2021四月月报

条件渲染

表单

Bytedance基础架构

- 存储
- 计算

反思与认识

数据流最后一次周会Leader的分享

- 大多数程序员的困境：只顾埋头拉车，全然不顾大局
- 你发现问题了，那然后呢?
- 补位意识
- 技术人员的傲气

关于面试

每月鸡汤

- 越强大，越不显山露水

发现新大陆

技术相关

后端通用框架介绍

- 代码骨架

```
awesomeProject
├── biz  # Hertz自带目录，业务逻辑文件夹（business --> biz）
|   ├── config  # 项目配置文件，拉取了动态配置中心yaml的结构（如redis配置等）
|   |   └── config.go  # 初始化项目配置
|   ├── consts  # 定义一些全局常量
|   |   ├── constants.go
|   |   └── enum.go
|   ├── dal  # data access layer 数据访问层
|   |   ├── cache_dal  # redis类型的数据访问层
|   |   ├── db_dal  # mysql类型的数据访问层
|   |   |   ├── private_info_dal.go  # 与我私人信息相关的数据库调用函数
|   |   |   ├── pubilc_info_dal.go  # 公共信息相关的数据库访问函数
|   |   └── error.go # 可以不用
|   ├── debug  # 与debug服务器建立连接
```

```
|   |   ├── idl.go
|   |   └── server.go
|   ├── handler  # 处理单元，每个接口会调用一个handler函数
|   |   ├── private_info_handler.go
|   |   ├── public_info_handler.go
|   |   └── utils.go  # 工具handler，放一些通用的处理器（读登录用户的信息）
|   ├── idl.go  # 不知道干嘛的
|   ├── middleware  # 中间件（比方说登陆认证）
|   |   └── user_info.go  # 确认用户信息
|   ├── model  # 数据模型（定义数据结构）
|   |   ├── dao  # 数据访问对象 Data Access Object
|   |   |   ├── private_info_dao.go
|   |   |   ├── public_info_dao.go
|   |   └── dto  # 数据传输对象 Data Transfer Object
|   |       ├── private_info_dto.go
|   |       ├── public_info_dto.go
|   ├── service  # 后端提供的服务一览，承接handler，下启dal
|   |   ├── private_info_service.go
|   |   ├── public_service.go
|   └── utils  # 工具类（生成当前时间诸如此类通用函数）
|       └── util.go
├── build.sh  # 解析代码脚本
├── conf  # 项目配置文件
|   ├── config.yaml # 空文件
|   └── data_dp_fort_backend.yaml  # Web框架初始化配置（端口等）
├── ddl  # sql表属性一览（不起任何作用，提供参照依据）
|   ├── private_info.sql  # 私人信息表
|   ├── public_info.sql  # 公共信息表
├── go.mod  # 项目依赖
├── go.sum  # 不晓得干嘛的
├── main.go  # 项目入口，初始化服务框架（如：Hertz）
├── output  # 编译文件存放目录
|   ├── __pycache__
|   |   └── settings.cpython-38.pyc
|   ├── bin
|   |   └── data.dp.fort_backend
```

```
|   ├── bootstrap.sh  # 运行此文件启动服务
|   ├── bootstrap_staging.sh
|   ├── conf
|   |   ├── config.yaml
|   |   └── data_dp_fort_backend.yaml
|   ├── log
|   └── settings.py
├── router.go  # 所有的接口在此定义
├── router_gen.go  # 个人感觉有些许的多余，这里的函数在main中被调用，接着在这里继续调用router.go的函数
└── script
    ├── bootstrap.sh
    └── settings.py
```

- 骨架解析

Git使用总结

- Git基础

版本库：又名仓库，英文repository，可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来的某个时刻可以“还原”

明确：所有的版本控制系统实际只能跟踪文本文件的改动，如果修改图片、视频这种文件，没法跟踪变化

```
git init  # 初始化一个Git仓库
# 添加文件到Git仓库，分两步
git add file1.txt  # 可反复多次使用，添加多个文件
git commit -m "bugFix"
```

```
# git status命令可以让我们时刻掌握仓库当前的状态
git status
```

```
# 如果修改了一个文件，但这次修改并没有被添加到Git仓库，就会报如下错误
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

```
# 我们把这次修改git add然后再看看git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   file1.txt
# git告诉我们将要被提交的修改包括file1.txt，下一步，可以放心提交了
```

