

# VRF + BFT 共识引起交易失败的问题

2022-09-03

昨天遇到一个问题，是 Ontology 的节点不出块了。节点使用的是 VBFT 共识，网络互通没有隔离，也就没有分叉，报的错是提案（proposal）过来的块，和预期的块哈希（MerkleRoot）不一样。由于种种原因，昨天的问题没有深入去查，用比较暴力的方法先让网络恢复正常。不过提到 VBFT，我想到了以前公司的一些事情。

我是在之前的公司开始接触区块链的，当时的项目号称自研区块链，也是用 VBFT（VRF + BFT）的共识，不过在共识方面不那么说，叫成 UBFT 还是什么。

我猜测 VRF + BFT 的主意是 Ontology 提出来的，我之前的公司把这种创意抄了过来，模仿着实现了一下。说来讽刺。

主要想说的是之前公司由于对 VRF + BFT 不靠谱的实现，引起的一个隐藏的 bug。那件事情距今快 2 年了，不记得当时为什么没有写博客记录一下，可能是在忙着做 PPT？昨天遇到共识相关的问题，我想起有那么一回事，正好现在有时间有心情写一下。由于过去时间太长了，细节上可能有出入。

## 背景

先介绍一下那个项目的情况，主打的特点有几个。

一个是异构多链，含义是可以在同一个节点上包含多条异构的链。异构是指一条链可以使用不同的共识机制、基于不同数据库运行起来。多链是指多条链可以在同一个节点上运行，因为觉得一条链不够用，一条链就相当于传统业务里的一张数据表，多条链可以方便地进行数据拆分，在联盟链的场景下更好地支持业务。异构多链，可以理解为把以太坊不同 chain id 的链，用同一个二进制包启动了。现在的某条开源联盟链，还在拿从异构多链演变来的灵活装配作为一大特点呢，猜猜为什么。

再一个是对多数据库的支持，同时支持很多关系型数据库和非关系型数据库，做法是针对各种数据库，写数据操作的中间层做适配。

还有就是共识机制方面，基于开源的 Tendermint 项目。Tendermint core 是一个对 BFT 类共识的实现，在那个基础上，做的改动是把轮询选择提案节点，用 VRF 函数，替换为随机选择提案节点。另外还增加了对分层共识的支持，也就是共识组的概念，每隔多少个块换一次共识组，换共识组的方式借鉴 BFT 的流程，保证换共识组过程的安全性。分层共识这个理念也不知道起源于哪儿，可能同时期的项目流行这个？

项目的这些特点都是在我接触之前就已经开发完成的，我也只是有所了解。

我当时遇到的问题是，如果向区块链发送一笔失败交易，节点会立即返回交易失败的结果，然后如果再次发送一笔失败交易，第二笔交易的结果将迟迟不返回，节点不出块了。接下来如果仍然是失败交易，第三笔、第四笔，都会是同样的现象。这个时候，如果发送一笔正确的交易，节点会立即返回结果，之后一切恢复正常。而且这种现象是概率性出现的，并不是每一次失败交易都会引起问题。

## 前提

首先是失败交易指合约返回执行结果为失败的交易。区块链系统的交易失败有两种，一种是交易不能被执行，另一种是交易能被执行，但是合约中返回了合约层面的失败。那个项目并没有严格区分这两种失败的类型，合约有权限返回交易层面的失败，这其实是有问题的设计。

不正确的交易，将会在提案的时候，被忽略掉，因为不正确交易没有必要记录在区块链上。加上项目对失败交易错误的处理，造成的现象就是，合约执行失败的交易，会被忽略掉。这是前提。

BFT 共识的基础，是投两轮票，最终确定一个块。不管是什 BFT，在前面加什么字母，不管通过多么复杂的流程决定出哪个节点提案、如何提案，不管对共识的效率做什么优化，是并行提案还是流水线共识什么的，只要是 BFT 类的共识，都是投票两轮。对两个阶段的命名可能不一样，不管是用 proposal 还是 prepare 来描述，都是那样一个过程。

BFT 的流程，是先有一个节点生成一个块，然后把这个块发送给其他节点，如果超过  $2/3$  节点同意，会进行下一轮投票。第二轮投票如果超过  $2/3$  节点同意，这个块就算确认下来了。两轮投票是理解 BFT 共识的关键。至于为什么投票两轮就可以达到  $3f+1$  的容错效果、为什么至少要两轮，我也不知道。

忽略掉失败交易的操作，是在检查交易的过程中完成的。项目里有两次检查，共识前检查和共识后检查。有一些交易是没办法在共识前进行检查的，比如在合约里进行的写数据库操作，如果共识前就写库了，然后共识失败了，数据不就乱套了吗。所以只能共识后进行检查。这是第二个前提。

## 第一笔失败交易

我们根据 bug 的现象分析一下，第一笔失败交易的流程是正常的。一笔失败交易进来，在共识之前是不会检查出失败的，所以预提案的节点正常提出了一个块，分发给其他节点，进行第一轮投票。之后正常进行第二轮投票，在确认块的阶段，写入块之前，会进行共识之后的检查，检查过程中发现交易失败，并且这个块只包含这一笔交易，这个块就作废了，没出块。同时，其他节点也都会返回消息告诉提案节点，这个块没出来，这笔交易失败了。所以第一笔失败交易是正常返回结果的。

第二笔失败交易进来，按照同样的处理流程，一切都应该是正常的才对。因为即使是失败交易，即使是在确认块的阶段，如果检查失败了，也会广播处理结果给其他节点。整个协议中投票失败或者落块失败，都是用空消息表示。其他节点不会因为交易失败，就收不到消息苦苦等待超时。那么既然 BFT 协议的流程没有问题，为什么还是出现 bug?

这个时候要提到项目在 VRF 方面的改造。

在 BFT 的协议中，是需要一个节点去生成一个块，分发给其他节点开始进行第一轮投票的。那么由哪个节点来进行这个生成块的操作呢？总不能是同一个节点吧，那就太中心化了。Tendermint 的做法是依次进行，比如有 4 个节点，第一次节点 A，第二次节点 B，这样轮询。

VRF (Verifiable Random Function) 做的事情，是改变依次选择节点的方式。因为如果按照顺序来，那很容易预测到下一轮要由哪个节点去生成块，顺序可以预测之后，就存在节点被贿赂、节点被攻击等安全隐患。VRF 的功能是参数相同结果一定相同，参数不同则结果随机。把块高度、投票的轮数作为参数，就可以很好地实现，每一个块都能由随机的节点来生成，无法预测。这个改动也是作为项目的一个亮点的。

不过 VRF 存在一个问题，既然是随机的，那就有一定可能，第一次随机到节点 A，第二次也随机到节点 A，这样的概率还是不小的。如果节点 A 是恶意节点，然后由节点 A 连续两次生成块，会给网络带来一些负担，虽然不至于破坏网络，但也是一点小小的麻烦。所以项目为了解决这个问题，在 VRF 的基础上加了黑名单的机制。

如果上一轮是节点 A 生成块，就把节点 A 放到黑名单里。如果 VRF 的结果在黑名单里，就再 VRF 一次，避免重复选择相同的节点。

## 第二笔失败交易

### 不返回结果

结合 VRF 和黑名单，再来看看第一笔失败交易发生了什么。节点 A 收到交易，会先把这笔交易广播给其他节点，然后打包成块进行投票的流程。此时节点 A 在黑名单里。投票失败后，节点 A 返回失败，并且这笔交易已经不在节点 A 的交易池里了，因为已经处理过了。

那么节点 B 呢？块里面的交易验证失败了，但是交易池里收到的交易还在，因为这笔交易还没有处理啊，处理的只是广播过来的块里面的交易。这个时候是不是应该把交易池里面的交易删掉？对，但是没删。所以造成一个问题，节点 B 被选作生成块的节点，把这笔交易打包了一下，广播了出去。这个块当然也是提案失败的。此时节点 A、节点 B 都在黑名单里。

以此类推，就这一笔交易，一轮下来，4 个节点全在 VRF 的黑名单里。但是对这笔交易结果的返回是没有影响的，因为交易结果在节点 A 的时候就已经返回了。

第二笔失败交易过来了，所有节点全在黑名单里，会发生什么？当然不能选不出节点，节点全在黑名单里，黑名单就失效了。VRF 的结果是哪个节点，就是哪个节点。

分析一下第二笔失败交易。同样是节点 A 收到交易，假如这一次是节点 B 负责生成块，然后这个块验证失败了，节点 B 就会删掉这笔交易，对吧，这个没问题。

注意，删掉交易的同时，通知客户端，交易失败了，返回交易结果。节点 B 被选中，节点 B 生成块，节点 B 返回通知。但提交这笔交易的客户端，连的是节点 A 啊！

第一笔失败交易为什么会收到响应？因为黑名单还没有失效，所有节点都处理了一遍交易，所有节点都返回了一遍交易结果。现在黑名单失效了，只有节点 B 会返回结果，所以节点 A 的客户端收不到交易结果。

那为什么黑名单失效，就不能像第一笔交易一样，所有节点都处理一遍？

### 阻塞后续交易

接着分析一下第二笔失败交易，节点 C 还有交易啊，节点 C 上面的这笔失败交易还没处理呢，节点 C 就开始用 VRF 选节点了。

刚才提到，此时黑名单失效，VRF 选出哪个就是哪个。刚才选出了节点 B，这一轮有没有可能再选一次节点 B？黑名单失效，就变得可能了。这个时候如果又是节点 B 负责生成块，会发生什么？

节点 B 生成不了块，因为节点 B 已经没有交易了，它已经把唯一的失败交易，在上一轮就删掉了。也就是说，在新一轮的共识过程中，4 个节点全部在等节点 B 生成块，节点 B 自己也知道该自己了，但是节点 B 拿不出块，节点 B 直接放弃这一轮共识，进入下一轮，并且节点 B 没有发出任何消息。

在分布式系统中，没有消息是一件可怕的事情。其他节点都在等节点 B 呢，节点 B 自己玩了。这个时候，节点 B 的轮数要比其他节点快一轮。

在 BFT 共识中，有两个索引值，一个是块高度，一个是共识的轮数。同一个块高度，有可能因为块没确认，就经过很多轮共识。由于节点 B 自己没生成块，轮数增加了，其他节点还不知道。

现在，所有节点都在 VRF 的黑名单里，节点 B 的共识轮数高于其他节点，

如果节点 A 再收到失败交易，有两种情况。一种情况是 VRF 又选中节点 B 了，节点 B 提出的块会被拒绝，因为其他节点还在等节点 B 上上轮的块，它拿出了高轮数的块，是对不上的。另一种情况是，VRF 选中了其他节点，那其他节点首先要等节点 B 上上轮的出块超时。超时之后，其他节点把轮数最高的数值同步一下，共识就算恢复正常了，然后 VRF 再选。

但是注意，这可是一笔失败交易，此时黑名单仍然失效，即使共识恢复正常，也还是有概率重蹈整个覆辙，节点 A 仍然收不到交易结果。至于具体的概率是多少，就懒得算了。

### 总结

可以看到，这个 bug 是由很多系统性的不合理设计共同造成的，直接原因在于 VRF 黑名单的失效，因为所有节点都在黑名单里了。或者说，黑名单没有及时清空，原先的错误之处在于，只有

在块高度变化的时候才清空黑名单，可能是认为每个块的产生都应该由不同的节点来处理。这种想法的失误在于忽略了相同块高度的时候，共识的轮数也会发生变化，每一轮都会产生一个新的块。所以只要在共识轮数发生变化的时候，也清一下黑名单就好了。实际的代码改动只有两行。

