推酷

编程狂人

Programming Madman

NO.41

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/540db244d91b142d9711ea53

欢迎下载推酷客户端体验更多阅读乐趣

目录

函数式思维和函数式编程

作者：Pedro Alvarez-Tabio

作为一个对Haskell语言彻头彻尾的新手，当第一次看到一个用这种语言编写的快速排序算法的优雅例子时，我立即对这种语言发生了浓厚的兴趣。下面就是这个例子：

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (p:xs) =
    (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser = filter (< p) xs
        greater = filter (>= p) xs
```

我很困惑。如此的简单和漂亮，能是正确的吗？的确，这种写法并不是“完全正确”的最优快速排序实现。但是，我在这里并不想深入探讨性能上的问题。我想重点强调的是，纯函数式编程是一种思维上的改变，是一种完全不同的编程思维模式和方法，就相当于你要重新开始学习另外一种编程方式。

首先，让我先定义一个问题，然后用函数式的方式解决它。我们要做的基本上就是按升序排序一个数组。为了完成这个任务，我使用曾经改变了我们这个世界的快速排序算法，下面是它几个基本的排序规则：

- 如果数组只有一个元素，返回这个数组

- 多于一个元素时，随机选择一个基点元素P，把数组分成两组。使得第一组中的元素全部 <p，第二组中的全部元素 >p。然后对这两组数据递归的使用这种算法。

那么，如何用函数式的方式思考、函数式的方式编程实现？在这里，我将模拟同一个程序员的两个内心的对话，这两个内心的想法很不一样，一个使用命令式 的编程思维模式，这是这个程序员从最初学习编码就形成的思维模式。而第二个内心做了一些思想上的改造，清洗掉了所有以前形成的偏见：用函数式的方式思考。事实上，这程序员就是我，现在正在写这篇文章的我。你将会看到两个完全不同的我。没有半点假话。

让我们在这个简单例子上跟Java进行比较：

```
public class Quicksort  {
  private int[] numbers;
  private int number;
public void sort(int[] values) {
  if (values == null || values.length == 0){
    return;
  }
  this.numbers = values;
  number = values.length;
  quicksort(0, number - 1);
}

private void quicksort(int low, int high) {
  int i = low, j = high;
  int pivot = numbers[low + (high-low)/2];
```

```
while (i <= j) {
  while (numbers[i] < pivot) {
    i++;
  }
  while (numbers[j] > pivot) {
    j--;
  }

  if (i <= j) {
    swap(i, j);
    i++;
    j--;
    }
  }
  if (low < j)
    quicksort(low, j);
  if (i < high)
    quicksort(i, high);
}

private void swap(int i, int j) {
  int temp = numbers[i];
  numbers[i] = numbers[j];
```

```
    numbers[j] = temp;
  }
 }
```

哇塞。到处都是i和j，这是干嘛呢？为什么Java代码跟Haskell代码比较起来如此的长？这就好像是30年前拿C语言和汇编语言进行比较！从某种角度看，这是同量级的差异。

让我们俩继续两个”我”之间的对话。

JAVA:

好，我先开始定义Java程序需要的数据结构。一个类，里面含有一些属性来保存状态。我觉得应该使用一个整数数组作为主要数据对象，针对这个数组进行排序。还有一个方法叫做sort，它有一个参数，是用来传入两个整数做成的数组，sort方法就是用来对这两个数进行排序。

```
public class Quicksort {
  private int[] numbers;

  public void sort(int[] values) {

  }
}
```

HASKELL:

好，这里不需要状态，不需要属性。我需要定义一个函数，用它来把一个list转变成另一个list。这两个list有相同 之处，它们都包含一样的元素，并有各自的顺序。我如何用统一的形式描述这两个list？啊哈！typeclass….我需要一个typeclass来实现 这个…对，Ord.

```
quicksort :: Ord a => [a] -> [a]
```

JAVA:

我要从简单的开始，如果是空数组，如果数组是空的，我应该返回这个数组。但是…该死的，当这个数组是null时，程序会崩溃。让我来在sort方法开始的地方加一个if语句，预防这种事情。

```
if (values.length == 0 || values == null) {
    return;
}
```

HASKELL:

先简单的，一个空list。对于这种情况，需要使用模式匹配。我看看如何使用，好的，非常棒!

```
quicksort [] = []
```

JAVA:

好的，现在让我用递归来处理正常的情况。正常的情况下，需要记录sort方法参数状态。需要它的长度，所以，我还需要在Quicksort类里添加一个新属性。

```
public void sort(int[] values) {
    if (values.length == 0 || values == null) {
        return;
    }
    this.numbers = values;
    this.length = values.length;
    quicksort(0, length - 1);
}
```

HASKELL:

这已经是递归了。不需要在再做任何事情。

No code. Nothing. Nada. That's good.

JAVA:

现在，我需要根据上面说明的规则实现快速排序的过程。我选择第一个元素作为基点元素，这不需要使用其它奇异方法。比较，递归。每次比较从两头同时遍历，一个从头至尾(i, 生成<p的list)，一个从尾至头(j, 生成>p的list)。每次在i方向遍历中发现有比j方向遍历的当前值大时，交互它们的位置。当i的位置超过j时，停止比较，对形成的两个新队列继续递归调用。

```
private void quicksort(int low, int high) {
    int i = low, j = high;
    int pivot = numbers[low];

    while (i <= j) {
        while (numbers[i] < pivot) {
            i++;
        }
        while (numbers[j] > pivot) {
            j--;
        }

        if (i <= j) {
            swap(i, j);
            i++;
            j--;
        }
    }
```

```
    if (low < j)
        quicksort(low, j);
    if (i < high)
        quicksort(i, high);
}
```

交换位置的方法：

```
private void swap(int i, int j) {
    int temp = numbers[i];
    numbers[i] = numbers[j];
    numbers[j] = temp;
```

使用Haskell

我先定义一个lesser和一个greater作为每次迭代的两个队列。等一下！我们可以使用标准的head和tail函数来获取第一个值作为基点数据。这样我们可以它的两个部分进行递归调用！

```
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
```

非常好，这里我声明了lesser和greater两个list，现在我将要用where——Haskell语言里一个十分强大的用来描述函数内部值(not 变量)的关键字——描述它们。我需要使用filter函数，因为我们已经得到除首元素之外的其它元素，我们可以调用(xs)，就是这样：

```
    where
        lesser = filter (< p) xs
        greater = filter (>= p) xs
```

我试图用最详细的语言解释Java里用迭代+递归实现快速排序。但是，如果在java代码里，我们少写了一个i++，我们弄错了一个while循环条件，会怎样？好吧，这是一个相对简单的算法。但我们可以想象一下，如果我们整

天写这样的代码，整天面对这样的程序，或者这个排序只是一个非常复杂的算法的第一步，将会出现什么情况。当然，它是可以用的，但难免会产生潜在的、内部的bug。

现在我们看一下关于状态的这些语句。如果出于某些原因，这个数组是空的，变成了null，当我们调用这个Java版的快速排序方法时会出现什么情况？还有性能上的同步执行问题，如果16个线程想同时访问Quicksort方法会怎样？我们就要需要监控它们，或者让每个线程拥有一个实例。越来越乱。

最终归结到编译器的问题。编译器应该足够聪明，能够“猜”出应该怎样做，怎样去优化。程序员不应该去思考如何索引，如何处理数组。程序员应该 思考数据本身，如何按要求变换数据。也许你会认为函数式编程给思考算法和处理数据增添的复杂，但事实上不是这样。是编程界普遍流行的命令式编程的思维阻碍 了我们。

事实上，你完全没必要放弃使用你喜爱的命令式编程语言而改用Haskell编程。Haskell语言有其自身的缺陷。只要你能够接受函数式编程思维，你就能写出更好的Java代码。**你通过学习函数式编程能变成一个更优秀的程序员。**

看看下面的这种Java代码?

```
public List<Comparable> sort(List<Comparable> elements) {
    if (elements.size() == 0) return elements;

    Stream<Comparable> lesser = elements.stream()
    .filter(x -> x.compareTo(pivot) < 0)
    .collect(Collectors.toList());

    Stream<Comparable> greater = elements.stream()
    .filter(x -> x.compareTo(pivot) >= 0)
    .collect(Collectors.toList());
```

```
    List<Comparable> sorted = new ArrayList<Comparable>();
    sorted.addAll(quicksort(lesser));
    sorted.add(pivot);
    sorted.addAll(quicksort(greater));

    return sorted;

}
```

是不是跟Haskell代码很相似？没错，也许你现在使用的Java版本无法正确的运行它，这里使用了lambda函数，Java8中引入的一种非常酷的语法。看到没有，函数式语法不仅能让一个程序员变得更优秀，也会让一种编程语言更优秀。

函数式编程是一种编程语言向更高抽象阶段发展的自然进化结果。就跟我们认为用C语言开发Web应用十分低效一样，这些年来，我们也认为命令式编程语言也是如此。使用这些语言是程序员在开发时间上的折中选择。为什么很多初创公司会选择Ruby开发他们的应用，而不是使用C++？因为它们能使开发周期更短。不要误会。我们可以把一个程序员跟一个云计算单元对比。**一个程序员一小时的时间比一个高性能AWS集群服务器一小时的时间昂贵的多**。通过让犯错误更难，让出现bug的几率更少，使用更高的抽象设计，我们能使程序员变得更高效、更具创造性和更有价值。

原译文链接：http://www.vaikan.com/programming-thinking-functional-way/

原文链接：http://peteratt.com/programming-thinking-functional-way/

StackOverflow热帖：如何用Java编写一段代码引发内存泄露?

作者:stackoverflow

本文来自StackOverflow问答网站的一个热门讨论：如何用Java编写一段会发生内存泄露的代码。

Q：刚才我参加了面试，面试官问我如何写出会发生内存泄露的Java代码。这个问题我一点思路都没有，好囧。

A1：通过以下步骤可以很容易产生内存泄露（程序代码不能访问到某些对象，但是它们仍然保存在内存中）：

1. 应用程序创建一个长时间运行的线程（或者使用线程池，会更快地发生内存泄露）。

2. 线程通过某个类加载器（可以自定义）加载一个类。

3. 该类分配了大块内存（比如new byte[1000000]），在某个静态变量存储一个强引用，然后在ThreadLocal中存储它自身的引用。分配额外的内存new by-
te[1000000]是可选的(类实例泄露已经足够了)，但是这样会使内存泄露更快。

4. 线程清理自定义的类或者加载该类的类加载器。

5. 重复以上步骤。

由于没有了对类和类加载器的引用，ThreadLocal中的存储就不能被访问到。ThreadLocal持有该对象的引用，它也就持有了这个类及 其类加载器的引用，类加载器持有它所加载的类的所有引用，这样GC无法回收ThreadLocal中存储的内存。在很多JVM的实现中Java类和类加载 器直接分配到permgen区域不执行GC，这样导致了更严重的内存泄露。

这种泄露模式的变种之一就是如果你经常重新部署以任何形式使用了ThreadLocal的应用程序、应用容器（比如Tomcat）会很容易发生内存泄露（由于应用容器使用了如前所述的线程，每次重新部署应用时将使用新的类加载器）。

A2:

静态变量引用对象

```
class MemorableClass {
    static final ArrayList list = new ArrayList(100);
}
```

调用长字符串的String.intern()

```
String str=readString(); // read lengthy string any source db,textbox/jsp etc..
// This will place the string in memory pool from which you cant remove
str.intern();
```

未关闭已打开流(文件，网络等

```
try {
    BufferedReader br = new BufferedReader(new FileReader(input-File));
    ...
```

```
    ...
} catch (Exception e) {
    e.printStacktrace();
}
```

未关闭连接

```
try {
    Connection conn = ConnectionFactory.getConnection();
    ...
    ...
} catch (Exception e) {
    e.printStacktrace();
}
```

JVM的GC不可达区域

比如通过native方法分配的内存。

web应用在application范围的对象，应用未重启或者没有显式移除

getServletContext().setAttribute("SOME_MAP", map);

web应用在session范围的对象，未失效或者没有显式移除

session.setAttribute("SOME_MAP", map);

不正确或者不合适的JVM选项

比如IBM JDK的noclassgc阻止了无用类的垃圾回收

A3：如果HashSet未正确实现(或者未实现)hashCode()或者equals(),会导致集合中持续增加“副本”。如果集合不能地忽略掉它应该忽略的元素，它的大小就只能持续增长，而且不能删除这些元素。

如果你想要生成错误的键值对，可以像下面这样做：

```
class BadKey {
   // no hashCode or equals();
   public final String key;
   public BadKey(String key) { this.key = key; }
}

Map map = System.getProperties();
map.put(new BadKey("key"), "value"); // Memory leak even if your threads die.
```

A4：除了被遗忘的监听器，静态引用，hashmap中key错误/被修改或者线程阻塞不能结束生命周期等典型内存泄露场景，下面介绍一些不太明显的Java发生内存泄露的情况，主要是线程相关的。

- Runtime.addShutdownHook后没有移除，即使使用了removeShutdownHook，由于ThreadGroup类对于未启动线程的bug，它可能不被回收，导致ThreadGroup发生内存泄露。
- 创建但未启动线程，与上面的情形相同
- 创建继承了ContextClassLoader和AccessControlContext的线程，ThreadGroup和InheritedThreadLocal的使用，所有这些引用都是潜在的泄露，以及所有被类加载器加载的类和所有静态引用等等。这对ThreadFactory接口作为重要组成元素整个j.u.c.Executor框架(java.util.concurrent)的影响非常明显，很多开发人员没有注意到它潜在的危险。而且很多库都会按照请求启动线程。
- ThreadLocal缓存，很多情况下不是好的做法。有很多基于ThreadLocal的简单缓存的实现，但是如果线程在它的期望生命周期外继续

运行ContextClassLoader将发生泄露。除非真正必要不要使用ThreadLocal缓存。

- 当ThreadGroup自身没有线程但是仍然有子线程组时调用ThreadGroup.destroy()。发生内存泄露将导致该线程组不能从它的父线程组移除，不能枚举子线程组。

- 使用WeakHashMap，value直接(间接)引用key，这是个很难发现的情形。这也适用于继承Weak/SoftReference的类可能持有对被保护对象的强引用。

- 使用http(s)协议的java.net.URL下载资源。KeepAliveCache在系统ThreadGroup创建新线程，导致当前线程的上下文类加载器内存泄露。没有存活线程时线程在第一次请求时创建，所以很有可能发生泄露。(在Java7中已经修正了，创建线程的代码合理地移除了上下文类加载器。)

- 使用InflaterInputStream在构造函数(比如PNGImageDecoder)中传递new java.util.zip.Inflater()，不调用inflater的end()。仅仅是new的话非常安全，但如果自己创建该类作为构造函数参数时调用流的close()不能关闭inflater，可能发生内存泄露。这并不是真正的内存泄露因为它会被finalizer释放。但这消耗了很多native内存，导致linux的oom_killer杀掉进程。所以这给我们的教训是：尽可能早地释放native资源。

- java.util.zip.Deflater也一样，它的情况更加严重。好的地方可能是很少用到Deflater。如果自己创建了Deflater或者Inflater记住必须调用end()。

原译文链接：http://www.importnew.com/12901.html

原文链接：http://stackoverflow.com/questions/6470651/creating-a-memory-leak-with-java

Redis设计与实现（一~五整合版）

作者：飘过的小牛

一

前言

项目中用到了redis，但用到的都是最最基本的功能，比如简单的slave机制，数据结构只使用了字符 串。但是一直听说redis是一个很牛的开源项目，很多公司都在用。于是我就比较奇怪，这玩意不就和 memcache 差不多吗？仅仅是因为memcache是内存级别的，没有持久化功能。而redis支持持久化？难道这就是它的必杀技？

带着这个疑问，我在 网上搜了一圈。发现有个叫做huangz的程序员针对redis写了一本书叫做《redis设计与实现》，而且业界良心搞了一个reids2.6版本的注 释版源码。这本书不到200页，估计2个星期能看完吧，之后打算再看下感兴趣部分的源码。当然，如果你不知道redis是干嘛的，请自行谷歌，简单说就是 Key-Value数据库，而且 value 支持5种数据结构：

1. 字符串
2. 哈希表（map）
3. 列表（list）
4. 集合（set）
5. 有序集

下面我们就从 redis 的内部结构开始说起吧：）

一、redis内部数据结构

首先需要知道，redis是用C写的。众所周知，任何系统对于字符串的操作都是最频繁的，而恰巧C语言的字符串备受诟病。然后作者就封装了一下C 语言的字符串 char *。

总之，根据redis的业务场景，整个redis系统的底层数据支撑被设计为如下几种：

- 简单动态字符串sds(Simple Dynamic String)
- 双端链表
- 字典(Map)
- 跳跃表

下面我们就分别来说说这4种数据结构。

1. 简单动态字符串sds

- redis的字符串表示为sds，而不是C字符串（以\0结尾的char*）
- 对比C字符串，sds有以下特性
  - 可以高效执行长度计算O(1)
  - 可以高效执行append操作(通过free提前分配）
  - 二进制安全
- sds会为追加操作进行优化，加快追加操作的速度，并降低内存分配的次数，代价是多占用内存，且不会主动释放

这个一看名字就能知道个大概了，因为字符串操作无非是增删查改，如果使用char[]数组，那是要死人的，任何操作都是O(N)复杂度。所以，要对某些频繁的操作实现O(1)级性能。但是我们还是得思考：

为什么要对字符串造轮子?

因 为redis是一个key-value类型的数据库，而key全部都是字符串，value可以是集合、hash、list等等。同时，在redis的各种 操作中，都会频繁使用字符串的长度和append操作，对于char[]来说，长度操作是O(N)的，append会引起N次realloc。而且因为 redis分为client和server，传输的协议

内容必须是二进制安全的，而C的字符串必须保证是\0结尾，所以在这两点的基础上开发sds

知 道了上面几点就可以看下实现了，其实实现特别简单。它通过一个结构体来代表字符串对象，内部有个len属性记录长度，有个free用于以后的append 操作，具体的值还是一个char[]。长度就不说了，只在插入的时候用一下，以后只需要维护len就可以O(1)拿到；对于free也很简 单，vector不也是这么实现的嘛。就是按照某个阈值进行翻倍叠加。

2. 双端链表

- redis自己实现了双端链表
- 双端链表主要两个作用:

1. 作为redis列表类型的底层实现之一
2. 作为通用数据结构被其他模块使用

- 双端链表及其节点的性能特征如下:
  - 节点带有前驱和后继指针
  - 链表是双向的，所以对表头和表尾操作都是O(1)
  - 链表节点可以会被维护，LLEN复杂度为O(1)

这玩意当时刷数据结构与算法分析那本书看过，但是没怎么用到过。说白了双端链表就是有2个指针，一个指向链表头，一个指向链表尾。对每个节点而言，记录自己的父节点和子节点，这样双向移动速度会快很多。

还是老问题:

为什么要有双端链表?

在Java或者C++中，都有现成的容器供我们使用，但是C没有。于是作者自己造了一个双端链表数据结构。而这个也是redis列表数据结构的基础之一（另外一个还是压缩列表）。而且双端链表也是一个通用的数据结构被其他功能调用，比如事务。

至于实现也是比较简单，双端链表，肯定有2个指针指向链表头和链表尾，然后内部维护一个len保存节点的数目，这样当使用LLEN的时候就能达

到O(1)复杂度了。其他的，额，对每个节点而言，都有双向的指针，另外还有针对双端链表的迭代器，也是两个方向。

3. 字典（其实说Map更通俗）

- 字典是由键值对构成的抽象数据结构
- redis中的数据库和哈希键值对都基于字典实现
- 字典的底层实现为哈希表，每个字典含有2个哈希表，一般只是用0号哈希表，1号哈希表是在rehash过程中才使用的
- 哈希表使用链地址法来解决碰撞问题
- rehash可以用于扩展或者收缩哈希表
- 对哈希表的rehash是分多次、渐进式进行的

这个虽然说经常用，但是对于redis来说确实是重中之重。毕竟redis就是一个key-value的数据库，而key被称为键空间(key space)，这个键空间就是由字典实现的。第二个用途就是用作hash类型的其中一种底层实现。下面分别来说明。

1. 键空间：redis是一个键值对数据库，数据库中的键值对就由字典保存：每个数据库都有一个与之相对应的字典，这个字典被称为键空间。当用户添加一个键值对到数据库（不论键值对是什么类型），程序就讲该键值对添加到键空间，删除同理。

2. 用作hash类型键的其中一种底层实现：hash底层实现是通过压缩列表和字典实现的，当建立一个hash结构的时候，会优先使用空间占用率小的压缩列表。当有需要的时候会将压缩列表转化为字典

对于字典的实现这里简单说明一下即可，因为很简单。

字 典是通过hash表实现的。每个字典含有2个hash表，分别为ht[0]和ht[1]，一般情况下使用的是ht[0],ht[1]只有在rehash的 时候才用到。为什么呢？因为性能，我们知道，当hash表出现太多碰撞的话，查找会由O(1)增加到O(MAXLEN),redis为了性能，会在碰撞过 多的情况下发生rehash，rehash就是扩大hash表的大小，从而将碰撞率降低，当hash表大小和节点数量维持在1：1时候性能最优，就是 O(1)。另外的rehashidx字段也比较有看头，redis支持渐进式hash，下面会讲到原理。

下面讲一下rehash的触发条件：

当新插入一个键值对的时候，根据used/size得到一个比例，如果这个比例超过阈值，就自动触发rehash过程。rehash分为两种：

- 自然rehash:满足used/size >= 1 && dict_can_resize条件触发的
- 强制rehash:满足used/size >= dict_force_resize_ratio(默认为5)条件触发的。

思 考一下，为什么需要两种rehash呢？答案还是为了性能，不过这点考虑的是redis服务的整体性能。当redis使用后台子进程对字典进行 rehash的时候，为了最大化利用系统的copy on write机制，子进程会暂时将自然rehash关闭，这就是dict_can_resize的作用。当持久化任务完成后，将dict_can_resize设为true，就可以继续进行自然rehash；但是考虑另外一种情况，当现有字典的碰撞率太高了，size是指针数组的 大小，used是hash表节点数量，那么就必须马上进行rehash防止再插入的值继续碰撞，这将浪费很长时间。所以超过 dict_force_resize_ratio后，无论在进行什么操作，都必须进行rehash。

rehash过程很简单，分为3步：

1. 给ht[1]分配至少2倍于ht[0]的空间
2. 将ht[0]数据迁移到ht[1]
3. 清空ht[0], 将ht[0]指针指向ht[1],ht[1]指针指向ht[0]

同样是为了性能（当用户对一个很大的字典插入时候，你不能让系统阻塞来完成整个字典的rehash。所以redis采用了渐进式rehash。说白了就是分步进行rehash。具体由下面2个函数完成：

1. dictRehashStep: 从名字可以看出，是按照step进行的。当字典处于rehash状态(dict的rehashidx不为-1)，用户进行增删查改的时候会触发dictRehashStep，这个函数就是将第一个索引不为空的全部节点迁移到ht[1]，因为一般情况下节点数目不会超过5(超过基本会触发强制 rehash），所以成本很低，不会影响到响应时间。

2. dictRehashMilliseconds:这个相当于时间片轮转rehash，定时进行redis cron job来进行rehash。会在指定时间内对dict的rehashidx不为-1的字典进行rehash

上面讲完了rehash过程，但是以前在组内分享redis 的时候遇到过一个问题：

当进行rehash 时候，我进行了增删查改怎么办？是在ht[0]进行还是在ht[1]进行呢？

redis采用的策略是rehash过程中ht[0]只减不增，所以增加肯定是ht[1]，查找、修改、删除则会同时在ht[0]和ht[1]进行。

Tips: redis为了减少存储空间，rehash还有一个特性是缩减空间，当多次进行删除操作后，如果used/size的比例小于一个阈值（现在是10%),那么就会触发缩减空间rehash，过程和增加空间类似，不详述了。

3. 跳跃表

- 跳跃表是一种随机化数据结构（它的层是随机产生的），查找、添加、删除操作都是O(logN)级别的。
- 跳跃表目前在redis 的唯一用处就是有序集类型的底层数据结构之一（另外一个还是字典）
- 当然，根据redis的特性，作者对跳跃表进行了修改
  - socre可以重复
  - 对比一个元素需要同时检查它的score和member
  - 每个节点带有高度为1的后退指针，用于从表尾方向向表头方向迭代

redis 使用了跳跃表，但是我发现。。。。我竟然不知道跳跃表是什么东东。亏我还觉得数据结构基础还凑合呢= =。于是赶紧去看了《数据结构与算法分析》，算是知道是啥玩意的。说白了，就是链表+二分查找的结合体。这里主要是研究redis的，所以就不细谈这个数 据结构了。

和双端链表、字典不同的是，跳跃表在reids中不是广泛使用的，它在redis中的唯一作用就是实现有序集数据类型。所以等到集合的时候再深入了解。

二

上一章我们介绍了redis的内部结构：

但是，创建这些完整的数据结构是比较耗费内存的，如果对于一个特别简单的元素，使用这些数据结构无异于大材小用。为了解决这个问题，redis在条件允许的情况下，会使用内存映射数据结构来代替内部数据结构，主要有：

1. 整数集合 intset
2. 压缩列表 ziplist

当然了，因为这些结构是和内存直接打交道的，就有节省内存的优点，而又因为对内存的操作比较复杂，所以也有操作复杂，占用的CPU时间更多的缺点。

这个要掌握一个平衡，才能使redis的总体效率更好。目前，redis使用两种内存映射数据结构。

1. 整数集合

整 数集合用于有序、无重复的保存多个整数值，它会根据元素的值，自动选择该用什么长度的整数类型来保存元素。比如，在一个int set中，最大的元素可以用int16_t保存，那么这个int set的所有元素都是int16_t，当插入一个元素是int32_t的时候，int set会先将所有元素升级为int32_t，再插入这个元素。总的来说，整数集合会自动升级。

看名字我们就知道它的用途：

1. 只保存整数元素
2. 元素的数量不多[因为它不费内存，费CPU。量多的话，肯定是CPU为第一考虑]

那么我们看一下 intset 的定义：