```
# 提交后，再用git status命令看看仓库的当前状态
On branch master
nothing to commit, working tree clean
# git告诉我们没有需要提交的修改，而且，工作目录是干净的
```

```
git log  # 使用log可以查看提交历史，以便确定要回退到哪个版本
git reflog  # 要重返未来，用git reflog查看命令历史以便确定要回到未来的哪个版本
```

```
#  head指向的版本就是当前版本，使用下面的命令允许在历史版本中穿梭
git reset --hard commit_id
```

工作区（Working Directory）就是电脑里能看到的目录，如项目文件夹；版本库（Repository）是所在文件夹下的.git文件，Git的版本库里存了很多东西，其中最重要的就是成为stage（或index）的暂存区，还有Git为我们自动创建的第一个分支 `master` ，以及指向 `master` 的一个指针叫 `head`

前面讲了把文件往Git版本库里添加的时候分了两步：

第一步用 `git add` 把文件添加进去，实际上就是把文件修改添加到暂存区

第二步用 `git commit` 提交更改，实际上就是把暂存区的所有内容提交到当前分支

因为我们创建Git版本库时，Git自动为我们创建了唯一一个 `master` 分支，所以，现在， `git commit` 就是往 `master` 分支上提交更改。简单理解：需要提交的文件修改统统放到暂存区，然后一次性提交暂存区的所有修改

```
# 撤销修改
# 1. 修改了但还没有add
git checkout -- file1.txt  # 丢弃了工作区的修改
# 2. 修改了并且add了
git reset head file1.txt  # head表示最新的版本，git reset既可以回退版本，也可以把暂存区的修改回退到工作区，运行指令后，暂存区的修改回退到了工作区
git checkout -- file1.txt  # 丢弃工作区的修改
# 3. 修改了而且commit了，回退版本
git reset --hard commit_id
# 4. 修改了而且推到远程了，请抓紧时间跑路。
```

```
# 如果误删除文件，可以利用版本库恢复文件
git checkout -- file1.txt
```

- Git远程仓库

```
# 从github复制远程库到本地
git clone git@github.com:bupt-zhenghui/learngit.git
# 上传本地代码到远程
git add .
git commit
git push
```

- Git分支

参考：https://learngitbranching.js.org/?locale=zh_CN

```
# Git仓库中的提交记录保存的是当前目录下所有文件的快照，就像是把整个目录复制，然后再粘贴一样，但比复制粘贴优雅许多。Git希望提交记录尽可能轻量，因此在每次进行提交时，不会盲目地复制整个目录
# Git还保存了提交的历史记录，对于项目组的成员来说，维护提交历史对大家都有好处
git commit
```

```
# Git的分支也非常轻量，他们只是简单的指向某个提交记录，并且按逻辑分解工作到不同的分支要比维护那些特别臃肿的分支简单多了
# 使用分支其实就相当于在说：我想基于这个提交以及它所有父提交进行新的工作
# 少建分支，多用分支
git branch bugFix # 新创建一个分支
git checkout bugFix # 将当前指向的节点从main移到新建的分支上
git commit # 提交更新到新的分支上
# 也可以简单一点
git checkout -b bugFix # 新创建一个分支并直接指向新的分支
git commit
```

```
# 利用merge将两个分支合并到一起（我们新建一个分支，在其上开发某个新功能，开发完成后再合并回主线）
git checkout main # 先切换回main分支
git merge bugFix # 将新分支合并到main
```

```
# rebase
# rebase取出一系列的提交记录，复制它们，然后在另外一个地方逐个的放下去
# rebase可以创造更线性的提交历史，如果只允许使用rebase的话，代码库的提交历史将会变得异常清晰
git checkout bugFix  # 切换到bugFix分支
git rebase main  # 将bugFix分支迁移到main分支下
```

```
# 指针HEAD
# HEAD是一个对当前检出记录的符号引用，HEAD总是指向当前分支上最近一次提交记录，大多数修改提交树的Git命令都是从改变HEAD的指向开始的；HEAD通常情况下指向分支名，在你提交时，改变了bugFix的状态，这一变化通过HEAD变的可见
git checkout HEAD^ # 将HEAD指针向上移动一个单位（提交）
```