上面写的东西，可能我自己也不想仔细去看，不好理解、抽象，而且文字的表达能力也弱，看起来费劲。这种类似状态机状态转换的文字描述，看起来是很痛苦的事情。尤其是内容和当时项目的耦合很深。总的来说，对于这个问题的分析和解决，我认为在逻辑上是自治的，能很好的解释成因和现象，以及用最简单的方式在表面上修复它。

时隔近 2 年的时间，我竟然还能记起来这些，感觉也是很奇怪。

# 一年的工作回顾

2022-08-27

来到公司一年多了，想要简单做一点阶段性的回顾，因为平时会把大致的工作内容记录在内部的 Confluence 上，所以总结起来会有据可循。

一年的时间内发生了很多事情，我交到了一个很好的朋友，也因此在工作上不是那么专心，相比以前不管是工作效率还是用心程度都大打折扣。不过总的来看仍然有很大收获。

## State channels

来到公司的第一个月主要是熟悉项目，这个项目比之前公司的项目规模大很多很多，功能上相当于整合了 Filecoin、IPFS、Raiden Network、Ontology 等公链项目，还自己实现了类似于 libp2p 的网络模块和网络代理。一开始看起来还是有点吃力的，一方面因为代码体量大，另一方面因为确实不了解公链，虽然知道区块链本身的技术模块，但不知道什么是 Layer 2、IPFS、PoC。第一个月把项目搭建运行起来，了解了上传下载的基本流程，主要看了 p2p 网络在协议层面的交互实现，第一次知道了 DHT 是什么意思。

第二个月除了深入熟悉代码细节外，做了一件事情就是把 State channels 中的路由查找从 DFS 换成了 Dijkstra 算法。项目里有一个类似于 Raiden Network 的 Layer 2，用来解决文件下载过程中要对其他节点频繁支付的问题。改路由查找是因为这个部分相对独立，不会给整个项目带来麻烦，至于 Dijkstra 算法可以考虑路径长度做出选择而 DFS 不能，这点优化其实没有意义，因为我们 DNS 节点不会那么多，这个和 Raiden Network 完全 P2P 的模式是不一样的。

后来两个月也就是第三、四个月，主要做的事情是在 State channels 中增加手续费，在中转节点上扣掉一部分转账金额。那是一个痛苦的过程，因为我当时不懂 Layer 2 也不懂 State channels。好在用了两个月的时间还是把协议搞明白了，由于项目场景的限制不能由发起转账的节点直接验证交易，引起一些 channel 状态不稳定的情况，不过无伤大雅。

进入公司后我几乎没有请教同事关于项目的情况，不管是项目的整体架构还是具体的代码细节，全部是自己去看代码、查文档，尤其是要面对很多自己完全不了解的概念，这样做当然是有意而为之。全部自己折腾效率低是理所当然的事情，不过好处就是，在那个过程之后我可以有足够的信心，仅凭自己的实力就可以搞明白那样规模的项目，完成该做的事情。

因为我是第一次跳槽，我对自己的能力是疑惑的，不知道在之前公司的感觉是错觉还是事实，我觉得之前公司的项目不行、技术也不行，而且跳槽是有工资的增长，我多少有点心虚，想知道在进入一家新的公司后，我能不能够称职地独立应付起这样的项目，有没有实力对得起工资。请教同事是多么简单的事情！在之前的公司我也受到了很多关照，不过我希望自己有独当一面的能力，这正好是个机会，必须不依赖外部帮助去解决问题了。

## Solidity 合约

第五个月，我在纠结 Layer 2 该往什么方向优化，有点难以下手，正准备解决性能低下的问题，但也没有思路。后来得到一个需求是把项目里的原生合约用 Solidity 写一遍，因为后续有想支持 EVM 的计划。当时我不了解以太坊也不了解 Solidity，花了大概一周时间看 Solidity 文档。

第六个月也就是今年一月份，用 Solidity 重写合约，那其实是一段愉快的时间，因为不太需要思考做什么、怎么做，照着现成的写就行，产出的代码量还大。只是用不同的编程语言，你知道的，换编程语言没什么压力。

第七个月今年二月份，在节点的 SDK 中加入对以太坊 SDK 的支持。折腾了一下以太坊的测试网发现不太好用，最后还是先用节点的开发模式了。

Solidity 虽然语法容易理解，但是由于 EVM 的限制，也有很多需要注意的问题，以及很多语言上的细节需要时间不断熟悉，当时用一个月写完合约，后面却断断续续用了不少时间去修改完善。总之在那个过程里，那个需求上，我学会了写 Solidity 合约，熟悉了以太坊智能合约的开

发，尽管对于生产级别的合约安全问题还缺少经验。

## 文件夹上传下载

今年三月份开始，做的一件事情是文件夹的上传下载。这个想法的起点是项目对 Git 的支持，想要支持 Git 协议，能够直接用 git 命令克隆 Git 仓库，结果发现普通文件夹的上传下载都没有，只有对文件的上传下载。

我们的项目用了一部分 IPFS 的 IPLD 协议，把文件转换成块进行传输，但是没有用 IPFS 的文件管理部分。IPFS 有一层针对文件系统操作的 API，可以统一处理文件和文件夹，数据结构之间还能方便地互相转化。我们是直接读取文件转化成块的，这也给文件夹上传下载的实现增加了难度。

这件事情一直断断续续持续至今，因为总是不断有各种各样的小问题出现，也感谢测试同事耐心的配合。文件夹的处理比单个文件复杂一点点，因为文件夹会存在无限的嵌套，文件夹内同时包含文件和文件夹，子文件夹内还会有文件和文件夹。以及其他问题像空文件夹、大文件的处理。

IPLD 节点储存在 Merkle DAG 的数据结构中，单个文件会生成一个 Merkle Tree，而文件夹需要做的是在上传的时候，把多个 Merkle Tree 组织起来成为同一个树，然后在下载的时候根据这个树反序列化成文件夹、把内容写入到磁盘上。

其实实现思路是简单的，这个树结构中有数据块 raw node 和用来做中间节点连接数据块的 proto node，只要把文件夹相关的额外信息写到 proto node 的 links 中，就可以把文件之间的关联信息储存传输到其他节点了。

不过在具体的实现过程中花了不少功夫，也绕了一些弯路，比如上传生成块的时候因为添加了额外的数据导致块数据验证不成功，不能生成完整的树结构，不得不深入到 IPLD 的代码里 debug；下载的时候对块数据的解析不熟悉，也没有意识到节点之间块数据的传输是没有顺序的，在功能不成功的时候一度怀疑整体思路出了问题。

由于块数据的传输是无序的，就需要在下载的时候自己整理块的顺序，排序过程中因为搞混了树的层序遍历和前序遍历，一开始深度优先生成顺序发现总是有几个块的位置错乱，debug 了好久才定位到问题改成广度优先。

这是一个无关紧要但是有意思的功能，在这个过程中加深了对文件的上传下载的了解。

## 文件的非对称加密

今年四五月份疫情严重，居家办公一个多月，做的事情是文件对非对称加密的支持，之前只支持 AES 的对称加密。

这个没太多可说的，就是要注意 ECDSA 是数字签名算法，没有加解密这回事，要用 ECIES 之类的混合模式，结合对称加密去实现对文件的非对称加解密。

## 支持以太坊账户

今年六七八月份，重点关注新 Layer 2 的方案，想要实现基于 Optimistic rollups 的 Layer 2。

Rollups 部分还没怎么动，目前停留在储存节点对以太坊账户的支持上。这一段不短的时间内，除了编译运行一下 Optimism 的项目、增加储存节点对多种网络模式的选项，还花了不少时间完善之前的 Solidity 合约、解决像合约大小超过限制需要拆分、节点真实运行过程中合约结果和预期不符等问题。

由于账户地址和公私钥都变成了另外一种格式，储存节点的协议消息也需要使用另外的签名方法，这些内容的改动都还在进行中。

## 总结

其实没多少事情，但也没怎么闲着。不算太认真，但也收获很多。

# 为什么要重视编程思想

2022-07-24

## 1

前两天遇到一个小问题，Solidity 写的智能合约超过 24 KB，不能部署到以太坊主网上，因为 EVM 对合约的代码大小有限制。于是考虑怎么减小合约的大小，当时对合约大小的概念都是模糊不清的。

其中注意到一个地方，合约是可以引入其他合约、调用其他合约方法的，只需要把部署后的合约地址作为参数传到合约里：

```
contract Demo {}  
contract Main {  
    Demo demo;  
    constructor(Demo _demo) public {  
        demo = _demo;  
    }  
}
```

合约大小包括引入的合约吗？EVM 在执行合约的时候，会不会先把其他合约的代码也加载进来，然后一起运行？代码大小的计算要包括所有合约？那就麻烦了。

后来注意到，可以使用接口替代合约：

```
contract IDemo {}  
contract Demo is IDemo {}  
contract Main {  
    IDemo demo;  
    contract(IDemo _demo) public {  
        demo = _demo;  
    }  
}
```

接口的代码量一定是少于具体实现的，因为接口不包含方法体，把引入的合约全部替换成接口，合约不就小多了？

当然，在这里纠结的不是 Solidity 合约怎么写或者合约代码大小怎么计算的问题，后来搞清楚了。比较在意的是，那个时候突然有点恍惚，用接口和直接用合约，有什么区别？

之前给合约定义接口是为了提供一个对外方法的描述，这里才意识到接口可以替代合约本身，直接用来定义变量，并且使用接口定义的变量，去调用合约里面的方法。但为什么可以呢，它不就是一个接口吗？

## 2

如果你刚学习过 Java，或者使用 Java 作为工作语言，一定会有哑然失笑的感觉，这个问题太幼稚了，这不就是多态吗？

上第一节 Java 的课程，老师就告诉我们，面向对象有三大特性，封装、继承、多态，这句话时至今日我都能想起来，这是多么基础的概念，结果在工作多年后的今天，我竟然在实际工作上因为如此简单的问题犯了难，一时没反应过来，用接口作为类型的写法是什么意思。这太荒唐了。可能也是因为很久没写 Java，现在一直在用 Golang。

不得不说 Java 是面向对象编程语言的标杆，Solidity 虽然是一种看似新的用于智能合约的脚本语言，揉杂了多种语言的特性，但基本的编程思想还是基于面向对象的。合约就是类，部署一个合约就是实例化了一个对象，合约地址就是对象的内存地址，合约调用就是对象的方法调用……

只要是支持面向对象的编程语言，就包含有面向对象的特性，就可以使用面向对象的写法，就离不开最基本的像多态一样的特性。从面向对象的角度去理解，Solidity 有什么难的呢？无非不就是换了一些表面上的形式，编程思路甚至可以一模一样，此外再添上一些区块链特有的概念，像转账、块高度之类，就没了。

从编程语言的角度看，Solidity 和 Java 那样成熟的语言自然没法比，面向对象的特性是残缺的，modifier、require 之类的写法看似好用却增加了很多理解成本，而且代码结构也变得不是太统一。EVM 怎么能和 JVM 相提并论呢？但作为一种轻量级的脚本语言，Solidity 又要使用静态类型那样冗余的写法。

当然要注意，编程思想是先于编程语言的，我仍然会认为形式上的编程语言[不值得学习](#)，但是不否认从学习编程语言的角度入手去学习编程思想。比如[多态](#)这个概念，含义是使用统一的符号去代表不同的类型，包括三种类型的解释，一是支持多种类型的参数，对应 Java 里方法的重载，二也是支持多种类型的参数，对应 Java 里的范型，三是子类型，也就是把接口作为类型，对应上面提到的场景中的多态的含义。