```
typedef struct intset {

    // 保存元素所使用的类型的长度
    uint32_t encoding;

    // 元素个数
    uint32_t length;

    // 保存元素的数组
    int8_t contents[];

} intset;
```

其中 encoding 保存的是 intset 中元素的编码类型，比如是 int16_t还是 int32_t等等。具体的定义在 intset.c 中：

```
#define INTSET_ENC_INT16 (sizeof(int16_t))

#define INTSET_ENC_INT32 (sizeof(int32_t))

#define INTSET_ENC_INT64 (sizeof(int64_t))
```

length 肯定就是元素的个数喽，然后是具体的元素，我们发现是 int8_t 类型的，实现上它只是一个象征意义上的类型，到实际分配时候，会根据具体元素的类型选择合适的类型。而且 contents 有两个特点：

1. 没有重复元素
2. 元素在数组中从小到大排序

所以，添加元素到intset有下面几个步骤：

1. 判断插入元素是否存在于集合，如果存在，没有任何操作(无重复元素)

2. 看元素的长度是否需要把intset升级，如果需要，先升级

3. 插入元素，而且要保证在contents数组中，从小到大排序

4. 维护length

简单总结一下整数集合的特点：

1. 保存有序、无重复的整数元素

2. 根据元素的值自动选择对应的类型，但是int set只升级、不降级

3. 升级会引起整个int set中的contents数组重新内存分配，并移动所有的元素（因为内存不一样了），所以复杂度为O(N)

4. 因为int set是有序的，所以查找使用的是binary search

2. 压缩列表

本 质来说，压缩列表就是由一系列特殊编码的内存块构成的列表，一个压缩列表可以包含多个节点，每个节点可以保存一个长度受限的字符数组（不以为\0结尾的 char数组）或者整数。说白了，它是以内存为中心的数据结构，一般列表是以元素类型的字节总数为大小，而压缩列表是以它最小内存块进行扩展组成的列表。 下面我来说一下。

压缩列表分为3个部分：

- header：10字节，保存整个压缩列表的信息，有尾节点到head的偏移量、节点个数、整个压缩列表的内存（字节）
- 节点：一个结构体、由前一个节点的大小（用于向前遍历）、元素类型and长度、具体值组成
- 哨兵：就是一个1字节的全为1的内存，表示一个压缩列表的结束

其中压缩列表的节点值得说一下，它可以存储两类数据：

那么，怎样实现呢？很简单，通过 encoding + length 就可以搞定。encoding 占2位，00，01，10，11表示不明的类型，只有11代表的是节点中存放的是整型，其他3个代表节点中存放的都是字符串。而根据这2位的不同，又对应着不同的长度。

所 以，由 encoding 可以知道元素的类型和这个元素的范围（比如 encoding 为01，包括 encoding 在内的2byte 代表长度，所以最长是214 - 1；如果 encoding 为00，包括 encoding 在内的1byte 代表元素的长度，所以最大值为26 -1 ）

然后添加元素大概是下面酱紫滴（对于列表来说，添加元素默认是加在列表尾巴的）：

- 首先通过压缩列表的head信息，找到压缩列表的尾巴到head的偏移量（因为可能重新分配内存，所以指针的话会失效）
- 根据要插入的值，计算出编码类型和插入值的长度。然后还有前一个节点所用的空间、然后对压缩列表进行内存充分配
- 初始化entry节点的所有相关信息：pre_entry_length、encoding、length、content
- 更新head中的长度啦、尾偏移啦、压缩列表总字节啦

上 面吐槽了压缩列表没有next指针，现在发现有了= =，但是不是指针，因为压缩列表会进行内存充分配，所以指针代表的内存地址需要一直维护，而当使用偏移量的话，就不需要更改一次维护一次。向后遍历是通过 头指针+节点的大小（pre_entry_length+encoding+length的总大小）就可以跳到下一个节点了

不过，说实话，压缩列表这个设计的好处我还没有看到，可能还需要和后面的东西结合吧。

重读之后看到了，(^__^) 嘻嘻……

本质上面已经说的很清楚了——节省内存。所以它不像上一章讲到的那种分配固定的大小，而 intset 和 ziplist 完全是根据内存定做的，一个字节也不多（当然，有些操作还是会有浪费的）。

三

前言

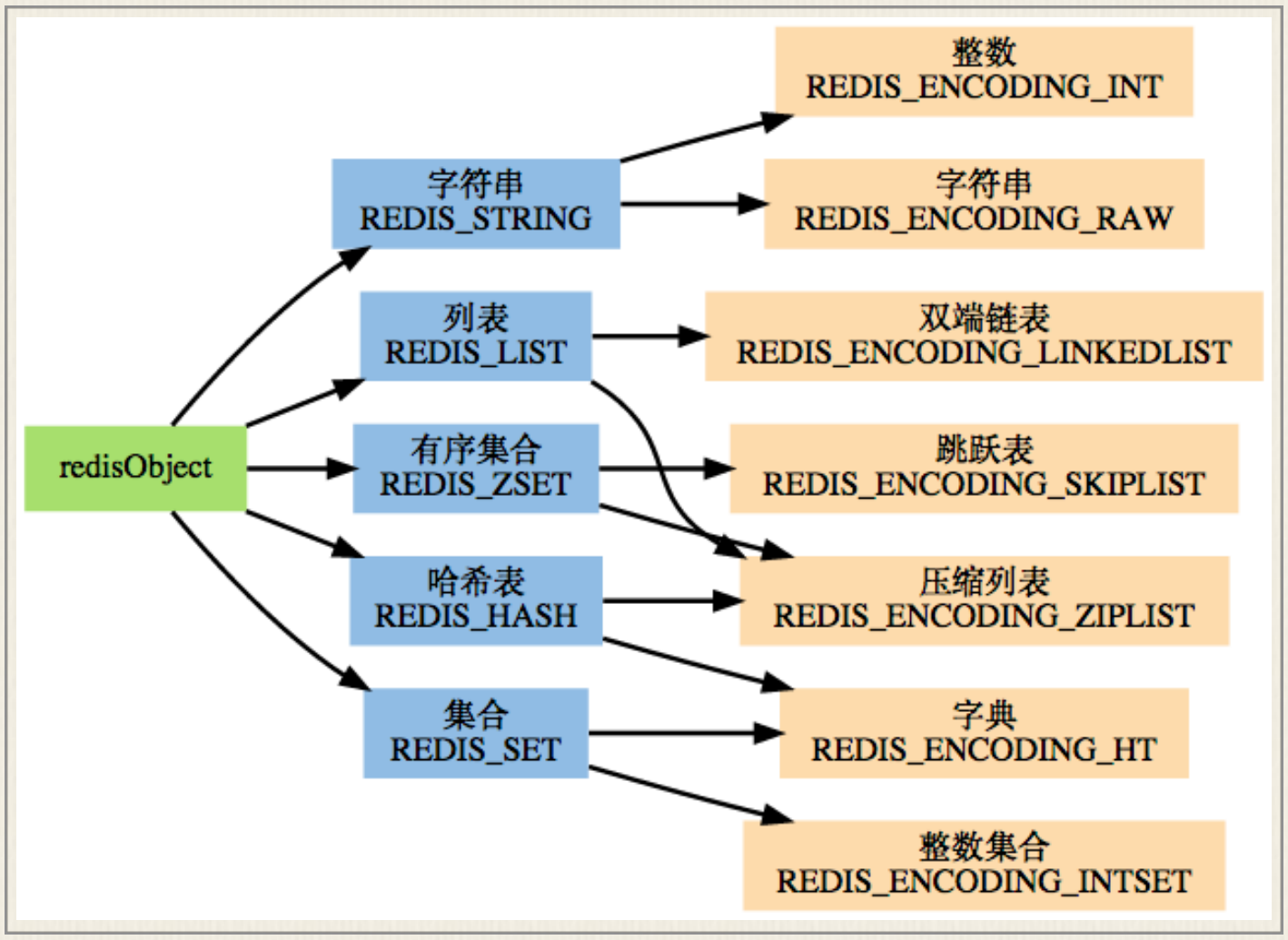
这一章主要是讲redis内部的数据结构是如何实现的，可以说是redis的根基，前面2章介绍了redis的内部数据结构：

redis的内存映射数据结构：

而这一章，就是具体将这些数据结构是如何在redis中工作的。

1. 总观redis内部实现

一张图说明问题的本质：



之后，我们再根据这张图来说明redis中的数据架构为什么是酱紫滴。前面我们已经说过，redis中有5种数据结构，而它们的底层实现都不是唯一的，所以怎样选择对应的底层数据支撑呢？这就需要“多态”的思想，但是因为redis是C开发的。所以通过结构体来模仿对象的“多态”（当然，本质来说这 是为了让自己能更好的理解）。

为了完成这个任务，redis是这样设计的：

- redisObject对象
- 基于redisObject对象的类型检查

- 基于redisObject对象的显式多态函数
- 对redisObject进行分配、共享和销毁的机制

下面看下redisObject的定义：

```
/*
 * Redis 对象
 */
typedef struct redisObject {

    // 类型
    unsigned type:4;

    // 对齐位
    unsigned notused:2;

    // 编码方式
    unsigned encoding:4;

    // LRU 时间（相对于 server.lruclock）
    unsigned lru:22;

    // 引用计数
    int refcount;

    // 指向对象的值
```

```
    void *ptr;

} robj;
```

其中type、encoding、ptr是最重要的3个属性：

- type：redisObject的类型，字符串、列表、集合、有序集、哈希表
- encoding：底层实现结构，字符串、整数、跳跃表、压缩列表等
- ptr：实际指向保存值的数据结构

举个例子就是：

如果一个 redisObject 的 type 属性为 REDIS_LIST ， encoding 属性为 REDIS_ENCODING_LINKEDLIST ，那么这个对象就是一个 Redis 列表，它的值保存在一个双端链表内，而 ptr 指针就指向这个双端链表；
如果一个 redisObject 的 type 属性为 REDIS_HASH ， encoding 属性为 REDIS_ENCODING_ZIPMAP ，那么这个对象就是一个 Redis 哈希表，它的值保存在一个 zipmap 里，而 ptr 指针就指向这个 zipmap ； 诸如此类。

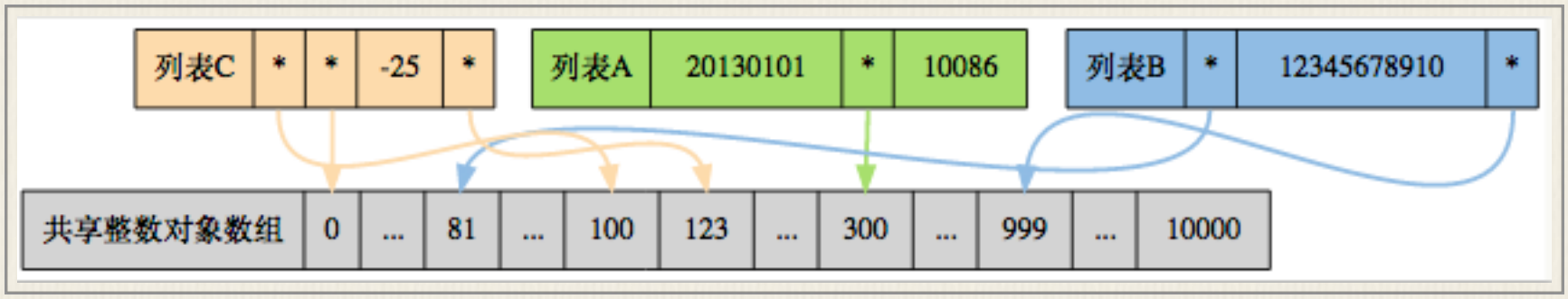
所以，当执行一个操作时，redis是这么干的：

1. 根据key，查看数据库中是否存在对应的redisObject，没有就返回null
2. 查看redisObject的type是否和要执行的操作相符
3. 根据redisObject的encoding属性选择对应的数据结构
4. 返回处理结果

然后reids还搞了一个内存共享，这个挺赞的：

对于一些操作来说，返回值就那几个。对于整数来说，存入的数据也通常不会太大，所以redis通过预分配一些常见的值对象，并在多个数据结构之间（很不幸，你得时指针才能指到这里）共享这些对象，避免了重复分配，节约内存。同时也节省了CPU时间

如图所示:



三个列表的值分别为:

- 列表 A： [20130101, 300, 10086]
- 列表 B： [81, 12345678910, 999]
- 列表 C： [100, 0, -25, 123]

最后一个：redis对对象的管理是通过最原始的引用计数方法。

2. 字符串

字符串是redis使用最多的数据结构，除了本身作为SET/GET的操作对象外，数据库中的所有key，以及执行命令时提供的参数，都是用字符串作为载体的。

在上面的图中，我们可以看见，字符串的底层可以有两种实现:

1. REDIS_ENCODING_INT使用long类型保存long的值
2. REDIS_ENCODING_ROW使用sdshdr保存sds、long long、double、long double等

说白了就是除了long是通过第一种存储以外，其他类型都是通过第二种存储滴。

然后新创建的字符串，都会默认使用第二种编码，在将字符串作为键或者值保存进数据库时，程序会尝试转为第一种（为了节省空间）

3. 哈希表

哈希表，嗯，它的底层实现也有两种:

- REDIS_ENCODING_ZIPLIST

- REDIS_ENCODING_HT(字典)

当创建新的哈希表时，默认是使用压缩列表作为底层数据结构的，因为省内存呀。只有当触发了阈值才会转为字典：

- 哈希表中某个键或者值的长度大于server.hash_max_ziplist_value（默认为64）
- 压缩列表中的节点数量大于server.hash_max_ziplist_entries(默认为512)

4. 列表

列表嘛，其实就是队列。它的底层实现也有2种：

- REDIS_ENCODING_ZIPLIST
- REDIS_ENCODING_LINKEDLIST

当创建新的列表时，默认是使用压缩列表作为底层数据结构的，还是因为省内存- -。同样有一个触发阈值：

- 试图往列表中插入一个字符串值，长度大于server..list_max_ziplist_value(默认是64)
- ziplist包含的节点超过server.list_max_ziplist_entries(默认值为512)

阻塞命令

对于列表，基本的操作就不介绍了，因为列表本身的操作和底层实现基本一致，所以我们可以简单的认为它具有双端队列的操作即可。重点讨论一下列表的阻塞命令比较好玩。

当我们执行BLPOP/BRPOP/BRPOPLPUSH的时候，都可能造成客户端的阻塞，它们被称为列表的阻塞原语，当然阻塞原语并不是一定会造成客户端阻塞：

- 只有当这些命令作用于空列表，才会造成客户端阻塞
- 如果被处理的列表不为空，它们就执行无阻塞版本的LPOP/RPOP/RPOPLPUSH

上面两条的意思很简单，因为POP命令是删除一个节点，那么当没有节点的时候，客户端会阻塞直到一个元素添加进来，然后再执行POP命令，那么，对客户端的阻塞过程是这样的：

1. 将客户端的连接状态更改为“正在阻塞”，并记录这个客户端是被那些键阻塞（可以有多个），以及阻塞的最长时间

2. 将客户端的信息加入到字典server.db[i].blocking_keys中，i就是客户端使用的数据库编号

3. 继续保持客户端和服务器端的连接，但是不发送任何信息，造成客户端阻塞

响应的，解铃须有系铃人：

1. 被动脱离：有其他客户端为造成阻塞的键加入了元素

2. 主动脱离：超过阻塞的最长时间

3. 强制脱离：关闭客户端或者服务器

上面的过程说的很简单，但是在redis内部要执行的操作可以很多的，我们用一段伪代码来演示一下被动脱离的过程：

```
def handleClientsBlockedOnLists():

    # 执行直到 ready_keys 为空
    while server.ready_keys != NULL:
        # 弹出链表中的第一个 readyList
        rl = server.ready_keys.pop_first_node()
        # 遍历所有因为这个键而被阻塞的客户端
        for client in all_client_blocking_by_key(rl.key, rl.db):

            # 只要还有客户端被这个键阻塞,就一直从键中弹出元素
            # 如果被阻塞客户端执行的是 BLPOP ,那么对键执行 LPOP
```

```
        # 如果执行的是 BRPOP ,那么对键执行 RPOP
        element = rl.key.pop_element()
        if element == NULL:
            # 键为空,跳出 for 循环
            # 余下的未解除阻塞的客户端只能等待下次新元素的进入了
            break
        else:
            # 清除客户端的阻塞信息
            server.blocking_keys.remove_blocking_info(client)
            # 将元素返回给客户端,脱离阻塞状态
            client.reply_list_item(element)
```

至于主动脱离，更简单了，通过redis的cron job来检查时间，对于过期的blocking客户端，直接释放即可。伪代码如下：

```
def server_cron_job():

    # cron_job其他操作 ...

    # 遍历所有已连接客户端
    for client in server.all_connected_client:

        # 如果客户端状态为“正在阻塞”,并且最大阻塞时限已到达
        if client.state == BLOCKING and \
        client.max_blocking_timestamp < current_timestamp():
```

```
        # 那么给客户端发送空回复, 脱离阻塞状态
        client.send_empty_reply()

