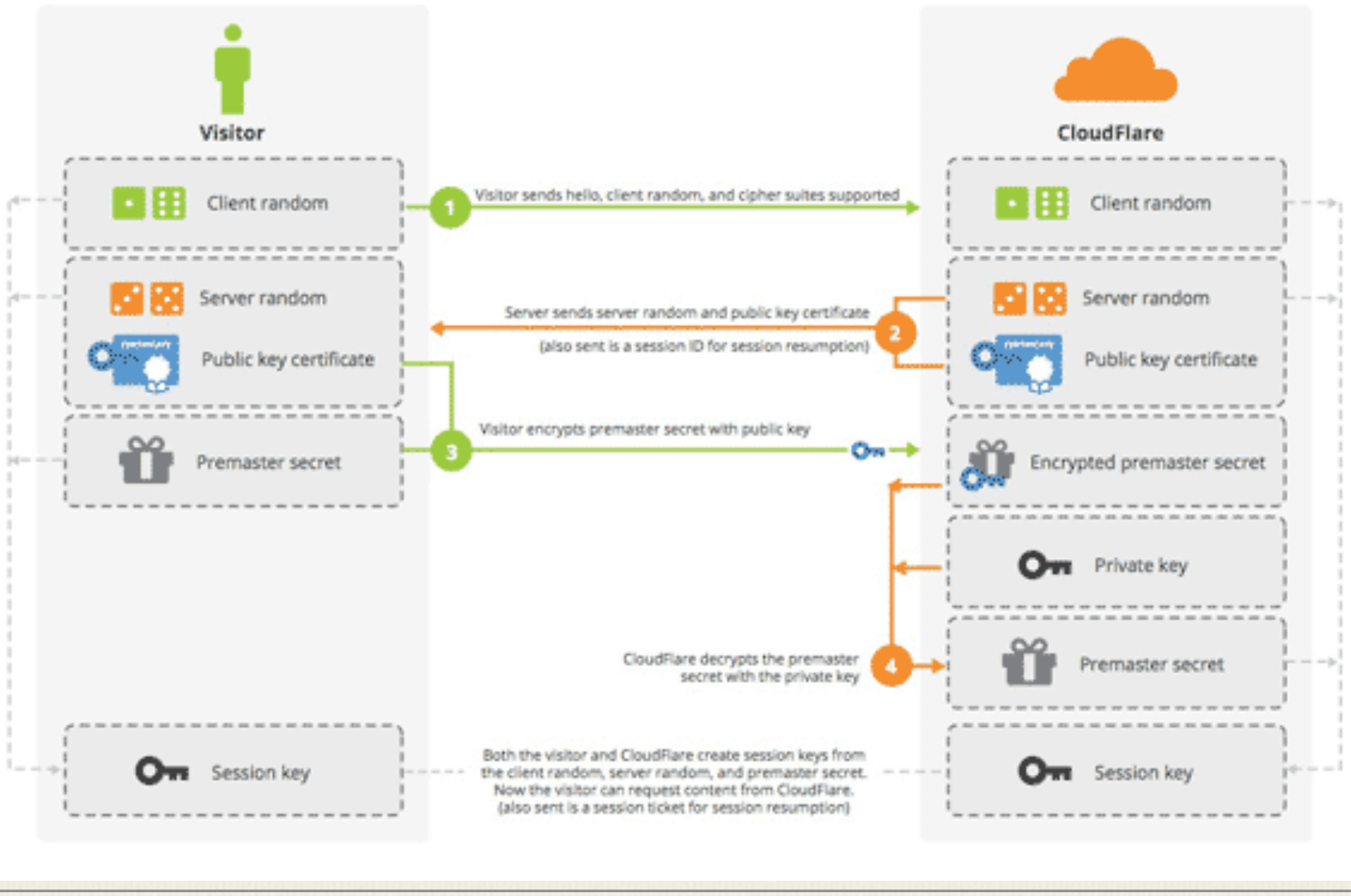
第二步，鲍勃确认双方使用的加密方法，并给出数字证书、以及一个服务器生成的随机数（Server random）。

第三步，爱丽丝确认数字证书有效，然后生成一个新的随机数（Premaster secret），并使用数字证书中的公钥，加密这个随机数，发给鲍勃。

第四步，鲍勃使用自己的私钥，获取爱丽丝发来的随机数（即Premaster secret）。

第五步，爱丽丝和鲍勃根据约定的加密方法，使用前面的三个随机数，生成"对话密钥"（session key），用来加密接下来的整个对话过程。

上面的五步，画成一张图，就是下面这样。



二、私钥的作用

握手阶段有三点需要注意。

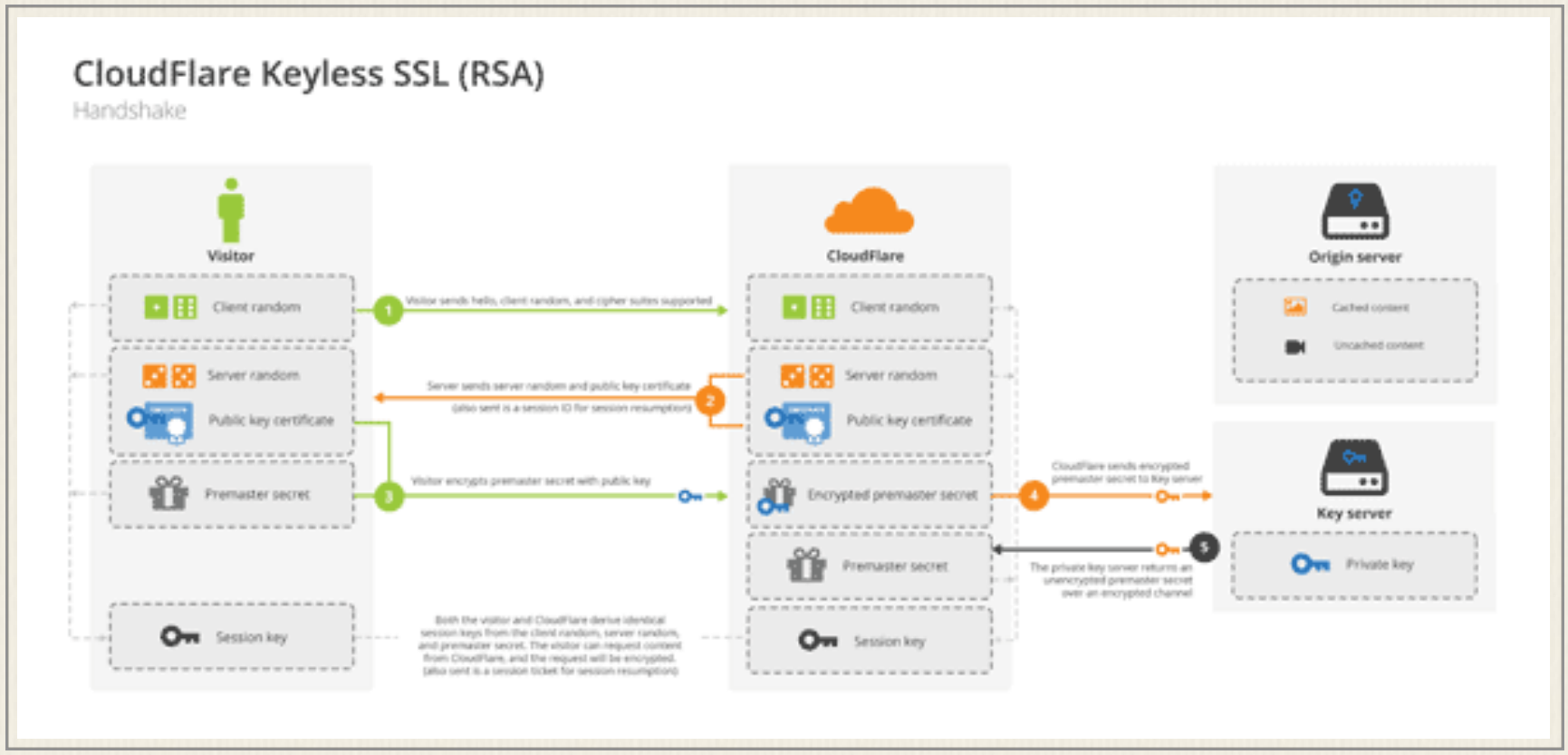
（1）生成对话密钥一共需要三个随机数。

（2）握手之后的对话使用"对话密钥"加密（对称加密），服务器的公钥和私钥只用于加密和解密"对话密钥"（非对称加密），无其他作用。

（3）服务器公钥放在服务器的数字证书之中。

从上面第二点可知，整个对话过程中（握手阶段和其后的对话），服务器的公钥和私钥只需要用到一次。这就是CloudFlare能够提供Keyless服务的根本原因。

某些客户（比如银行）想要使用外部CDN，加快自家网站的访问速度，但是出于安全考虑，不能把私钥交给CDN服务商。这时，完全可以把私钥留在自家服务器，只用来解密对话密钥，其他步骤都让CDN服务商去完成。