面向对象是一种编程思想，包含很多计算机科学的概念，而 Java 是一种完全的面向对象的编程语言，不但涵盖众多有用的特性，而且实现的完整漂亮，如果你学习了 Java，自然也就知道面向对象是怎么回事了，受用无尽。从这个角度看，和 Java 相比，Golang 有什么值得学习的地方吗？是 struct 的写法还是 \* 号的用法？可能 Golang 更像是一种快餐式的语言吧，可以很方便地 go func()。不过要是为了学习，就不是太推荐了。

花了几分钟看 Java 文档的目录，倒是能很快想起来那些内容，毕竟实在是太基础了。也是要告诫自己，别忘了代码怎么写。

# 对 Web 3.0 的理解

2022-06-19

## Web 3.0 开发

Web 3.0 是一个几年前就存在的概念，可能随着区块链的推广越来越有名了。当人们还不知道 3.0 版本的 Web 会是什么样子的时候，区块链出现了，尤其是以太坊的 dApp 提供了一种很大的可能性，于是 Web 3.0 就和区块链、去中心化、自我主权这些概念绑定在一起。

前几天提到说。「Web 3.0 开发」是可以作为一种职业定位去描述的，而且这个词可以涵盖区块链开发的范畴，立意比「区块链开发」这个描述高一点。

Web 2.0 时代，我们说的 Web 开发指普通的前后端开发，前端用所谓的三大框架 React.js、Vue.js、Angular.js 结合组件库，后端用 Spring 全家桶，加上各种中间件 Zookeeper、Kafka、Elasticsearch 之类，还有常用的数据库 MySQL、Oracle，就是 Web 开发的常用技术栈。

Web 3.0 开发的技术栈，可能会演进为 Remix、Hardhat、Ruffle 这些智能合约的开发工具和框架，人们像关注 Java 的语言特性一样去关注 Solidity，以及各种区块链节点的搭建运行调用、二次开发，甚至及区块链节点本身的开发等等。当然，很难简单地把这些技术栈去和 Web 2.0 一一对应。一个简单的例子是，当你从事区块链开发的工作，你已经很难用前端开发或者后端开发来形容自己的工作内容了，就只能是区块链开发，或者智能合约开发，或者其他的方式。

现在已经有一些岗位在用「web3 开发」的形容了，不过我们要区分清楚 Web 3.0 和 web3 不是一回事，目前很多岗位说的 web3 指以太坊的那个 web3 框架。我们需要一个新的描述，同时我们需要有一个更好的、更有前景的职业定位，那就是 Web 3.0 开发。

我这里想说的是，要相信我们走在正确的道路上。

## 一级市场和二级市场

区块链的一级市场，指比特币、以太坊这种原生的链。二级市场更多是基于这些链，衍生出的一些项目，基于以太坊的项目尤其多，Layer 2、预言机、NFT、ENS，都属于二级市场。

有一些团队是做一级市场也就是区块链开发的，Solana、Filecoin、Neo 都是在以太坊之后出现的，Dfinity 的 IC 也是处于活跃的一种一级市场的例子，再比如像 Bitcoin SV 是在做“支持智能合约的比特币”这样的事情。

还有很多创业团队是在做二级市场，比如有从 360 出来的去做智能合约的安全，把合约扫描一遍报出安全漏洞给你；还有做 NFT 的交易协议，去定制一些类似 NFT 交易所的 API，想建立通用的交易网络；还有炒元宇宙概念的，给虚拟人物定制不同样式的衣服；也有基于 IC 做去中心化邮箱的等等。以太坊的各种扩容方案当然也算二级市场，OP 前段时间还发行 token 了。

从商业角度没有什么高下之分，从技术角度也不好说简单和难，不过我觉得还是一级市场更基础一些，但是技术上的发展相对缓慢，花样没那么多。具体倾向于哪一种看个人意愿了，这里想提醒的是，Web 3.0 开发是统称，要了解这些不同层级市场的区别。

## 去中心化是历史的倒退

刚才说希望 Web 3.0 是有前景的方向，这个部分想说的是 Web 3.0 的前景也没有那么好。

想到这个话题是在关心钱包安全的时候，意识到一个问题，就是账户的私钥一旦泄露，你就永远失去了对账户资产的控制权，或者说别人永远拥有了你账户资产的控制权。

因为我们知道，账户地址是可以从私钥解码出来的，私钥就是你的资产，在备份钱包的时候，备份的就是私钥。你的私钥泄露，就相当于把金钱摆到别人手里，至于别人会不会及时拿走，你能

不能在对方动手之前抢回来，那就是另外的问题了。

这个和传统的账户模型是不一样的，你不可能说你的用户密码是你的财产，因为中心化账户是基于 KYC 的，你只要能证明自己的身份，身份证或者指纹或者长相，都可以找回你的财产，因为财产是和你绑定在一起，而不是你的账户，你的账户密码是可以修改的，即使泄露，别人也只能在短时间内拥有你账户的控制权，你把密码改掉，别人就没办法了。

私钥是不可能更改的，你能做的，就是及时把资产转移到另外的私钥。去中心化的世界有意区分了身份和数字身份的概念，增加了数字身份的主权，但同时也削弱了身份对数字身份的控制能力。

这里衍生出的问题就是，去中心化的资产安全吗？把钱拿在自己手里，比把钱存到银行，更加安全吗？考虑到比特币诞生的背景，是出于对中心化机构的不信任，才有了去中心化的理念。

想想吧！一开始就是没有中心化机构的，人们以物换物，打一开始，就是去中心化的世界。后来为了降低个人保护自己财产的成本，为了增加对坏人作恶更有力的惩罚机制，人们共同组建起中心化机构，保护大部分人的利益。

现在炒作去中心化的理念，不正是一种历史的倒退吗？去中心化并非新产生的事物，而是早就已经存在的、被人们选择性抛弃的东西。

不过现在的去中心化和以前的去中心化，最大的不同就是现在的技术手段更为先进，有可能做到之前做不到的事情，把世界推到一种新的愿景上。但是也要注意现在的技术不是那么先进，还远没有发展到那种程度，区块链的技术瓶颈非常多。

所以我的观点是，现在的去中心化理念不是中心化世界的演进，而是中心化世界的补充。在接下来的时间，中心化和去中心化会同时存在。

要注意的是，去中心化不等于 Web 3.0，Web 3.0 是 Web 2.0 的演进，因为版本号增加了。Web 3.0 将是中心化和去中心化同时存在的时代。

## LUNA 归零

前段时间有一件搞笑的事情，有一天，LUNA 的价格早上还是 80 美元一个，晚上的时候就跌到 1 美元一个了。在接下来的三四天，LUNA 的价格从 1 美元，跌到了 0.00001 美元。几天之内，近万倍的跌幅。曾经号称前十的加密货币，突然归零了。

我粗浅的理解是，UST 有一个交易池，当短时间有大量卖出的时候，交易池会有小幅的倾斜。当时先是小幅的波动，然后随着社交媒体的传播，大量散户失去了对 UST 的信任，开始大幅卖出，越卖价格越低。Terra 团队是有 5 万个比特币作为储备的，当时也及时打进去想把平衡拉回来，结果比特币也在跌，质押进去的比特币在结算的时候已经不值预期那么多钱了，没能把价格拉回来，后来 Terra 团队也放弃了，任由价格下跌。

LUNA 的事情发生后不久，看到有的人讨论说，LUNA 还有机会吗？有一种机会是，Terra 团队还有 20 亿，等 UST 的价格跌到总市值小于 20 亿的时候，Terra 团队可以把市场上所有 UST 都买下来，销毁掉多余的 UST，只留 20 亿个，UST 的价格就可以回到 1 美元了。不过 Terra 团队可能没打算那么做，后来发行了新的 LUNA。

LUNA 的失败不意味着算法稳定币的失败，有的团队也在研发新的算法稳定币，据说是想把美联储的运行模式，用算法模拟出来，正在写白皮书。

在 Web 3.0 宏大的时代背景下，LUNA 的事情就算是先行的笑料吧。

## Web5

最近新出一个 Web5 的概念，就是 Web 5.0 的意思。提出这个概念的人说，跳过 Web4 是因为 Web2 + Web3 = Web5。好家伙，不愧是 Web5，提出 Web5 的能是一般人吗？但凡对软件工程有了解的敢这么说？

简而言之，我的结论是，Web5 一定不会成功。不管它叫 Web5 还是 Web6、Web7，它的理念还是围绕去中心化、SSI 那一套，还在我理解的 Web 3.0 的范畴之内。如果认真了解过 DIDs 的理念，就知道现阶段所谓的 Web5 完全是噱头了。

# 人际交往中的心态问题

2022-05-25

1

我刚才想到一个问题，有点想不通。

假如你有一个价值 10 块钱的东西，另一个人拿出价值 5 块钱的东西，非常想和你交换，那么这个时候，你愿意换吗？

那我肯定不会跟他交换呀，你既然要跟我交换，你肯定得等价交换呀，那凭什么那个什么，我这个比你的这个价值高，你拿价值低的跟我交换，这对于我来说不公平啊，我不跟跟他交换。

2

这个时候你有两种做法，

一种是，拿出价值 5 块钱的东西，和他交换，剩 5 块，得到一个东西；

另一种是，不交换，剩 10 块，没有得到东西，等有人愿意拿出 10 块钱的东西，再换。

选哪一种？

这个就是因情况而定，如果我特别着急，想跟人家交换东西嘛，然后我可能选择第一个，因如果说我不想着急交换，我对这个东西还挺喜欢的，那么慢慢儿等呗。

3

换另一个问题，

假如一个人有 10 块钱，愿意把 10 块钱都拿出来，在某一个时间点拿了 3 块出来；

另一个人也有 10 块钱，总共愿意拿 5 块钱出来，在相同的时间点，把 5 块钱都拿出来了。

表面上拿出来的 5 块大于 3 块，但是他们后面的意愿是不一样的。

你觉得他们，谁的代价更大一点？

应该是第一个，代价大呀，他愿意把他的全部都给我，只是可能那一段时间有困难吧，只能拿出三块钱，但他之后也愿意拿出七块钱都给我呀，所以他付出的代价更大一点呀。第二个，我感觉他怎么说呢，虽然有，但他并不是全部都给我呀。

4

如果你只需要 5 块呢，多了也没用。

情况就不一样了。

我觉得越多越好。我比较贪心。

分析

如果你只需要 5 块钱，但是有人愿意给你 10 块钱。

他知道你只需要 5 块钱吗?

如果知道，他的意愿就是自作多情。

如果不知道，你为什么没有告诉他?

## 结论

没有朋友的人，有 10 块钱。有朋友的人，只剩 5 块钱。没有朋友的人，应该拿 5 块出来，还是等 10 块钱的人?

没有对象的人，有 10 块钱。有对象的人，只剩 5 块钱。没有对象的人，如果愿意拿 10 块出来，即使现在只拿出 3 块，你的 5 块怎么比?

## 备注

上述钱的单位是相同的，不存在汇率问题。

## 延伸

如果考虑汇率的不同，每个人钱总量、剩余数量的不同，以及随着时间的变化，这些变量都会改变，问题更复杂一点。

# 补充一个案例

2022-06-18

## 5

假如你现在被关在监狱里，痛苦不堪，经年累月，你逐渐习惯了这样的痛苦。

你一直等啊等，等一个能为你打开牢门、放你出去的人。

终于有一天，你等来一个人，他为你打开牢门，只不过，是有时间限制的，两个小时之后，你还需要回到牢房里。

他答应你，每天都会来这里，为你打开牢门，虽然只有两个小时。

此时，你会感谢他为你带来的短暂的自由吗?

你越是能够体会自由的快乐，就越难以忍受在牢房里的痛苦。你多了两个小时的快乐，但其余时间的痛苦也加倍了。

也许你甚至会怨恨，为什么来的是一个只能打开两小时牢门的人，而不是一个，能够真正放你出去的人?