        # 并清除客户端在服务器上的阻塞信息
        server.blocking_keys.remove_blocking_info(client)

    # cron_job其他操作 ...
```

5. 集合

这个就是set，底层实现有2种：

- REDIS_ENCODING_INTSET
- REDIS_ENCODING_HT(字典)

对于集合来说，和前面的2种不同点在于，集合的编码是决定于第一个添加进集合的元素：

1. 如果第一个添加进集合的元素是long long类型的，那么编码就使用第一种
2. 否则使用第二种

同样，切换也需要达到一个阈值：

- intset保存的整数值个数超过server.set_max_intset_entries(默认值为512)
- 从第二个元素开始，如果插入的元素类型不是long long的，就要转化成第二种

然后对于集合，有3个操作的算法很好玩，但是因为没用到过，就暂时列一下：

6. 有序集

终于看到最后一个数据结构了，虽然只有5个- -。。。。。首先从命令上就可以区分这几种了：

- GET/SET是字符串
- H开头的是哈希表
- L开头的是列表
- S开头的是集合
- Z开头的是有序集

继续说有序集，这个东西我还真的没用过。。。其他最起码都了解过，这个算是第一次接触。现在看来，它也算一个sort过的map，sort的依据就是score，对score排序后得到的集合。

首先还是底层实现，有2种:

- REDIS_ENCODING_ZIPLIST
- REDIS_ENCODING_SKIPLIST

这个竟然用到了跳跃表，不用这个的话，跳跃表好像都快被我忘了呢。。对于编码的选择，和集合类似，也是决定于第一个添加进有序集的元素:

- 如果满足：1.服务器属性server.zset_max_ziplist_entries值大于0（默认为128）2.元素的member长度小于服务器属性server.zset_max_ziplist_value(默认为64)，就以第一种作为底层数据结构
- 否则使用第二种

对于编码的转换阈值是这样的:

- ziplist保存的元素数量超过服务器属性server.zset_max_ziplist_entries的值（默认为128）
- ziplist的元素长度大于服务器属性server.zset_max_ziplist_value(默认为64)

那 我们知道，如果有序集是用ziplist实现的，而ziplist终对于member和score是按次序存储的，如 member1,score1,member2,score2...这样的。那么，检索时候就蛋疼了，肯定是O(N)复杂度，既然这样，效率一下子就没有 了。庆幸的是，转换成跳跃表之后，redis搞的很高明:

它用一个字典和一个跳跃表同时来存储有序集的元素，而且因为member和score是在内存区域其字典有指针，就可以共享一块内存，不用每个元素复制两份。

通过使用字典结构,并将 member 作为键,score 作为值,有序集可以在 O(1) 复杂度内:

- 检查给定member是否存在于有序集(被很多底层函数使用)
- 取出 member 对应的 score 值(实现 ZSCORE 命令)

通过使用跳跃表,可以让有序集支持以下两种操作:

- 在 O(logN) 期望时间、O(N) 最坏时间内根据 score 对 member 进行定位(被很多底层函数使用)
- 范围性查找和处理操作,这是(高效地)实现 ZRANGE 、ZRANK 和 ZINTERSTORE等命令的关键

通过同时使用字典和跳跃表,有序集可以高效地实现按成员查找和按顺序查找两种操作。所以，对于有序集来说，redis的思路确实是很流弊的。

7. 总结

上面几个小节讲述了redis的数据结构的底层实现，但是没有涉及到具体的命令，如果调研后发现redis的某种数据结构满足需求，就可以对症下药，去查看redis对应的API即可。

四

前言

这一章主要讲解redis内部的一些功能，主要分为以下4个：

那么，我们就来逐个击破！

1. 事务

事务对于刚接触计算机的人来说可能会比较抽象。因为事务是对计算机某些操作的称谓。通俗来说，事务就是一个命令、一组命令执行的最小单元。事务一般具有ACID属性（redis只支持两种，下文详细说明）：

- 原子性（atomicity）：一个事务是一个不可分割的最小工作单位，事务中包括的诸操作要么都做，要么都不做。

- 一致性（consistency）：事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。

- 隔离性（isolation）：一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

- 持久性（durability）：持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

那么，redis是通过MULTI/DISCARD/EXEC/WATCH这4个命令来实现事务功能。对事务，我们必须知道事务安全性是一个非常重要的。

事务提供了一种“将多个命令打包，然后一次性、按顺序执行”的机制，并且在事务执行期间不会中断——意思就是在事务完成之前，客户端的其他命令都是阻塞状态。

以下是一个事务的例子,它先以 MULTI 开始一个事务,然后将多个命令入队到事务中,最后 由EXEC 命令触发事务,一并执行事务中的所有命令:

```
redis> MULTI
OK
redis> SET book-name "Mastering C++ in 21 days"
81
QUEUED
redis> GET book-name
QUEUED
redis> SADD tag "C++" "Programming" "Mastering Series"
QUEUED
redis> SMEMBERS tag
```

QUEUED

redis> EXEC

1) OK

2) "Mastering C++ in 21 days"

3) (integer) 3

4) 1) "Mastering Series"

2) "C++"

3) "Programming"

一个事务主要经历3个阶段：

1. 开始事务
2. 命令入队（看，上面都有QUEUED这个返回值）
3. 执行事务

这 几个过程都比较简单，开始事务就是切换到事务模式；命令入队就是把事务中的每条命令记录下来，包括是第几条命令，命令参数什么的（当然，事务中是不能再嵌 套事务的，所以再有事务关键字（MULTI/ DISCARD/WATCH）会立即执行的）；执行事务就是一下子把刚才那个事务的命令执行完。

- DISCARD: 取消一个事务，它会清空客户端的整个事务队列，然后将客户端从事务状态调整回非事务状态，最终返回字符串OK给客户端，说明事务已经取消

- MULTI：因为redis不允许事务嵌套，所以，当在事务中输入MULTI时，redis服务器会简单返回一个错误，然后继续等待该事务的其他操作，就好像没有输入过MULTI一样

- WATCH：WATCH用于在事务开始之前监视任意数量的键，当调用EXEC执行事务时，如果任意一个监视的键被修改了，那么整个事务就不再执行，直接返回失败。【事务安全性检查】

对于上面的WATCH来说，我们可以看成一个锁。这个锁在执行期间是不可以修改（类比为打开锁）的，这样才能保证这次事务是隔离的，安全的。那么，WATCH是如何触发的呢?

在 任何对数据库键空间进行修改的命令执行成功后，multi.c/touchWatchKey函数都会被调用——它会检查数据库的watch_keys字 典，看是否有客户端在监视被修改的键，如果有的话，就把这个监视的是客户端的REDIS_DIRTY_CAS打开。之后，执行EXEC前，会对这个事务的 客户端检查是否REDIS_DIRTY_CAS被打开，打开的话就说明事务的安全性被破坏，直接返回失败；反之则正常进行事务操作。

事务的ACID性质

前面说到，事务一般具有ACID属性，但是redis只保证两种机制：一致性和隔离性。对于原子性和持久性并没有支持，下面说明redis为什么这样做。

1. 原子性：redis的单条命令是原子性的，但是redis没有对事务进行原子性保护。如果一个事务没有执行成功，是不会进行重试或者回滚的。
2. 一致性【redis保证】：这个要分三个层次：

1. 入队错误：如果执行一个错误的命令（比如命令参数不对：set key），那么会被标记为REDIS_DIRTY_EXEC,执行会直接返回错误

2. 执行错误：对某个类型key执行其他类型的操作，不会影响结果，所以不会影响事务的一致性。事务会继续进行

3. redis进程被冻结：简单来说，redis有持久化功能。但是这个持久化是建立在执行成功的基础上，如果不成功是不会进行持久化的。所以，出问题时都会保证要么事务没有执行；要么事务执行成功。所以保证了数据的一致性。

3. 隔离性【redis保证】：因为redis是单进程程序，并在执行事务时不会中断，一直执行到事务对列为空，所以隔离性是可以保证的。
4. 持久性：不管是单纯的内存模式，还是开启了持久化文件的功能，事务的每条命令执行过程中都会有时间间隙，如果这时候出现问题，持久化

还是无法保证。所以，redis使用的是事务没执行或者事务执行完成才会进行持久化工作（AOF模式除外，虽然现在还没有看到- -）

2. 订阅与发布

这 个东西没有仔细看，但是大概知道是啥功能的。我想了一下，可以使用这个功能来完成跨平台之间消息的推送。比如我开发了一个app，分别有web版本、 ios版本、Android版本、Symbian版本。那么，我可以结合模式+频道，将消息推送到所有安装此应用的平台上。

3. Lua脚本

这是redis2.6版本最大的亮点。但是我们好像木有用过- -所以，以后有需求的时候再好好研究一下吧。

4. 慢查询日志

慢 查询日志是redis系统提供的一个查看系统性能的功能。它的每一条记录的是一条命令的执行时间。所以，你可以在redis.conf中设置当超过slowlog_log_slower_than的时候，将这个命令记录下来；因为慢查询日志是一个FIFO队列（用链表实现的），所以还有一个 slowlog_max_than来限制队列长度，如果溢出，就从队头删除最旧的，将最新的添加到队尾。

五

前言

这一章是讲redis内部运作机制的，所以算是redis的核心。在这一章中，将会学习到redis是如何设计成为一个非常好用的nosql数据库的。下面我们将要讨论这些话题:

- redis是如何表示一个数据库的？它的操作是如何进行的?
- redis的持久化是怎样触发的？持久化有什么作用（mem-cache就没有）
- redis如何处理用户的输入？又试如何将运行结果返回给用户呢?
- redis 启动的时候，都需要做什么初始化工作？传入服务器的命令又是以什么方法执行的?

带 着这几个问题，我们就来学习一下redis的内部运作机制，当然，我们重点是学习它为什么要这样设计，这样设计为什么是最优的？有没有可以改进的地方呢？ 对细节不必太追究，先从整体上理解redis的框架是如何搭配的，然后对哪个模块感兴趣再去看看源码，好像2.6版本的代码量在5W行左右吧。

1. 数据库

嗯，好像一直用的都是默认的数据库。废话不说，直接上一个数据库结构：

```
typedef struct redisDb {
    //数据库编号
    int id;

    //保存数据库所有键值对数据，也成为键空间（key space）
    dict *dict;

    //保存着键的过期信息
    dict *expires;

    //实现列表阻塞原语，如BLPOP
    dict *blocking_keys;
    dict *ready_keys;

    //用于实现WATCH命令
    dict *watched_keys
}
```

主要来介绍3个属性：

- id：数据库编号，但是不是select NUM这个里面的，id这个属性是为redis内部提供的，比如AOF程序需要知道当前在哪个数据库中操作的，如果没有id来标识，就只能通过指针来遍历地址相等，效率比较低

- dict：因为redis本身就是一个键值对数据库，所以这个dict存放的就是整个数据库的键值对。键是一个string，值可以是redis五种数据结构的任意一种。因为数据库本身是一个字典，所以对数据库的操作，基本都是对字典的操作

- 键的过期时间：因为有些数据是临时的，或者不需要长期保存，就可以给它设置一个过期时间（当然，key不会同时存在在key space和expire的字典中，两者会公用一个内存块）

这其中比较好的一个是redis对于过期键的处理，我当时看到这里想，可以弄一个定时器，定期来检查expire字典中的key是否到了过期时间，但是这个定时器的时间间隔不好控制，长了的话已经过期的键还可以访问；短了的话，又注定会影像系统的性能。

- 定时删除：定时器方法，和我想法一致

- 懒惰删除：这个类似线段树的lazy操作，很巧妙（总算数据结构没白学啊。。。）

- 定期删除：上面2个都有短板，这个是结合两者的一个折中策略。它会定时删除过期key，但是会控制时间和频率，同时也会减少懒惰删除带来的内存膨胀

lazy机制：

当 你不用这个键的时候，我才懒得删除。当你访问redis的某个key时，我就检查一下这个key是否存在在expire中，如果存在就看是否过期，过期则 删除（优化是标记一下，直接返回空，然后定时任务再慢慢删除这个）；反之再去redis的dict中取值。但是缺点也有，如果用于不访问，内存就一直占 用。加入我给100万个key设置了5s的过期时间，但是我很少访问，那么内存到最后就会爆掉。

所以，redis综合考虑后采用了懒惰删除和定期删除，这两个策略相互配合，可以很好的完成CPU和内存的平衡。

2. RDB

因为当前项目用到了这个，必须要好好看看啊。战略上藐视一下，就是redis数据库从内存持久化到文件的意思。redis一共有两种持久化操作：

逐个来说，先搞定RDB。

对于RDB机制来说，在保存RDB文件期间，主进程会被阻塞，直到保存成功为止。但是这也分两种实现：

- SAVE：直接调用rdbSave，阻塞redis主进程，直到保存完成，这完成过程中，不接受客户端的请求
- BGSAVE：fork一个子进程，子进程负责调用rdbSave，并在保存完成知乎向主进程发送信号，通知保存已经完成。因为是fork的子进程，所以主进程还是可以正常工作，接受客户端的请求

整个流程可以用伪代码表示：

```
def SAVE():
    rdbSave()

def BGSAVE():
    pid = fork()
    if pid == 0:
        # 子进程保存 RDB
```

```
    rdbSave()

elif pid > 0:

    # 父进程继续处理请求，并等待子进程的完成信号
    handle_request()

else:

    # pid == -1
    # 处理 fork 错误
    handle_fork_error()
```

当然，写入之后就是load了。当redis服务重启，就会将存在的dump.rdb文件重新载入到内存中，用于数据恢复，那么redis是怎么做的呢?

额，这一节重点是RDB文件的结构，如果有兴趣，可以自己去看下dump.rdb文件，然后对照一下很容易就明白了。

3. AOF

AOF是append only file的缩写，意思是追加到唯一的文件，从上面对RDB的介绍我们知道，RDB的写入是触发式的，等待多少秒或者多少次写入才会持久化到文件中，但是AOF是实时的，它会记录你的每一个命令。

同步到AOF文件的整个过程可以分为三个阶段:

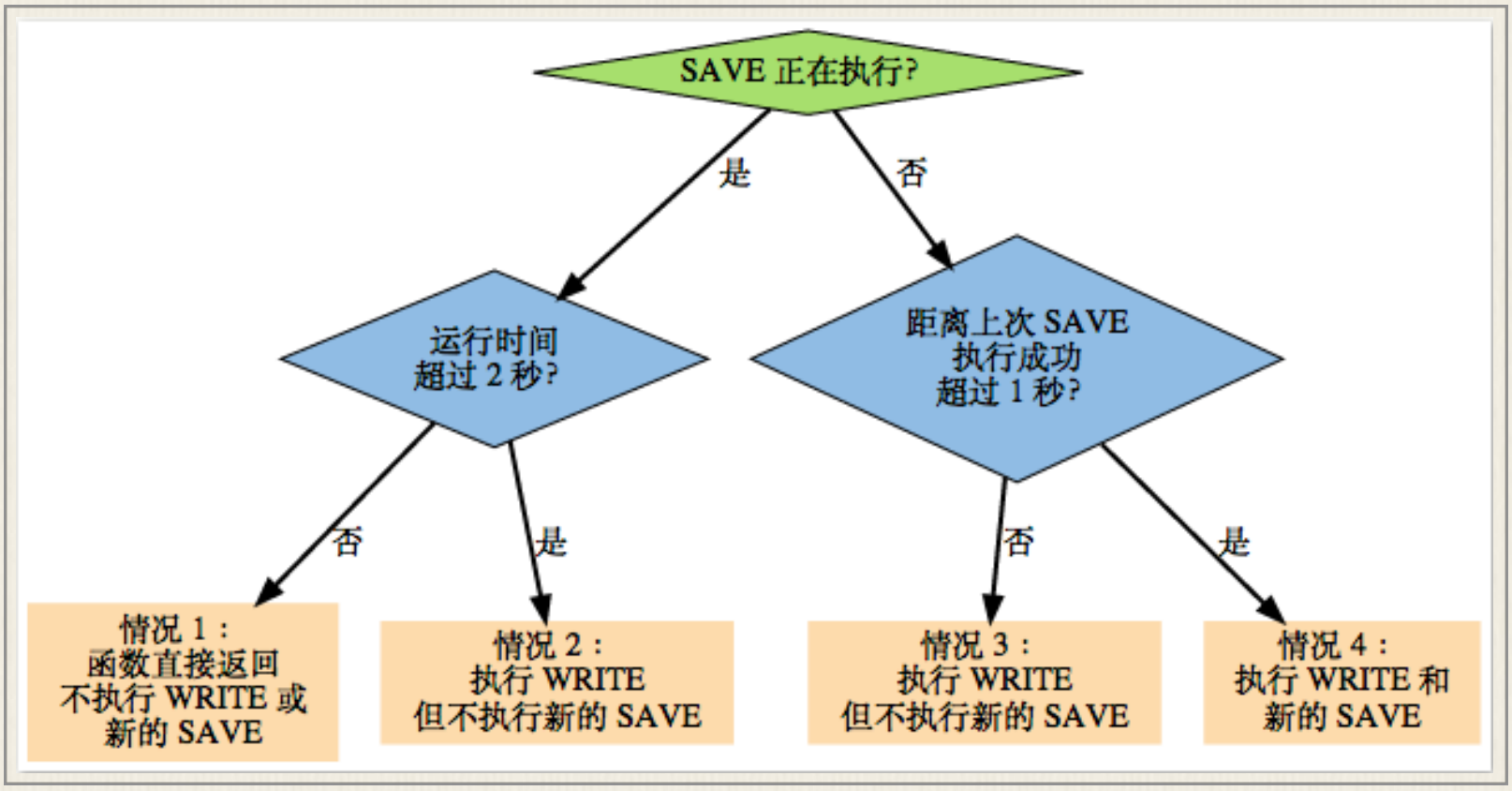
- 命令传播：redis将执行的命令、参数、参数个数都发送给AOF程序
- 缓存追加：AOF程序将收到的数据整理成网络协议的格式，然后追加到AOF的内存缓存中

- 文件写入和保存：AOF缓存中的内容被写入到AOF文件的尾巴，如果设定的AOF保存条件被满足，fsync或者或者fdatasync函数会被调用，将写入的内容真正保存到磁盘中

对于第三点我们需要说明一下，在前面我们说到，RDB是触发式的，AOF是实时的。这里怎么又说也是满足条件了呢？原来redis对于这个条件，有以下的方式：

- AOF_FSYNC_NO： 不保存。这时候，调用flushAppendOnlyFile函数的时候WRITE都会执行（写入AOF程序的缓存），但SAVE会(写入磁盘)跳过，只 有当满足：redis被关闭、AOF功能被关闭、系统要刷新缓存（空间不足等），才会进行SAVE操作。这种方式相当于迫不得已才会进行SAVE，但是很 不幸，这三种操作都会引起redis主进程的阻塞
- AOF_FSYNC_EVERYSEC：每一秒保存一次。因为SAVE是后台子线程调用的，所有主线程不会阻塞。
- AOF_FSYNC_ALWAYS：每执行一个命令保存一次。这个很好理解，但是因为SAVE是redis主进程执行的，所以在SAVE时候主进程阻塞，不再接受客户端的请求

补充：对于第二种的流程可能比较麻烦，用一个图来说明：



如果仔细看上面的条件，会发现一会SAVE是子线程执行的，一会是主进程执行的，那么怎样从根本上区分呢?

我 个人猜测是区分操作的频率，第一种情况是服务都关闭了，主进程肯定会做好善后工作，发现AOF开启了但是没有写入磁盘，于是自己麻溜就做了；第二种情况， 因为每秒都需要做，主进程不可能用一个定时器去写入磁盘，这时候用一个子线程就可以圆满完成；第三种情况，因为一个命令基本都是特别小的，所以执行一次操 作估计非常非常快，所以主进程再调用子线程造成的上下文切换都显得有点得不偿失了，于是主进程自己搞定。【待验证】

对于上面三种方式来说，最好的应该是第二种，因为阻塞操作会让 Redis 主进程无法持续处理请求，所以一般说来，阻塞操作执行得越少、完成得越快，Redis 的性能就越好。

- 模 式 1 的保存操作只会在AOF 关闭或 Redis 关闭时执行， 或者由操作系统触发， 在一般情况下， 这种模式只需要为写入阻塞， 因此它的写入性能要比后面两种模式要高， 当然， 这种性能的提高是以降低安全性为代价的： 在这种模式下， 如果运行的中途发生停机， 那么丢失数据的数量由操作系统的缓存冲洗策略决定。
- 模式 2 在性能方面要优于模式 3 ， 并且在通常情况下， 这种模式最多丢失不多于 2 秒的数据， 所以它的安全性要高于模式 1 ， 这是一种兼顾性能和安全性的保存方案。
- 模式 3 的安全性是最高的， 但性能也是最差的， 因为服务器必须阻塞直到命令信息被写入并保存到磁盘之后， 才能继续处理请求。

AOF文件的还原

对于AOF文件的还原就特别简单了，因为AOF是按照AOF协议保存的redis操作命令，所以redis会伪造一个客户端，把AOF保存的命令重新执行一遍，执行之后就会得到一个完成的数据库，伪代码如下:

```
def READ_AND_LOAD_AOF():


    # 打开并读取 AOF 文件
```

```
file = open(aof_file_name)
while file.is_not_reach_eof():

    # 读入一条协议文本格式的 Redis 命令
    cmd_in_text = file.read_next_command_in_protocol_format()

    # 根据文本命令，查找命令函数，并创建参数和参数个数等对象
    cmd, argv, argc = text_to_command(cmd_in_text)

    # 执行命令
    execRedisCommand(cmd, argv, argc)

    # 关闭文件
    file.close()
```

AOF重写

上面提到，AOF可以对redis的每个操作都记录，但这带来一个问题，当redis的操作越来越多之 后，AOF文件会变得很大。而且，里面很大一部分都是无用的操作，你如我对一个整型+1，然后-1，然后再加1，然后再-1（比如这是一个互斥锁的开 关），那么，过一段时间后，可能+1、-1操作就执行了几万次，这时候，如果能对AOF重写，把无效的命令清除，AOF会明显瘦身，这样既可以减少AOF 的体积，在恢复的时候，也能用最短的指令和最少的时间来恢复整个数据库，迫于这个构想，redis提供了对AOF的重写。

所谓的重写呢，其 实说的不够明确。因为redis所针对的重写实际上指数据库中键的当前值。AOF 重写是一个有歧义的名字，实际的重写工作是针对数据库的当前值来进行的，程序既不读写、也不使用原有的 AOF 文件。比如现在有一个列表，push了1、2、3、4，然后

删除4、删除1、加入1，这样列表最后的元素是1、2、3，如果不进行缩减，AOF会记录4次 re-
dis操作，但是AOF重写它看的是列表最后的值：1、2、3，于是它会用一条rpush 1 2 3来完成，这样由4条变为1条命令，恢复到最近的状态的代价就变为最小。

整个重写过程的伪代码如下：