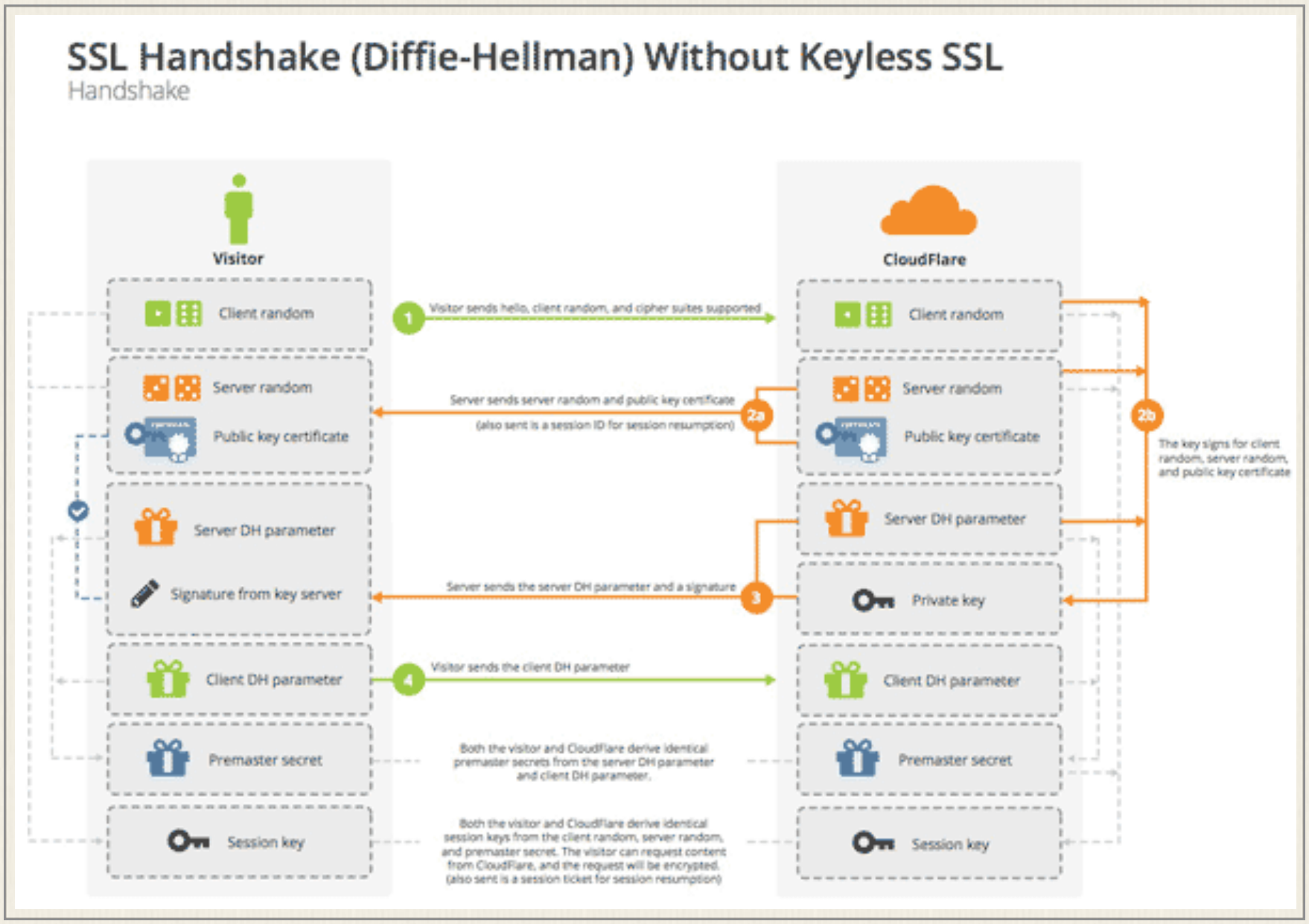
上图中，银行的服务器只参与第四步，后面的对话都不再会用到私钥了。

三、DH算法的握手阶段

整个握手阶段都不加密（也没法加密），都是明文的。因此，如果有人窃听通信，他可以知道双方选择的加密方法，以及三个随机数中的两个。整个通话的安全，只取决于第三个随机数（Premaster secret ）能不能被破解。

虽然理论上，只要服务器的公钥足够长（比如2048位），那么Premaster secret可以保证不被破解。但是为了足够安全，我们可以考虑把握手阶段的算法从默认的RSA算法，改为 Diffie-Hellman算法（简称DH算法）。

采用DH算法后，Premaster secret 不需要传递，双方只要交换各自的参数，就可以算出这个随机数。



上图中，第三步和第四步由传递Premaster secret 变成了传递DH算法所需的参数，然后双方各自算出Premaster secret。这样就提高了安全性。

四、session的恢复

握手阶段用来建立SSL连接。如果出于某种原因，对话中断，就需要重新握手。

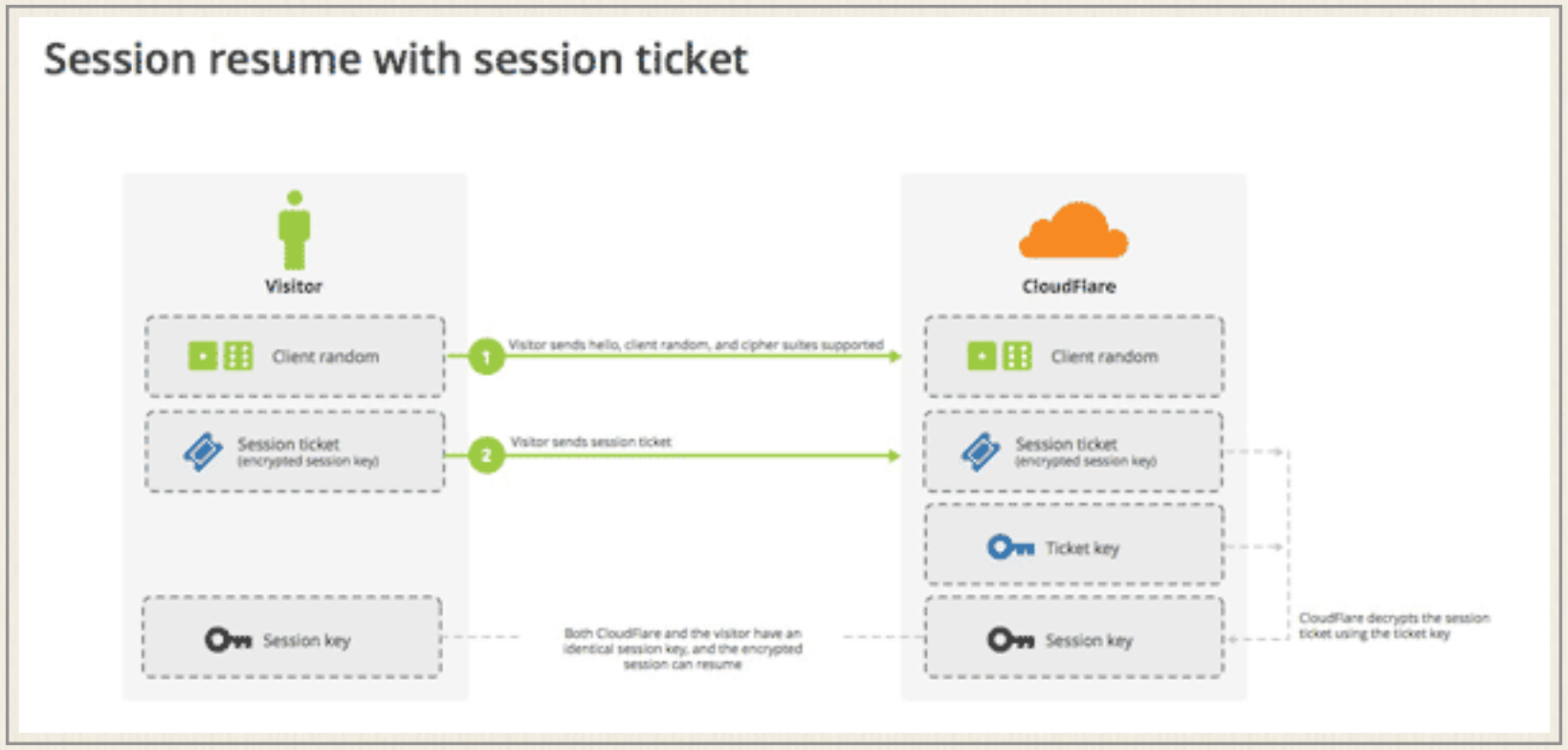
这时有两种方法可以恢复原来的session：一种叫做session ID，另一种叫做session ticket。

session ID的思想很简单，就是每一次对话都有一个编号（session ID）。如果对话中断，下次重连的时候，只要客户端给出这个编号，且服务器有这个编号的记录，双方就可以重新使用已有的"对话密钥"，而不必重新生成一把。

上图中，客户端给出session ID，服务器确认该编号存在，双方就不再进行握手阶段剩余的步骤，而直接用已有的对话密钥进行加密通信。

session ID是目前所有浏览器都支持的方法，但是它的缺点在于session ID往往只保留在一台服务器上。所以，如果客户端的请求发到另一台服务器，就无法恢复对话。session tick-
et就是为了解决这个问题而诞生的，目前只有Firefox 和Chrome浏览器支持。

下图中，客户端不再发送session ID，而是发送一个服务器在上一次对话中发送过来的session ticket。这个session ticket是加密的，只有服务器才能解密，其中包括本次对话的主要信息，比如对话密钥和加密方法。当服务器收到session ticket以后，解密后就不必重新生成对话密钥了。