此情此景，你会是什么样的感想?

## 6

随着时间的累加，你对打开牢门的人有所了解。

你发现，原来他不只是给你打开牢门，你隔壁的狱友，牢门也会打开两个小时，而且也是他。

这个时候，你会嫉妒，你会害怕。为什么不只是你，为什么还有别人，会不会有一天，他不再为你，打开牢门。

你本想过主动放弃，但你没有放弃。也许以后连放弃的机会都没有了，你同样抓狂。

你质问他，为什么？

他一脸无辜，完全不知道做错了什么。

他再次承诺，不会忘记你。

7

再后来，你发现，不只你，不只你隔壁的狱友。

还有很多人，也是两个小时，也是他。

他的时间被分散在很多人身上，所以你只有两个小时。

其他人的时问，有的多，有的少，反正你，是两个小时。

你期待的，是一个能放你出去、给你二十四小时的人，结果等来的，是只能给你两小时的人。

怎么办？

## 延伸

如果不是单方面的保释一样的需求，而是双向的、相互为对方打开牢门，情况会有什么样的不同？

# 心理疾病诊疗记录

2022-05-02

1

- 你好。
- 主教你好，我来忏悔我的罪行，我最近不太开心，我想我精神虐待了一个女孩。
- 别着急，慢慢说，一点一点来。你是说，你虐待别人，然后自己不开心了？
- 是的，这说来复杂。我认为在行为上是我的错，但我的行为因她而起，所以归根结底，是她的错。
- 你把我搞糊涂了。从现在的情况说起吧，你们现在是什么关系，关系怎么样？
- 我们是朋友，也许……如果她愿意的话。我们之前吵架了，后来和好了，她说没事了。但是我不确定她到底怎么想，不知道她的主意会不会改变，你知道，女人嘛，总是变来变去的。
- 你们是情侣吗？之前是情侣吗？
- 不是，她有男朋友了。一提到这个我就伤心。
- 你的意思是，作为普通朋友，吵架了，然后和好了，你不确定她有没有原谅你是吗？这个精神虐待有什么关系呢？你又为什么伤心呢？
- 你问的问题太多了，我不知道先回答哪个。

2

- 是我的失职。一个一个来，你不确定她有没有原谅你是吗？
- 她原谅我了。我虽然担心她会改变主意，但是又没什么理由怀疑她。或者说，我害怕她改变主意。
- 你害怕她改变主意，是因为害怕失去她这个朋友？
- 是的。
- 你为什么没有理由怀疑她？
- 因为她说要相信她，她不会骗我。
- 她骗过你吗？
- 几乎没有，也不算骗，就是有过一些事情。
- 所以你不相信她，害怕她骗你。
- 我觉得她不会故意骗我。
- 你说有过一些事情，是什么事情呢，这些事情导致你不相信她吗？
- 是的。她并不总是说到做到。我担心她做不到她说的，比如原谅我。
- 言行一致倒是个可贵的品质。

- 我本以为，这是做人的基本素养。
- 说一说具体的事情吧，她到底怎么说到做不到了。
- 她答应我什么时候去玩，但到了约定的时间，由于种种原因，就没有玩。
- 她违约了？
- 也不算违约，中间发生了很多事情。
- 你刚才说的“种种原因”，你知道是什么吗？你觉得那些原因合理吗？
- 合理倒是合理，但都是因为她，她总是说一些自己的原因。我在怀疑，她不想和我一起玩了，于是找了一些借口，编造出那些理由。
- 你怀疑她骗你。
- 对。
- 她不是说，她不会骗你吗？你刚才说，你相信她不会故意骗你。
- 啊，倒也是。信还是不信，这是个问题。

### 3

- 你的逻辑是矛盾的，如果你相信她，就不应该怀疑她骗你。如果你不相信她，就……不相信她，还和她交朋友？
- 我只是害怕，她不想和我一起玩。
- 为什么不直接问她，她是不是不想和你一起玩？
- 她的回答一定是：“不是”。我没必要问。
- 你怀疑她的回答是在骗你。
- 不排除这样的可能。
- 为什么？
- 因为她几乎没有过，真的和我一起玩。如果只是问问，她说话可好听了。
- 所以你想让她和你一起玩，证明她说的话是真的。不然你没有安全感是吗？
- 安全感……我只是拿不定主意，该不该相信她。我想要相信她，和她做朋友，但是我需要一些事实，证明她说的话是真的，让我能够相信她。这确实是安全感的问题，我没有安全感。
- 换个角度想，她为什么要骗你？
- 也许她说谎成性，或者习惯说善意的谎言。善意的谎言也是谎言，我讨厌谎言。可能她为了不伤害我，才说好话骗我的。但是她不明白，这同样会伤害我。
- 无论是不是谎言，她的本意都是好的。
- 我希望她直说。
- 有区别吗？
- 区别太大了，如果她说不想和我玩，那我什么都不会去想。如果她编造了一些理由，我还

是会想着以后约她玩。

- 你觉得她吊着你?
- 倒也……我相信她不是吊着我，至少不是故意的。
- 你还是相信她的。
- 她本意不坏。
- 你还怀疑她吗?
- 不了。如果我不相信她的回答，但是相信她的人品，我的逻辑就是混乱的。我应该相信她。

#### 4

- 回到一开始的问题，你说的“精神虐待”是什么意思?
- 是我的错，我做了很多对她不好的事情。
- 不具体说说吗?
- 忽然不想说了，我承认是我的错，只希望有机会能够弥补我的错误。
- 那关于你的“伤心”呢?
- 也没有必要提了。
- 我们的对话会不会结束得太潦草了?
- 已经足够了，

#### 附

#给闹掰朋友发的短信#

我最近明白了一些道理，我知道我哪儿错了，但是我不会告诉你，因为我怕理解的不对、不到位、不深刻，因为 talk is cheap。我只希望有机会，用实际行动证明，我可以做得比以前更好。

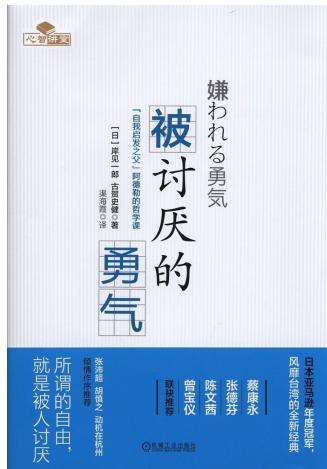
前段时间玩九不思议街，我拿的角色是汉诺和莎士比亚，故事以我的角色写的一句诗结尾，当时玩本的时候就有点破防。

那句诗是：“你若呼唤，我必归来。”

– 2022.04.21（新浪微博）

#### 附

推荐一本书 《[被讨厌的勇气：“自我启发之父”阿德勒的哲学课](#)》，作者是岸见一郎。



内容非常值得一看，带给我很大启发，主要是用“阿德勒心理学”（个体心理学）来解释诸多心理问题，是我最近看的几十本情感类和心理学相关书籍中，个人认为最好的一本。

# 如何面对误解

2022-04-21

所有误解都来源于，用自己的思维方式，去理解别人的行为。

每个人都是独立的个体，善良、自由、快乐。



每个人都有属于自己的世界观，那是在经历过曲折人生之后沉淀下来的，用以面对残酷世界和复杂人性的武器，也许像一面魔法镜，有丰富而神奇的魔力。

世界观给予每个人不同程度的理解能力，可以把别人的话在内心中翻译成，自己能够理解的含义。



当你想要表达某种想法，你会将想法转换为语言，转换的过程由表达能力负责。你的魔法镜是红色的，你说出的话也是红色的，这句红色的话是善意的。



经过一些媒介的传播，你的话会到达对方的位置。



不幸的是，对方的魔法镜，是蓝色的。你理解这样的情况，因为很难要求每个人，都拥有和你一样的理解能力、一样颜色的魔法镜，这个世界本就是多姿多彩的。

所以很可惜，你红色的话，经由蓝色魔法镜的解析，被理解为了粉色的含义。红色是不带恶意的，但粉色，就包含了一些其他意味在里面，偏离了你原本要表达的心意。



对方基于粉色的含义，用蓝色的话语回复了你。对方在回复你的时候，就已经有一些偏颇了，因为对方误解了你的意思。



蓝色的内容也许没有恶意，也许带有一点恶意，总之会朝你飞来。



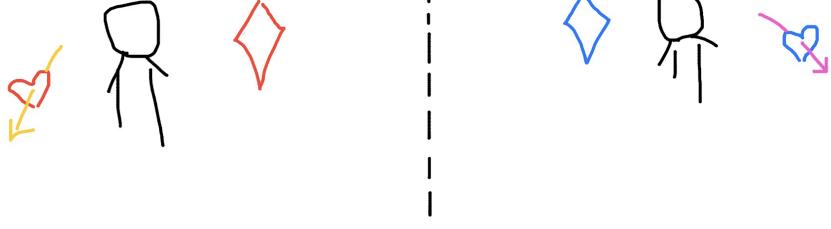
再次不幸的是，你的魔法镜是红色的。对于对方蓝色的内容，你的魔法镜，会把它解析为黄色。



这实在是加深了误解。原本赤红、纯蓝的两颗心，都感觉受到了不可原谅的伤害。对方怎么可以这样！



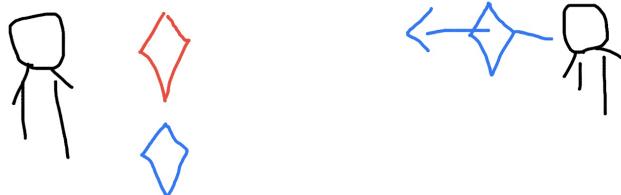
久而久之，两个人的隔阂越来越大。



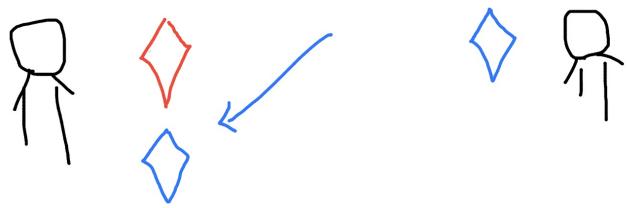
作为聪颖的魔法师，你力图改变这种状况。经过自己的勤学苦练，你想办法拥有了蓝色的魔法镜。



今后，当对方使用蓝色魔法镜和你交流。



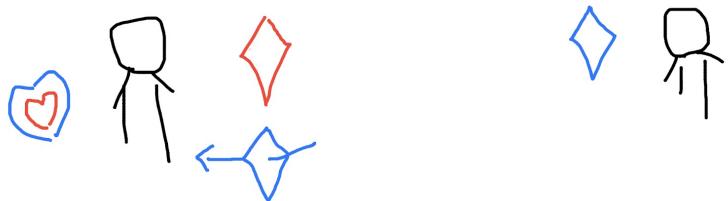
你可以拿出自己的蓝色魔法镜，来解析对方的内容。



经过蓝色魔法镜的解析，对方善意的内容可以原封不动到达你的脑海里。



你不会因此而受伤了。



当然，你没有理由去要求别人，拥有和自己一样的红色魔法镜。这太难了，且不说别人愿不愿意，对方也不一定有足够的实力，驾驭你的红色魔法镜。

如果你想和对方沟通，只要拿出自己的蓝色魔法镜就可以了。

总而言之，就是让自己变得强大。

# 之前公司（UMF）存在的问题

2022-03-11

## 2021.08.31 原文

我之前任职的公司是传统类型的电信服务类企业，话费支付起家。区块链部门是创新业务部门，内部孵化项目和产品，属于成本部门，公司给钱，但也有一些收入预算。部门收入的来源主要是：