```
# 强制修改分支位置
# 这里要注意，什么叫强制修改？强制修改与HEAD无关，而与分支的绝对位置有关
git branch -f main HEAD~3  # 将main分支强制指向HEAD的第三级父提交
```

```
# 撤销变更
# git reset通过把分支记录回退几个提交记录来实现撤销改动（注意：是修改分支绝对位置）
# 虽然在本地使用git reset很方便，但这种方法对远程分支无效，为了撤销更改并分享给别人，需要使用git revert
git reset local^  # 本地分支回退一个提交
git revert HEAD  # 撤销HEAD提交回到之前的提交（实际上它新建了一次提交，恢复成之前那一次）
```

- Git conflict

```
# 1. 与自己的代码产生冲突
# 实际上只有新手会产生这种问题，产生的原因在于：自己当前的提交与上一次提交产生了冲突，因为本地的master并没有切到上次提交后的主分支上
# 最好的解决（避免）方法是，每天打开IDE之前，把本地的master和服务器最新的master同步，同时，在每次创建新分支以前再同步一次，确保自己是在最新的主分支上进行提交
git pull origin master # 拉取服务器master分支到本地
```

- Git提交

```
# 发现push的内容出现问题
git push origin --delete f_20210330_multi_conditions # 删除远程分支
git reset VersionID # 本地git提交回退到指定版本
# 接下来重新创建分支提交即可
```

mysql使用总结

- 建表

```
# 创建一个标准sql表
CREATE TABLE fort_govern_rule_condition (
  'id'          int unsigned NOT NULL AUTO_INCREMENT COMMENT '自增id',
  'rule_id'     int unsigned NOT NULL  COMMENT '规则id',
  'sequence'    int unsigned NOT NULL  COMMENT '筛选条件序列号',
  'rule'        int unsigned NOT NULL  COMMENT '执行哪条规则',
  'operator'    int unsigned NOT NULL  COMMENT '算子',
  'value'       varchar(128)           COMMENT '具体比较数值',
  'sign'        int unsigned NOT NULL  DEFAULT 0 COMMENT '与为0或为1',
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci
COMMENT='规则顺序执行表';
```

- 常见操作

```
# 修改表名
rename table fort_govern_rule_sequence to fort_govern_rule_condition
# 复制一张旧表到新表里
create table fort_govern_rule_temp select * from fort_govern_rule
# 设置字段为主键，自增
alter table fort_govern_rule_temp add primary key (id);
alter table fort_govern_rule_temp modify id int auto_increment
# 删除一列
ALTER TABLE fort_govern_rule_temp DROP rule
# 删除多列
ALTER TABLE fort_govern_rule_temp DROP operator, DROP value
# 删除行
DELETE FROM fort_govern_rule_temp WHERE id = 0
```

代码的发布流程

React-Typescript

```
# 前期知识准备
1. Javascript
2. HTML+CSS
3. 构建工具：Webpack
4. 安装node，cnpm
```

- Webpack

本质上，Webpack是一个现代JavaScript应用程序的静态模块打包器（module bundler）。当webpack处理应用程序时，它会递归的构建一个依赖关系图（dependency graph），其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个bundle

入口（entry）

入口起点指示webpack应该使用哪个模块，来作为构建其内部依赖图的开始。进入入口起点后，webpack会找出有哪些模块和库时入口起点（直接和间接）依赖的。每个依赖项随即被处理，最后输出到称之为bundles的文件中

可以在webpack配置中配置 `entry` 属性，来指定一个入口起点，默认值为 `./src` ，看一个例子：

```
module.exports = {
    entry: './path/to/my/entry/file.js'
}
```

出口（output）

output属性告诉webpack在哪里输出它所创建的bundles，以及如何命名这些文件，默认值为 `./dist` 。基本上，整个应用程序结构，都会被编译到你指定的输出路径的文件夹中。你可以通过在配置中制定一个 `output` 字段，来配置这些处理过程

```
const path = require('path')
module.exports = {
    entry: './path/to/my/entry/file.js',
    output: {
        path: path.resolve(__dirname, 'dist'),
        filename: 'my-first-webpack.bundle.js'
    }
};
```

loader

loader让webpack能够去处理那些非JavaScript文件（webpack自身只理解JavaScript）。loader可以将所有类型的文件转换为webpack能够处理的有效模块，然后就可以利用webpack的打包能力，对它们进行处理。本质上，webpack loader将所有类型的文件，转换为应用程序的依赖图（和最终的bundle）可以直接引用的模块

在更高层面，在webpack的配置中**loader**有两个目标：

1. `test` 属性，用于标识出应该被对应的loader进行转换的某个或某些文件
2. `use` 属性，表示进行转换时，应该使用哪个loader