原文链接：http://www.ruanyifeng.com/blog/2014/09/illustration-ssl.html?utm_source=tuicool

GC对吞吐量的影响

作者：deepinmind

在看内存管理术语表的时候偶然发现了”Pig in the Python（注：有点像中文里的贪心不足蛇吞象）”的定义，于是便有了这篇文章。表面上看，这个术语说的是GC不停地将大对象从一个分代提升到另一个分代的情景。这么做就好比巨蟒整个吞食掉它的猎物，以至于它在消化的时候都没办法移动了。

在接下来的这24个小时里我的头脑中充斥着这个令人窒息的巨蟒的画面，挥之不去。正如精神病医生所说的，消除恐惧最好的方法就是说出来。于是便有了这篇文章。不过接下的故事我们要讲的不是蟒蛇，而是GC的调优。我对天发誓。

大家都知道GC暂停很容易造成性能瓶颈。现代JVM在发布的时候都自带了高级的垃圾回收器，不过从我的使用经验来看，要找出某个应用最优的配置真 是难上加难。手动调优或许仍有一线希望，但是你得了解GC算法的确切机制才行。关于这点，本文倒是会对你有所帮助，下面我会通过一个例子来讲解JVM配置 的一个小的改动是如何影响到你的应用程序的吞吐量的。

示例

我们用来演示GC对吞吐量产生影响的应用只是一个简单的程序。它包含两个线程:

PigEater - 它会模仿巨蟒不停吞食大肥猪的过程。代码是通过往java.util.List中添加 32MB字节来实现这点的，每次吞食完后会睡眠100ms。

PigDigester - 它模拟异步消化的过程。实现消化的代码只是将猪的列表置为空。由于这是个很累的过程，因此每次清除完引用后这个线程都会睡眠2000ms。

两个线程都会在一个while循环中运行，不停地吃了消化直到蛇吃饱为止。这大概得吃掉5000头猪。

Java代码

```
1.  package eu.plumbr.demo;
2.
3.  public class PigInThePython {
4.    static volatile List pigs = new ArrayList();
5.    static volatile int pigsEaten = 0;
6.    static final int ENOUGH_PIGS = 5000;
7.
8.  public static void main(String[] args) throws InterruptedException {
9.      new PigEater().start();
10.     new PigDigester().start();
11.   }
12.
13.   static class PigEater extends Thread {
14.
15.     @Override
16.     public void run() {
17.       while (true) {
18.         pigs.add(new byte[32 * 1024 * 1024]); //32MB per pig
19.         if (pigsEaten > ENOUGH_PIGS) return;
```

```
20.          takeANap(100);
21.        }
22.      }
23.    }
24.
25.    static class PigDigester extends Thread {
26.      @Override
27.      public void run() {
28.        long start = System.currentTimeMillis();
29.
30.        while (true) {
31.          takeANap(2000);
32.          pigsEaten+=pigs.size();
33.          pigs = new ArrayList();
34.          if (pigsEaten > ENOUGH_PIGS)  {
35.
     System.out.format("Digested %d pigs in %d ms.%n",pigsEaten, Sys
tem.currentTimeMillis()-start);
36.            return;
37.          }
38.        }
39.      }
40.    }
41.
42.    static void takeANap(int ms) {
```

```
43.      try {
44.        Thread.sleep(ms);
45.      } catch (Exception e) {
46.        e.printStackTrace();
47.      }
48.    }
49.  }
```

现在我们将这个系统的吞吐量定义为“每秒可以消化的猪的头数”。考虑到每100ms就会有猪被塞到这条蟒蛇里，我们可以看到这个系统理论上的最大吞吐量可以达到10头/秒。

GC配置示例

我们来看下使用两个不同的配置系统的表现分别是什么样的。不管是哪个配置，应用都运行在一台拥有双核，8GB内存的Mac（OS X10.9.3）上。

第一个配置：

4G的堆（-Xms4g -Xmx4g）

使用CMS来清理年老代（-XX:+UseConcMarkSweepGC）使用并行回收器清理新生代（-XX:+UseParNewGC）

将堆的12.5%（-Xmn512m）分配给新生代，并将Eden区和Survivor区的大小限制为一样的。

第二个配置则略有不同：

2G的堆（-Xms2g -Xms2g）

新生代和年老代都使用Parellel GC(-XX:+UseParallelGC)

将堆的75%分配给新生代（-Xmn 1536m）

现在是该下注的时候了，哪个配置的表现会更好一些(就是每秒能吃多少猪，还记得吧)？那些把筹码放到第一个配置上的家伙，你们一定会失望的。结果正好相反：

第一个配置（大堆，大的年老代，CMS GC）每秒能吞食8.2头猪

第二个配置（小堆，大的新生代，Parellel GC）每秒可以吞食9.2头猪

现在我们来客观地看待一下这个结果。分配的资源少了2倍但吞吐量提升了12%。这和常识正好相反，因此有必要进一步分析下到底发生了什么。

分析GC的结果

原因其实并不复杂，你只要仔细看一下运行测试的时候GC在干什么就能发现答案了。这个你可以自己选择要使用的工具。在jstat的帮助下我发现了背后的秘密，命令大概是这样的：

<quote>

jstat -gc -t -h20 PID 1s

</quote>

通过分析数据，我注意到配置1经历了1129次GC周期（YGCT_FGCT），总共花了63.723秒：

Java代码

```
1.  Timestamp         S0C    S1C    S0U    S1U      EC       EU        OC          OU       P
    C     PU    YGC     YGCT    FGC    FGCT     GCT
2.  594.0 174720.0 174720.0 163844.1  0.0   174848.0 131074.1 3670016.0  2621693.5  21248.0
     2580.9   1006   63.182  116 0.236   63.419
3.  595.0 174720.0 174720.0 163842.1  0.0   174848.0 65538.0  3670016.0  3047677.9  21248.0
     2580.9   1008   63.310  117 0.236   63.546
4.  596.1 174720.0 174720.0 98308.0 163842.1 174848.0 163844.2 3670016.0   491772.9  21248.
    0 2580.9   1010   63.354  118 0.240   63.595
5.  597.0 174720.0 174720.0  0.0   163840.1 174848.0 131074.1 3670016.0   688380.1  21248.0
     2580.9   1011   63.482  118 0.240   63.723
```

第二个配置一共暂停了168次（YGCT+FGCT），只花了11.409秒。

```
1.  Timestamp         S0C    S1C    S0U    S1U      EC       EU        OC          OU       P
    C     PU    YGC     YGCT    FGC    FGCT     GCT
2.  539.3 164352.0 164352.0  0.0    0.0   1211904.0 98306.0   524288.0   164352.2  21504.0
    2579.2 27    2.969  141 8.441   11.409
3.  540.3 164352.0 164352.0  0.0    0.0   1211904.0 425986.2  524288.0   164352.2  21504.0
    2579.2 27    2.969  141 8.441   11.409
4.  541.4 164352.0 164352.0  0.0    0.0   1211904.0 720900.4  524288.0   164352.2  21504.0
    2579.2 27    2.969  141 8.441   11.409
5.  542.3 164352.0 164352.0  0.0 0.0   1211904.0 1015812.6  524288.0   164352.2  21504.0 25
    79.2 27 2.969  141 8.441   11.409
```

考虑到两种情况下的工作量是等同的，因此——在这个吃猪的实验中当GC没有发现长期存活的对象时，它能更快地清理掉垃圾对象。而采用第一个配置的话，GC运行的频率大概会是6到7倍之多，而总的暂停时间则是5至6倍。

说这个故事有两个目的。第一个也是最主要的一个，我希望把这条抽风的蟒蛇赶紧从我的脑海里赶出去。另一个更明显的收获就是——GC调优是个很需要 技巧的经验活，它需要你对底层的这些概念了如指掌。尽管本文中用到的这个只是很平常的一个应用，但选择的不同结果也会对你的吞吐量和容量规划产生很大的影 响。在现实生活中的应用里面，这里的区别则会更为巨大。因此，就看你如何抉择了，你可以去掌握这些概念，或者，只关注你日常的工作就好了，让Plumbr 来找出你所需要的最合适的GC配置吧。

原文链接：http://deepinmind.iteye.com/blog/2118983

余锋（褚霸）：RDS数据通道的挑战和实践

作者：阿里云技术团队

大家下午好，今天很高兴在这里跟大家分享一下RDS数据通道的挑战和实践。RDS大家都知道，是云产品三架马车之一。数据服务一般在整个生态链最底 层的。RDS就是数据通道的一个资料，会直接影响到整个生态链的稳定性。RDS我们做了三年，在里面其实踩过很多坑，一开始我们用户规模也是比较小，那时 候其实很多的选择或者做出的判断，现在想想也是比较幼稚的。今天下面的时间我给大家讲讲我们过去踩过的坑，还有后面我们面临的挑战，以及我们如何解决这些 挑战的一些经验。

功能定位

我们整个数据通道的用途和功能定位很明确。因为RDS是一个数据库的集合，我们现在支持MySQL和MSSQL，我们后面有其他数据库的加入。这些 东西包括不同的数据库版本，对用户来讲我们不希望这些变化对用户有很大的一个干扰，所以我们第一个比较注重数据通道屏蔽后端的变化。第二个数据通道一定高 可用。第三我们做公有云服务的，整个数据通道必须是安全的。因为所有用户不一定是安全专家，他们不一定有安全意识，我们有责任和义务去帮到大家。