```
def AOF_REWRITE(tmp_tile_name):

  f = create(tmp_tile_name)

  # 遍历所有数据库
  for db in redisServer.db:

    # 如果数据库为空，那么跳过这个数据库
    if db.is_empty(): continue

    # 写入 SELECT 命令，用于切换数据库
    f.write_command("SELECT " + db.number)

    # 遍历所有键
    for key in db:

      # 如果键带有过期时间，并且已经过期，那么跳过这个键
      if key.have_expire_time() and key.is_expired(): continue
```

```
if key.type == String:

  # 用 SET key value 命令来保存字符串键

  value = get_value_from_string(key)

  f.write_command("SET " + key + value)

elif key.type == List:

  # 用 RPUSH key item1 item2 ... itemN 命令来保存列表键

        item1, item2, ..., itemN = get_item_from_list(key)

        f.write_command("RPUSH " + key + item1 + item2 + ... + itemN)

      elif key.type == Set:

        # 用 SADD key member1 member2 ... memberN 命令来保存集合
键

        member1, member2, ..., memberN = get_member_from_set(key)

        f.write_command("SADD " + key + member1 + member2 + ... +
memberN)
```

```
        elif key.type == Hash:

            # 用 HMSET key field1 value1 field2 value2 ... fieldN valueN 命令
来保存哈希键

            field1, value1, field2, value2, ..., fieldN, valueN =\
            get_field_and_value_from_hash(key)

            f.write_command("HMSET " + key + field1 + value1 + field2 + val-
ue2 +\
                            ... + fieldN + valueN)

        elif key.type == SortedSet:

            # 用 ZADD key score1 member1 score2 member2 ... scoreN mem-
berN
            # 命令来保存有序集键

            score1, member1, score2, member2, ..., scoreN, memberN = \
            get_score_and_member_from_sorted_set(key)

            f.write_command("ZADD " + key + score1 + member1 + score2 +
member2 +\
                            ... + scoreN + memberN)
```

```
        else:

          raise_type_error()

        # 如果键带有过期时间，那么用 EXPIREAT key time 命令来保存键的过期时间
        if key.have_expire_time():
          f.write_command("EXPIREAT " + key + key.expire_time_in_unix_timestamp())

        # 关闭文件
        f.close()
```

AOF重写的一个问题：如何实现重写?

是使用后台线程还是使用子进程（redis是单进程的），这个问题值得讨论下。额，对进程线程只是概念级的，等看完之后得拿redis的进程、线程机制开刀好好学一下。

redis肯定是以效率为先，所以不希望AOF重写造成客户端无法请求，所以redis采用了AOF重写子进程执行，这样的好处有：

1. 子进程对AOF重写时，主进程可以继续执行客户端的请求
2. 子进程带有主进程的数据副本，使用子进程而不是线程，可以在避免锁的情况下，保证数据的安全性

当然，有有点肯定有缺点：

- 因为子进程在进行AOF重写时，主进程没有阻塞，所以肯定继续处理命令，而这时候的命令会对现在的数据修改，这些修改也是需要写入AOF文件的。这样重写的AOF和实际AOF会出现数据不一致。

为 了解决这个问题，redis增加了一个AOF重写缓存（在内存中），这个缓存在fort出子进程之后开始启用，redis主进程在接到新的写命令之后，除了会将这个写命令的协议内容追加到AOF文件之外，还会同时追加到这个缓存中。这样，当子进程完成AOF重写之后，它会给主进程发送一个信号，主进程接收 信号后，会将AOF重写缓存中的内容全部写入新AOF文件中，然后对新AOF改名，覆盖老的AOF文件。

在整个AOF重写过程中，只有最后的写入缓存和改名操作会造成主进程的阻塞（要是不阻塞，客户端请求到达又会造成数据不一致），所以，整个过程将AOF重写对性能的消耗降到了最低。

AOF触发条件

最后说一下AOF是如何触发的，当然，如果手动触发，是通过BGREWRITEAOF执行的。如果要用redis的自动触发，就要涉及下面3个变量（AOF的功能要开启哦 appendonlyfile yes）：

- 记录当前AOF文件大小的变量aof_current_size
- 记录最后一次AOF重写之后，AOF文件大小的变量aof_re-write_base_size
- 增长百分比变量aof_rewrite_perc

每当serverCron函数（redis的crontab）执行时，会检查以下条件是否全部满足，如果是的话，就会触发自动的AOF重写：

1. 没有 BGSAVE 命令在执行
2. 没有 BGREWRITEAOF 在执行
3. 当前AOF文件大小 > server.aof_rewrite_min_size(默认为1MB)
4. 当前AOF文件大小和最后一次AOF重写后的大小之间的比率大于等于指定的增长百分比(默认为1倍，100%)

默认情况下，增长百分比为100%。也就是说，如果前面三个条件已经满足，并且当前AOF文件大小比最后一次AOF重写的大小大一倍就会触发自动AOF重写。

原文链接：http://github.thinkingbar.com/tags/#redis

AFNetworking2.0源码解析

作者：bang

本篇我们继续来看看AFNetworking的下一个模块 — AFURLRequestSerialization。

AFURLRequestSerialization用于帮助构建NSURLRequest，主要做了两个事情：

1.构建普通请求：格式化请求参数，生成HTTP Header。

2.构建multipart请求。

分别看看它在这两点具体做了什么，怎么做的。

1.构建普通请求

A.格式化请求参数

一般我们请求都会按key=value的方式带上各种参数，GET方法参数直接加在URL上，POST方法放在body 上，NSURLRequest没有封装好这个参数的解析，只能我们自己拼好字符串。AFNetworking提供了接口，让参数可以是 NSDictionary, NSArray, NSSet这些类型，再由内部解析成字符串后赋给NSURLRequest。

转化过程大致是这样的：

```
@{
    @"name" : @"bang",
    @"phone": @{@"mobile": @"xx", @"home": @"xx"},
    @"families": @[@"father", @"mother"],
```

```
    @"nums": [NSSet setWithObjects:@"1", @"2", nil]
}
->
@[
    field: @"name", value: @"bang",
    field: @"phone[mobile]", value: @"xx",
    field: @"phone[home]", value: @"xx",
    field: @"families[]", value: @"father",
    field: @"families[]", value: @"mother",
    field: @"nums", value: @"1",
    field: @"nums", value: @"2",
]
    ->
    name=bang&phone[mobile]=xx&phone[home]=xx&families[]=father&families[]=mother&nums=1&num=2
```

第一部分是用户传进来的数据，支持包含NSArray,NSDictionary,NSSet这三种数据结构。

第二部分是转换成AFNetworking内自己的数据结构，每一个key-value对都用一个对象AFQueryStringPair表示，作用是最 后可以根据不同的字符串编码生成各自的key=value字符串。主要函数是AFQueryStringPairsFromKeyAndValue，详见 源码注释。

第三部分是最后生成NSURLRequest可用的字符串数据，并且对参数进行url编码，在AFQueryStringFromParametersWithEncoding这个函数里。

最后在把数据赋给NSURLRequest时根据不同的HTTP方法分别处理，对于GET/HEAD/DELETE方法，把参数加到URL后面，对于其他如POST/

PUT方法，把数据加到body上，并设好HTTP头，告诉服务端字符串的编码。

B.HTTP Header

AFNetworking帮你组装好了一些HTTP请求头，包括语言Accept-Language，根据[NSLocale preferredLanguages]方法读取本地语言，高速服务端自己能接受的语言。还有构建User-Agent，以及提供Basic Auth认证接口，帮你把用户名密码做base64编码后放入HTTP请求头。详见源码注释。

C.其他格式化方式

HTTP请求参数不一定是要key=value形式，可以是任何形式的数据，可以是json格式，苹果的plist格式，二进制protobuf格式等，AFNetworking提供了方法可以很容易扩展支持这些格式，默认就实现了json和plist格式。详见源码的类AFJSONRequestSerializer和AFPropertyListRequestSerializer。

2.构建multipart请求

构建Multipart请求是占篇幅很大的一个功能，AFURLRequestSerialization里2/3的代码都是在做这个事。

A.Multipart协议介绍

Multipart是HTTP协议为web表单新增的上传文件的协议，协议文档是rfc1867，它基于HTTP的POST方法，数据同样是放在body上，跟普通POST方法的区别是数据不是key=value形式，key=value形式难以表示文件实体，为此Multipart协议添加了分隔符，有自己的格式结构，大致如下：

—AaB03x
content-disposition: form-data; name=“name"

bang
--AaB03x

content-disposition: form-data; name="pic"; filename=“content.txt"
Content-Type: text/plain

... contents of bang.txt ...
--AaB03x--

以上表示数据name=bang以及一个文件，content.txt是文件名，… contents of bang.txt …是文件实体内容。分隔符—AaB03x是可以自定义的，写在HTTP头部里：

Content-type: multipart/form-data, boundary=AaB03x

每一个部分都有自己的头部，表明这部分的数据类型以及其他一些参数，例如文件名，普通字段的key。最后一个分隔符会多加两横，表示数据已经结束：—AaB03x—。

B.实现

接下来说说怎样构造Multipart里的数据，最简单的方式就是直接拼数据，要发送一个文件，就直接把文件所有内容读取出来，再按上述协议加上头 部和分隔符，拼接好数据后扔给NSURLRequest的body就可以发送了，很简单。但这样做是不可用的，因为文件可能很大，这样拼数据把整个文件读 进内存，很可能把内存撑爆了。

第二种方法是不把文件读出来，不在内存拼，而是新建一个临时文件，在这个文件上拼接数据，再把文件地址扔给NSURLRequest的bodyStream，这样上传的时候是分片读取这个文件，不会撑爆内存，但这样每次上传都需要新建个临时文件，对这个临时文件的管理也挺麻烦的。

第三种方法是构建自己的数据结构，只保存要上传的文件地址，边上传边拼数据，上传是分片的，拼数据也是分片的，拼到文件实体部分时直接从原来的文件 分片读取。这方法没上述两种的问题，只是实现起来也没上述两种简单，AFNetworking就是实现这第三种方法，而且还更进一步，除了文件，还可以添 加多个其他不同类型的数据，包括NSData，和Input-Stream。

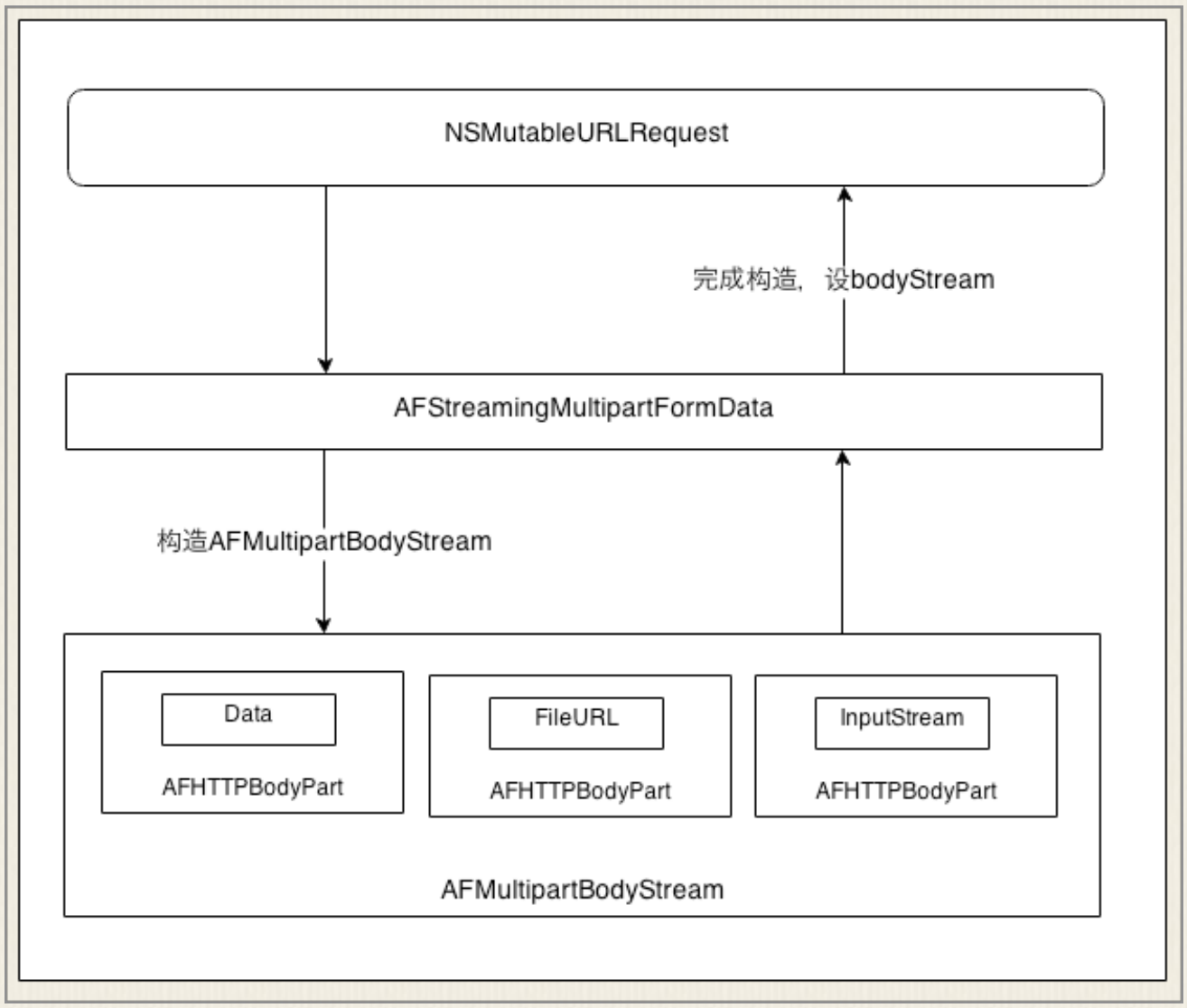
AFNetworking里multipart请求的使用方式是这样：

```
AFHTTPRequestOperationManager *manager = [AFHTTPRequestOperationManager manager];
NSDictionary *parameters = @{@"foo": @"bar"};
NSURL *filePath = [NSURL fileURLWithPath:@"file://path/to/image.png"];
[manager POST:@"http://example.com/resources.json" parameters:parameters constructingBodyWithBlock:^(id formData) {
    [formData appendPartWithFileURL:filePath name:@"image" error:nil];
} success:^(AFHTTPRequestOperation *operation, id responseObject) {
    NSLog(@"Success: %@", responseObject);
} failure:^(AFHTTPRequestOperation *operation, NSError *error) {
    NSLog(@"Error: %@", error);
}];
```

这里通过constructingBodyWithBlock向使用者提供了一个AFStreamingMultipartFormData对象， 调这个对象的几种append方法就可以添加不同类型的数据，包括FileURL/NSData /NSInputStream，AFStreamingMultipartFormData内部把这些append的数据转成不同类型的AFHTTPBodyPart，添加到自定义的AFMultipartBodyStream里。最后把AFMultipartBodyStream赋给原 来NSMutableURLRequest的bodyStream。NSURLConnection发送请求时会读取这个bodyStream，在读取数 据时会调用这个bodyStream的-read:maxLength:方法，AFMultipartBodyStream重写了这个方法，不断读取之前 append进来的AFHTTPBodyPart数据直到读完。

AFHTTPBodyPart封装了各部分数据的组装和读取，一个AFHTTPBodyPart就是一个数据块。实际上三种类型(FileURL/NSData/NSInputStream)的数据在AFHTTPBodyPart都转成NSInputStream，读取数据时只需 读这个inputStream。inputStream只保存了数据的实体，没有包括分隔符和头部，AFHTTPBodyPart是边读取变拼接数据，用 一个状态机确定现在数据读取到哪一部份，以及保存这个状态下已被读取的字节数，以此定位要读的数据位置，详见AFHTTPBodyPart的-read:maxLength:方法。

AFMultipartBodyStream封装了整个multipart数据的读取，主要是根据读取的位置确定现在要读哪一个 AFHTTPBodyPart。AFStreamingMultipartFormData对外提供友好的append接口，并把构造好的 AFMultipartBodyStream赋回给NSMutableURLRequest，关系大致如下图：



C.NSInputStream子类

NSURLRequest的setHTTPBodyStream接受的是一个NSInputStream*参数，那我们要自定义 inputStream的话，创建一个NSInputStream的子类传给它是不是就可以了？实际上不行，这样做后用NSURLRequest发出请求会导致crash，提示[xx _scheduleInCFRunLoop:forMode:]: unrecognized selector。

这是因为NSURLRequest实际上接受的不是NSInputStream对象，而是CoreFoundation的 CFReadStreamRef对象，因为CFReadStreamRef和NSInputStream是toll-free bridged，可以自由转换，但CFReadStreamRef会用到CFStreamScheduleWithRunLoop这个方法，当它调用到这 个方法时，object-c的toll-free bridging机制会调用object-c 对象NSInputStream 的相应函数，这里就调用到了 _scheduleInCFRunLoop:forMode:，若不实现这个方法就会crash。详见这篇文章。

原文链接：http://blog.cnbang.net/tech/2371/

Android 涂鸦最佳实践

作者：JackCho

Android中实现手势绘图一般都两种方式，一是直接在View上绘制，而是使用SurfaceView。两者还是有一些区别的，简单介绍下。 View：显示视图，内置画布，提供图形绘制函数、触屏事件、按键事件函数等；必须在UI主线程内更新画面，速度较慢。 SurfaceView：基于view视图进行拓展的视图类，更适合2D游戏的开发；是view的子类，使用双缓机制，在新的线程中更新画面所以刷新界面 速度比view快。所以呢，要实现涂鸦的功能优先选择后者。

在开始码代码之前，先简单理下要实现的功能。

1、可以自定义画笔的颜色

2、可以自定义画笔的粗细

3、可以实现各种常见形状的绘制

4、允许画布的回退，就是回到上一步

5、要支持橡皮擦功能

6、已作完的画，要支持保存

下面我们就逐步去实现这五个功能点。

一、关于自定义画笔的颜色和粗细，这个最简单，只须调用Paint的setColor(int color)和setStrokeWidth(float width)这两个方法即可。需要主要的是，使用SurfaceView绘图需要注意是通过SurfaceHolder获得Canvas实例，这时可以通 过Canvas实例去绘图，绘制结束调用unlockCanvasAndPost（canvas）去提交改变。

```
@Override
public void surfaceCreated(SurfaceHolder holder) {
    Canvas canvas = mSurfaceHolder.lockCanvas();
    canvas.drawColor(Color.WHITE);
    mSurfaceHolder.unlockCanvasAndPost(canvas);
    mActions = new ArrayList<Action>();
}
```

二、支持自由曲线、直线、矩形、圆形、实心矩形、实心圆形，很方便的进行扩展。这里先抽象出一个基类Action，每一次的绘制都是一个action实例，我们的画板就是一个action的列表，这样就能很好的支持回退功能。

```
//基础类
public abstract class Action {
    public int color;

    Action() {
        color = Color.BLACK;
    }

    Action(int color) {
        this.color = color;
    }

    public abstract void draw(Canvas canvas);

    public abstract void move(float mx, float my);
}
```

三、画布的回退。如果画布上的action列表大小不为0，表示画布目前是支持回退的，只须把列表中最后一个action给remove掉，重新绘制就OK了

```java
/**
 * 回退
 * @return
 */
public boolean back() {
    if (mActions != null && mActions.size() > 0) {
        mActions.remove(mActions.size() - 1);
        Canvas canvas = mSurfaceHolder.lockCanvas();
        canvas.drawColor(Color.WHITE);
        for (Action a : mActions) {
            a.draw(canvas);
        }
        mSurfaceHolder.unlockCanvasAndPost(canvas);
        return true;
    }
    return false;
}
```

四、橡皮擦。这里我取了个巧，画布的背景是白色的，所以橡皮擦的实现也是一个action，形状为自由曲线，颜色也为白色，这样就营造了一种被擦除的效果，其实只是被白色的曲线给遮盖住了。按照第三点的实现，橡皮擦也支持回退。

```java
case R.id.eraser_picker:
    mDoodle.setSize(15);
    mDoodle.setColor("#ffffff");
    break;
```

五、保存画板。画布上画满了你的各种图形，最后一步就是保存了，但是View和SurfaceView的截取是不同的，View是静态的被动 的，SurfaceView是主动的动态的，如果使用View的截图方法只能得到一个黑屏。这时好办法就是把咱们保存的action列表重新绘制出来。代码如下