- 政府补贴
- 招投标项目
- 熟人介绍项目

都是交付形式的项目，招投标项目是按照指定要求开发软件然后全权交付出去，熟人介绍项目是把内部开发的区块链产品交付出去，都要提供开发联调和售后支持。如果没有项目中标、没有熟人介绍，就没有收入了，存在一定收入压力。

这种营利方式属于相对好理解的模式。很早以前我还没有了解部门收入来源的时候，就无意间和同事说，“现在已经不是靠卖软件挣钱的时代了”，暗指部门领导的挣钱思路有问题。同事问，那靠什么？我随口说，“卖服务啊”。我当时并没有关注相关问题，只是闲聊而已。

想到这个话题是因为，今天问到 HR 公司盈利模式的问题，“我们又不是像卖煎饼一样，卖一个挣一个的钱。为什么互联网公司的工资高？靠的不是直接营利的收入，是投资人砸钱，花投资人钱。互联网公司的钱都不是真实的钱，很多互联网公司，为什么财报亏损但还是很多投资人愿意投资？他们都是先占有市场和用户，然后上市，最后由股民掏钱了。我们也是有 IPO 的计划……”

当然，对这些内容的真实性和准确性存疑。但能肯定的，和之前公司的销售模式完全不同。

HR 和我提到说，有什么问题可以随时问，我们的企业文化就是“坦诚沟通”。我说，可能直接问只会得到一些「道貌岸然」的回答，有些事需要不断「试探」才能知道真相。我想到，我这样的行为方式，可能和之前的经历有关。

如果你提出一个现实的具体的问题，一个人在回答的时候说了很多，扯得很远，某银行的金融方式什么，某大学的教授说了什么话，某体制内机构的高官是什么样的经历，公司某高层怎么要求，公司某员工有什么样的行为，名人名言、历史典故，等等。说了一圈，你发现他并没有正面回答你的问题，好像回答了，又好像没有。偶尔一两次你会觉得，这个人水平很高，接触的都是一些高级的人，说出来的都是大道理方法论。

时间长了以后，你发现，每一次具体的实际的问题，他都会用方法论来回答，顾左而言他，就是不正面回答你。甚至很多次说出来的都是相同的事例不同角度的道理，从具体问题总结出方法论是一种高级的技能，但如果一味讲究方法论，也会是一种灾难。所有问题一说出来啥都懂啥都明白，怎么回事别人是怎么做的我们应该怎么做，但就是解决不了现实的问题。后来你也许会觉得，可能他不是不想回答，是他也不知道问题的答案。有人说，其实他也很迷茫。

在具体的工作内容上，他唯一的要求就是“客户认可”，也就是能赚钱的、最好有直接收益的。你可以把产品相关的技术方向全部列出来，但是发现在所有的技术方向上，没有一个是所谓“客户认可”的，甚至全国有吗？是什么？客户怎么会因为你使用了某一种技术而买账呢。你明知道项目存在各种各样的问题，知道该怎么改善那些问题，但是在“客户认可”的判断依据下，这些问题没有一个是需要解决的，没有一个产品的技术方向是可以深入的。

“客户认可”绝对是再正确不过的判断标准，公司做什么事情一定是为了营利，付出的成本要体现在收入上无可厚非。但是，但就是感觉这玩意儿很难啊。公司本身是那样的营利模式，但是在那种营利模式下，国内挣钱的都没有几家。

尤其是看到国内某开源项目如火如荼势如破竹，部门的产品寻找突破口更加困难。在近几年都没

有显著创新的情况下，没有技术储备、没有发展路线、没有商业资源，甚至工程质量都一言难尽，当你说产品的竞争力不足，又会听到说“其实我们公司不是专门做这个的”，公司的高层几度放弃部门和产品，这些都预示着部门在走向灭亡。

## 2022.03.11 更新

最近注意到服务集成这个软件开发的领域，以及在这个方向上有所成就的[创业公司](#)，同时看到有人来公司面试，偶然想起一些以前的事情，回想起我在上家公司的经历。之前文中没有具体说的是，部门的市场定位不清晰。当然造成那样的结果，有很多历史原因，公司一开始是有靠山的子公司，作为甲方立场很足，后来转变为乙方的形式，没有活跃在市场上的基因。这里只说结果。

如果部门的定位是拥有技术基础去自研产品，那么问题就很严重。

我在正式接手核心产品的开发后，发现项目的代码在工程上有凌乱不堪，不是代码乱，是没有顶层设计，没有明确的模块划分，没有清晰的目录结构，没有靠谱的软件设计。项目支持四五种数据库，然而数据库的配置方式竟然不统一，分散在不同配置文件的不同位置，哪儿生效哪儿不生效，哪个功能好使哪个功能不好使，要么靠经验，要么靠猜。API 的设计也很糟糕，我在《[随想](#)》中提到过关于 URL 参数的问题，另外 URL 的配置和参数的校验是写在单个的配置文件中，意味着如果在智能合约中想新增 URL，就需要改配置文件然后重启节点。至于配置规则的热加载，好像没有人关心。

智能合约的机制也有问题。交易在提交到合约后，交易的检查和执行分为两个步骤，检查函数的入参和出参都是 bitMap，必须要严格保证入参和出参的长度一致，否则节点就会 panic，因为外面是用循环处理的，会 out of index。问题在于，那可是智能合约，怎么会用那么生硬的写法。后来前面的人告诉我，写智能合约的原则就是，“绝对不能出错”，因为说是智能合约，其实是和项目耦合很深的功能模块，美其名曰系统合约，是底层链的开发人员去开发的，而不是交给用户使用。开发合约，就需要对底层链有足够的了解。当时的人似乎还对这种事情有一点自豪感，感觉像是，“我们能写，因为我们比较熟悉”，丝毫不认为那是一种功能的不健全，而认为是有门槛的 feature。听说本来是没有打算支持智能合约的，由于需要的不断扩张，就硬生生加上了。当时的某人还拿 leader 的某篇文章奉为真理，说，其实区块链也不一定需要智能合约，对那种和广义智能合约理念背道而驰的设计大加赞赏。

多写几个合约后，就会注意到每个合约的检查函数上，都会有一个判断交易是否为空指针的语句。本以为系统内部的函数调用，怎么会凭空出现空交易呢，经过复现和排查，发现在并发情况下，队列偶尔会出现异常，push 一批交易进去，pop 出来就有空交易了，这纯粹是数据结构的问题。然后虽然定位到问题，但没有去解决，因为大不了每次都在合约上写个判断，算是我偷懒。也不知道前面的人，还埋了多少隐性 bug 在里面。

当用合约处理业务的时候，就会发现数据库的读写性能会成为交易性能的瓶颈，比如 MySQL，而且存在一个我一直没想通的问题：合约里的检查函数，怎么判断交易是成功还是失败？因为合约是要针对业务去开发的。合约对交易的检查和执行，都是是在 BFT 共识的 commit 阶段，这个时候已经完成共识了，检查函数并不能去预执行数据库的写操作（如果合约依赖数据库的特性比如事务，区块链就没有意义了），难道要把所有有可能失败的场景全排除一遍？是语义层面的排除，还是执行层面的排除？即使能够枚举出异常，又会损耗多少性能？那可能会产生疑问，为什么不在共识前检查？共识前的检查也是有的，但不管在共识前还是共识后，对数据库的操作总量不会变，对性能的损耗不会变。（延伸思考：为什么公有链不存在这样的问题，联盟链存在。）

版本管理的混乱也是在工程方面的问题之一，甚至都没有人能说清楚，当前的版本号到底是多少，是 2.0 吗？配置文件里可还写的是 1.4，是 2.0.1 吗？仓库里可还有 2.0.3，但不知道是谁改的，有哪些变动。甚至主干代码中会出现用于测试的 case，有些需要异常场景测试的情况，比如在 BFT 共识过程中恶意投票，只能通过改代码的方式观察效果，结果那部分代码就保留在了项目中，可以通过配置启用。其他原因的代码冗余也存在，比如智能合约的公用接口，为了兼容 UTXO，就不得不增加对应的接口，但其他合约完全用不到那些，又不得不实现。

项目在技术方面是存在各种各样问题的，包括我之前解决的由于使用 VRF 导致共识在提案阶段黑名单失效的问题，都说明系统尤其是在比较核心的地方都不够完善，而重构项目的成本又非常高。项目存在的最大价值和意义，就是参加一个行业内知名的测试活动，测试通过后某机构会为企业颁发证书，以证明这个软件是合格的、有资质的、符合标准的。然后企业拿着这个证书就可

以宣传、投标、卖钱。至于软件本身好不好，并不重要。我当时光看测试项还感觉没啥，都是一些对区块链的基本要求，能通过测试没什么大不了的。直到亲手操作后发现，测试过程中存在大量的困难，都是人为困难，由于之前开发人员的不专业、团队管理的松散，以及项目本身很多不合理的设计，加上文档和人员的流失，都大大增加了测试过程的准备难度。也许大家没有意识到，什么是有效的困难，才造成了一种项目很好的假象。

如果部门的定位是服务集成，提供解决方案和技术支持，也是不合格的。

除了区块链底层，部门还有 Bass、中间件、SDK、浏览器之类的项目，会涉及到 Kafka、Zookeeper、Redis、普罗米修斯等组件，但用的很浅，本身技术含量低，整体上做的也不到位，没有产品、没有 UI、没有用户思维、没有 owner 意识，全是在有业务需求的情况下临时改进，结果每次都手忙脚乱，总是以优先应付客户为要务。服务集成的事情，可以简单也可以难，可以做好也容易做不好。然而部门另一方面又不太想做服务集成，比如在用 Hyperledger Fabric 做一些事情的时候，leader 就说：客户会问，你们底层用的 Fabric，只是拿来用，也没干什么呀，为什么要收钱？至少能体现出部门的定位有多混乱了。

# 我为什么不学习编程语言

2022-02-16

## 2021.06.27 原文

这里的“编程语言”特指编程语言具体的实现形式，而不是编程语言的设计思想或者语言特性。

- 因为我早已知晓顶级编程语言专家的样子。
- 因为学习某种编程语言的某个特性，对我而言已经是手到擒来的事情。

至少在短时间内，我会保持和坚持目前的观点和做法。

## 2021.08.08 更新

昨天剧本杀，遇到一个 00 后的女学生，拼场来的，但同时也是那家店的兼职 DM。稍有感慨，自己 20 岁的时候在干嘛？

旧博客的链接一直挂在博客最下面，一般也不去看。稍微认真回忆了一下，感慨的事情先不管，我忽然明白，为什么我在毕业的时候会看不起同学的编程水平，为什么我刚参加工作的时候可以以至少平等的水平面对同事，为什么最近在面试过程中即使代码能力受到质疑我也不为所动，为什么现在我已经不屑于学习编程语言在形式上的那些东西。

站在国家图书馆的书架前，我忍不住回想起自己大学时候在图书馆里度过的日子。

## 2021.08.29 更新

我又来补充这个问题了。/捂脸

在 QQ 空间上看到一个大学同学发的动态，是一个 Geek 风格的同学。最近的一条是「重学 Unix 高级编程」，上一条是「B 站上某技术会议的视频」。我曾经也尝试去找一些技术会议、meetup，后来发现除了演讲者本人受益，听众都是在陪跑而已，你不会真的指望从中学到什么。会议是一种圈子里面的事情。

对于 Unix 编程或者编程语言这样的东西，我心底是喜欢的，我能想象到，亲手写出一些命令按照自己的意愿达到预期效果，是一件多么原始的能让程序员开心的事情。如果我想学习这些内容，现在的我有足够的可靠的信息源、足够有效和明确的方式方法，只要假以时日就可以足够全面系统地掌握这些内容以及其中的细节。