```
const path = requre('path')
const config = {
    output: {
        filename: 'my-first-webpack.bundle.js'
    },
    module: {
        rules: [
            { test: /\.txt$/, use: 'raw-loader' }
        ]
    }
};
module.exports = config
```

以上配置中，对一个单独的module对象定义了 `rules` 属性，里面包含两个必须属性： `test` 和 `use` 。这告诉webpack编译器如下信息：嘿，webpack编译器，当你碰到「在 `require()` / `import` 语句中被解析为 '.txt' 的路径」时，在你对它打包之前，先**使用** `raw-loader` 转换一下。

插件（plugins）

loader被用于转换某些类型的模块，而插件则可以用于执行范围更广的任务。插件的范围包括，从打包优化和压缩，一直到重新定义环境中的变量。插件接口功能极其强大，可以用来处理各种各样的任务

```
const HtmlWebpackPlugin = require('html-webpack-plugin');  // 通过 npm 安装
const webpack = require('webpack');  // 用于访问内置插件
const config = {
    module: {
        rules: [
            { test: /\.txt$/, use: 'raw-loader'}
        ]
    },
    plugins: [
        new HtmlWebpackPlugin({template: './src/index.html'})
    ]
};
module.exports = config
```

```
// 模式：设置mode参数启用相应模式下webpack内置的优化
module.exports = {
    mode: 'production'
}
```

- React

初始化React项目

```
npx create-recact-app react-demo
cd react-demo
npm start
```

```
# 骨架介绍
react-demo
├── README.md
├── package-lock.json
├── package.json  # 配置文件
├── node_modules  # 非常大的依赖文件
├── public  # 入口文件（index）
```

```
|   ├── favicon.ico
|   ├── index.html
|   ├── logo192.png
|   ├── logo512.png
|   ├── manifest.json
|   └── robots.txt
└── src  # 源码
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js  # 主入口文件
    ├── logo.svg
    ├── reportWebVitals.js
    └── setupTests.js
```

```
// 一个最简单的react项目
// index.js
import React from 'react'  // 引入react自带的资源
import ReactDOM from 'react-dom'

// 渲染函数 render(content, DOM)
// content: 要插入的内容 (html)
// DOM: 要插入的容器 (把content放在哪里)
ReactDOM.render(<h1>Hello React!</h1>, document.getElementById('root'));
// 解释一句: 实际上在public的index.html里, 又一个标签就是root
```

JSX

JSX是一个JavaScript的语法扩展，JSX可以很好的描述UI应该呈现出它应有交互的本质形式。JSX可能会使人联想到模版语言，但它具有JavaScript的全部功能

```
import React from 'react'
import ReactDOM from 'react-dom'
const name = 'zhenghui'
ReactDOM.render(<h1>Hello, { name }</h1>,
document.getElementById('root'))
```

JSX语法解析策略

- 遇到 `<>`，用HTML语法解析
- 遇到 `{}`，用JavaScript语法解析

元素渲染

```
// 页面上动态更新时间
import React from 'react'
import ReactDOM from 'react-dom'

function tick() {
    // (): 如果存在标签结构，并且标签结构要换行，需要用()括起来
    const element = (
        <div>
            <h1>Hello, world!</h1>
            <h2>It is { new Date().toLocalTimesString() }</h2>
        </div>
    )
    ReactDOM.render(element, document.getElementById('root'))
}
setInterval(tick, 1000);
```

组件

组件的后缀可以是js，也可以是jsx，一个React项目，可能由成千上万个组件组成

```jsx
// app.jsx
import React from 'react'
import ReactDOM from 'react-dom'
// 用类的形式创建组件
class App extends React.Component{
    // 渲染函数
    render() {
        return <h1>Hello React component</h1>
    }
}
export default App
```

```jsx
// index.js
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App'
ReactDOM.render(<App />, document.getElementById('root'))
```