```java
/**
 * 获取画布的截图
 * @return
 */
public Bitmap getBitmap() {
    bmp = Bitmap.createBitmap(getWidth(), getHeight(),
            Bitmap.Config.ARGB_8888);
    Canvas canvas = new Canvas(bmp);
    canvas.drawColor(Color.TRANSPARENT);
    for (Action a : mActions) {
        a.draw(canvas);
    }
    canvas.drawBitmap(bmp, 0, 0, mPaint);
    return bmp;
}
```

代码如下：https://github.com/JackCho/AndroidDoodle

原文链接：http://www.cnblogs.com/jack-1900/p/3952511.html

快速了解Scala技术栈

作者：张逸

我无可救药地成为了Scala的超级粉丝。在我使用Scala开发项目以及编写框架后，它就仿佛凝聚成为一个巨大的黑洞，吸引力使我不得不飞向它，以至于开始背离Java。固然Java 8为Java阵营增添了一丝亮色，却是望眼欲穿，千呼万唤始出来。而Scala程序员，却早就在享受lambda、高阶函数、trait、隐式转换等带来 的福利了。

Java像是一头史前巨兽，它在OO的方向上几乎走到了极致，硬将它拉入FP阵营，确乎有些强人所难了。而Scala则不，因为它的诞生就是OO与FP的混血儿——完美的基因融合。

“Object-Oriented Meets Functional”，这是Scala语言官方网站上飘扬的旗帜。这也是Scala的野心，当然，也是Martin Odersky的雄心。

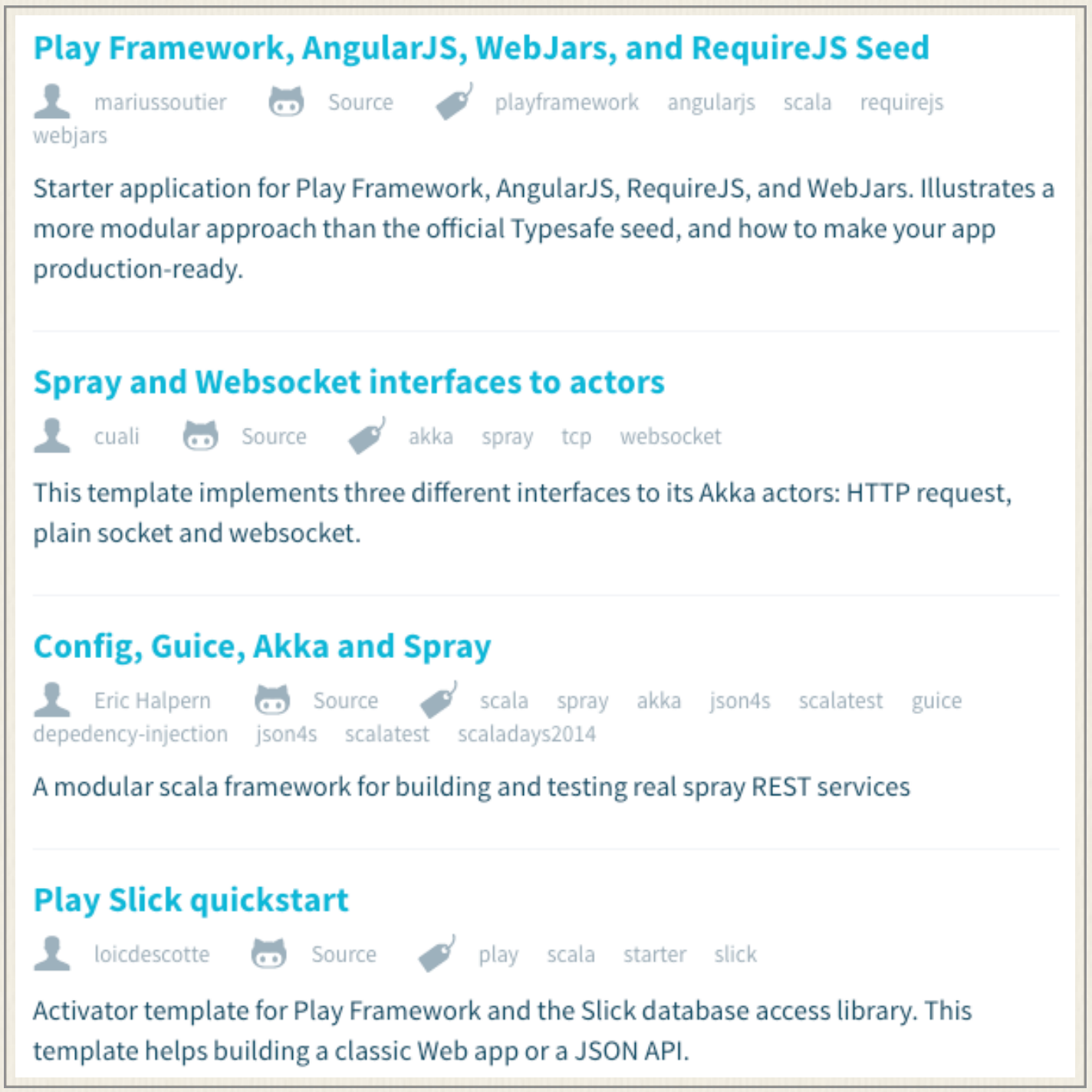
Scala社区的发展

然而，一门语言并不能孤立地存在，必须提供依附的平台，以及围绕它建立的生态圈。不如此，语言则不足以壮大。Ruby很优秀，但如果没有Ruby On Rails的推动，也很难发展到今天这个地步。Scala同样如此。反过来，当我们在使用一门语言时，也要选择符合这门语言的技术栈，在整个生态圈中找到 适合具体场景的框架或工具。

当然，我们在使用Scala进行软件开发时，亦可以寻求庞大的Java社区支持；可是，如果选择调用Java开发的库，就会牺牲掉Scala给我们 带来的福利。幸运的是，在如今，多数情况你已不必如此。伴随着Scala语言逐渐形成的Scala社区，已经开始慢慢形成相对完整的Scala技术栈。无 论是企业开发、自动化测试或者大数据领域，这些框架或工具已经非常完整地呈现了Scala开发的生态系统。

快速了解Scala技术栈

若要了解Scala技术栈，并快速学习这些框架，一个好的方法是下载typesafe推出的Activator。它提供了相对富足的基于Scala 以及Scala主流框架的开发模板，这其中实则还隐含了typesafe为Scala开发提供的最佳实践与指导。下图是Activator模板的截图：



那么，是否有渠道可以整体地获知Scala技术栈到底包括哪些框架或工具，以及它们的特性与使用场景呢？感谢Lauris Dzilums 以及其他在Github的Contributors。在Lauris Dzilums的Github上，他建立了名为awesome-scala的Repository，搜罗了当下主要的基于Scala开发的框架与工具，涉及到的领域包括：

- Database
- Web Frameworks
- i18n
- Authentication
- Testing
- JSON Manipulation
- Serialization
- Science and Data Analysis
- Big Data
- Functional Reactive Programming
- Modularization and Dependency Injection
- Distributed Systems
- Extensions
- Android
- HTTP
- Semantic Web
- Metrics and Monitoring
- Sbt plugins

是否有“乱花渐欲迷人眼”的感觉？不是太少，而是太多！那就让我删繁就简，就我的经验介绍一些框架或工具，从持久化、分布式系统、HTTP、Web框架、大数据、测试这六方面入手，作一次蜻蜓点水般的俯瞰。

持久化

归根结底，对数据的持久化主要还是通过JDBC访问数据库。但是，我们需要更好的API接口，能更好地与Scala契合，又或者更自然的ORM。如 果希望执行SQL语句来操作数据库，那么运用相对广泛的是框架

ScalikeJDBC，它提供了非常简单的API接口，甚至提供了SQL的DSL语法。例如：

```
val alice: Option[Member] = withSQL {
  select.from(Member as m).where.eq(m.name, name)
}.map(rs => Member(rs)).single.apply()
```

如果希望使用ORM框架，Squeryl应该是很好的选择。我的同事杨云在项目中使用过该框架，体验不错。该框架目前的版本为0.9.5，已经比较 成熟了。Squeryl支持按惯例映射对象与关系表，相当于定义一个POSO（Plain Old Scala Object），从而减少框架的侵入。若映射违背了惯例，则可以利用框架定义的annotation如@Column定义映射。框架提供了org.squeryl.Table[T]来完成这种映射关系。

因为可以运用Scala的高阶函数、偏函数等特性，使得Squeryl的语法非常自然，例如根据条件对表进行更新：

```
update(songs)(s =>
  where(s.title === "Watermelon Man")
  set(s.title := "The Watermelon Man",
      s.year  := s.year.~ + 1)
)
```

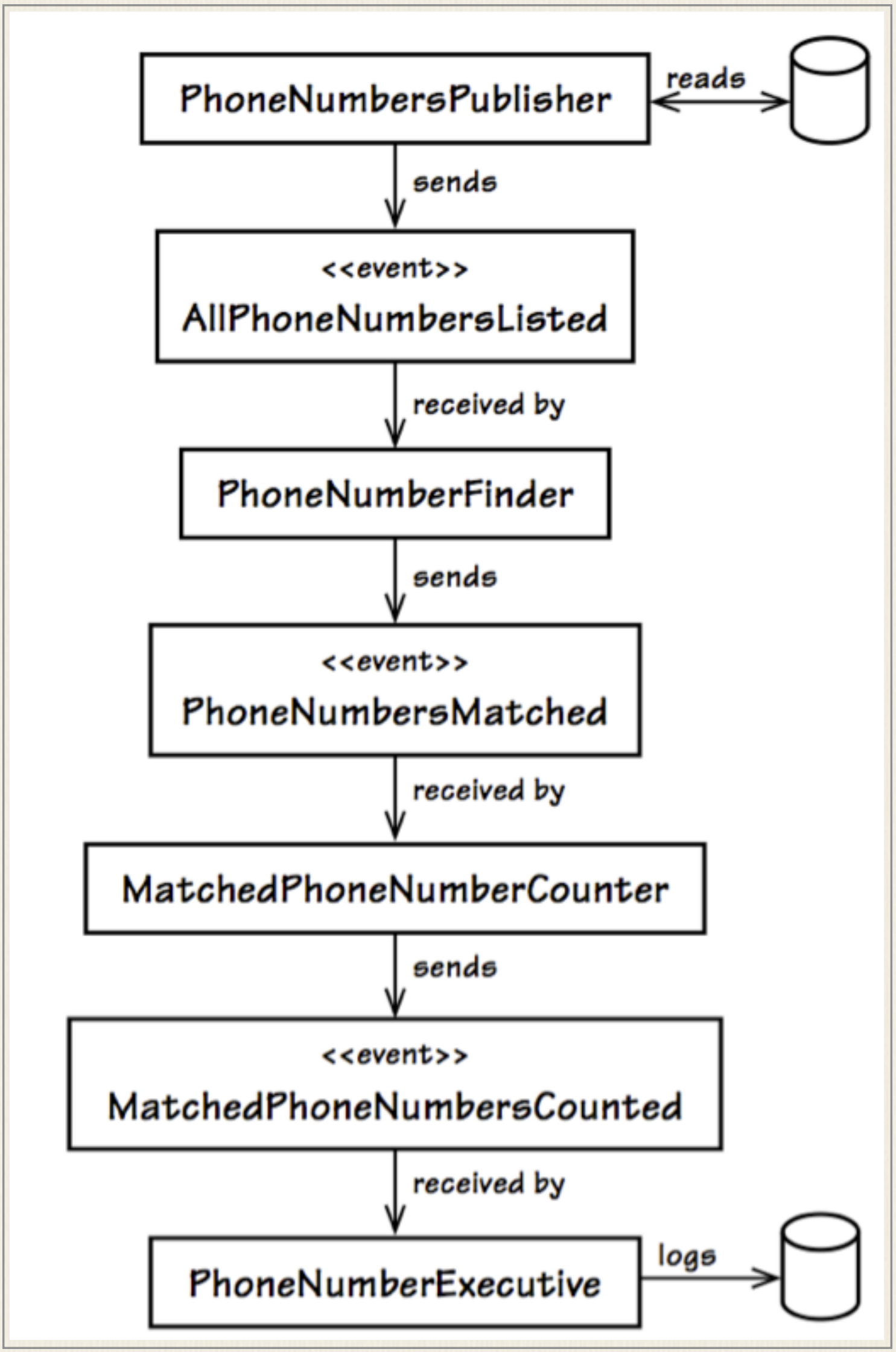
分布式系统

我放弃介绍诸如模块化管理以及依赖注入，是因为它们在Scala社区的价值不如Java社区大。例如，我们可以灵活地运用trait结合cake pattern就可以实现依赖注入的特性。因此，我直接跳过这些内容，来介绍影响更大的支持分布式系统的框架。

Finagle的血统高贵，来自过去的寒门，现在的高门大族Twitter。Twitter是较早使用Scala作为服务端开发的互联网公司，因而 积累了非常多的Scala经验，并基于这些经验推出了一些颇有影响力的框架。由于Twitter对可伸缩性、性能、并发的高要求，这些框架也极为关注这些 质量属性。Finagle就是其中之一。它是一个扩展的RPC系统，以支持高并发服务器的搭建。我并没有真正在项目中使用过Finagle，大家可以到它 的官方网站获得更多消息。

对于分布式的支持，绝对绕不开的框架还是AKKA。它产生的影响力如此之大，甚至使得Scala语言从2.10开始，就放弃了自己的Actor模 型，转而将AKKA Actor收编为2.10版本的语言特性。许多框架在分布式处理方面也选择了使用AKKA，例如Spark、Spray。AKKA的Actor模型参考了Erlang语言，为每个Actor提供了一个专有的Mailbox，并将消息处理的实现细节做了良好的封装，使得并发编程变得更加容易。AKKA很好地 统一了本地Actor与远程Actor，提供了几乎一致的API接口。AKKA也能够很好地支持消息的容错，除了提供一套完整的Monitoring机制 外，还提供了对Dead Letter的处理。

AKKA天生支持EDA（Event-Driven Architecture）。当我们针对领域建模时，可以考虑针对事件进行建模。在AKKA中，这些事件模型可以被定义为Scala的case class，并作为消息传递给Actor。借用Vaughn Vernon在《实现领域驱动设计》中的例子，针对如下的事件流：


PhoneNumbersPublisher
reads
sends
<<event>>
AllPhoneNumbersListed
received by
PhoneNumberFinder
sends
<<event>>
PhoneNumbersMatched
received by
MatchedPhoneNumberCounter
sends
<<event>>
MatchedPhoneNumbersCounted
received by
PhoneNumberExecutive
logs

我们可以利用Akka简单地实现：

```
case class AllPhoneNumberListed(phoneNumbers: List[Int])
case class PhoneNumberMatched(phoneNumbers: List[Int])
case class AllPhoneNumberRead(fileName: String)

class PhoneNumbersPublisher(actor: ActorRef) extends ActorRef {
  def receive = {
        case ReadPhoneNumbers =>
        //list phone numbers

        actor ! AllPhoneNumberListed(List(1110, ))
  }
}

class PhoneNumberFinder(actor: ActorRef) extends ActorRef {
  def receive = {
        case AllPhoneNumberListed(numbers) =>
              //match

              actor ! PhoneNumberMatched()
  }
}

val finder = system.actorOf(Prop(new PhoneNumberFinder(...)))
```

val publisher = system.actorOf(Prop(new PhoneNumbersPublish-er(finder)))

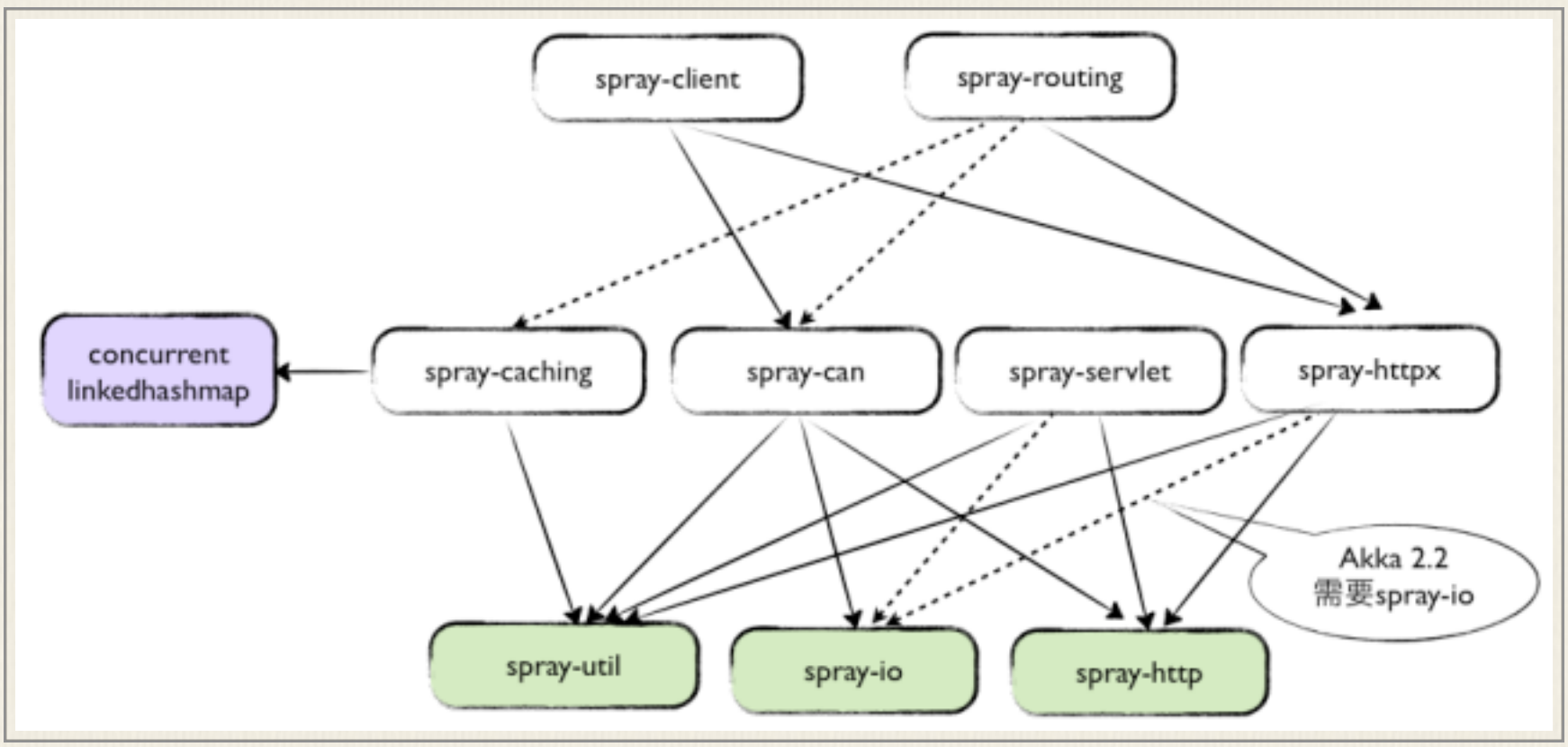
publisher ! ReadPhoneNumbers("callinfo.txt")

若需要处理的电话号码数据量大，我们可以很容易地将诸如PhoneNumbersPublisher、PhoneNumberFinder等Actors部署为Remote Actor。此时，仅仅需要更改客户端获得Actor的方式即可。

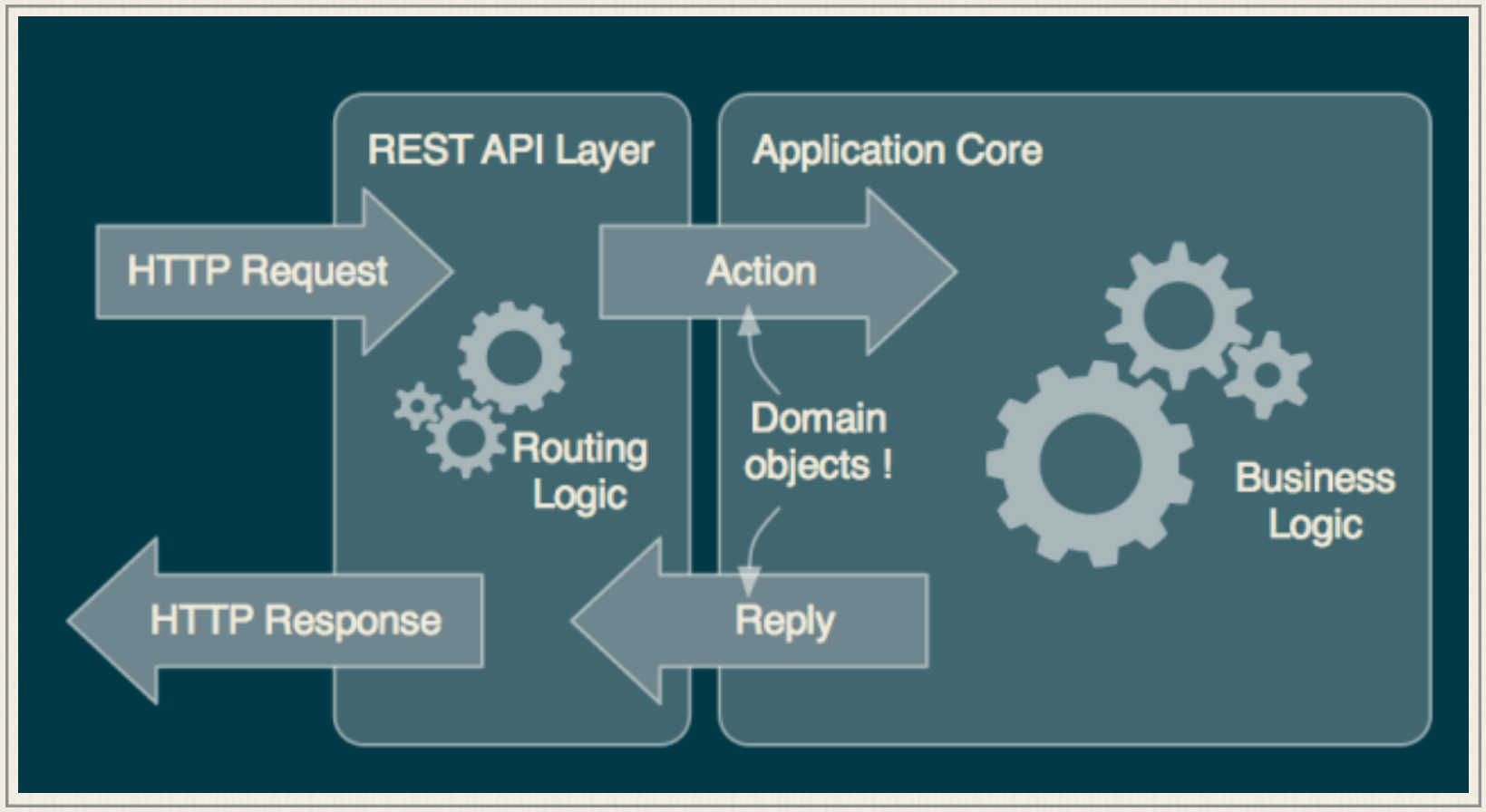
Twitter实现的Finagle是针对RPC通信，Akka则提供了内部的消息队列（MailBox），而由LinkedIn主持开发的 Kafka则提供了支持高吞吐量的分布式消息队列中间件。这个顶着文学家帽子的消息队列，能够支持高效的Publisher-Subscriber模式进 行消息处理，并以快速、稳定、可伸缩的特性很快引起了开发者的关注，并在一些框架中被列入候选的消息队列而提供支持，例如，Spark Streaming就支持Kafka作为流数据的Input Source。

HTTP

严格意义上讲，Spray并非单纯的HTTP框架，它还支持REST、JSON、Caching、Routing、IO等功能。Spray的模块及其之间的关系如下图所示：



我在项目中主要将Spray作为REST框架来使用，并结合AKKA来处理领域逻辑。Spray处理HTTP请求的架构如下图所示：



Spray提供了一套DSL风格的path语法，能够非常容易地编写支持各种HTTP动词的请求，例如：

