函数式编程思想

赵杨，复旦大学

摘要

近年来，由于计算机单体性能不再符合摩尔定律[1]，而用户量仍然不断地在继续增长，为了解决对于高并发的需求，越来越多的分布式软件[2]被开发出来。同时，人工智能的火爆，也使得函数式编程思想在处理数据流上的优势得到了体现。函数式编程的引用透明与没有副作用的特性使得其在许多分布式软件构建体系中发挥着重要的作用。在软件变得越来越庞大，愈发倾向于分布式体系的今日，能够高度抽象并且无状态的函数式编程思想也随之变得愈发关键。

1.简介

本篇论文主要是为了阐述了函数式编程思想的一些显著特征，并且介绍一些拥有者函数式编程思想的语言和与之相关的已被投入实践应用的现实项目。

函数式编程语言自从上世纪以来就已经被人们所提及，在过去，计算机最初被设计出来就是为了解决数学上的计算问题[3]，而在数学领域内，最普遍可以看到的便是随处可以看到的函数，这也被应用到了程序设计的理念中，例如最为古老的 lisp[4]。一般来说，使用函数式思想设计的程序是没有副作用的，即其没有状态，在任何时候对其使用一段程序，它只是计算出了表达式的值，在任何时候都可以对其进行求值操作，并且值不会发生改变，这也被称作“引用透明”。这一特性使得程序员可以不用过多地将心思控制程序的流序列。这一特点在当今的分布式系统中尤为有效。

函数式编程的另一特点是其高度的抽象，它可以很好地将一个个非常细小的步骤分割后并写成一段独立的更为精细的函数，这样的好处就是我们可以将程序分解成许多特定的模块，进行随意的分拆与组装。举例来说，在工业中，为了制造一辆车子，我们通常会将其拆分成许多模块，轮子、车门、发动机等等，许多零件还会更加细分地进行拆分，而将其各个零部件生产好后，再到流水线上进行统一组装。这样做的好处就是之后任一组件想要更替都可以对其进行修改，而不用将整辆车的组装流程更改，只需要更改一个小部件即可。而函数式编程的思想也是如此，我们将我们的程序分为各个细小的模块，完成其需要完成的较小的工

作，这样在程序变大后，我们仍然可以很好地进行维护，若有更改的需求，可以对某一部分的函数进行更改就可以，达到了低耦合性的效果。除此以外，这样的设计可以让我们更加良好地理解程序的流程，我们将许多部分分解成特定功能的函数后，能够让我们更加清晰地理解程序的某一部分完成了什么。这样的优势在大型项目上尤为显著，可以令我们非常良好地理解程序的构造，并且更好地设计程序的结构体系。

2.函数式编程特性

函数式编程的语言有着许多特性，例如函数式编程中的高阶函数、惰性计算、Monad[5]等概念，这些函数式编程的特性使得函数式编程具有着更加高度的抽象能力，也能够令函数式语言有着更加高效的运行效率，帮助我们的代码变得更为简洁有效。

2.1 高阶函数

在函数式编程中，高阶函数基本是随处可见的。函数式编程的一大特点就是可以使用高阶函数，即一个函数可以将函数作为参数传入，也可以返回一个函数作为函数的结果。这样的特性使得人们使用函数式语言时可以非常轻松地将各类过程进行模块化地拆分与组装，使得程序实现非常高的可复用性与低耦合性。

为了更好地理解高阶函数的原理，我们可以通过基本所有函数式编程语言中都有的函数 map（f， list）来理解，在函数式语言中，map（f， list）的作用是传入一个函数与一个列表，而后将传入的函数作用与各个列表之上，这样子的抽象使得我们可以非常轻松地实现 map（f， list）构造出非常有用的其他函数，例如我们可以使用 map（f， list）实现诸如 filter、reduce 等功能。

高阶函数的一大优势就是使得我们可以进行更好层次的抽象，我们可以通过高阶函数，将微分与积分很好地通过一步步拆分成一个个过程后组装起来，非常清晰地讲述了如何写成一个微分器与积分器，这种简洁与优雅的代码在非函数式语言中是很难看到的[6]。

2.2 惰性计算

函数式编程还具有惰性计算的特性，在惰性计算中，表达式不是在绑定到变量时立即计算，而是在求值程序需要产生表达式的值时进行计算。这帮助我们很好地生成无限数据结构，例如斐波那契数列、自然数等等，我们只需要在程序运行计算时取我们需要的前面几位数字；还可以实现短路特性，例如在许多语言中，

&&与｜｜都是按照短路运算，即前面的判断若可以判断出结果，则后面的不需要再进行运算，这种思路可以很轻松地通过惰性计算来实现，帮助我们提高程序运行效率，减少不需要的计算，。

3.函数式编程思想的应用

随着函数式编程思想的日渐流行，许多软件都开始支持函数式编程的模块，例如 C#、python 都通过加入 Lambda 来实现高阶函数，以此支持函数式编程。除了语言层面使用函数式编程的思想外，许多软件的构成其实也是按照函数式的模块化来写成，将各种功能进行模块化，并且使用控制流的思想来将一个个过程拼接后实现完整功能。