```jsx
import React from 'react'
export default class Home extends React.Component{
    render() {
        return ( <div>Home</div> )
    }
}
```

Props属性

组件的复用性很重要

比方说我在多处要用到一个结构，但结构内的数据不同，怎么处理

```jsx
// MyNav.jsx
import React from 'react'
export default class Mynav extends React.Component {
    render() {
```

```
        return (
            <div>
                <ul>
                    {
                        this.props.nav.map((element, idx) => {
                            return <li key={ idx }>{ element }</li>
                        })
                    }
                </ul>
            </div>
        )
    }
}
```

```
// app.jsx
import React from 'react'
import Home from './Home'
import Nav from './MyNav'
// 用类的形式创建组件
class App extends React.Component{
    // 渲染函数
    render() {
        const nav1 = ["首页", "视频", "学习"]
        const nav2 = ["Web", "Java", "Node"]
        return (
            <div>
                <h1>Hello React component</h1>
                <h2><Home /></h2>
                <Nav nav={ nav1 }/>
                <Nav nav={ nav2 }/>
            </div>
        )
    }
}
export default App
```

State

```
// StateComponent.jsx
import React from 'react'
export default class StateComponent extends React.Component{
    /**
     * 组件中的状态：state
     * 以前我们操作页面元素的变化，都是修改DOM，操作DOM
     * 但是有了React优秀的框架，我们不再推荐操作DOM，页面元素的改变使用State进行处
理
    */
    constructor(props) {
        super(props);
        this.state = {
            count: 10
        }
    }
    increment() {
        // setState
        this.setState({
            count: this.state.count += 1
        })
    }
    decrement() {
        this.setState({
            count: this.state.count -= 1
        })
    }
    clickHandler = () =>{
        console.log(this);
    }
    render() {
        return (
            <div>
                <h3><div>组件的State</div></h3>
                <p>{ this.state.count }</p>
```

```
                <button onClick={ this.increment.bind(this) }>增
加</button>
                <button onClick={ this.decrement.bind(this) }>减
少</button>
                <button onClick={ this.clickHandler }>关于this</button>
            </div>
        )
    }
}
```

生命周期函数（高级）

随着对React的理解和使用越来越多，生命周期的参考价值越来越高

```
// ComponentLife.jsx
import React from 'react'
export default class ComponentLife extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            count: 10
        }
    }
    componentWillMount(){
        console.log("component will mount")
    }
    componentDidMount() {
        console.log("component did mount")
    }
    shouldComponentUpdate() {
        return true
    }
    componentWillUpdate() {
        console.log("component will update")
    }
    componentDidUpdate() {
```

```
        console.log("component did update")
    }
    componentWillReceiveProps() {
        console.log("component will receive props")
    }
    componentWillUnmount() {
        console.log("component will unmount")
    }
    changeHandler = () => {
        this.setState({
            count: this.state.count += 1
        })
    }
    render() {
        const { count } = this.state.count
        return (
            <div>生命周期函数: { count }</div>
            <button onClick={ this.changeHandler }>增加</button>
        )
    }
}
```

setState的更新时机

```
// setStateDemo.jsx
import React from 'react'
export default class SetStateDemo extends React.Component{
    constructor(props) {
        super();
        this.state = {
            count: 0
        }
    }
    increment = () => {
        this.setState({
            count: this.state.count += 1
```

```
        })
        console.log(this.state.count)
    }
    render() {
        return (
            <div>
                setState同步还是异步问题
                <p>{ this.state.count }</p>
                <button onClick={ this.increment }>修改</button>
            </div>
        )
    }
}
```

条件渲染

```
// ifDemo.jsx
import React from 'react'
export default class IfDemo extends React.Component{
    constructor() {
        super()
        this.state = {
            isLogin: false,
            names: ['zhangyiming', 'leijun', 'liyanhong'],
            value: 10
        }
    }
    login = () => {
        this.setState({
            isLogin: !this.state.isLogin
        })
    }
    render() {
        const { names, value } = this.state
```

```
        let showView = this.state.isLogin ? <div>请登录</div> : <div>退出登录</div>
        return (
            <div>
                条件渲染: { showView }
                <button onClick={ this.login }>登录</button>
                <div>
                    {
                        (value > 10) ?
                        <div>
                            {
                                names.map((element, idx) => {
                                return <p key={ idx }>{ element }</p>
                            })
                            }
                        </div>
                        :
                        <div>没有数据......</div>
                    }
                </div>
            </div>
        )
    }
}
```