```
trait HttpServiceBase extends Directives with Json4sSupport {
    implicit val system: ActorSystem
    implicit def json4sFormats: Formats = DefaultFormats
    def route: Route
}

trait CustomerService extends HttpServiceBase {
    val route =
```

```
            path("customer" / "groups") {
                get {
                    parameters('groupids.?) {
                        (groupids) =>
                            complete {
                                groupids match {
                                    case Some(groupIds) =>
                ViewUserGroup.queryUserGroup(groupIds.split(",").toList)
                                    case None =>
ViewUserGroup.queryUserGroup()
                                }
                            }
                    }
                }
            } ~
            path("customers" / "vip" / "failureinfo") {
                post {
                    entity(as[FailureVipCustomerRequest]) {
                        request =>
                            complete {
                                VipCustomer.failureInfo(request)
                            }
                    }
                }
```

```
        }
    }
```

我个人认为，在进行Web开发时，完全可以放弃Web框架，直接选择AngularJS结合Spray和AKKA，同样能够很好地满足Web开发需要。

Spray支持REST，且Spray自身提供了服务容器spray-can，因而允许Standalone的部署（当然也支持部署到Jetty和 tomcat等应用服务器）。Spray对HTTP请求的内部处理机制实则是基于Akka-IO，通过IO这个Actor发出对HTTP的bind消息。 例如：

```
IO(Http) ! Http.Bind(service, interface = "0.0.0.0", port = 8889)
```

我们可以编写不同的Boot对象去绑定不同的主机Host以及端口。这些特性都使得Spray能够很好地支持当下较为流行的Micro Service架构风格。

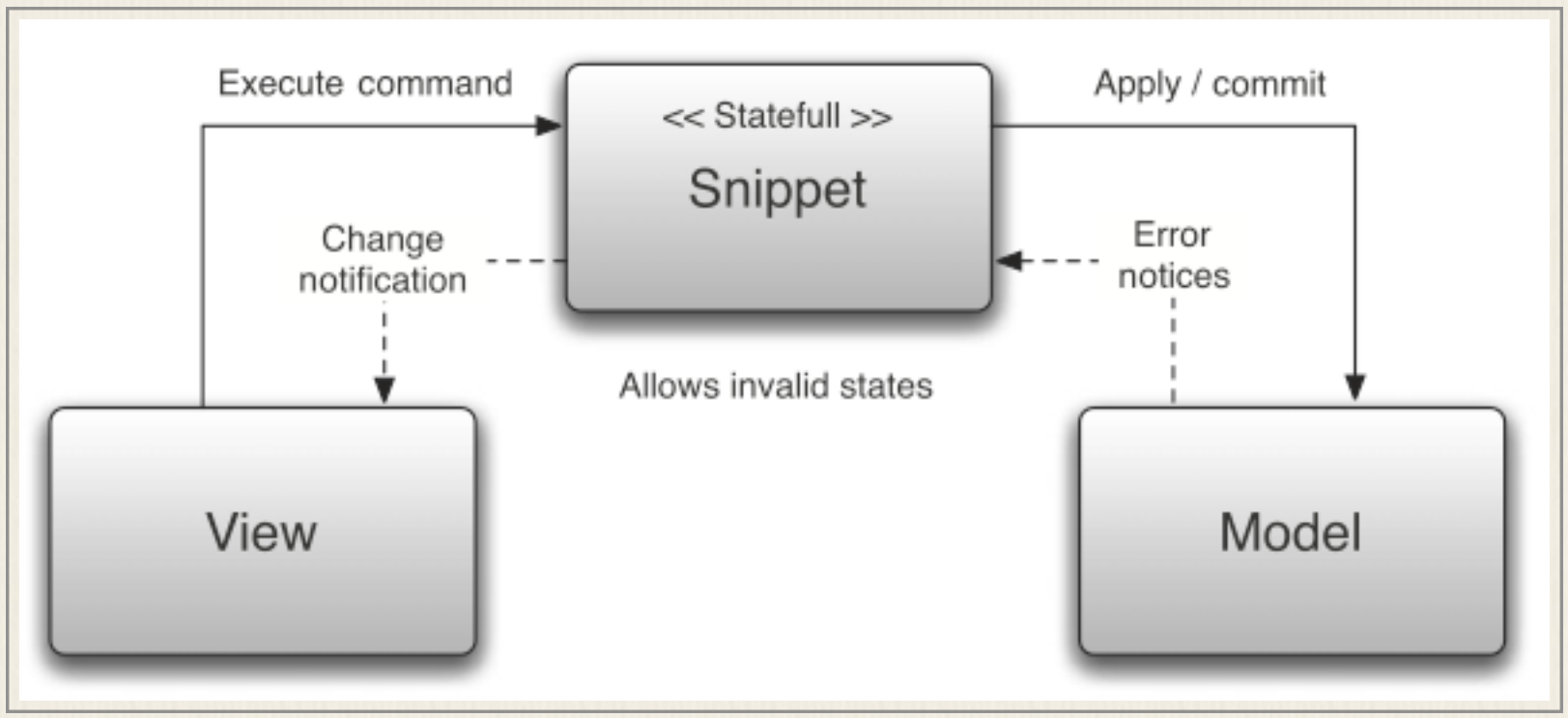
Web框架

正如前面所说，当我们选择Spray作为REST框架时，完全可以选择诸如AngularJS或者Backbone之类的JavaScript框架 开发Web客户端。客户端能够处理自己的逻辑，然后再以JSON格式发送请求给REST服务端。这时，我们将模型视为资源（Resource），视图完全 在客户端。JS的控制器负责控制客户端的界面逻辑，服务端的控制器则负责处理业务逻辑，于是传统的MVC就变化为VC+R+C模式。这里的R指的是 Resource，而服务端与客户端则通过JSON格式的Resource进行通信。

若硬要使用专有的Web框架，在Scala技术栈下，最为流行的就是Play Framework，这是一个标准的MVC框架。另外一个相对小众的Web框架是Lift。它与大多数Web框架如RoR、Struts、Django以 及Spring MVC、Play不同，采用的并非MVC模式，而是使用了所谓的View First。它驱动开发者对内容生成与内容展现（Markup）形成“关注点分离”。

Lift将关注点重点放在View上，这是因为在一些Web应用中，可能存在多个页面对同一种Model的Action。倘若采用MVC中的 Controller，会使得控

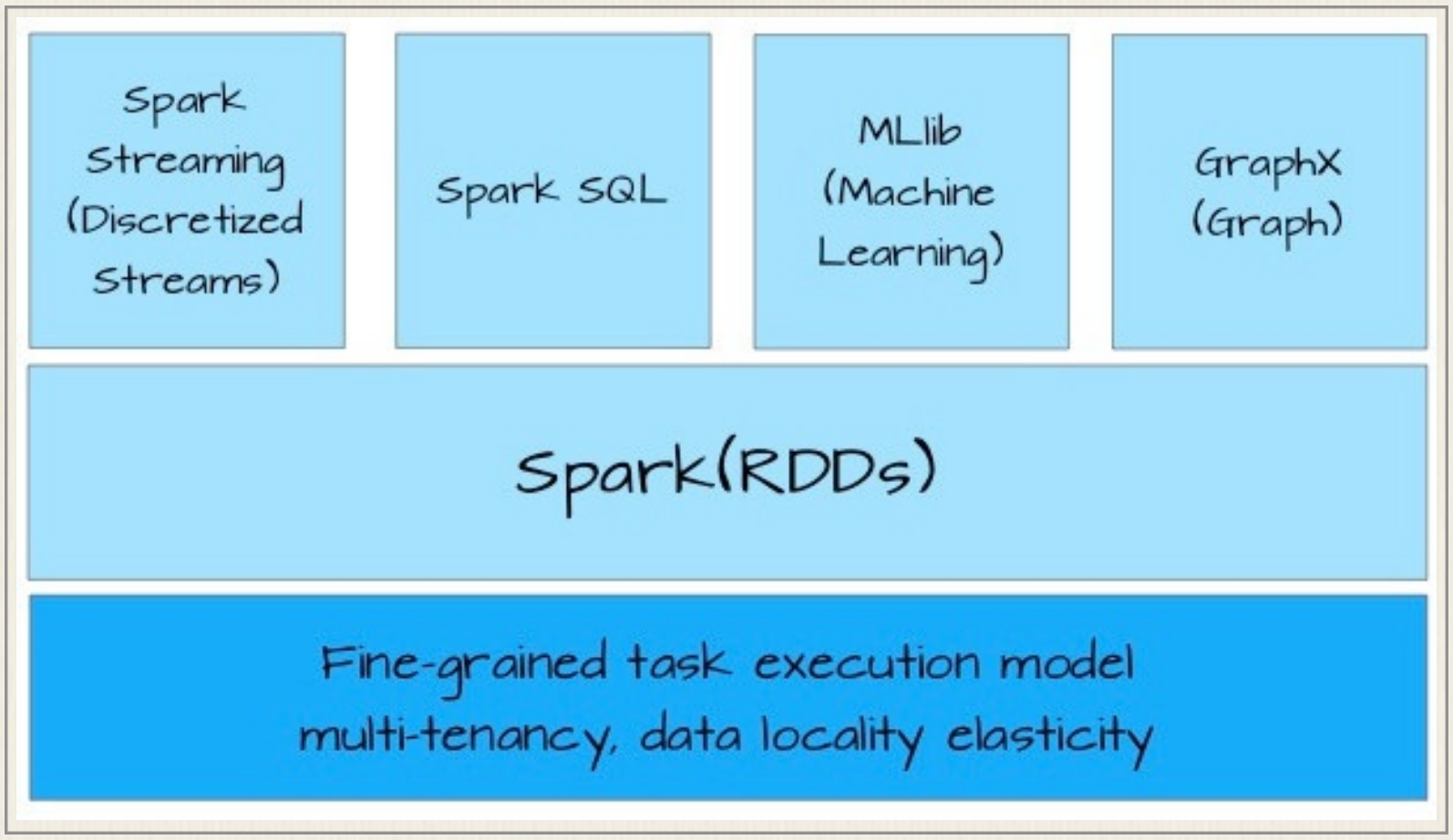
制变得非常复杂。Lift提出了一种所谓view-snippet-model（简称为VSM）的模式。



View主要为响应页面请求的HTML内容，分为template views和generated views。Snippet的职责则用于生成动态内容，并在模型发生更改时，对Model和View进行协调。

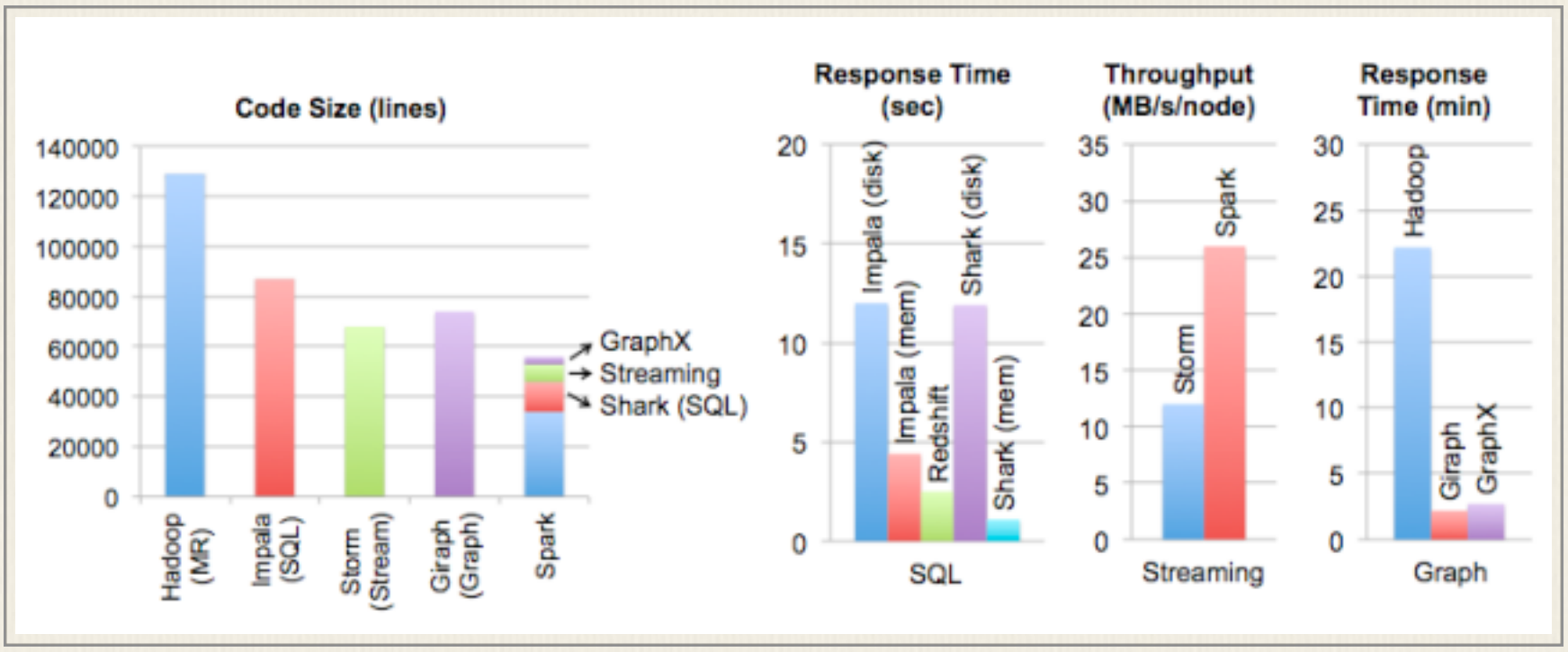
大数据

大数据框架最耀眼的新星非Spark莫属。与许多专有的大数据处理平台不同，Spark建立在统一抽象的RDD之上，使得它可以以基本一致的方式应对不同的大数据处理场景，包括MapReduce，Streaming，SQL，Machine Learning以及Graph等。这即Matei Zaharia所谓的“设计一个通用的编程抽象（Unified Programming Abstraction）。



由于Spark具有先进的DAG执行引擎，支持cyclic data flow和内存计算。因此相比较Hadoop而言，性能更优。在内存中它的运行速度是Hadoop MapReduce的100倍，在磁盘中是10倍。

由于使用了Scala语言，通过高效利用Scala的语言特性，使得Spark的总代码量出奇地少，性能却在多数方面都具备一定的优势（只有在Streaming方面，逊色于Storm）。下图是针对Spark 0.9版本的BenchMark：



由于使用了Scala，使得语言的函数式特性得到了最棒的利用。事实上，函数式语言的诸多特性包括不变性、无副作用、组合子等，天生与数据处理匹配。于是，针对WordCount，我们可以如此简易地实现：

```
file = spark.textFile("hdfs://...")


file.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
```

要是使用Hadoop，就没有这么方便了。幸运的是，Twitter的一个开源框架scalding提供了对Hadoop MapReduce的抽象与包装。它使得我们可以按照Scala的方式执行MapReduce的Job：

```
class WordCountJob(args : Args) extends Job(args) {
  TextLine( args("input") )
    .flatMap('line -> 'word) { line : String => tokenize(line) }
    .groupBy('word) { _.size }
    .write( Tsv( args("output") ) )

  // Split a piece of text into individual words.
  def tokenize(text : String) : Array[String] = {
    // Lowercase each word and remove punctuation.
    text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]", "").split("\\s+")
  }
}
```

测试

虽然我们可以使用诸如JUnit、TestNG为Scala项目开发编写单元测试，使用Cocumber之类的BDD框架编写验收测试。但在多数情 况下，我们更倾向于选择使用ScalaTest或者Specs2。在一些Java开发项目中，我们也开始尝试使用ScalaTest来编写验收测试，乃至 于单元测试。

若要我选择ScalaTest或Specs2，我更倾向于ScalaTest，这是因为ScalaTest支持的风格更具备多样性，可以满足各种不同的需求，例如传统的JUnit风格、函数式风格以及Spec方式。我的一篇博客《ScalaTest的测试风格》详细介绍了各自的语法。

一个被广泛使用的测试工具是Gatling，它是基于Scala、AKKA以及Netty开发的性能测试与压力测试工具。我的同事刘冉在InfoQ发表的文章《新一代服务器性能测试工具Gatling》对Gatling进行了详细深入的介绍。

ScalaMeter也是一款很不错的性能测试工具。我们可以像编写ScalaTest测试那样的风格来编写ScalaMeter性能测试用例，并 能够快捷地生成性能测试数据。这些功能都非常有助于我们针对代码或软件产品进行BenchMark测试。我们曾经用ScalaMeter来编写针对 Scala集合的性能测试，例如比较Vector、ArrayBuffer、ListBuffer以及List等集合的相关操作，以便于我们更好地使用 Scala集合。以下代码展示了如何使用ScalaMeter编写性能测试：

```
import org.scalameter.api._


object RangeBenchmark
extends PerformanceTest.Microbenchmark {
  val ranges = for {
    size <- Gen.range("size")(300000, 1500000, 300000)
  } yield 0 until size