第二个整合数据通道。我们希望它能做更多，比如说数据路由。举个例子，我们可以帮助用户区分冷热数据，冷热数据存在不同的介质，这样可以给用户省 钱。还有类似的数据操作，今天我们可以做分库分表，对用户是透明的，或者我们今天用Mysql协议接不同的后端，用户对这个没有感知的，数据通道有这个功 能定位。

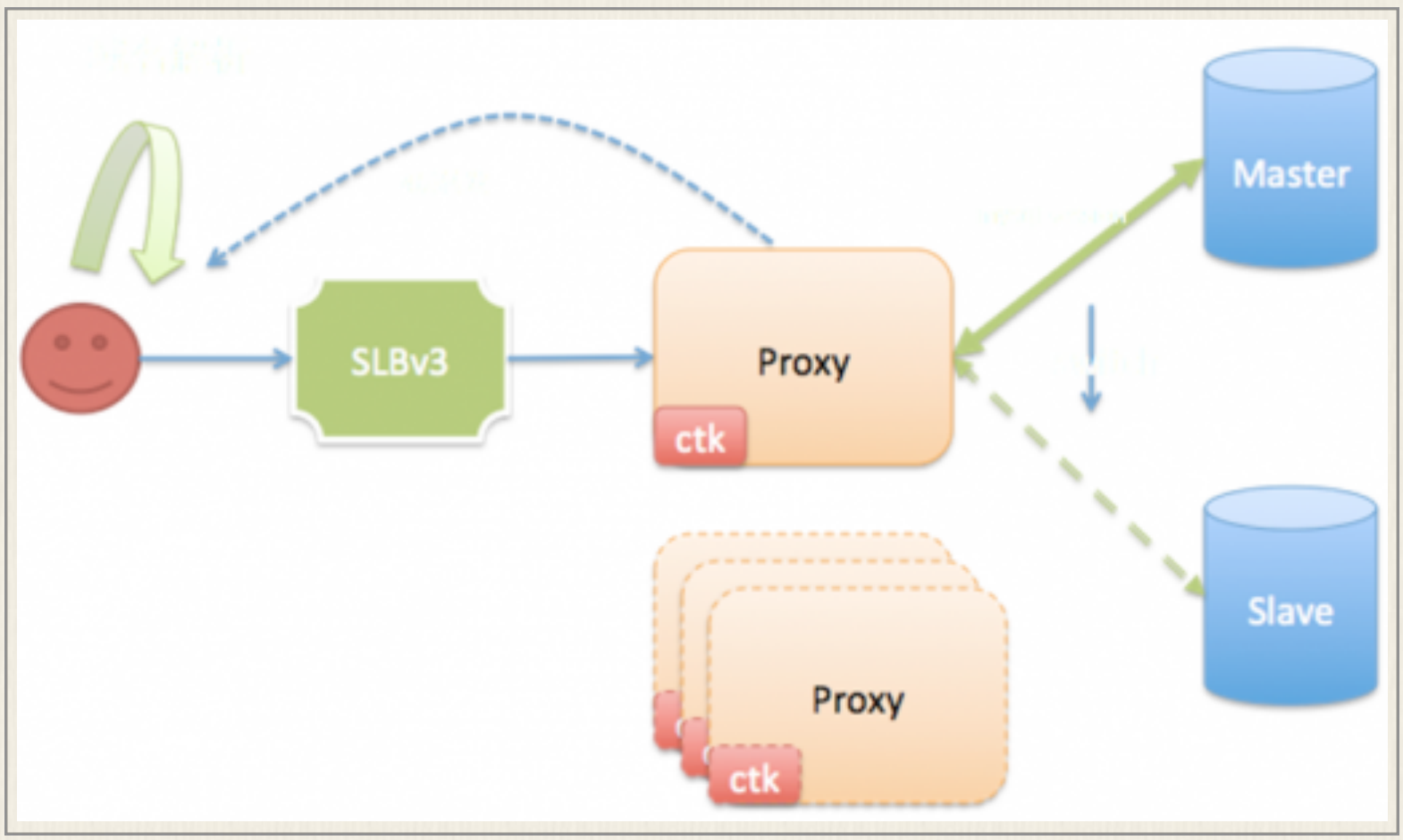
约束

整个数据通道我们可能跑在普通的PC服务器上，跟你们用的PC服务器没有什么区别。

第二个必须高可用，我们知道宕机一秒钟几秒钟对用户意味着什么，有成千上万的用户，每一个成千上万的用户可能服务成千上万他自己的客户，当你一个公有服务不能做到高可用的话，我们知道意义了。

第三个规模。我们规模越来越大的时候我们系统一定是要能够横向拓展。你现在飞机已经启动起来已经飞了，你不可能说等等我把系统调整一下，花一个月时 间或者多少时间把它弄得可以用。我们有时候觉得开着飞机换轮胎，在地上一个轮胎几分钟换好了，空中可能换需要一年，所以说特别的痛苦。RDS整个系统是可 运维的。如果你今天一个系统不能作为可诊断可自省的，问题都是用户帮你曝出来的，我觉得这个系统可能没有生命力。

三层架构



RDS的数据通道是一个非常经典的三层架构，就跟大家平常看到的Web服务器三层架构一模一样的。WEB服务器一个简单的HTTP服务器，为什么有 这么多的中间件，这个是标准的三层架构。今天三层架构为什么这么流行？我不知道大家想过没有，但是这里面肯定有它很重要的一个原因，如果是两层的话你会发 现过于简单，层数过多的话你链条太长，三层是刚好，在灵活性跟简单性中间有一个很重要的平衡。所以说我们整个RDS后续的架构就是基于这样一个三层的架 构。三层架构业界有很多很多的经验我们可以借鉴的，Web服务器前人积攒几十年的经验我们都可以用。我们整个系统都是基于这样一个架构。把域名算上去有四 层的，就是用户如果直接用这个数据库的话，实际上就是这一层，我们加上中间的PROXY加上四层的SLB，前面还有一个域名，每一层的引入对用户都带来非 常大的影响。

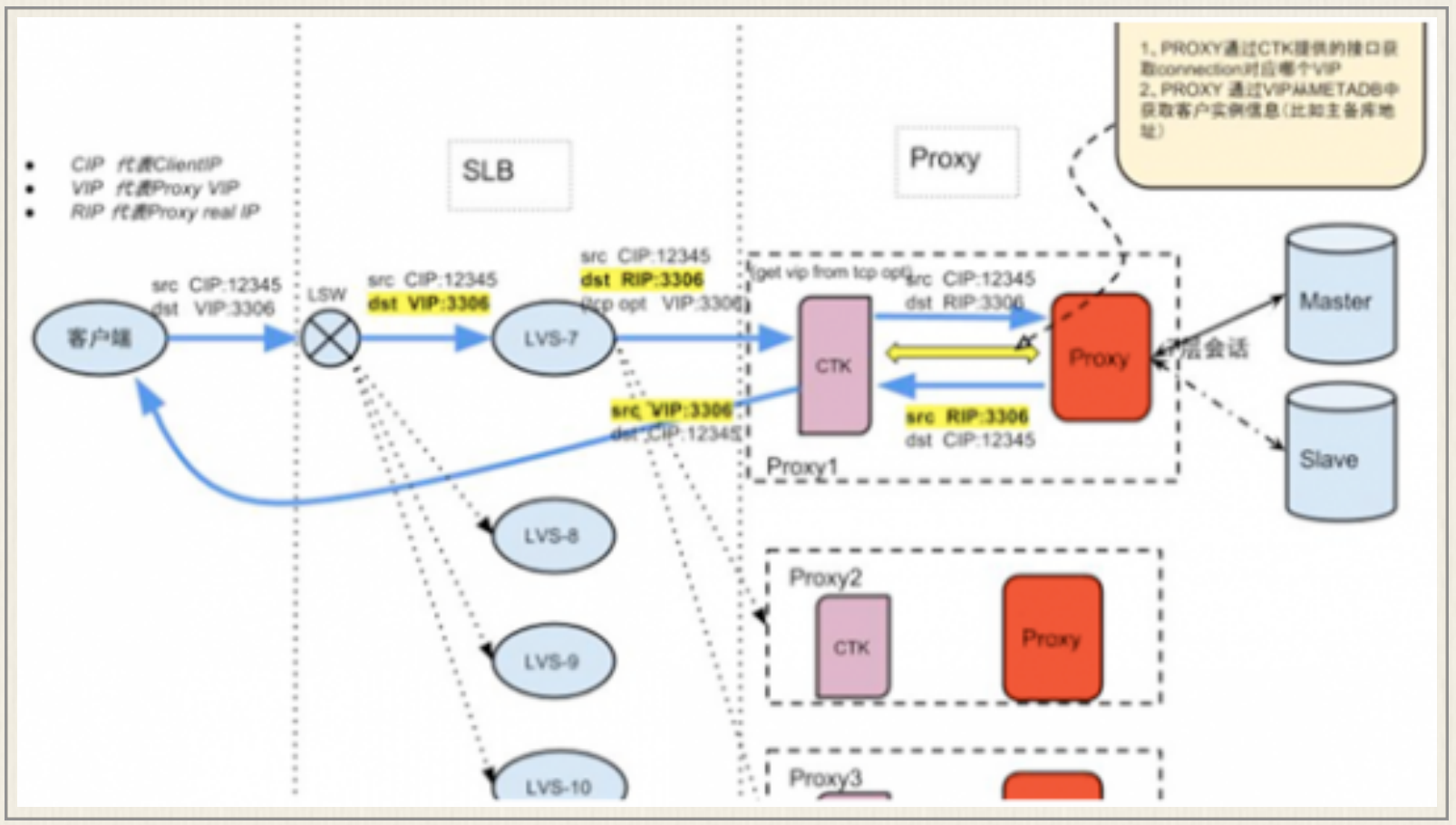
我举个最简单的例子来讲，域名解释大家觉得有问题吗？今天我们数据库给大家提供一个域名，域名可能对应一个VIP，我们提供域名最基本的想法是说换 VIP很容易，但是实际情况不是这样，今天你是公有云的用户，用户对这个系统的支持不尽相同的。很多时候你给他一个域名，他直接要你提供的VIP，还有有 些用户主机上设了域名，它绑定起来很长时间不生效，也就是说即使域名这么简单的事情，我们真的切换域名的时候，你发现95%用户是OK的，5%是切不动 的。第二个它直接用VIP，这两点把你搞死，我们实际上有这层但是没有办法用。我们第一天想这么一个域名切换很简单，今天连域名简单的事情我们不能切我们 用VIP漂移，我们去后端系统怎么变更，我们对用户提供的VIP不变的，极端情况下我们没有办法让它去变。当它用户的规模大，你可以看到各种各样的问题， 这些问题都是挑战。