3.1WEB 前端中的函数式编程思想

WEB 前端指的是创建 Web 页面或 app 等前端界面呈现给用户的过程，通过 HTML，CSS 及 JavaScript 以及衍生出来的各种技术、框架、解决方案，来实现互联网产品的用户界面交互。在过去，WEB 前端主要展示的是静态的内容，在近年来，WEB 前端大多需要做成动态的页面，使得用户可以与服务器进行交互，并且前端负责将数据给更加完善地展示出来。

函数式编程的思想在 WEB 前端中可以很好地发挥其作用[7]。首先，在 WEB 前端的实现过程中，程序设计主要的目的是为了将从后端取得的数据进行展示，而函数式编程可以使用纯函数[8]，纯函数的概念就是指没有副作用的函数，在理论上它等价于我们数学世界里面的函数概念。纯函数的优势在于，它向我们保证了其"纯粹性"，同样的输入无论调用多少次，都会是一样的输出，并且不用担心调用过程会修改外部环境。每个函数都是足够独立足够抽象的个体，我们可以放心地将函数进行组合，这让我们在做代码复用或者重构时，不用去担心函数是否会影响到其他地方。在许多场景下，由于纯函数没有副作用，不能记录变量的状态，并不会采取使用纯函数。其次，出于可以得到更快的响应速度，减少访问后端数据，提高软件实行效率等等原因，前端有着非常多的异步问题，而函数式编程的无副作用特性正可以使得我们可以随意地组合各种方法。

目前前端的主流框架中，如 react，vue 等，都支持函数式编程，甚至已经开始有一些以函数式编程为主范式的框架开始出现。究其原因，是因为在当今，前端页面的展示内容越来越丰富，原本的前端页面设计虽然比较易学，然而其抽象层次仍然不够，使用函数式编程可以更好地将数据展示的过程抽象出来，使得我

们更关注于如何将视图展示出来。

例如，如果我们想写一个展示图片的页面，则我们的应用将做四件事：

1.根据特定搜索关键字构造 url
2.向 flickr 发送 api 请求
3.把返回的 json 转为 html 图片
4.把图片放到屏幕上

我们可以将代码写成如下形式：

```
var _ = R;
var Impure = {
        getJSON: _.curry(function(callback, url) {
          $.getJSON(url, callback);
        }),

        setHtml: _.curry(function(sel, html) {
          $(sel).html(html);
        })
      };

      var img = function (url) {
        return $('<img />', { src: url });
      };

      //////////////////////////////////////////

      var url = function (t) {
        return   'https://api.flickr.com/services/feeds/photos_public.gne?tags='
+ t + '&format=json&jsoncallback=?';
      };

      var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

      var srcs = _.compose(_.map(mediaUrl), _.prop('items'));
```

```
var images = _.compose(_.map(img), srcs);

var renderImages = _.compose(Impure.setHtml("body"), images);

var app = _.compose(Impure.getJSON(renderImages), url);

app("cats");
```

3.2 分布式系统中的函数式编程思想

分布式系统是由一组通过网络进行通信、为了完成共同的任务而协调工作的计算机节点组成的系统。分布式系统的出现是为了用廉价的、普通的机器完成单个计算机无法完成的计算、存储任务。其目的是利用更多的机器，处理更多的数据。当单个节点的处理能力无法满足日益增长的计算、存储任务的时候，且硬件的提升，如加内存、加磁盘、使用更好的 CPU 等高昂到得不偿失的时候，应用程序也不能进一步优化的时候，我们需要考虑使用分布式系统。在近年来，由于大数据的火爆与 CPU 主频不再大幅提升等原因，人们越来越偏向于使用分布式系统。

MapReduce[9]是一种编程模型，用于大规模数据集的并行运算。概念"Map（映射）"和"Reduce（归约）"，是它们的主要思想，都是从函数式编程语言里借来的，还有从矢量编程语言里借来的特性。它极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。 当前的软件实现是指定一个 Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的 Reduce（归约）函数，用来保证所有映射的键值对中的每一个共享相同的键组。它借助于函数式程序设计语言 Lisp 的设计思想，提供了一种简便的并行程序设计方法，用 Map 和 Reduce 两个函数编程实现基本的并行计算任务，提供了抽象的操作和并行编程接口，以简单方便地完成大规模数据的编程和计算处理 。MapReduce 的一个经典实例是 Hadoop。

图 3-1 展示了 MapReduce 的工作流程，它将一个集群上的服务器氛围一个 master 与多个 worker，程序启动时，master 将首先启动，随后 worker 通过 RPC 请求向 master 请求 task，master 随后会将任务分割成一个个 split 送给不给的 worker，而 worker 在它们本地服务器上做好各自工作后，会在各自服务器上产生临时文件，并且随后通过 RPC 请求告诉 master 已经完成。master 可以进行 reduce 步骤，将各个 worker 产生的文件进行第二次传送，每个 worker 负责各