可是我总觉得，我应该关注更加「高级」、更加「上层」的东西，我甚至不知道这里的「高级」和「上层」是什么含义，但是我总觉得，我有更加重要的需要了解和掌握的内容，我不可以专门花费时间在这些东西上。

## 2022.02.16 更新

最近在工作上遇到一个可能搞不定的问题。大体是需要把 Golang 的智能合约用 Solidity 写一遍，在以太坊上运行。其中有一部分内容是基于 Bulletproofs（零知识证明的一种）来实现文件完整性的证明。且不说 Solidity 语言的表达能力好不好实现 Bulletproofs，零知识证明一直是我的黑洞，用什么语言都不会写，而且源文件中存在 magic number，有一个二进制文件和 seed 列表暂时不知道来源是什么。

当然工作问题还是交给工作来解决。我只是由此想到一些问题，感到困惑。例如，什么样的人可以写出 Bulletproofs 的实现？我为什么不可以？差距在哪儿，有多大？Bulletproofs 的学习成本有多大，门槛有多高？

于是几天前我在一个聊天室问了一下，这个是我的发言 (<https://t.me/c/1711254099/593>)：

大家好，想借聊天室这个机会，请教一下大家的看法。

最近在了解零知识证明的时候，发现原理比较难理解。想要用代码实现像 bulletproofs 这样的系统 (<https://cathieyun.medium.com/building-on-bulletproofs-2faa58af0ba8>)，仅仅了解编程语言、会编程是不够的，需要一些其他方面（加密学）的前提知识。

由此想到，程序员的工作，或者设计一种软件、协议，很重要的是把编程的技能应用到各种领域中，对编程语言的掌握可能是最基础的技能。

《解密计算机科学》的文章中，最后提到文章已经包含了计算的全部内容。bulletproofs 的本质也是计算，但是要复杂更多吧。

所以我不确定这种看法：计算机科学其实位于比较“基础”的一层，研究编程语言是专业性很强的事情，并不适用于大多数人？即使编程技能很好，也还是需要花费很多时间去了解其他领域的。

这是一个专门讨论编程语言的聊天室，所以发言中强调了编程语言的概念。紧接着有人回复道 (<https://t.me/c/1711254099/595>)：

我觉得不同的编程语言类似于钢铁、木材这种基础材料，我如果想做个凳子，简单了解下材质的用法特性就可以用木材制作一个，如果要做把刀，就需要用钢铁来进行制作，也很简单。如果我要做一个汽车底盘，就需要了解更多的钢铁材质特性和力学知识来进行制作。如果建造摩天大楼就需要学习建筑相关知识，再结合材质特性进行制作。计算机 + 音频知识 = 数字音乐、计算机 + 图形学 + 设计知识 = ps 等…

他形象地比喻了编程语言和领域知识的关系。几天之后，另外一个人同意了我的说法 (<https://t.me/c/1711254099/757>)：

计算的定义，我想引用 yin 的博客：计算就是“机械化的信息处理”

编程语言就是一种表达“计算”的工具/手段。用 sicp 的话来讲是“capture how to knowledge”。

所以会用编程语言就可以表达 domain knowledge 了。而研究编程语言就是为了创造更好的工具（让表达力更强）。

即使编程技能很好，也还是需要花费很多时间去了解其他领域的。

很赞同

总而言之就是，光会写代码是不够的！只了解编程语言是不够的！

## 2022.03.05 更新

编程语言应该学习到什么程度呢？我的观点是，学习到可用的程度。对于不同的人，“可用”有不一样的含义，需要自己掂量。

前几天看到一个关于 Personal development 的视频，YouTube 搜应该第一个就是，里面提到几句话，大概意思是，“当你寻找一份工作的时候，不要关心你会得到什么，而是关心你会成为什么（What you become）”。

如果你想成为编程语言专家，就竭尽全力研究编程语言。如果你想成为专精的资深开发者，就钻研在自己岗位的技术方向上。如果你没有目标，或者志在别处，就降低对编程技能的要求，“不学习编程语言”。

所以最重要的问题还是，你想成为什么样的人？

# 除夕

2022-02-01

堆在墙角的快递很久都没拆  
想换掉的被子到现在都没买

剧本杀的拼场没有往日红火  
工作日的玩家没有往日沉浸  
游戏像往常无数次一样进行  
新的故事新的剧本新的店家  
脑子里却只是想起过往场景

开源项目像往常一样没有动态  
没有什么事情非春节才可以做  
总是想不去虚度光阴浪费生命  
可是提升自我已经是日常生活

玩不感兴趣的去消磨时间  
但焦虑时间能否真的用来消遣  
可是不消耗时间又能做些什么

有的年纪轻轻就已生子成家  
有的年纪不小还是对影成双  
相同的年龄有人已经被迫成人  
相同的年龄有人不想承担责任

街上的车辆没有往日繁华  
除夕的当天格外不想出门  
想让自己一个人体会烦扰  
主要是出门也做不了什么

合租的邻居说记得关好门窗  
但家乡的善意让人猝不及防  
在这盛世的繁荣美景之下  
掩盖着多少挣扎的牛与马

窗外的风景看了很多遍  
风云变幻  
一成不变

也许我写东西水平有所降低  
最近确实很多事情没有头绪  
有太多的事情想不清楚结局  
不过至少我会一直都在这里

# This year

2021-12-31

Maybe I should write something today, but I don't want to write anything recently. I use a grammar tool named "Grammarly" to help me correct my grammar error this moment. As for why I started to write this in English because I found Simplified Chinese is limited for some reason. And I am difficult to write something about emotion directly. I am trying a new way, although I could just use simple sentences.

Changing the job is the most correct and biggest thing this year I did. No need to talk about it more.

I make a new friend this month when playing script-killing. She was once an overseas study in Korea. She will be my first friend if she is exactly to be my friend. I recalled It's full 3 years today from I come to Beijing. It's full 3 years I don't have any friends.

This is life, it keeps going on.

# 中式英语

2021-11-25

最近注意到几个典型的中式英语的词汇：

- 这本书里没有 magic，只是一些词法表达式的教程
- 这种做法太 low 了；那个方案太 low 了
- 我是不是 out 了

这些都是中文句子里夹杂英文单词的例子，类似的现象很多，包括很多（国内的）影视剧、综艺节目都频繁出现。

问题在于，这些英文单词，在这些句子里面，怎么翻译？直译肯定是不能表达全部含义的（为什么？）。有一种悲哀的可能性就是，这样说话的人，英文不好，中文也不好，无法把中文全部替换为英文，也无法把英文全部替换为中文。

如果把中式英语也理解为一种语言的话，就属于第三种混合的语言了，不会英文的人无法理解，不会中文的人也无法理解。看到一个视频说，也许是段子，一个英国人在中国生活，回到英国后他和家人聊天，说“这个太 low 了、那个太 low 了”。他的家人问他，“low”是什么意思？他说，“low 就是 low ……”

语言的含义也会受到意识形态的影响。在国内的一些平台，很多中文词汇都打不出来了，但是并不影响人们理解在谈论什么。

有的人宣称自己有“语言洁癖”，无法容忍中英混杂的句子，认为这是一种不干净的表达，应该用纯粹的中文或者英文。这样的人是有民族主义的嫌疑的。

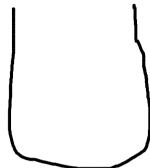
无论是象形文字还是拉丁字母，都是人类用来表达自己想法的工具，都是人类社会在思想上达成共识的媒介。90 后的火星文字、00 后的拼音简写，都是属于一代人的集体记忆，同样属于一种另类形式的语言。

言尽其意就好。

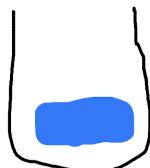
# 知识的诅咒

2021-11-24

一开始，每个人都是一个空的容器：



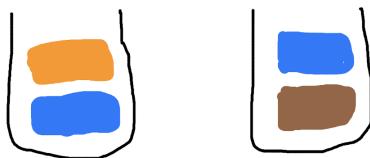
随着经历的增长，容器内增添新的内容：



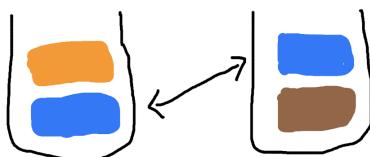
并且会不断增加新的内容：



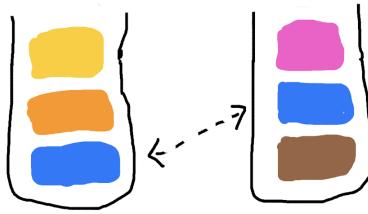
当两个容器相遇，容器内装着不同的内容：



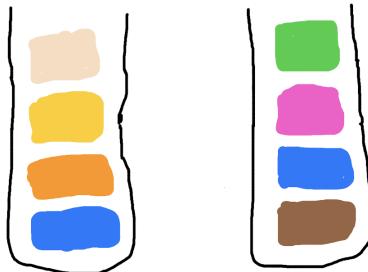
两个容器会尽可能拿出内容相同的一面，产生关联，在某些问题上达成一致，并且形成更密切的关系：



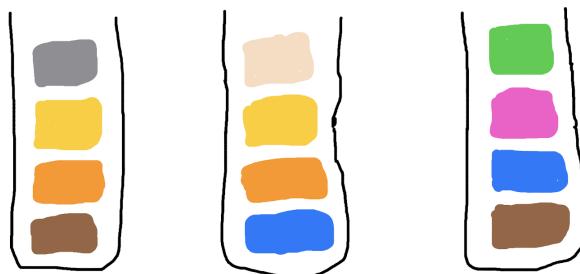
随着各自内容的继续增长，它们相同的一面占比由  $1/2$  变为  $1/3$ ，它们之间的联系变弱了：



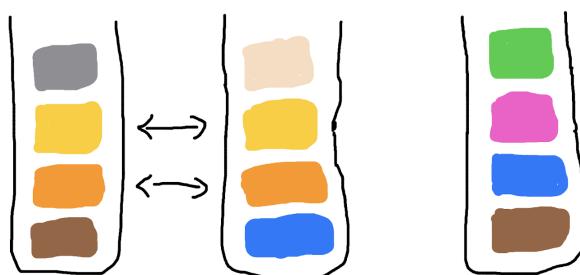
相同内容占比变为  $1/4$  的时候，它们不再继续有联系：



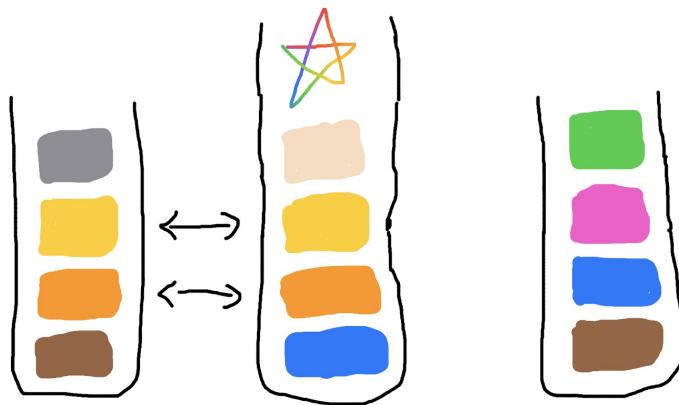
此时第三个容器出现了：



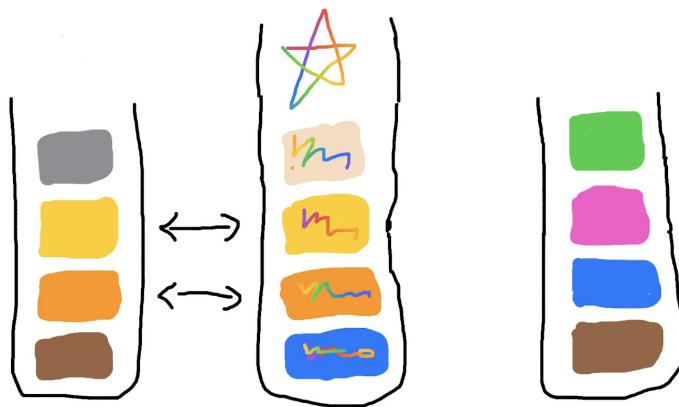
最左边和中间的容器，相同之处占比为  $2/4 = 1/2$ ，它们产生了联系：