虽然是三层，看起来特别简单的三层架构，但是对我们整个系统部署不是这样子的，我们前面说过整个的系统目标是可规模的扩展。所以说部署上第一个尽可 能的自治，比如说我们以集团为单位管理用户。客户过来通过VIP路由到一个交换机去，客户请求同交换机到不同的LVS去，任何一台LVS挂掉的话，数据连 接不受影响，哪个LVS到不同的地方也是1比N的关系，Proxy对这个数据也是1比N的关系，大家奇怪为什么弄得这么复杂，我们引入SLB最初的感觉， 因为每台DB的容量有限的，对这么大用户来讲你达到流量的切分，我把流量导到不同的后端。

这里面就是VIP，通过VIP的归属我们可以导到不同机房不同的交换机，通过这个路由我们导入不同的LVS，通过LVS作用这是四层，我可以导入不

同的proxy，通过这个七层导流我们可以导到DB节点上。我们可以把一个大容量通过这几层一级一级分解，分解到不同的地方去。中间加了一层对这个有点影 响，比如说我们用三角模式，用户回数据我们不走LVS，直接走更短的路径，这是我们的优化。

这是我们的数据流图。我们今天一个用户给他一个VIP，比如说一个用户还有不同的业务，每个业务给他一个不同的VIP，一个用户可能有多个VIP， 这是我们过去几年踩下来一个最最重要的经验。如果我们通过VIP把用户区分的话很多事情非常好做，过去三年我学会这个东西，通过一个维的VIP把这些用户 真正区别开，后面东西围绕这个做。



数据流图

可用性之SLB/99999

前面我把这个数据通道的背景跟大家说了一下。接下来我们讲挑战，今天SLB是五个9，发展到今天特别的稳定，除了稳定以外其实整个能达到五个9实际 上要做很多很多事情。比如说刚才一个挑战，每个用户有N个VIP，我们前面提到DNS不能动的，如果动可能有5%的用户切不过去的。比如说一个用户的数据 库，云计算是一个操卖，或者它的资源可以动态的

漂移，今天资源从一个机房飘到另外一个机房，或者从一个机架飘到另外一个机架，可能VIP发生变化，我们最 初设计改一下域名就好了。我们做了很多方案让我们自己后端设备中间如何去屏蔽这些变化，如何做漂移。我可以给大家一个数字，我们最近做一个漂移的动作，我 们大概从半年前开始做到今天没有做完。然后开了无数的会，做了无数周密的计划，能够让用户不受影响。这是我们很大很大的挑战。

第二个就是说今天SLB运行在普通的硬件上，比如说几万块钱的机器上，它过能量不需要多大内存，CPO计算力，但是它的稳定性很重要，从DB节点往 前挂了一个部件对用户影响特别大。所以我们设计上保证你单台宕机没有问题，这里面有单台座机除了我们软件上控制它可以允许你宕机的，但是从硬件质量上我们 选择这种机型，可能不需要那么高豪华配置，只要皮实一点稳定一些，我们从硬件跟软件都要保证。今天我们单台机器宕机的概率很小，即使宕掉了我可以从软件层 面恢复的。

第三个就是说你今天一个挖掘机下去把你光纤给挖断了，是不是存在这个问题。我们做公有云服务你不能停的，所以我们会有双机房，VIP两个机房，这个 机房挖掘机挖断的时候，另外一个机房可以应用。对每个VIP来讲有一个储备的作用，当我们系统发生问题我们切到备机房。开始我们没有考虑这么多， SLB这层我们换过四种架构，这是V3版，今天我们定下来就是这种，也是我们踩过很多坑，最终发现这样子比较好。

可用性之Proxy/9999

另外，我们中间有一个proxy，这层我们要求做到四个9。我们要做分库分表，数据路由等各种业务都在里面，你可以预期它升级非常频繁，今天有软件 升级意味着你有Bak。第一个要求是四个9。怎么样达到这个目标？我们做了一个很重要的判断，我们今天整个的数据通道是由erlang做的，Erlang 可能历史比大部分人的年纪都大，它做了二三十年了，它以前号称四个9，我们最后事实证明这个是对的。我们从来没有发现过Erlang自己宕机，有宕机都是 我们自己资源使用不当导致宕机的。我们已经有三年的时间，我们可以把资源使用有可能超标的问题监控起来，所以我们从来没有发现它宕机。一个空指针可以搞挂 一个进程，我们这个代码大概有五六万

行，六七万行Erlang代码，折成C代码是6比1，有30多万行的C代码，这是很高的稳定性。

首先我们用的是ERLANG这个东西，第二个我们有计划的软件升级。这里面Erlang有非常重要的特性就是支持热升级，热升级的概念就是不影响用户，用户该怎么样怎么样，今天我把软件升级了用户一点不知道。我们做云服务，体验级质量，很多用户需求很快很快的，我们一个月发布两次版本都可能。所以说 有一个热升级的特性非常有帮助， erlang跟其他系统不太一样，erlang系统是我们链接层面，跑两个版本的代码，C有人做了，java有人做了，你只能说这个进程跑同一套的代 码，erlang可以做到每一条TBC链接跑不同的代码，也就是说每一个用户可以让你跑不同的代码，可以做到这个级别的热升级。我们可以把力度做到用户级 别，支持两种，一种只要这个代码改动不算非常非常复杂，我们都可以用热升级。比如说这次升级很大，我们就用暖升级，我们通过切流量的方式，旧用户在旧进程 上，我再写一个新进程，把旧流量导到新进程上面去，旧进程上的用户让它自生自灭这样达到目的，大部分人是这么干的。说暖升是很简单的事情，其实操作起来很 麻烦的，三天以后旧的用户在旧进程上，30%还在操作，7天之后可能剩10%，你发现一个月之后还有2%的钉子户在那里不动，这个VIP连在上面你尽可能 不对他有影响，如果你再想升只有一个可能就是把它杀掉，理论上只有2%的是钉子户，它真的对连接特别敏感。所以这个事情还挺麻烦的。

第二件事情就是我们减少宕机这种状况。我们自己高可用的系统发现这个事情发生了我们把它切走，今天可用性可以达到四个9。你做这个东西不就是Mysql proxy吗？今天这两个系统感觉上不是一个数量级上的。

可用性之DB/9995

再后面是DB，DB我们保证高可用，任何一个软件都有bug的，今天我们用Mysql是甲骨文来的，它全球客户很多，今天我想问你个位数的bug你 修不修，你用户给你报了一个故障你修不修？甲骨文全球这么多的用户每天的故障不太多它不会修的，但是我们会修个位数的，而且极低极低的概率都修，只要是 bug发现的我一定会修。第二个在部署上，master这个结构大家都知道，我们DB部署上尽可能的跨机房跨机架，没有条件我跨机架，有条

件我跨机房。比 如说复制上我们会做主备同步，你是标准版的主备同步很失望，我们做了很多工作主备工作很快，原来没有做这个的时候发现用户主备同步有超过好几个9，就是主 备和备控延迟好几个9，做了这个之后可能在几秒，达到你主备同步达到四个9你所有都没有意义。可行性有了，数据不对了，所以你所有的东西都没有意义。我们 在数据安全和可用这个中间，让我有一个平衡我选数据安全。今天我切主备可能不到一秒，但是我们在那犹豫不决，我们判断切换主备有一箩筐，每次其中一个策略 调整我们一大票人进来吵一下午最后没有结论。数据安全跟可用性之间的平衡是一个很麻烦的事情。

兼容性挑战

另外一个兼容性的挑战，今天可以看到proxy架在DB和用户之间，每个版本甲骨文可以向下兼容，但是有细微的差别，对大部分用户来讲体验不出来这 个差别，但是对我们来讲可以体会到的。客户端版本今天PHD推出一个什么库用甲骨文的新特性，明天java弄一个什么库，后面谁弄一个连接层，这些都在变 不一定符合规范，有的驱动违反协议规程，但是有的就这么干。所以我们要去规避这个事情。