```
// map遍历套路
{
    names.map((element, idx) => {
        return <p key={ idx }>{ element }</p>
    })
}
// 一道经典面试题：这里的key有什么用？不写会怎样？
// 答：key为元素添加了索引，添加了索引的列表不会参与重新渲染时
```

表单

在React里，HTML表单元素的工作方式和其他的DOM元素有些不同，这是因为表单元素通常会保持一些内部的state

```
import React from 'react'
export default class FormDemo extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            value: ""
        }
    }
    handleSubmit =() => {

    }
    handleChange =(e) => {
        this.setState({
            state: e.target.value
        })
    }
    render() {
        return (
            <div>
                <form onSubmit={ this.handleSubmit }>
                    <input type="text" onChange={ this.handleChange }/>
                    <input type="submit" value="submit" />
                </form>
            </div>
            )
    }
}
```

Bytedance基础架构

机房设施 —— 物理机房设施，供电、冷却、安全等

网络 —— 网络层包含路由器，交换机，防火墙，负载均衡器等

存储 —— NAS，SAN，及其他分布式存储

服务器 —— 服务器层，包含本地部署物理或者虚拟化服务器以及虚拟化的数据中心环境

基础设施管理和服务 —— 关键类服务如DHCP，DNS，服务器配置管理，监控，告警，认证等

- 存储

LAMP —— Linux + Apache + MySQL + PHP（2005年以前）实现一个网站

在2005年左右，三层架构被提出，把最前端的HTTP服务器和后端的控制逻辑层解耦（常见的Apache Tomcat），后端的MySQL数据库，那个时候在数据库层面需要提供High Reliability，所以一般底层会使用一个共享式的泛存储（两台数据库，一主一备，主挂了就切换到备）但随着访问的不断膨胀，数据库成了瓶颈

2006年，Amazon启动AWS，标志着进入云存储时代，2007年世界各个大型企业开始构建云存储

- 对象存储
    1. 适用场景

       HTTP REST访问

       并发访问高

       不需要层级结构

       相对稳定不变的非结构化数据（海量视频，图片，文档，备份包等）
    2. 限制

       不太适合频繁增删改场景

       对延迟比较敏感也不建议

不支持目录层次结构，Rename功能

- KV存储（ByteKV）（Hbase）

  传统的DBMS太重了，扩展性受限，我们使用的很多场景并不需要那么强的关系型数据库支撑，很多场景只需要K和V就可以实现功能

  分布式KV，由于可能存在大量的KV数据，单一机器无法存储这么多，于是采用分布式存储，每个机器会有一个Key Range，请求发送后上层会根据Key的Value自动分配到其所在的主机完成查找

- ByteDoc/Mongo文档数据库

  1. 适用场景

     Schema Free：文本数据，爬虫数据，嵌套数据（传统关系数据库怎么存文档？JSON格式？没法存）

     高度变化的数据：游戏用户数据

     地理位置（LBS）

     事先模型没确定的快速开发迭代

  2. 限制

     不适合复杂关联查询

     不适合跨文档事务的场合

- ByteSql/Table

  可以认为是一个带主键和多级索引的分布式大宽表场景

  很多业务不需要很复杂的sql模型，只需要KV存储就可以

  由于基于ByteKV，所以单行数据量太大会影响性能

  索引不是没有代价的，不宜过多

- ByteGraph图存储

  一张图由万亿条点和边组成，点是实体，边是实体之间的关系

  举个例子：抖音里用户就是实体，用户关注另一用户的行为可以理解为关系