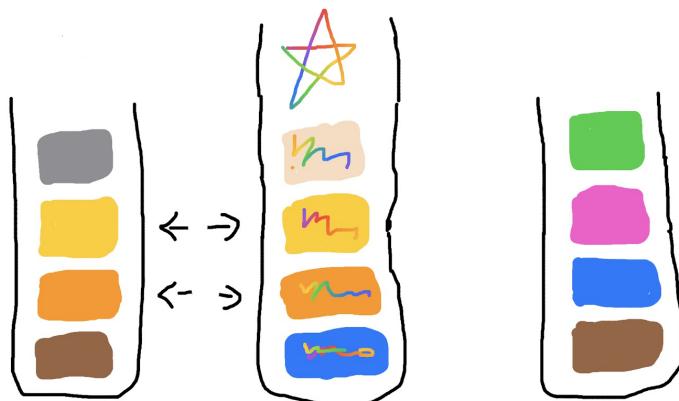
随着时间的继续增加和内容的继续增长，中间的节点获得了一些特殊的独一无二的内容，这个独特的内容是一个容器区别于其他容器的关键，俗称独立思考能力：



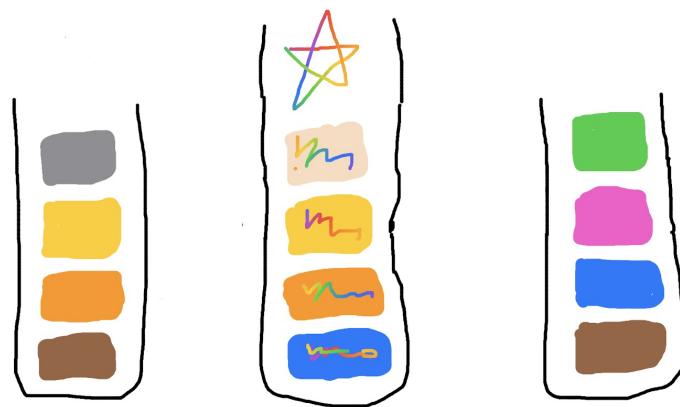
这一部分特殊的内容不仅自己特殊，还会影响之前存在的已有的内容，将之前的内容变为更加丰富的、带有独立思考的内容：



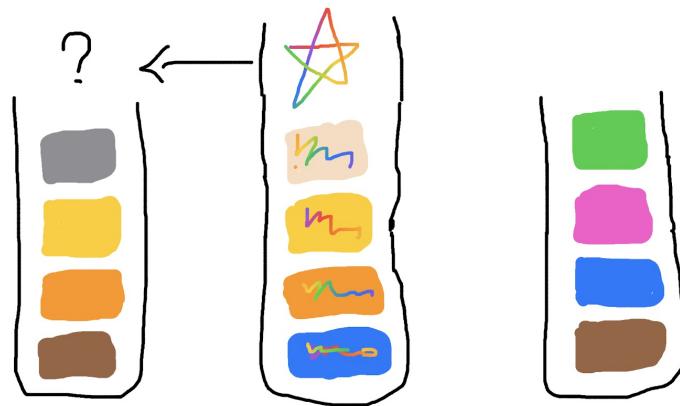
由于特殊内容对之前内容产生了干扰，混杂在之前的内容中，之前内容相同的占比已经降低，最左边容器和中间容器的关系变弱了：



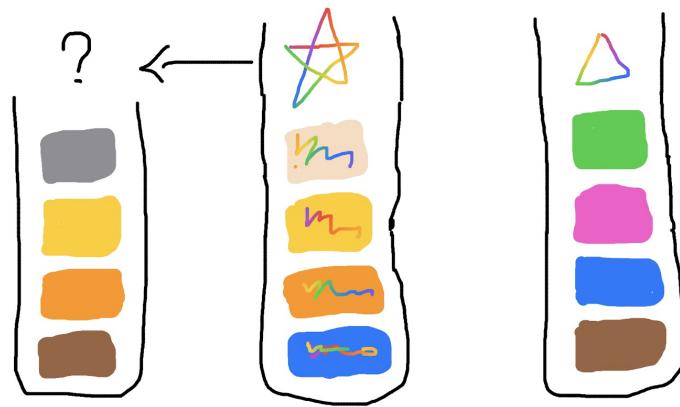
甚至再没有关联：



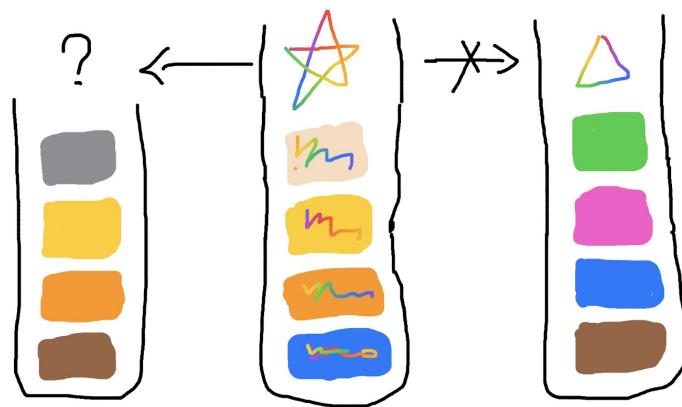
最左边的容器，始终没有获得那样特殊的内容：



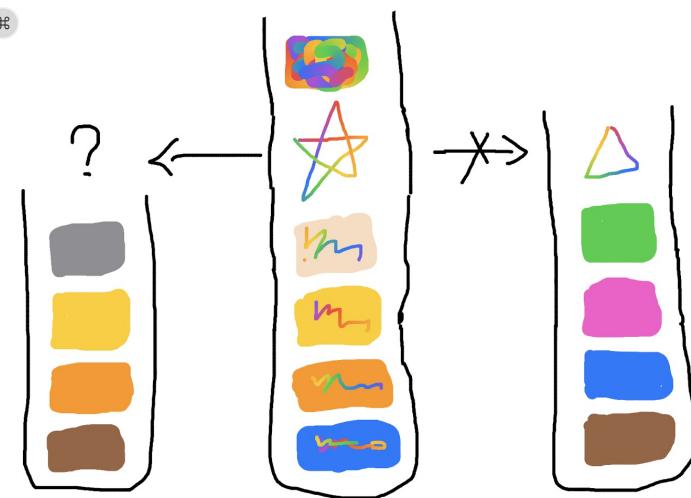
直到有一天，最右边的节点获得了一些特殊的内容，只不过是三角形的：



中间的容器和右边的容器无法达成一致：

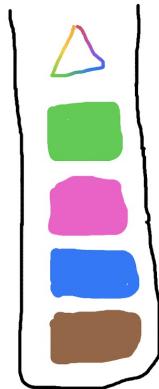
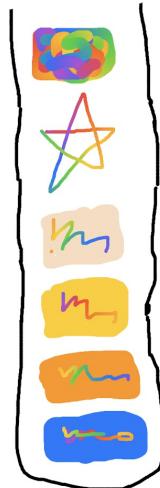


中间容器的内容没有停止增长，在有了特殊内容之后，再增添的内容也变成彩色了：



中间的容器和左右两边的节点，都没有联系了，也难以再产生联系：

⌘

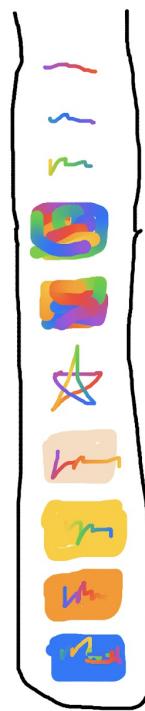


中间的容器，只剩自己了，和一开始一样：

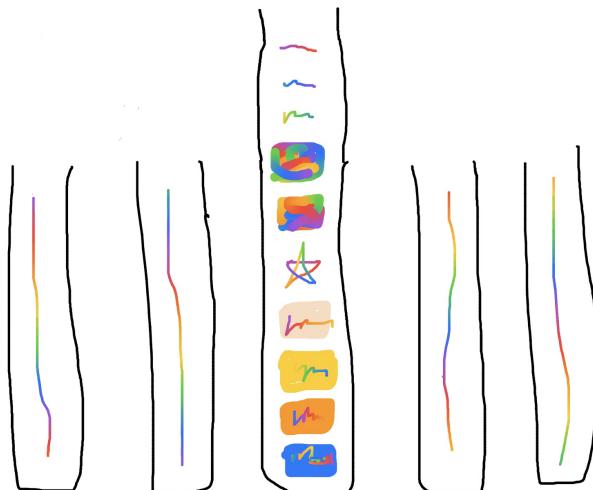
⌘



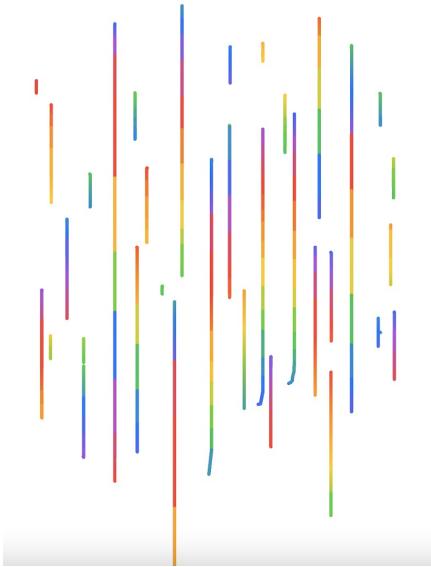
中间容器的内容继续增长着：



和其他容器一起，各自独立地继续累加内容：



林林总总，花花绿绿：



# 随想

2021-11-23

现在是凌晨快 1 点，睡不着，稍微写写。今天白天（昨天），测试提醒我，REST 接口的输入参数应该以大写字母开头。

我一开始还紧张了一下，我 out 了？我没有按照该有的规范开发？REST 接口的参数大写开头？  
(接口的 Content-Type 是 application/json)

然后就去查了一下 GitHub 的 API 文档，又看了一眼 DIDs (Decentralized Identifiers) 的标准文档，以及结合长时间接触使用 json 的经验，没有啊，json 的 key，哪有大写字母开头的习惯？

后来想到了一个吐血的原因，就问了一句，“参数首字母大写，是公司的规范吗？”测试回复的原话是：

我之前问过，就是你们开发自己都这么写，好像是，公有变量还是私有变量，就要首字母大写

果然是。（微笑）

## 权限控制

Java 里的权限修饰符是很基础的概念，用来控制类的访问权限，一共有 3 个关键词、4 种情况，比如 public 表示公开，private 表示私有：

```
public class aaa {  
    public int a;  
    private String b;  
}
```

具体内容可以参考 Java 的文档：[Controlling Access to Members of a Class](#)

像 Python 那样的脚本语言本身没有访问权限控制的概念，所以约定俗成地认为以 \_ 开头命名的变量是私有的，不过没有代码层面的保护，只是命名上的区别。

```
class aaa:  
    a = 1  
    _b = 2
```

Python 里 \_\_ 开头的变量名可以达到代码层面私有变量的效果，但把 \_\_ 用做私有变量并不完全是 Python 的本意。参考 Python 的文档：[9.6. Private Variables](#)