  measure method "map" in {
```

```
    using(ranges) curve("Range") in {
      _.map(_ + 1)
    }
  }
}
```

根据场景选择框架或工具

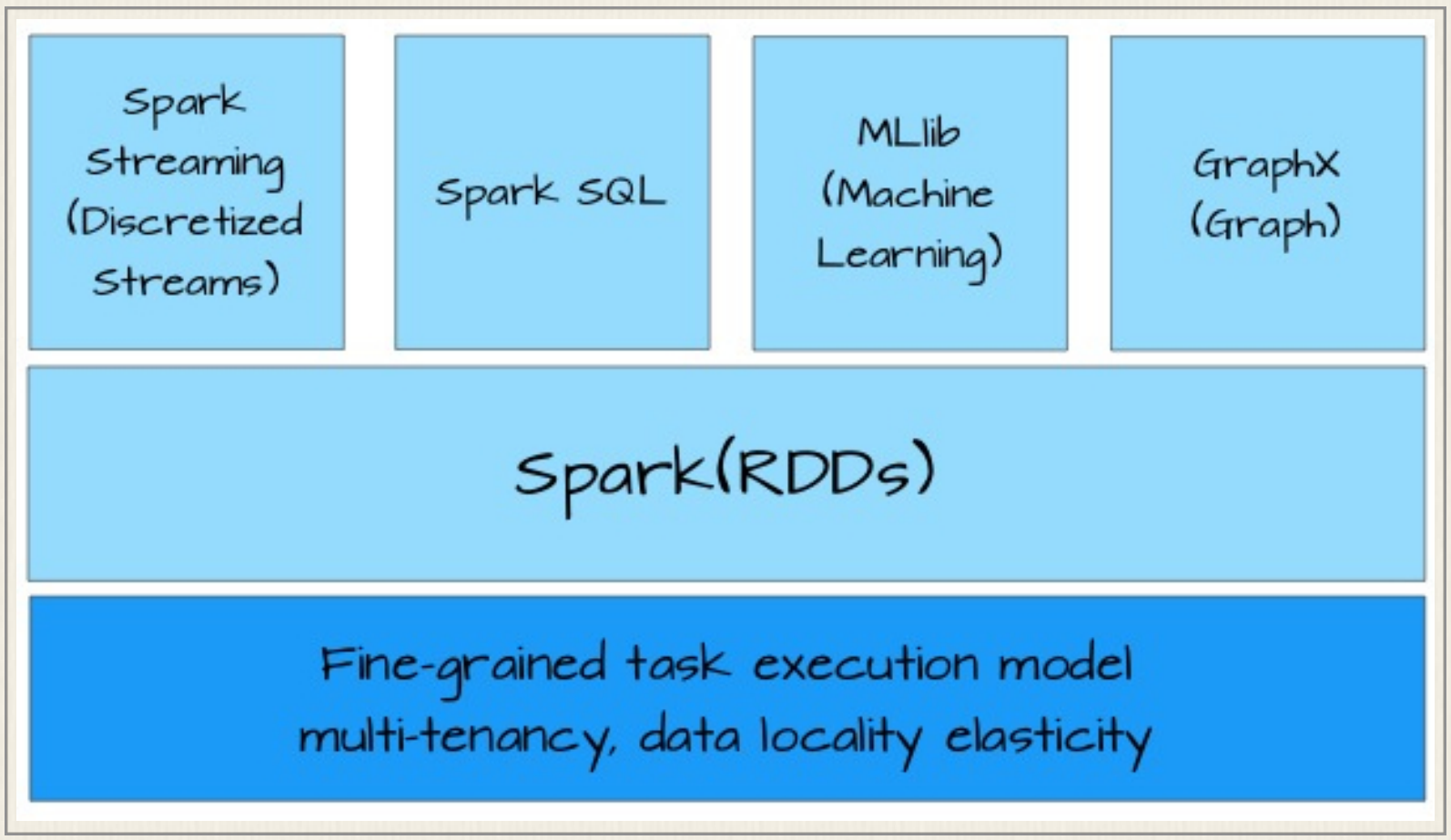
比起Java庞大的社区，以及它提供的浩如烟海般的技术栈，Scala技术栈差不多可以说是沧海一粟。然而，麻雀虽小却五脏俱全，何况Scala以 及Scala技术栈仍然走在迈向成熟的道路上。对于Scala程序员而言，因为项目的不同，未必能涉猎所有技术栈，而且针对不同的方面，也有多个选择。在选择这些框架或工具时，应根据实际的场景做出判断。为稳妥起见，最好能运用技术矩阵地方式对多个方案进行设计权衡与决策。

我们也不能固步自封，视Java社区而不顾。毕竟那些Java框架已经经历了千锤百炼，并有许多成功的案例作为佐证。关注Scala技术栈，却又不局限自己的视野，量力而为，选择合适的技术方案，才是设计与开发的正道。

原文链接：http://www.infoq.com/cn/articles/scala-technology?utm_source=tuicool

理解Spark的核心RDD

作者：张逸



与许多专有的大数据处理平台不同，Spark建立在统一抽象的RDD之上，使得它可以以基本一致的方式应对不同的大数据处理场景，包括 MapReduce，Streaming，SQL，Machine Learning以及Graph等。这即Matei Zaharia所谓的“设计一个通用的编程抽象（Unified Programming Abstraction）。这正是Spark这朵小火花让人着迷的地方。

要理解Spark，就需得理解RDD。

RDD是什么?

RDD，全称为Resilient Distributed Datasets，是一个容错的、并行的数据结构，可以让用户显式地将数据存储到磁盘和内存中，并能控制数据的分

区。同时，RDD还提供了一组丰富的操作 来操作这些数据。在这些操作中，诸如map、flatMap、filter等转换操作实现了monad模式，很好地契合了Scala的集合操作。除此之 外，RDD还提供了诸如join、groupBy、reduceByKey等更为方便的操作（注意，reduceByKey是action，而非transformation），以支持常见的数据运算。

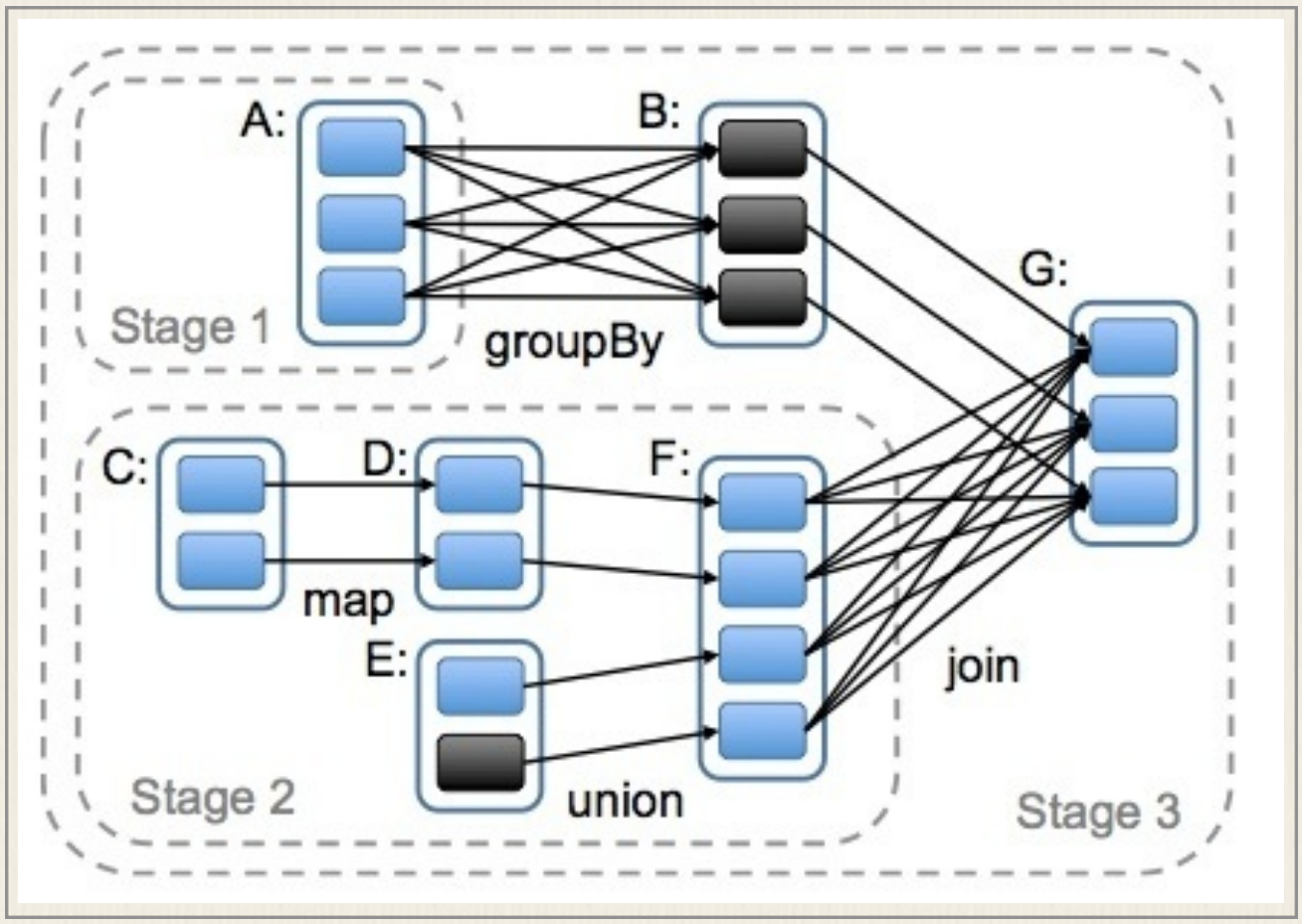
通常来讲，针对数据处理有几种常见模型，包括：Iterative Algorithms，Relational Queries，MapReduce，Stream Processing。例如Hadoop MapReduce采用了MapReduces模型，Storm则采用了Stream Processing模型。RDD混合了这四种模型，使得Spark可以应用于各种大数据处理场景。

RDD作为数据结构，本质上是一个只读的分区记录集合。一个RDD可以包含多个分区，每个分区就是一个dataset片段。RDD可以相互依赖。如 果RDD的每个分区最多只能被一个Child RDD的一个分区使用，则称之为narrow dependency；若多个Child RDD分区都可以依赖，则称之为wide dependency。不同的操作依据其特性，可能会产生不同的依赖。例如map操作会产生narrow dependency，而join操作则产生wide dependency。

Spark之所以将依赖分为narrow与wide，基于两点原因。

首先，narrow dependencies可以支持在同一个cluster node上以管道形式执行多条命令，例如在执行了map后，紧接着执行filter。相反，wide dependencies需要所有的父分区都是可用的，可能还需要调用类似MapReduce之类的操作进行跨节点传递。

其次，则是从失败恢复的角度考虑。narrow dependencies的失败恢复更有效，因为它只需要重新计算丢失的parent partition即可，而且可以并行地在不同节点进行重计算。而wide dependencies牵涉到RDD各级的多个Parent Partitions。下图说明了narrow dependencies与wide dependencies之间的区别：



本图来自Matei Zaharia撰写的论文An Architecture for Fast and General Data Processing on Large Clusters。图中，一个box代表一个RDD，一个带阴影的矩形框代表一个partition。

RDD如何保障数据处理效率?

RDD提供了两方面的特性persistence和patitioning，用户可以通过persist与patitionBy函数来控制RDD的 这两个方面。RDD的分区特性与并行计算能力(RDD定义了parallerize函数)，使得Spark可以更好地利用可伸缩的硬件资源。若将分区与持 久化二者结合起来，就能更加高效地处理海量数据。例如:

input.map(parseArticle _).partitionBy(partitioner).cache()

*partitionBy*函数需要接受一个*Partitioner*对象，如:

val partitioner = new HashPartitioner(sc.defaultParallelism)

RDD本质上是一个内存数据集，在访问RDD时，指针只会指向与操作相关的部分。例如存在一个面向列的数据结构，其中一个实现为Int的数组，

另一个实现为Float的数组。如果只需要访问Int字段，RDD的指针可以只访问Int数组，避免了对整个数据结构的扫描。

RDD将操作分为两类：transformation与action。无论执行了多少次transformation操作，RDD都不会真正执行运 算，只有当action操作被执行时，运算才会触发。而在RDD的内部实现机制中，底层接口则是基于迭代器的，从而使得数据访问变得更高效，也避免了大量 中间结果对内存的消耗。

在实现时，RDD针对transformation操作，都提供了对应的继承自RDD的类型，例如map操作会返回MappedRDD，而 flatMap则返回FlatMappedRDD。当我们执行map或flatMap操作时，不过是将当前RDD对象传递给对应的RDD对象而已。例如：

def map[U: ClassTag](f: T => U): RDD[U] = new MappedRDD(this, sc.clean(f))

这些继承自RDD的类都定义了compute函数。该函数会在action操作被调用时触发，在函数内部是通过迭代器进行对应的转换操作：

```
private[spark]
class MappedRDD[U: ClassTag, T: ClassTag](prev: RDD[T], f: T => U)
  extends RDD[U](prev) {