```
点 Vertex
# id可以是用户的id，value可以是用户画像
id  type --> value
```

```
边 Edge
id1 edgeType id2 timestamp --> value
```

- 适用场景

  社交网络关系（点赞、2度好友关系、共同关注）

  知识图谱构建/血缘关系

  金融欺诈风控检测

  实时推荐引擎

- 使用注意点

  关注大点（边很多，如大V）

  多度查询需要关注数据量，常规都在3度内（6度人脉基本就关联地球上所有人了）

  热点写入或者查询问题

- 消息队列

  - 适用场景

    解耦生产/处理

    兜底，失败重试

    弹性，缓冲

    异步/批量处理

  - 使用注意点

    NSQ：侧重吞吐，不保序，消息不持久化

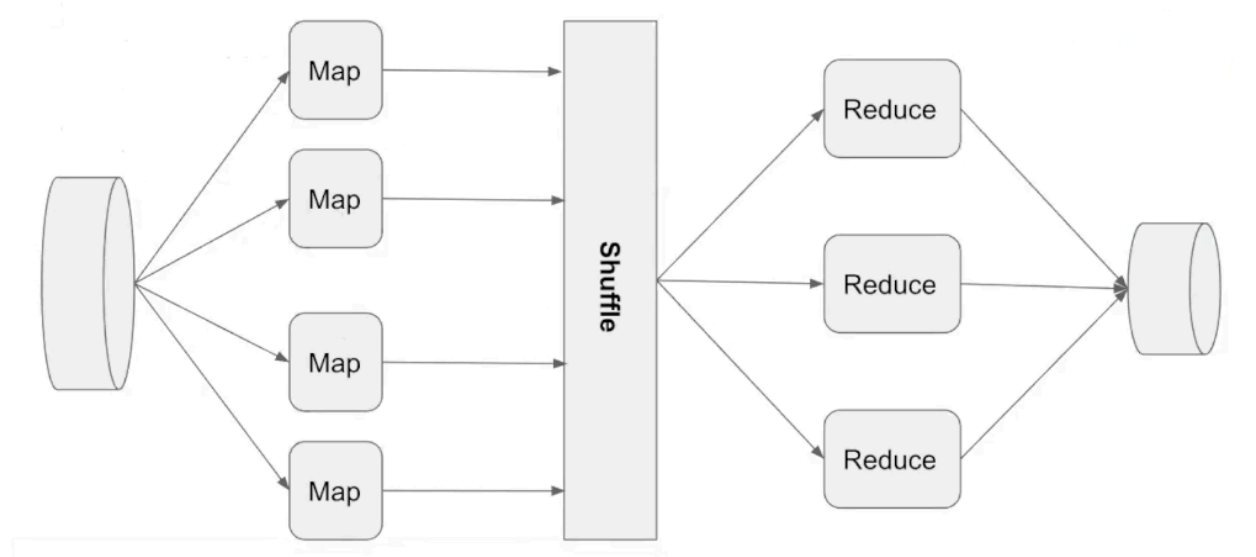
    RocketMQ：分区层面有序；Topic/分区多对性能影响不大

    Kafka：分区多了性能影响厉害

- 计算

- MapReduce

  如何存储大量的信息？如何轻而易举地创建海量索引？



- 分治思想

  任务可拆分，相互依赖度小

- 处理模型抽象（Map & Reduce）

  Map：对一组元素进行某种重复式的处理

  Reduce：对中间结果进行进一步的整理，并产生最终的结果

- 隐藏细节复杂度，统一框架

- 优点

  接口简单，易于编程

  扩展性强

  可以处理PB级以上数据

- 缺点

  实时计算，无法像MySQL那样毫秒级返回

  适合批处理，无法很好的处理动态变化的流式输入

  DAG（后一个应用程序的输入为前一个的输出）计算下IO开销中

- Spark
- Flink
- K8S

反思与认识

数据流最后一次周会Leader的分享

讲了很多，在此记录一下感触比较深的，供自己参考学习

- 大多数程序员的困境：只顾埋头拉车，全然不顾大局

埋头拉车，对程序员来说就是写代码。车能拉起来，拉的快，表明你代码写的不错。实际上，对一个刚来一个月的实习新人来说，能埋头把车拉起来已经可以了🤦

但是，对于一个致力于长期从事这个行业的工程师来说，只顾埋头拉车，会使你无法打开眼界，丢掉创造力，慢慢的就转化为一个只会写代码的机器人了。如果希望在技术领域有所建树，你不能把眼界局限于自己手头的工作，你至少需要花一些时间抛开代码，从大局看清你所做的事情的终极目的是什么，为了达到这个目的有没有可能有更优的策略？你要像这样考虑问题

- 你发现问题了，那然后呢?

你和你的同事共同完成一项开发任务，此时PM提了一个Bug给你，但你发现这个Bug根本不是你写的，此时你应该怎么办？下下策：告诉PM这个Bug是某某某负责的，请直接找他。

这个问题应该还是非常常见的，因为光在我自己身上就已经发生了无数次.....因为每个人负责的部分不同，把具体问题划分到个人本无可厚非，但在划分以前，至少也应该想一想（或者说帮别人想一想）问题出在哪里？有没有可能自己直接把问题就解决了？当然，如果对这个领域确实是一无所知，那么去跟相关同学联系是必要的。简单点说，发现了问题，不要立刻把皮球踢给ABCD，请先尝试解决问题。你需要时刻谨记：你和你的队友是一条船上的，一荣俱荣，一损俱损，遇到了问题，能不能扛住，能不能主动站出来，也是一个工程师是否优秀的非常重要的衡量标准。