我给大家举两个例子，Mysql认证是挑战认证，用户发一个用户名，发一个密码过来，三字挑战码看看过没过，比如说密码错误，错误码是007，我们 中间做一些安全，IP过滤，我们除了密码错误，可能还有IP符合不符合，我们可能给标准的错误码加1加2，我们为跟标准区分开我们改成1007，这个实际 上是一个灾难。很多的客户端，比如说甲骨文的客户端会防钓鱼，用户名进去，什么密码，只要有用户名就过了，甲骨文会防止这个事情，第一个发一个用户名空密 码过去，这样如果你是防钓鱼一定是OK的，它希望你认证不通过007错误码，实际上我们改成1007了。我们后端跟DB标准TB连的，这个数据是通的，有 的把数据通道建立起来因为走SLB，他会确认一下你是不是真的走SLB，我们后端走DB过去没有这一条，后端就说你造假，类似这样的事情逃不过的。兼容性 我们没有办法第一天把这个事情做到位的，但是有一样东西可以做，就是我们时刻做好兼容准备，我们整个系统是erlang做，兼容性我会后知后觉，我怎么样 发现这个问题，我们做了很多工作。我们实际运行数据来看是亿分之三，对用户来讲这个数

字是感知不到的，我们自己知道。我们一个月可能做两个版本，直接把它修掉。这里面最大挑战就是你有很多未知东西，你怎么样发现这个问题。

公平调度

第二个就是公平性，你是如何保证这是多客户的问题。今天连在你上面用户是各种各样的，有非常守规矩的好公民有特别狠的，有的一秒钟发一万多张图片， 你见过一个QOS上传过来几十G吗？我见过。比如说我们做分库分表，后面是几个T的数据，用户来一个全扫描，他合规吗？他合规。你不能不让用户做这个事 情。所以很重要的事情就是你要保证它的资源合理。今天一个不合规的用户不能把合规的用户剥夺了，就跟我很有钱，我把其他人都给杀了，可以吗？不能。每个人 都有公平生存机会。CPU内存、网络、IO，CPU跟IO、内存强隔离的，每个用户的系统保证是可以公平调度的，基本上原理按时间片来的，对IO的使用， 或者对计算的使用它会折成一个数，然后它跟我们操作系统的调度一样，你这个时间片用完它把你切出去再回来，对于我们程序员是透明的不知道这个事情，但是它 会帮你做这个事情。

这个问题在这里是一个链条，我相信你做一件事情两件事情很重要，但是你把这些链条做起来就贵了，你知道虚拟机有20多万代码，它大概有3万代码在做 如何保证公平的合规性，最新提交了三千多行修正原来的算法，让公平调度更公正一些。所以说我碰到很多云产品，很多的东西公平调度没办法解决。所以它会把很 多转换成其他的形式，让一个用户一个进程，让操作系统隔离这个事情。

另外一个就是刚才Erlang对网络隔离比较粗，我们做的层面更细致。今天我给大家举个例子，我们上次发布压缩一个会导致死循环。死循环造成这个用 户CPU100%，上线第五秒钟你会发现这个问题，但是我们没有回滚没有下线，因为其他用户都OK服务好好的。最极端的情况下相当于一个用户把我们所有的 吃掉，这种事情我们没有回滚，我们把问题查出来以后再回过去。我们系统可以公平服务大家，这是很重要的。

安全

另外一个安全的挑战，数据库前两年发生一个用户名密码验证的漏洞，你多输几遍就可以突破论证体系过去，今天我们是公有云的服务，发生这样的事情真的 是灾难。即使我今天第一时间把bug找到，我bug如何上线。做这个事情一周都不够，您那个bug怎么解决，因为我们有中间层，今天我们在中间层可能几个 小时把所有漏洞堵住。第二个我们有系统审核，有的用户拖库了，我们要帮用户保证这个安全。我们今天是百亿级别的规模，今天是这个数过几天不是这个数了，每 一条sql我们都拿去运算，拿到用户的一条SQL看看这是不是符合规则应该放过去，今天是百亿级别我要花多少CPU算这个事情，它的挑战是多少？刚才前面 讲合规的用户一条sql是几十字节，一两百字节，不合规有几十兆甚至上G的，规则引擎对我们来讲是很大的问题，可能有几个用户过来把你搞死了。

比如说密码帮你破解，这些都是安全这边要做的，今天你不做这个事情也没事，但是你真正有事就死了，你做不来。

性能

另外一个就是性能，今天我们怎么样性能对用户最小，今天对数据库来讲很重要一个指标就是RT。我们用的是Erlang帮我们做这个事情，Erlang第一天设计就是准实时的系统。第一天设计是毫秒准实时，第一天定义在毫秒级别。所以说这个系统帮助我们做很多。

第二个吞吐量，因为整个系统你要想吞吐量尽可能的并发，这又是Erlang帮我们做的。今天我发现很多很多系统都在改造，说我们要异步法，今天纯并行法，最后得出这样一个结论。但是Erlang几年前就已经是这样子了，而且实践的非常好，所以我们搭乘这个快车。

我举个极端的例子，比如说短连接，用户连接它代理的，他直接异步到位，这中间除了代理带来的延迟还有一个网络延迟，原来直接到DB，你现在到SLB 然后到DB，SLB到DB你要用很长时间，有些用户是短连接，对用户来讲不可接受的。今天所有的系统都做了连接，但是有一件事情你克服不了的，今天连接词 呼应，数据库有上下文的，比如说有上下文的变量，这个连接呼应你刚才用的，我如果拿去用的话我会受到污染，你刚才用中文字

符我用的是英文字符，连接符本身 很简单但是怎么做到简单，你要做到透明，跟用户连接DB一样的。我们把数据库那边后端改造，因为我们连接就是一条DB连接好，所以我们会把DB改造，连接 词直接过去让DB把我所有用户清空。

第二个挑战就是用户刚才给我发这条QPS的时候，他刚才什么上下文你要重复过来，第二件事情怎么把用户的变量全部重复一遍，这是两个挑战。这两个挑战把大部分人挡住了，做不了，我们需很长时间做这个事情，有这个上下字呼应我们可以三层架构的使用，甚至比两层架构更快。

桥接

我们数据库断了重连一下就好了吗？但是游戏规则不是这样，跟钱有关系的不能重连，这个逻辑乱掉了，当这个断掉了，会网上推。如果正常你重连一下就好 了，但是他们不是。我们经常人家半夜打游戏可以收钱，我们半夜做变更。闪断伤害这是最大的问题。今天我们可以把闪断这个问题成功桥接到99.7%，前端迁 移、网络闪断对用户影响我们可以控制在99.7%，这个数据非常小。用户从不同交换机，IOS，不同的proxy过去，从不同路径过来的用户都切到同一个 进程，要么不切，所有你的proxy你要让它能够同时切，所以这是一个系统。

第二件事情你怎么判断边界，正常一个事务几毫秒，有一个长事务几十秒，你如何判断它是长事务。用户请求先到我手里，我可以让它放在主或者备，这个是 30秒，桥接时间不能超过这个数，留给我们时间是15秒，15秒你决定这个事情能不能做，做到什么程度，我们失败0.03%，有些数据库不能切，有在同 步。

包括用户session变量重放，这是非常复杂的事情，我们不做这个事情对用户有没有感知，大部分用户感知不大，我们真正后面做迁移也是比较少的，但是你为了用户，相对有一些敏感的用户对他体验好我们做这个事情，这个事情我们大概做了一年，所以是非常痛苦的。

部署

部署，因为是云服务，你资源要能够动态的扩展，扩展要求你要对容量有评估，你知道什么时候扩什么时候下，容量怎么算出来的，服务怎么样快速，比如说 我一分钟把几十个节点加上去了，系统要做的足够完善。第二个降级，今天有可能我遭到攻击了，今天可能有些情况，我在里面大概做了十几的开关，CPU紧张的 时候用户不要压缩了，把压缩功能关掉。关掉以后还不行CPU降不下来，我把流量的关掉，再不行我把桥接开关关掉，最后我可以降到透支处理。我要保证数据通 道是通的。

第三个proxy的分组，今天所有用户从不同的通道过来，今天如果一个机器挂掉影响全面用户，我们做到影响部分用户，就是进行分组。

另外一个就是灰度，我们今天上线一个新版本，我们怎么做？肯定拿一个最小集群把这个部署上去，看没有什么问题，再布第二个第三个，这个很慢，有的用 户没有用到，所以我们要做到用户级别的灰度，我把长链接用户一打包这个流量可以切到新版本上。我们proxy要能够区分这个事情做到灰度。

用户行为洞察挑战

另外用户行为洞察，这个很重要的事情。一个系统能够可运维、可自省非常重要的。所以我们希望了解用户行为。用户行为就是数据采集，数据采集代码占到 系统可能百分之三四十，我们从链接到集群到proxy到用户到压缩，按照树形数据组织。我们一台机器上有很多用户，几万个用户你加一遍已经吐血了，当一个 用户退出来，QPS可以加到proxy级别，用户级别，链接级别，一层一层往下推的。

第二个今天我们用户里面几个人是短连接，几个人长连接，这个影响我今天版本发布对用户影响多少，客户端有什么版本，类似这样的一些业务统计，老板们 最喜欢问这些东西，大部分人回答不出来，现在我可以回答出来。比如说异常行为今天有人暴力破解你知道吗？昨天这个数据库IP是一个毫秒，今天变成三毫秒， 我昨天一毫秒今天三毫秒你帮我查查什么原因，这个容易查，但是你要牛在它变成三毫秒的时候用户没有知道你知道了。第一个你有足够的设施，我们把所有的用户 级别连接级的我们记下来，我们