  override def getPartitions: Array[Partition] = firstParent[T].partitions

  override def compute(split: Partition, context: TaskContext) =
    firstParent[T].iterator(split, context).map(f)
}
```

RDD对容错的支持

支持容错通常采用两种方式：数据复制或日志记录。对于以数据为中心的系统而言，这两种方式都非常昂贵，因为它需要跨集群网络拷贝大量数据，毕竟带宽的数据远远低于内存。

RDD天生是支持容错的。首先，它自身是一个不变的(immutable)数据集，其次，它能够记住构建它的操作图（Graph of Operation），因此当执行任务的Worker失败时，完全可以通过操作图获得之前执行的操作，进行重新计算。由于无需采用replication 方式支持容错，很好地降低了跨网络的数据传输成本。

不过，在某些场景下，Spark也需要利用记录日志的方式来支持容错。例如，在Spark Streaming中，针对数据进行update操作，或者调用Streaming提供的window操作时，就需要恢复执行过程的中间状态。此时，需要 通过Spark提供的checkpoint机制，以支持操作能够从checkpoint得到恢复。

针对RDD的wide dependency，最有效的容错方式同样还是采用checkpoint机制。不过，似乎Spark的最新版本仍然没有引入auto check-pointing机制。

总结

RDD是Spark的核心，也是整个Spark 的架构基础。它的特性可以总结如下：

- 它是不变的数据结构存储
- 它是支持跨集群的分布式数据结构
- 可以根据数据记录的key对结构进行分区
- 提供了粗粒度的操作，且这些操作都支持分区
- 它将数据存储在内存中，从而提供了低延迟性

原文链接：http://www.infoq.com/cn/articles/spark-core-rdd?utm_source=tuicool

10个顶级的CSS UI开源框架

作者：小峰

随着CSS3和HTML5的流行，我们的WEB页面不仅需要更人性化的设计理念，而且需要更酷的页面特效和用户体验。作为开发者，我们需要了解一些 宝贵的CSS UI开源框架资源，它们可以帮助我们更快更好地实现一些现代化的界面，包括一些移动设备的网页界面风格设计。本文分享了10个顶级的CSS UI开源框架，有几个确实不错，一起来看看。

1、Bootstrap – 最流行的Web前端UI框架

Bootstrap是由twitter推出的Web前端UI框架，它由Twitter的设计师Mark Otto和Jacob Thornton合作开发,是一个CSS/HTML框架。它使用了最新的浏览器技术，Bootstrap提供了时尚的排版样式，表单，buttons，表 格，网格系统等等。

官方网站：http://getbootstrap.com/

2、jQuery UI - 基于jQuery的开源Javascript框架

jQuery UI是一款基于jQuery的开源Javascript框架，jQuery UI框架主要提供了用户交互、动画、特效和可更换主题的可视控件，让开发者可以更方便地实现网页交互界面，jQuery UI的整个框架比较庞大，但你也可以根据自己需要使用的功能生成适合自己的框架底层。jQuery UI界面设计非常漂亮，值得一试。

官方网站：http://jqueryui.com/

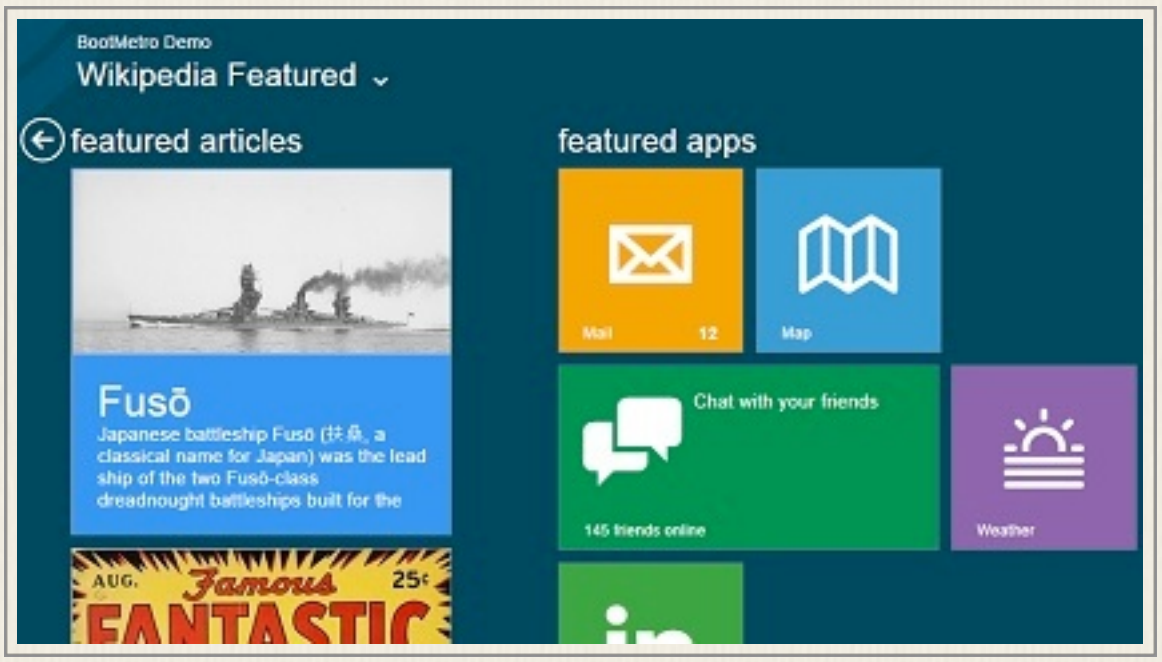
3、jQuery UI Bootstrap

它是jQuery UI和Bootstrap的集成，它是Bootstrap样式的，因此外观比较漂亮，同时它拥有jQuery UI的控件功能，这也方便开发者快速地创建一个网页控件。

官方网站：https://github.com/jquery-ui-bootstrap/jquery-ui-bootstrap/

4、BootMetro - Metro风格的CSS框架

BootMetro是一款基于Bootstrap的前端UI框架，BootMetro的特点是可以很方便地构建类似Windonws 8扁平化风格的网页界面，效果非常不错。



官方网站：http://aozora.github.io/bootmetro/

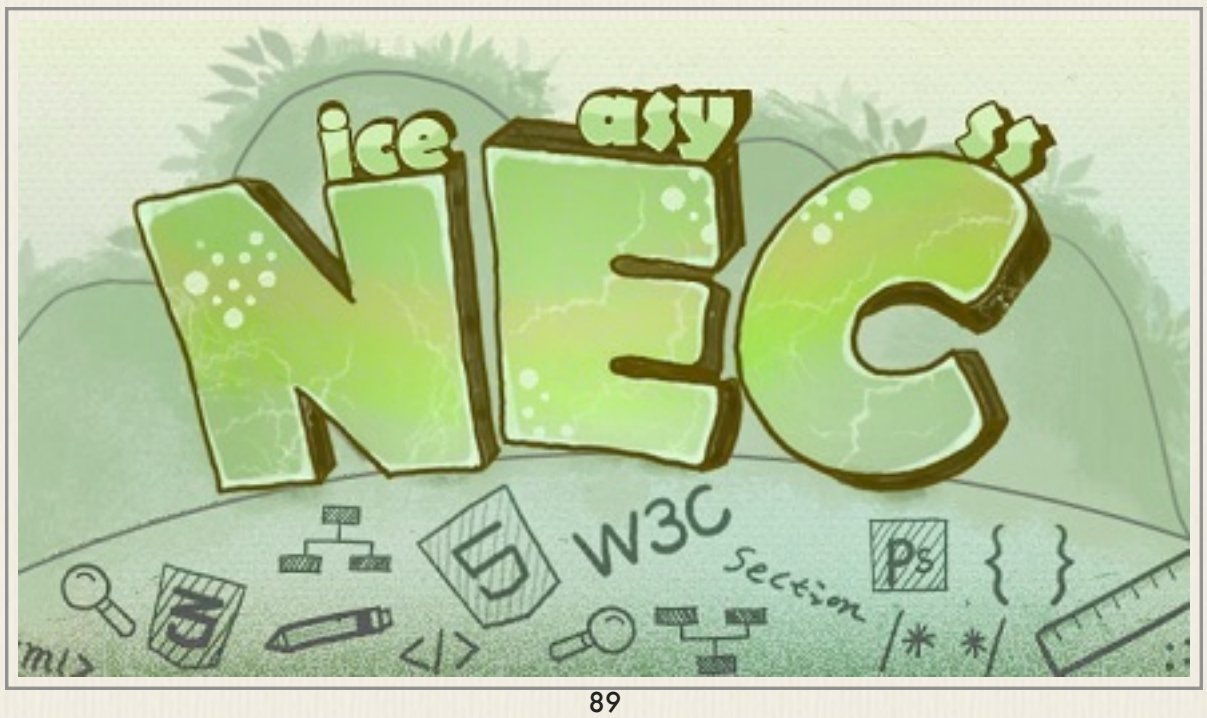
5、Flat UI - 扁平风格 UI 工具包

Flat UI是一套精美的扁平风格 UI 工具包，基于 Twitter Bootstrap 实现。这套界面工具包含许多基本的和复杂的 UI 部件，例如按钮，输入框，组合按钮，复选框，单选按钮，标签，菜单，进度条和滑块，导航元素等等。

官方网站：https://github.com/designmodo/Flat-UI

6、网易CSS框架 NEC

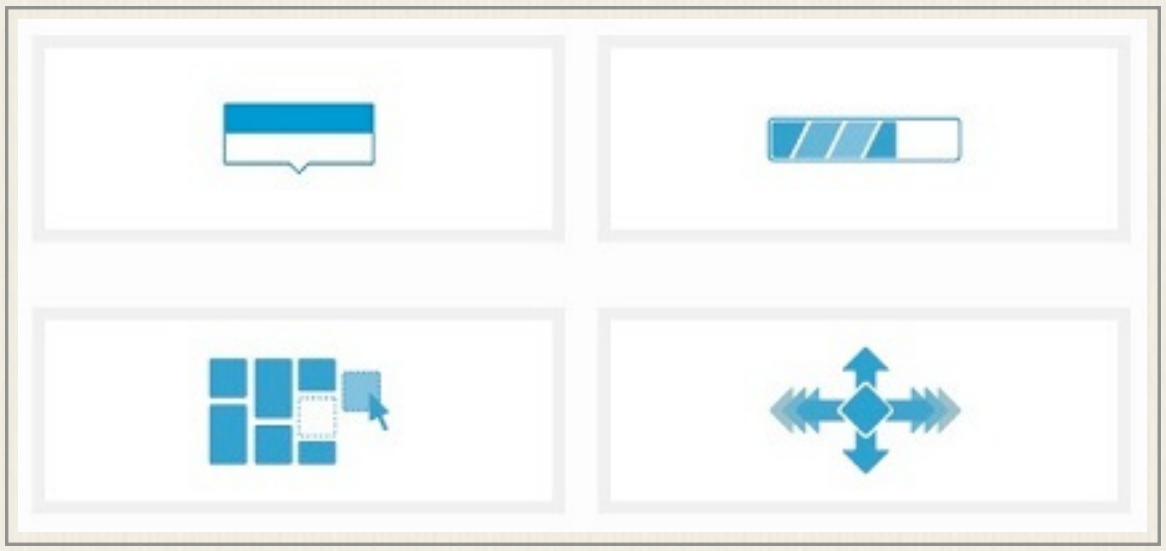
NEC是网易推出的开源前端CSS框架，NEC提供了丰富UI代码库和插件，可以极大的帮助开发人员提高开发效率。即使你并非前端专业开发人员，利用NEC你也可以快速地构建属于自己的网页应用。



官方网站：http://nec.netease.com/

7、Alloy UI – 功能强大的CSS UI框架

Alloy UI是基于YUI 3的前段UI框架，包含一套丰富的（超过60）UI 部件，如图片库，对话框，树形结构，面板，自动完成，按钮，日历控件，工具条等。



官方网站：http://alloyui.com/

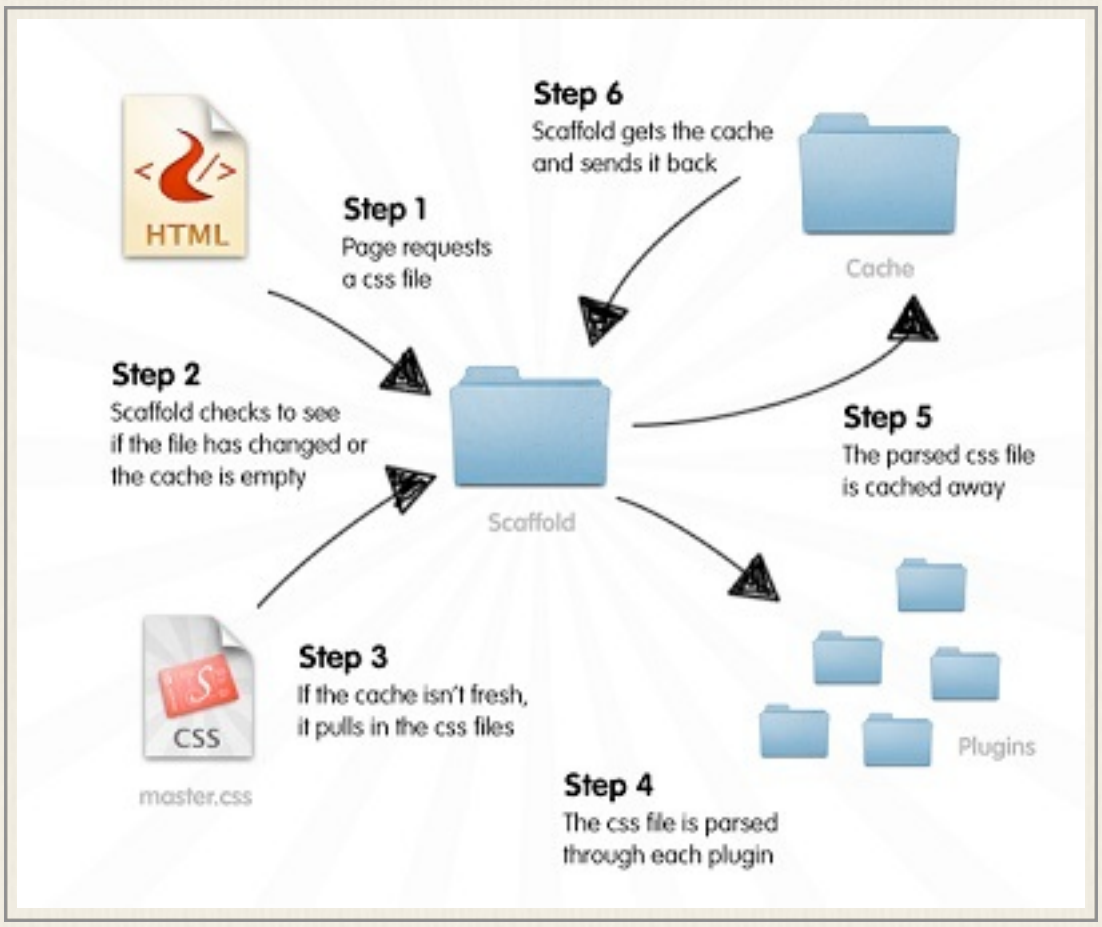
8、Cardinal – 移动端的CSS UI框架

Cardinal 是一个小型的移动优先的 CSS 框架，提供很多有用的默认样式、可缩放排版、可重用模块和一个简单的响应式表格系统。

官方网站：http://cardinalcss.com/

9、快速开发CSS的框架 CSScaffold

不同于许多CSS框架，它必须依靠PHP与Apache的mod_rewrite来执行，但也因为需要这两种东西，让CSScaffold变得很神奇、很方便，写起CSS来又快又轻松！



官方网站：http://www.w3avenue.com/2009/10/13/csscaffold-php-based-css-framework/

10、后台UI开发框架 MuseUI

一款基于bootstrap 风格，兼容于主流浏览器（包括IE6）的后端UI开发组件。

官方网站：http://git.oschina.net/muse/museui

原文链接：http://www.codeceo.com/article/top-10-css-ui-framework.html

GitHub迁移数据库，借助MySQL大行其道！

作者：伍昆

GitHub，作为广泛使用的开源代码库以及版本控制系统，其数据库MySQL性能的优劣对整个网站平台有着举足轻重的影响。接下来我们一起跟随GitHub基础架构团队的步伐，来重温去年8月做的一次重大MySQL更新，看是如何使得GitHub运行得更畅顺的。

任务简述

自去年开始，我们陆续地把GitHub 主体架构迁移到新的数据中心，与之配套的是世界级的硬件和网络环境。我们十分希望这次升迁对后端系统基石MySQL的性能也有所提高。不过在一个新环境重 新建立一个新的服务器集群和硬件平台，并不是件轻易的事情，我们必须做好计划与测试，确保迁移工作顺利完成。

准备工作

每当要进行类似的重大升级工作，对每个测量和指标量度步骤都会提出严格的要求。为新机器安装好操作系统后，接下来需要根据不同的配置来进行测试。为了得到真实的负载测试数据，我们使用了tcpdump对从旧集群系统到新系统执行的SELECT查询进行抓包分析。

MySQL性能调整可谓是细节决定成败，例如众所周知的innodb_buffer_pool_size的设置，对整体有着举足轻重的影响。为了尽更全面地管控升级过程，我们把 innodb_thread_concurrency、innodb_io_capacity、innodb_buffer_pool_instances等参数也一并进行分析和研究。

每次测试时我们都只改变某一个参数，然后让系统连续运行至少12 小时。在这过程中不断观察SHOW ENGINE INNODB STATUS带来的统计信息，其中SEMAPHORES栏目，能很好地反映工作负荷竞争情况。当相关

设置测试通过后，接下来我们将尝试把其中一个最大的数 据表迁移到一个单独的集群上。作为前期测试的一部分，这样的迁移工作能为日后更大更核心的变更带来指引。

除了对基础硬件部分进行了升级，我们还对流程和拓扑进行了优化。例如：延后复制，更快速和高频的备份，提高备带读取能力。一切就绪后，将进入最后的升级阶段。

制定升级项目清单，进行二次检查

作为每天服务上百万用户的平台，任何差错都将是毁灭性的。我们在进行真实切换前，列出了一个任务清单，确保各项工作有序执行：

- 确保缓冲池在新集群中成功预热
- 在推特等社交平台公告维护开始
- 把网站转为维护等待模式
- 等候所有与旧MySQL服务器相关的通信终止
- 把旧服务器设为只读模式
- 从旧集群中移除主要和复制的VIPs数据
- 确认所有写入操作已经终止
- 终止cluster1的复制
- 获取cluster1复制的位置，并告知当前线程
- 重置cluster1的复制
- 关闭cluster1的只读模式
- 把旧集群连接到新的cluster1集群
- 按照cluster1连接配置进行应用程式部署
- 确保新连接能通过新集群
- 检查后台工具resque的任务(workers)

- 进行阶段检查并确保一切正常
- 把网站转为正常模式
- 在推特等社交平台公告维护结束
- 把https://github.com/github/xxxx/pull/xxx整合到主页面

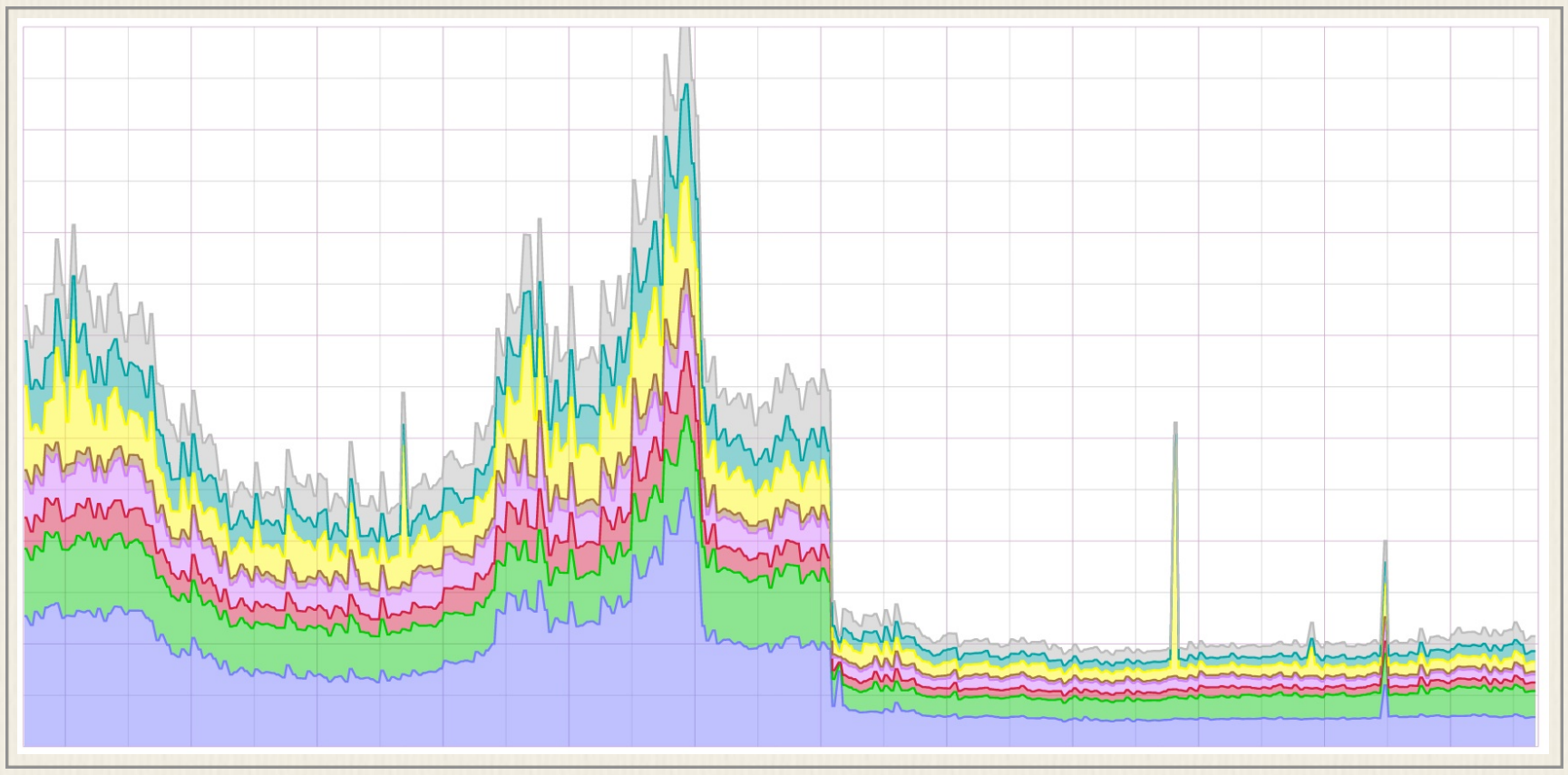
迁移当天

在周六的上午5点太平洋时间)，团队成员就位后，迁移工作正式开始。当用户在这段时间访问网站时，会收到如下的提示:

13分钟后，新集群即将开始正常运作。我们终止了网站的维护模式，并告知大众网站将回复正常。

效果评估

在接下来的几个星期，我们密切关注了整体性能和响应时间方面的变化。结果是令人欣喜的，页面载入时间减少了将近三分之二：



经验总结：

1. 功能划分

在本次操作中，我们把较大的历史数据记录表放入单独的集群，事后证明这是明智的做法—很好地释放了存储空间和缓冲池空间。同时，能够把更多资源放在活跃数据处理上，连接逻辑的划分也使得程序可以在多个集群间进行查询。以后我们还将采取该方法进行升级。

2. 不断测试

罗马非一天建成，整个过程需要不断进行验收和回归测试，避免意外的发生。

3. 团队的力量

如此重大的架构升级需要很多小伙伴协力工作，我们主要使用GitHub上的拉请求功能来进行互动交流。部署团队来自世界各地：

当开启一个拉请求后，我们将进行实时交流，例如：错误处理，回归处理等信息的交流。每个交流环节都生成一个URL，方便进行历史查询和反馈。

一年后……

路遥知马力，一年后，实践证明这是一次成功的操作。MySQL持续表现符合预期，系统可靠性进一步提高。还有个附加好处是：新系统的扩展性得到提升，将来可进行更大规模的升级和改造。

原译文链接：http://www.csdn.net/article/2014-09-08/2821581-GitHub-MySQL

原文链接：https://github.com/blog/1880-making-mysql-better-at-github

Swift内存管理-示例讲解

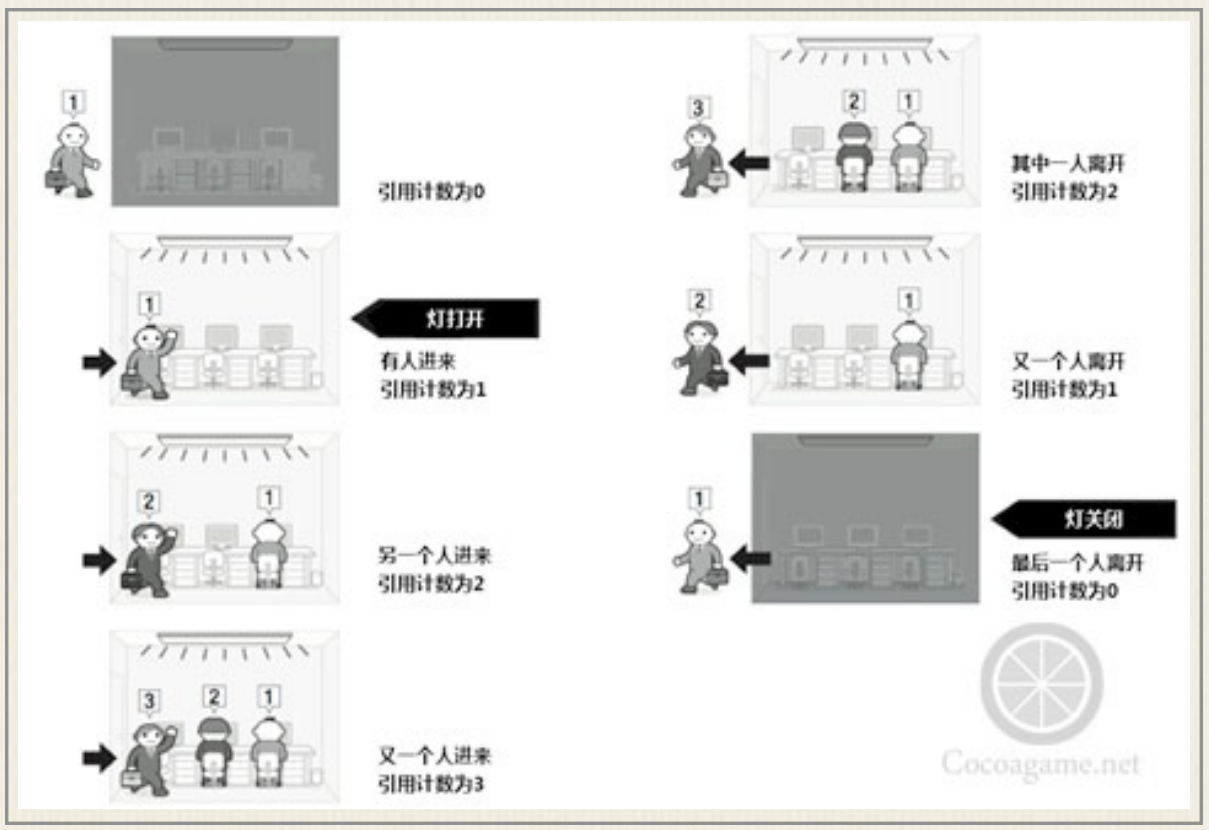
作者：关东升

具体而言，Swift中的ARC内存管理是对引用类型的管理，即对类所创建的对象采用ARC管理。而对于值类型，如整型、浮点型、布尔型、字符串、元组、集合、枚举和结构体等，是由处理器自动管理的，程序员不需要管理它们的内存。

一、引用计数

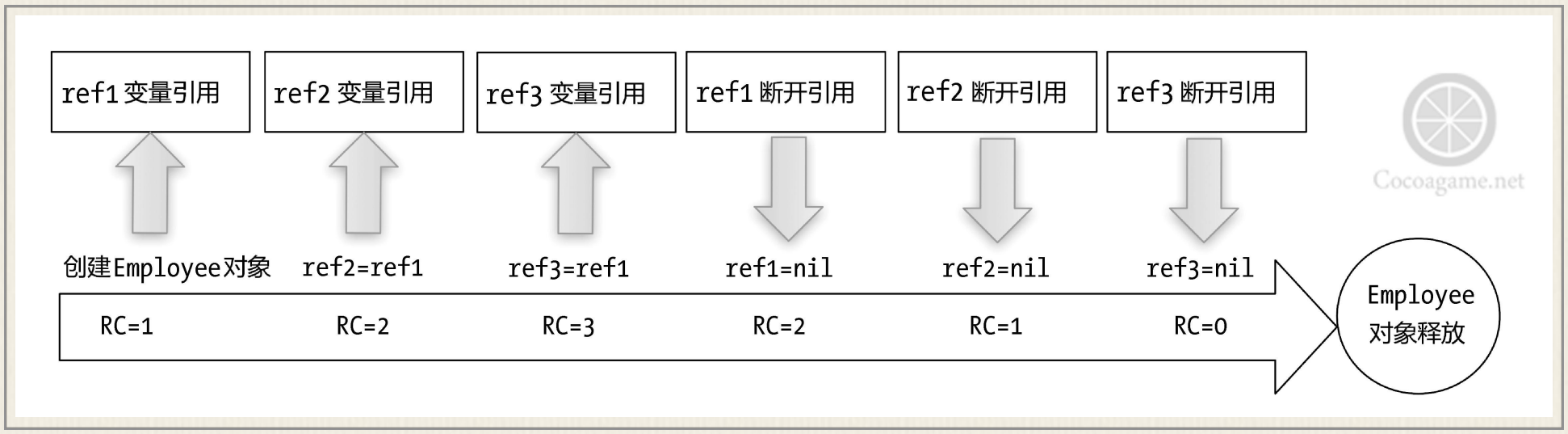
每 个Swift类创建的对象都有一个内部计数器，这个计数器跟踪对象的引用次数，称为引用计数（Reference Count，简称RC）。当对象被创建的时候，引用计数为1，每次对象被引用的时候会使其引用计数加1，如果不需要的时候，对象引用断开（赋值为 nil），其引用计数减1。当对象的引用计数为0的时候，对象的内存才被释放。

下图是内存引用计数原理示意图。图中的房间就好比是对象的内存，一个人进入房间打开灯，就是创建一个对象，这时候对象的引用计数是1。有人进入房间，引用计数加1；有人离开房间，引用计数减1。最后一个人离开房间，引用计数为0，房间灯关闭，对象内存才被释放。



二、示例：Swift自动引用计数

下面我们通过一个示例了解一下Swift中的自动引用计数原理。下图是Employee类创建的对象的生命周期，该图描述了对象被赋值给3个变量，以及它们的释放过程。



示例代码如下：

```
1.  class Employee {              ①
2.  var no : Int
3.  var name : String
4.  var job : String
5.  var salary : Double
6.
7.
8.  init(no : Int, name: String, job : String, salary : Double) {   ②
9.      self.no = no
10.     self.name = name
11. self.job = job
12. self.salary = salary
```

```
13.  println("员工\(name) 已经构造成功。")    ③
14.      }
15.  deinit {                          ④
16.  println("员工\(name) 已经析构成功。")    ⑤
17.      }
18.  }
19.
20.
21.  var ref1: Employee?          ⑥
22.  var ref2: Employee?          ⑦
23.  var ref3: Employee?          ⑧
24.
25.
26.
ref1 = Employee(no: 7698, name: "Blake", job :"Salesman", salary : 1600)
  ⑨
27.
28.
29.  ref2 = ref1                  ⑩
30.  ref3 = ref1                  ⑪
31.
32.
33.  ref1 = nil                   ⑫
34.  ref2 = nil                   ⑬
```

35. ref3 = nil ⑭

上述代码第①行声明了Employee类，第②行代码是定义构造器，在构造器中初始化存储属性，并且在代码第③行输出构造成功信息。第④行代码是定义析构器，并在代码第⑤行输出析构成功信息。

代 码第⑥~⑧行是声明3个Employee类型变量，这个时候还没有创建Employee对象分配内存空间。代码第⑨行是真正创建Employee对象分配内存空间，并把对象的引用分配给ref1变量，ref1与对象建立“强引用”关系，“强引用”关系能够保证对象在内存中不被释放，这时候它的引用计数是1。第⑩行代码ref2 = ref1是将对象的引用分配给ref2，ref2也与对象建立“强引用”关系，这时候它的引用计数是2。第⑪行代码ref3 = ref1是将对象的引用分配给ref3，ref3也与对象建立“强引用”关系，这时候它的引用计数是3。

然后在代码第⑫行通过ref1 = nil语句断开ref1对Employee对象的引用，这时候它的引用计数是2。以此类推，ref2 = nil时它的引用计数是1，ref3 = nil时它的引用计数是0，当引用计数为0的时候Employee对象被释放。

我们可以测试一下看看效果，如果设置断点单步调试，会发现代码运行完第⑨行后控制台输出：

员工Blake 已经构造成功。

析构器输出的内容直到运行完第⑭行代码才输出：

员工Blake 已经析构成功。

这说明只有在引用计数为0的情况下才调用析构器，释放对象。

原文链接：http://blog.csdn.net/tonny_guan/article/details/39058205