自不同的内容并进行 reduce，最终将产生的结果文件传送给 master 后进行整合输出最终结果。

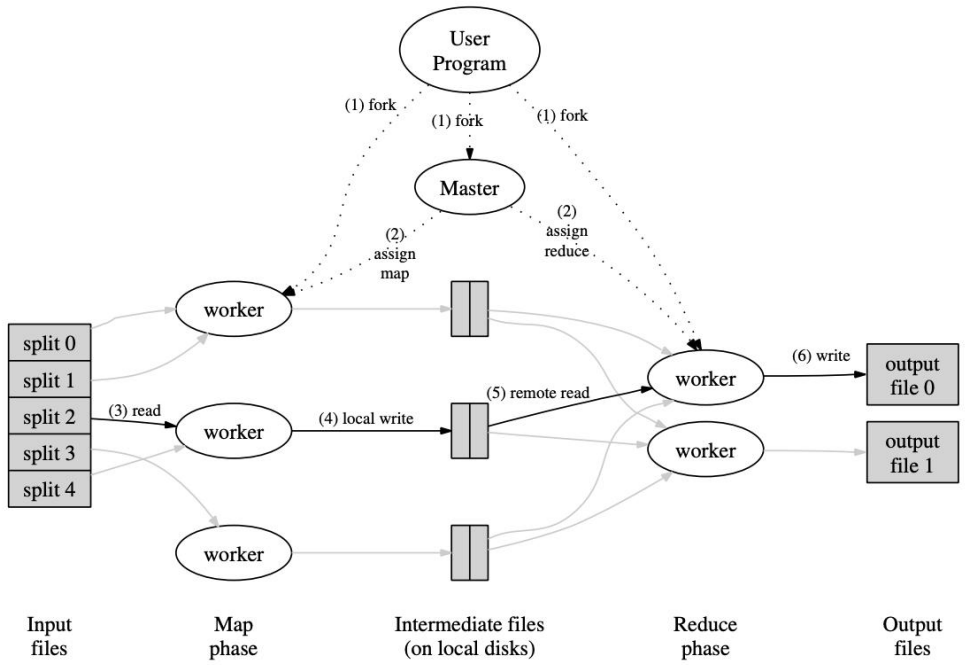

图 3-1 MapReduce 流程

在许多分布式的系统中，流处理的方式是一种很普遍的处理手段，因此函数式编程的思想可以很好地与之结合。在 MapReduce 模型中，我们可以在 Map 与 Reduce 中实现自己需要的内容，很方便地使用分布式系统实现自己想要的函数效果，提升对于大数据处理的效率。

在函数式编程的语言中，普遍都会实现 Map 与 Reduce，而 MapReduce 正是将这种函数式处理的方式应用于分布式系统，将数据进行流处理，分成一个个模块，每个步骤解决一个特定的问题，将一个大问题分成一个个小问题后，逐步解决，使得代码逻辑变得清晰，且耦合性更低，更易更改，拥有很高的可复用性。

4.结论

在本论文中，我们介绍了函数式编程的思想，它的高阶函数、流处理的思想使得我们可以写出拥有极高可复用性的程序。除此以外，如果我们编写纯函数，那么它将没有副作用，对于特定输入，输出结果不随时间状态的改变而改变，可以确定其输出结果，在一些特定场景，如前端程序编写中，我们需要编写异步页面时，有着非常好的效果。在分布式系统中，由于许多分布式系统都是为了处理数据，进行各种分布式计算任务，这正符合了函数式编程的初衷（lisp 语言设计

就是为了解决计算问题），使得我们通过函数式语言能够很好地解决分布式系统的构成。

函数式编程在许多领域的实现非常优雅与简洁，让我们可以清晰地了解到各个模块完成的内容，并通过结合，组成更加复杂的模块，但是它仍有着不可忽略的问题。对于程序设计者来说，我们需要有着更好的抽象思维，才能够写出符合函数式编程思想的代码，并且许多设计中，很难脱离状态，完全使用纯函数的内容进行编写。除此以外，即使拥有惰性计算的特点，但很多时候函数式程序仍然由于过多的递归等问题，效率上与内存占用上仍有着各类问题。

参考文献：

[1]https://www.zhihu.com/question/23306821

[2]Leonard Kleinrock. 1985. Distributed systems. Commun. ACM 28, 11 (Nov. 1985), 1200–1213.

[3]Adam cibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional programming for modular Bayesian inference. Proc. ACM Program.

[4]Hal Abelson, Gerald Jay Sussman. Structure and Interpretation of Computer Programs - 2nd Edition. The MIT Press; second edition (September 1, 1996)

[5]Philip Wadler. 1992. The essence of functional programming. In Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '92). Association for Computing Machinery, New York, NY, USA, 1–14.

[6]John Hughes Chalmers, Tekniska Hogskola. 1997. Why Functional Programming Matters.

[7]Atze van der Ploeg. 2013. Monadic functional reactive programming. In Proceedings of the 2013 ACM SIGPLAN symposium on Haskell (Haskell '13). Association for Computing Machinery, New York, NY, USA, 117–128.

[8]https://blog.csdn.net/qq_42497250/article/details/93625668

[9]Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113.