有三层，所以我们有三条线，从用户角度看到RT，从连接看到RT，从DB看到RT。

有了这些数据以后你才有可能做这种，我们叫做牛式计算，实时的算这些东西，你今天用户级别的RT变化我是马上可以知道的。

诊断

还有一个挑战就是诊断，今天这些数据驱动去做这些事情，你没有这些数据你怎么发现系统存在问题。比如说用户连接慢，我也知道你慢，但是我根本不知道 你慢在哪，什么环节慢？但是我今天可以每一个阶段发哪去都很精确地用图表方式算出来。如果是一个表格你是没有记录不可回溯的，我们是图表，就是说你所有精 确度量的时间，从你一个月前一年前我都知道了，所以我需要一个精确的度量，每天工程师吃饱上班看看昨天图表有什么异常。比如说有一些异常发现，我的CPU 高，我知道你CPU高，用户也知道，但是这个不是目的。我今天报CPU高，真的把CPU高的问题解决掉这才是根本问题，怎么把这个解决掉？我们做一个类似 扁鹊的系统，从CPU硬件系统捕CPU干吗，我们可以做到原码级别。工程师一看这个是怎么回事，这个事情就很好解决了。很多事情你不知道是很恐怖的，你知 道了就很好解决了。

有了时间消耗，实时分析趋势你就有条件自省你的系统，这是很重要的能力。今天我讲座就到这了。希望大家有机会来一起做这个事情。

原文链接：http://blog.aliyun.com/1705

阿里搜索离线技术团队负责人谈Hadoop：阿里离线平台、YARN和iStream

作者：张天雷

Hadoop从互联网诞生，但近些年在整个大数据领域呈现爆发式发展和进化，尤其是在2013年Hadoop 2.0正式Release后，Hadoop有了正式的 Operation System—YARN，从此Hadoop不再只是MapReduce的代名词，Storm、Spark、Graph，MPI等越来越多的计算模型可以运 行在YARN上，批处理计算、实时流式计算、迭代交互计算等都可以同时运行在Hadoop集群上，Hadoop已经成为大数据计算的全能平台。HBase 随着近几年的高速发展和应用，已经成为大数据技术领域最主流的NoSQL数据库；Tez和Spark的出现让Hive拥有了更高效的计算引擎可以选 择；Impala和Stringer更是将大数据SQL带入到了Realtime时代；Ambari的诞生和快速发展也大幅降低了Hadoop集群的运维 门槛。随着Hadoop开源社区不断涌现出各种令人兴奋的新技术，逐步完善的Hadoop生态系统已经成为大数据行业发展的核心动力。

本次QCon上海的Hadoop专题出品人王峰（莫问）接受了InfoQ邮件采访，谈到自己在阿里的工作，YARN的优势以及Stream和Spark等平台的比较。

InfoQ：为什么会做这次QCon上海“Hadoop，超越MapReduce ”的出品人?

王峰：我在阿里的8年中一直从事搜索和分布式技术研发，自2010年开始基于Hadoop生态技术构建阿里的搜索离线技术平台，统一支持淘宝、天猫、1688、一淘和云搜索等多条搜索业务线的后台数据处理，亲自带领团队经历了Hadoop从1.0到2.0的平台演化之路，本次受我们阿里的朱鸿老师邀请，有幸成为“Hadoop，超越MapReduce”的出品人。

InfoQ：您一直负责为阿里集团服务业务提供平台数据支持，请给大家简要介绍一下整体情况?

王峰：我负责的阿里搜索离线技术团队，为阿里集团的搜索业务提供统一的离线基础数据平台支持，目前我们基于YARN构建了统一的计算平台，支持批处 理、实时流式等多种计算模型支持；基于HBase构建了统一存储平台，支持KV，SQL，Queue等多种存储模型，计算+存储共享集群资源，同一套基础 架构同时支持淘宝、天猫、1688、一淘和云搜索等多条搜索业务线，为阿里的搜索引擎提供实时、增量、全量的全数据支持。

InfoQ：2013年阿里搜索全面升级YARN，比起之前来讲有什么优势呢？

王峰：第一次接触YARN是在2011年低在美国参加Hadoop World，随即造访了Hortonworks，更加详细深入的理解了YARN的设计思路，感觉这个东西如果成熟了，就是Hadoop OS，Hadoop的计算能力将产生飞越。但如果只在YARN上单纯运行MapReduce，其价值将不会有质的变化，最大的好处也就是把集群规模可以做 的更大了，这个意义就大打折扣了。升级到YARN的最终目标应该是让计算模型更加丰富，并产出统一的计算平台，降低维护成本，更大程度的扩大集群资源利用 率，发挥云计算的效果。我们阿里搜索的Hadoop升级到YARN以后，不仅运行了传统的MapReduce、Hive，还自主研发了iStream（流 式计算引擎）、iCall（基于Thrift的分布式RPC服务），后续还计划尝试Tez，Spark等新式计算模型，统一的计算平台相比之前的 MapReduce Job，无论是效率，成本还是对业务支持的灵活性都实现了质的飞跃。

InfoQ：我们还看到淘宝自主研发了iStream流式计算引擎，这方面的工作也想请您简要介绍一下。

王峰：其实我们当初最早是尝试storm，但storm最大的问题是无法和hadoop集群复用，单独存在的storm集群让我们运维成本增加，同 时资源利用率也上不去，出现各种问题也无法根本解决，YARN的出现让我们有了新的思路。iStream天然是基于YARN来设计的，因此其在设计理念上 最大的亮点就是考虑了如何和其他计算模型共存，达到实时计算效果的同时，还可以实现计算平台的全局最优化，例如：iStream可以自动感知流处理的进度 快慢，智能调整计算节点的数量，即高峰期可以自动扩容节点保证处理速度，低峰期也可以在保证进度的条件下合理释放节点，让资源在多计算模型场景下真正按需 分配。现在阿里搜索的hadoop集群上，iStream承担了流式数据处理的角色，为搜索引擎提供实时增量数据，MapReduce承担了

全量或者批量 数据处理的角色，为搜索引擎提供全量数据，两种计算模型可以自动合理的配合，无需人工运维干预。

InfoQ：Spark平台目前挺火的，您在这方面是否有所涉及?

王峰：Spark目前可以算是最火的计算模型，不过我们还没有将Spark投入生产，原因不是我们不认可Spark，而是Spark强在迭代计算和 实时SQL，这块在搜索主流程中的场景不是特别明显。简单来说，Spark Streaming在我们这里有了iStream，实时性和资源管理更加专业；Spark SQL在我们这里有了Phoenix（SQL On HBase），因为我们的数据基本都在HBase，基础的SQL场景，我们用Phoenix可以轻量级的解决了；迭代运算都是算法训练的纯离线过程，都在 阿里的云梯和ODPS上运行了。

InfoQ：作为行业翘楚，您对Hadoop的认识非常深刻，不知您有没有比较好的图书、社区推荐给广大读者?

王峰：其实我个人阅读的Hadoop相关的图书并不多，除了几本英文经典之外，大部分信息都是通过社区文档、 hortonworks/cloudera的blog、slideshare上的各种会议slides以及微博/twitter获取的，当然经常去 hadoop社区的jira上看看issue，阅读一些源码也是必不可少的。

InfoQ：最后一个比较八卦的小问题，请您谈谈花名“莫问”的由来?

王峰：虽然我在阿里已经超过8年了，但我前几年在雅虎中国和阿里云，2010年转到淘宝的时候，好的花名已经都没有了，“莫问”这个名字是“七剑”中傅青主拿的那把剑的名字，是“七剑”智慧的象征，同时也挺喜欢七剑中“莫问前程有愧，但求今生无悔“这句话，所以就起名“莫问”了。

原文链接：http://www.infoq.com/cn/news/2014/09/hadoop-alibaba-yarn?utm_source=tuicool

火币CTO巨建华访谈：数字货币交易平台的基础架构和技术挑战

作者：崔康

在最近举办的“比特币产业峰会暨火币网一周年庆”活动期间，InfoQ就数字货币交易平台的基础架构和技术挑战采访了火币CTO巨建华，他分享了自己在行业领域的宝贵经验和精彩观点。

InfoQ:首先请做一下自我介绍。

巨建华:我先后毕业于电子科技大学和中国人民大学，曾就职于海虹控股、中国搜索、大麦网和YOKA时尚网等知名IT企业，多年以来一直从事互联网电子商务、搜索和门户网站相关技术研发工作。

在工作期间成功构建了目前国内最大票务网站大麦网的全国联网票务系统，部署在数千个场馆节点，支撑了全国70%以上的演出和 体育赛事的售票；5年互联网金融软件领域创业经历，主持开发了多款基于上证所Level-2高速行情数据的软件产品；2013年进军医疗行业应用开发，推 出基于知识库的艾默康智能处方评估系统被应用于100多家大型医院获得了较高的行业知名度；2014年加入火币网后，对火币网的比特币交易系统进行了完全 的重构，使之具备了更加高速，稳定和开放的能力。

InfoQ:相比传统金融交易平台，火币网比特币交易系统（以下简称火币网）的不同点在哪里?