- 补位意识

很多时候，机会就摆在你的面前，可惜你没有意识到，让它一次又一次地与你擦肩而过。很长一段时间以后，你就开始抱怨：啊！凭什么别人的运气那么好！实际上只是你没有把握住罢了。

慢慢的我好像意识到，困难和机遇往往是相伴而生的。领导家里临时有急事，于是他把一项艰巨的任务托付给你，你为了完成这个任务可能要加班加点一周甚至好几周，这是幸运还是不幸呢？这很难说。对自己要求严格的人如果接受了这项任务，有可能超出领导预期完成；但我相信更多的人，心里会产生怨念：你拍拍屁股走人，脏活累活都留给我，我tm好烦！也是这种心态，导致最后完成的不理想。恰恰是这种危险与机遇并存的任务，能帮助你的上级挑选出真正的精英，真正有能力接棒的“下一代”。

- 技术人员的傲气

大多数做技术的同学都是有傲气的，往往个人能力越强的人，与人合作的能力就越差。但是如果你真的想走这条路，很遗憾，双拳一定难敌四手。

刘康讲了一个历史故事：时任国民党陆军少将的徐复观希望拜入哲学大师熊十力门下，希望熊给自己推荐一本值得看的书。熊十力给他推荐了《读通鉴论》，徐复观看完以后熊让他谈谈心得，徐复观就谈了许多对王夫之的批评。熊十力还未听完就破口大骂：你这个东西，怎么会读得进书！任何书的内容，都是有好的地方，也有坏的地方。你为什么不先看出他的好的地方，却专门去挑坏的；这样读书，就是读了百部千部，你会受到书的什么益处？读书是要先看

出他的好处，再批评他的坏处，这才像吃东西一样，经过消化而摄取了营养。譬如《读通鉴论》，某一段该是多么有意义，又如某一段理解是如何深刻，你记得吗？你懂得吗？你这样读书，真太没有出息！（个人理解：潜意识里通过贬低别人来拔高自己）

关于面试

最近也参与到字节的内推中，跟很多同学有聊这方面的问题。首先有一点是肯定的：面试的结果是有一定的运气成分的。我当初的面试官现在就坐在我旁边，所以我也时不时向她讨教面试方面的问题。就比方说实习生的选择，有时候一个非常好的人选（面试官的学弟，而且能力确实不错），仅仅是因为觉得他实习时间短了一点（3个月？）就在一面挂了他（面试官其实是很惋惜的，但没有办法），这就是运气问题。

大多数学生都以为“只要我回答对面试官提出的每一个问题，我就一定能过”，实际上并不是这样。有时候学生的个人气质和状态，自信谦虚从容沉稳比会不会答题更关键，因为这些是体现底层素质和个性特点的。“我们在实际面校招生的时候，很多学生讲完自我介绍我们就觉得这人不行了” —— 腾讯的面试官

面试时候表现出的（过度）紧张也会导致印象分降低，因为你紧张，本质上就是不自信，你觉得我们公司很牛逼，你害怕表现不好。那你自己都不相信自己的能力，自己都觉得自己很难进我们公司，那我们凭什么相信你优秀？另一个极端：你想表现出自己很优秀，你要自己先认可自己很优秀，但是这个表现稍微不注意就会变成自负，所以整个面试是非常考验学生的底层素质的。

每月鸡汤

– 越强大，越不显山露水

当能力提升、经验丰富和成果显著之后，自身的存在感也会跟着增强。然而，越是感到一切都顺风顺水，越容易发生意想不到的反转。无论你认为自己多么成功，也一定会有人不以为然，这是世间常识。因此，越是感到一帆风顺之时，越要谦虚谨慎行事。

发现新大陆