巨建华:由于市场性质的不同，导致火币网和传统的金融交易平台在业务营运方式较大差异，比特币是一种全球性的 数字资产，它的价格行情会受到全球市场的影响，启动了金融杠杆交易后这种价格波动的影响被放大了多倍，这导致了交易系统必满足全年24x7任何时刻不间断 的交易服务，系统服务的中断意味着行情将受到剧烈的波动加大交易的风险，因此我们在系统设计上需要尽可能的避免停机维护，从硬件到软件都做到了完整的冗余 和

高可用，每一个系统都充分围绕着高可用和容错来设计和实现，我们做到了所有系统模块的高可用和核心交易系统的快速故障转移。

相对于传统金融交易平台，火币做数字加密货币金融业务的同时也是一家互联网公司，众所周知互联网产品的创新迭代是非常快速 的，在快速迭代的产品开发过程我们推出了大量不同的创新金融业务，这也导致了非常复杂的资产结算逻辑，在高峰期时每天的交易额达到了10个亿人民币的规模下，我们不能像传统证券交易所那样当天卖出证券资产后第二天才能提现，因为比特币领域太新，在没有法律认可的第三方资金监管的情况下，用户希望自己的资产 随时掌握在自己手里才踏实，这带来了用户对法币和虚拟货币充值提现的实时结算需求，使我们结算和风控系统的设计也变得非常复杂，实现结算提现加比特币资产 的不可追回特性，导致出现任何问题非常致命的，因此我们比传统金融平台来说在技术上存在更大的挑战。

InfoQ:火币网的技术难点有哪些？ 如何解决的?

巨建华:火币网的技术难点主要在于对比特币的撮合交易引擎和比特币数字资产的管理这两方面。

和传统证券交易所同时进行数千个商品的撮合交易不同，在火币网目前只有比特币和莱特币这两种主流数字货币在交易，这意味着每 天10个亿的交易额到未来交易额不断增长的时候，所有用户都是对这两种商品在报价交易，由于撮合交易的规则要求完全遵循时间优先和价格优先的原则进行撮合成交，这导致我们在单个品种每天的交易额在很大程度上超过了大部分现在流通的股票证券交易，不能像传统的证券交易所通过增加集群节点将不同的商品放到不同 的服务器上实现交易系统容量的提升，这个难题我们通过选择对单进程计算最占优势的硬件，在对算法进行了大量的改进实现了单机撮合性能的数量级提升，确保了 对未来交易业务增长的支撑。

比特币本身是一个分布式网络，普通用户使和比特币只需要下载和安装一个钱包客户端，或者使用在线钱包服务即可满足使用比特币 的需求，但是由于比特币网络还处于发展初期，并没有非常有效的开源软件可以支撑住比特币交易所这样集中式的大量比特币资产充值和提现的业务需求，因此我们 需要按照比特币协议，重新设计并量身打造出交易所钱包软件，高效安全的满足数十万用户的充值和提现需求。

InfoQ:火币网的技术栈是怎样的?

巨建华:火币网最初是基于LNMP搭建的交易平台，在关键的钱包和撮合引擎方面使用C++实现，随着业务的发展和业务增长带来的营运压力提升，我们逐渐根据业务的特点进行了相应的技术升级。

首先我们升级到了面向服务的架构，原来的PHP作为Web层实现与用户的交付，将业务层转向用Java开发的后端服务中，这种架构下我们可以在保持非常高效的前端产品迭代周期上，并确保整个服务平台的稳健。

在服务框架方面我们分别采用了Rest.li和Thrift，根据不同的应用场景灵活选用，并通过ZooKeeper实现了 服务的配置管理和集群管理。 和大部分互联网企业一样，我们在实际业务中大量使用了Redis做持久化的存储和数据缓存，通过Haproxy和LVS结合Keepalived实现关键 系统的软件负载均衡。

消息服务方面我们分别使用了RabbitMQ和Appolo，用于实现订单和行情数据的发布管理，通过Node.js和QuickFix这两个开源项目我们实现了实时的行情推送，并为用户提供了可靠的交易API服务。

Python承担了部份运维管理和日常数据处理任。

数据库层面我们使用MongoDB承担了全部行情数据的存储和分发，通过MySQL、InnoDB实现了业务数据的存储。同时我们的交易终端覆盖Windows/Mac OS X/Android/
iOS，在桌面和移动端为用户提供了更好的交易体验。

InfoQ:安全性对于数字货币来说至关重要，火币网是如何从技术方面保障安全的?

巨建华:安全对于交易所来说是非常重要的一面，日本最大交易所MtGox因为安全问题的倒闭在世界范围内影响了比特币的价格走势，为整个比特币交易领域敲响了警钟。

火币网在成立初期就建立了安全部，由经验丰富的安全专家带队参与到了火币网开发和运营的方方面面，从代码安全到系统监控甚至 社交攻击防护，到处都有安全部门的影子。 在火币网发展速度最快的时候，我们面临了大规模的DDOS攻击，最高时攻击的流量达到80多G，我们在改进自身系统和的同时，也引入乌云、安全宝、加速乐 等安全领域的公司的专业服

务，这些工作使火币网一直以来未发生过安全隐患。 在解决自身安全问题的同事，我们也在风控系统中增加了对用户的安全监控，比如有用户帐号被盗以后如果存在异常的登陆提现等行为，我们客服系统上会有相应的 报警，客服人员会在第一时间和用户进行电话核实。

InfoQ:对于7x24的服务承诺，火币网在架构和运维方面是如何做的?

巨建华:我们在所有的系统架构都为高可用做了大量的设计，在前端Web层面和后台数据缓存和业务服务层均允许做任意的节点失效。在数据库层面我们通过复制和数据分区的方式实现了主备层面的高可用，在出现故障后通过相应的业务日志检查即可迅速通过ip漂移实现数据库的故障恢复。

运维方面，从硬件层面的IDC机房线路到防火墙等网络设我们都实现了自动化的主备切换的能力，使用Zibbix完善了监控系 统，除了对所有服务器和网络设备的监控外，还根据业务场景提供了数百个监控点，使我们可以在第一时间获得系统的运行状况和问题报告。 工作时间上我们运维和客服都是24小时待命的，确保了不会出现管理上的空档期带来的意外故障，并为用户提供了随时可以联系报故障的渠道，使我们能快速响应 用户的问题。

InfoQ:对于火币网的重构过程，你有哪些经验收获可以分享给大家?

巨建华:火币网的重构工作主要体现对技术平台的升级和技术团队的建设上，通过初期的问题分析、方案制定和人员 招聘，我们只花了比较短的时间，这个过程中管理团队的支持显得非常重要。 重构过程最重要一环，是火币网重构完成的升级工作主要的挑战在于过程中不能暂停业务，因此从重构初期到完成成级，我们没有发布过一次停机服务的公告。 不停机的重构加上复杂的业务环境，使我们在重构升级的过程中经常面临一些两难和折中的选择，在这些过程中也产生和遗留下了一些问题，在这个过程中我所获得的主要经验以下几点比较重要:

- 充分和原业务和技术团队沟通，深入了解业务了解是重构的前提
- 架构设计和实现上需要支持灰度升级，随时回滚
- 建立尽可能真实的测试环境
- 对可能产生的问题做好规划和演练是重构成功的前提

InfoQ:火币网下一步的发展路线图是什么?

巨建华:火币网下一步将通过完成在整个产业链的布局，推动比特币整个产业务链的发展。具体在技术层面来说，我 们将进行新一代交易系统的设计，使之具备更好的性能和自动灾难切换的能力，同时我们将加大在矿场投资管理、比特币钱包及比特币支付、虚拟货币基金、虚拟商 品期货等金融衍生品领域的技术研发力度，为提升比特币在金融领域和现实世界中的应用提供有力的技术支持。

InfoQ:目前互联网金融很火，很多IT人员都想进入该领域，你从自己的职业生涯角度，会有哪些建议?

巨建华:目前互联网行业普遍缺少高级IT人才，在互联网金融领域更是如此，但是和社区门户、搜索以及电商等领域不同，互联网金融领域对于安全和稳定有着超乎寻常的需求，对于基础技术的掌握要求更加深入。

举个例子，很多人都非常熟悉的MySQL数据库，在传统互联网领域只需要了解一般的主从复制和sql语句编写调优即可，但是 在将MySQL用到存储用户资产、时刻进行高并发的交易时就要求对于数据库事务有着深入的了解才能在确保数据的准确性和一致性的前提下，保证计算的精度和 性能。在金融领域对于主从复制成功的数据库还需要数据正确性的较验在其它领域是很少见的。 多数中小型的互联网金融企业因为成本方面的原因在大量的使用MySQL数据库而不是Oracle这样的解决方案，因此值得希望进入互联网金融领域的人员深 入学习，但是当成长到一定规模以后Oracle、DB2又会在一定程度上成为主流，也是需要掌握的。

Java平台是互联网金融领域技术平台的主力军，几乎所有主流的金融平台都是使用Java开发的，对于想进入互联网金融领域的IT人员来说最好能够系统的掌握Java平台的开发和设计。

对于金融知识的掌握也是进入互联网金融领域的一大挑战，如果不了解金融产品的基础知识，就无法理解复杂的业务逻辑，难以胜任高级的系统架构师和开发人员的职位，这些都是挑战同时也是机遇。

原文链接：http://www.infoq.com/cn/articles/huobi-cto-interview?utm_source=tuicool