众所周知，Golang 是一种奇葩的语言，不但要求你大括号必须写在函数名同一行，还会要求你必须做一些什么事情，比如，大写字母开头的变量意味着公开，小写字母开头的变量意味着私有。

Golang 对变量大小写的区分，可不仅仅是命名上的区别，而是代码层面的控制。下面的代码会输出什么？

```
func main() {  
    a := struct {  
        i int  
    }{  
        i: 1,  
    }  
    s, _ := json.Marshal(a)  
    fmt.Println(string(s))  
}
```

当然是 {}，由于 i 是小写的，在 marshal 的时候自动忽略了。

这其实算是比较不正常的要求了。Golang 为了设计上的“简洁”，很大程度上减少关键字、关键字复用，但是又不得不提供某些能力，就需要一些奇怪的方式做变通。

我在上面的 Java 代码中，刻意用了小写的 aaa 做类名。Java 的类一般使用大写字母开头，但是不会在代码层面限制你把代码写成什么样子。

虽然我工作使用 Golang，但我不是它的教徒，当然，也不是其他语言的教徒。如果你仍然是某种语言的教徒，祝愿你可以尽早改变一下观念。

所以，由于 Golang 的一些“特性”，为了在序列化的时候方便一点，项目的 REST 接口就习惯于使用大写字母开头的变量名作为 key 了。

睡觉了，明天再写。

## 没有问题

json 的 key 大写或者小写开头，其实都没有问题。输出参数为了省去写 `json` 的麻烦，就默认大写开头，所以输入参数也大写开头了。这是一种工程上的对称，并没有什么问题。

想起之前有一次，我把 REST 接口设计为，输入参数用驼峰命名 myName，输出参数用下划线分割 my\_name。这样做的原因，是参数输入的时候，Java 的 Entity 能够把 POST Body 里驼峰命名的变量自动识别到对象上。返回参数时，程序直接把从数据库查出来的数据返回出去，是下划线形式的（不符合开发规范是真的）。这样做可以达到代码上的最简洁，但确实输入输出不对称不合适。

## 一件事

如果只是这么点事，就不值得写什么了。昨天测试说我什么地方该怎么样做的时候，我想到另一件事，在之前的公司里发生的事。

URL 的输入参数有两种形式，一种是 .com/?a=1&b=2，另一种是 .com/:a/:b。当时的项目里全是第二种形式，也就是路由形式的参数。

全部是第二种形式，有时候特别长，比如 .com/:a/:b/:c/:d/:e。参数很多时候是查询条件，如果中间的某个参数不需要作为条件，也就是不需要传参数，但是这种路由形式的参数又不能跳过中间的一段，就用了一种很奇葩的方式变通：all。用 all 来代替不需要传参的情况，.com/:a/:b/all/:d/:e。程序里面的判断也是各种离谱。

REST 接口本身是一种 Web 服务，但是如果缺少 Web 开发的基础，就容易做的有一些欠缺。

我当时怀疑开发那个项目的人技术水平有问题，也就是我当时名义上的小组长。好吧，也许是一段不怎么开心的往事。

顺便解释一下，我也不是有事没事就会对别人有意见。我当时所在的公司，有一种绩效的制度，每个季度开始前都需要写绩效计划，大概是接下来一段时间要做什么之类，一般是组长制定一个大方向上的目标，然后组员分配分别完成一些节点。我因为质疑他的水平，担心他制定的目标比较低，限制我的发挥，就借题发挥试探一下。对别人有意见对我自己没有任何好处，我也是出于一种恐惧，担心自己接下来会置于难以发挥的处境。

我当时并没有想离职，所以还是在极力想办法改善自己周围的环境和情况，给自己争取一个自己可以适应的状况。记得当时有其他同事离职的时候，（以前好像说过？好像说过，不记得了，再说一遍），我开玩笑说，工作中最开心的时候，可能就是提离职的时候和收到 offer 的时候了。同事说，最开心的时候是年底发奖金的时候……也是开玩笑。后来接着说，如果不是到没有办法的地步，不会有人愿意离职的。

我知道参数形式这种问题是小问题，但是我坚持认为，在当时的接口设计上，应该用参数形式的

参数而不是路由形式的参数。参数形式的参数明明更好用，为什么不用？说说你的理由，我已经拿出了话题，还把话头递给了你，你说一说啊，你的理由，除了“以前的人就是这么写的”之外，你有没有其他的在技术方面的思考？能不能够在技术方面碾压我，以小见大，体现一下自己的技术能力？

我的基本观点是，他缺少 Web 开发的经验，不是认为路由形式的参数好用，而是完全不知道有参数形式的参数存在。我分析有两种可能，一种是他自己有实力，但是不轻易和我说，另一种就是自己没有实力。所以啊，我从一个很小的问题，制造了话题，给他说话的机会，甚至之后用比较强硬的态度在 leader 面前，表达了我对他的能力的质疑。我心想，我都这么挑事了，要真有实力，总该表现出来了吧。

唉再之后就是一些收场环节了。

## 设身处地

我当时也想过，如果我是他，我会怎么办？

如果以后有小伙子那样对我，我能处理地比他更好吗？

如果有人质疑我的技术实力，或者揪着我的失误不放，借题发挥，我该怎么办？

我也知道，我好像破坏了一种和谐的宁静。

## 然后呢

现在的我，已经没有兴趣在类似那样的小问题上较真了，怎么样是对的，什么样是错的，有什么所谓呢。

不过过去的经历好像也是必要的。如果不是过去认真过，可能现在也还是会好奇：如果在一些问题上，我坚持自己的观点，或者非要争个对错，会是什么样的结果呢。

由于过去的经历，我已经知道是什么样的结果了。

其实结果不太好，也有点无聊。

我不太喜欢也并不期望是那样。

暂时没有然后了。

# GitBook 好用吗?

2021-11-21

10月15日，我写下一句话：

用 cloudflare cloud 的 DNS，把子域名 gub 从原来 CNAME 到 gitbook 改为指向到 github，到现在已经超过 72 小时，开启 DNS proxy 的情况下依然跳转到 gitbook，看样子是一个 302 forward。

不是 cacheing 的问题，已经很多次打开 dev mode 并且 purge 所有内容了。猜测 cloudflare cloud proxy 服务对于 forward 记录的更新非常慢，甚至有 bug。现在经过的时长一定超过 TTL 了。

一个月过去了，问题无意间得到了解决。想展开详细描述一下我遇到的问题。

## 背景

这里会涉及到两个概念，DNS（Domain Name System）和 CDN（Content Delivery Network）。如果你不太接触 Web service 领域，可以先了解一下它们的联系和区别。

## 问题

一开始的时候，我计划写一本开源书，选择 GitBook 作为写作平台。GitBook 名声在外，又有 GitbookIO/gitbook 那样广为人知的开源渲染工具，是开源书的不二选择。经过短暂的试用后，在平台的使用上没有感觉到异常。我创建了 Workspace，然后在 GitHub 上新建仓库，把仓库关联到 GitBook 上，一切都很顺利。我简单测试了一下 GitBook 和 GitHub 自动同步的能力，有可能会出现一点点冲突，但还是容易解决的。

我在 GitBook 上绑定了自定义的域名。smallyu.net 这个域名托管在 Cloudflare 上，子域名 gub.smallyu.net 也是在 Cloudflare 上设置 DNS 记录。全世界都知道，Cloudflare 会提供免费的 CDN 服务，只要在 DNS 记录上打开 Proxy 的橙色按钮开关就可以了：

CNAME	gub	smallyunet.github.io
Type	Name	Target
CNAME	gub	smallyunet.github.io
<button>Delete</button>		

当时在解析到 GitBook 的时候，开关是打开的。之后没几天，正好遇到了 GitBook 改版大升级，写作界面完全改变了。改版后一两天，我想要更新一些页面的内容，发现改版后的 GitBook 操作流程反直觉、bug 满天飞。每次修改都相当于 Git 的 Pull Request，而且每次点编辑按钮，都会新增一个 Pull Request 的条目。当同时存在多个 Pull Request 记录时，页面状态会完全不可控。这个是 1、那个是 2、另一个是 3，还能不能增量合并。因为你无法区分两个 PR 之间，一些内容是没修改过还是被删除了。重点在于，PR 没有删除选项，稍微有点强迫症都受不了。网速不好的时候，多刷新两下编辑页面，草稿箱就会多出好几个 PR 的条目，还不知道哪个是刚刚修改的。包括一些其他使用体验上的小问题，当时我还吐槽说：

现在 gitbook 的在线编辑难用过头了，不能删除 commit，不能新建文档，光标会自动跳转……他们是怎么对用户负责的。

后来决定放弃 GitBook，换成了 docsify，页面部署在 GitHub Pages 上，sub.smallyu.net 域名的解析也换到了 GitHub 上。更改 DNS 的解析记录后，发现解析没有生效（Cloudflare 上的 CDN Proxy 开着），访问 <https://gub.smallyu.net> 总是跳转到原来的 GitBook 页面上。

一开始怀疑是 DNS TTL 的问题，因为在 Proxied 的状态下，TTL 的值只能是 Auto。毕竟 Cloudflare 的 CDN 节点多，我的域名访问量又低，可能 DNS 记录更新比较慢。幸苦苦苦等了 3 天，这个时间足够长了，发现解析依然不生效，因为域名还是跳转到了旧的页面。

dig 域名的记录，是这样的结果：

```
gub.smallyu.net. 300 IN A 104.21.81.212
gub.smallyu.net. 300 IN A 172.67.146.253
```

此时域名是查不到 CNAME 记录的。对比之后，发现这就是 Cloudflare CDN 的 IP。域名已经解析到了 CDN 上，问题是 CDN 没有返回预期的新页面的内容。

然后偶然发现，把 CDN Proxy 关了，域名解析正常了，A 记录是 GitHub 的，CNAME 也是 GitHub 的，页面是新的。

是什么问题呢？Cloudflare 的 CDN 没有刷新内容。

Cloudflare 有一个 Caching 的配置，也提供了 Purge 的能力：

## Purge Cache

Clear cached files to force Cloudflare to fetch a fresh version of those files from your web server. You can purge files selectively or all at once.

**Note:** Purging the cache may temporarily degrade performance for your website and increase load on your origin.

---

在点过很多次 Purge Everything 的按钮后，CDN 内容仍然没有刷新，即使打开其他人都说有效的开发者模式，也是徒劳：

---

## Development Mode

Temporarily bypass our cache allowing you to see changes to your origin server in realtime.

**Note:** Enabling this feature can significantly increase origin server load. Development mode does not purge the cache so files will need to be purged after development mode expires.

This setting was last changed 13 days ago

---

---

包括自定义页面规则，不走任何缓存，也无济于事：

# Create a Page Rule for smallyu.net

**If the URL matches:** By using the asterisk (\*) character, you can create a page rule that matches many URLs, rather than just one. All URLs are case insensitive. [Learn more](#)

gub.smallyu.net

**Then the settings are:**

The screenshot shows a configuration interface for a page rule. On the left, there's a dropdown menu labeled "Cache Level" with a "▼" icon, currently set to "First". Below it is a button labeled "+ Add a Setting". At the bottom, there are three buttons: "Cancel", "Save as Draft", and a blue "Save and Deploy" button which is highlighted. To the right of the Cache Level dropdown, a vertical list of options is shown in a modal or dropdown menu:

- Select Cache Level
- Bypass
- No Query String
- Ignore Query String
- Standard
- Cache Everything

甚至为了让内容更新生效，我在 GitBook 上删除了原有的 Workspace，还注销了账号。仍然没有用。

之后就不不了了之了，只要不开 Proxy，域名解析就是能用的。不过，我当时认为可能是 CDN 的 bug，也许 GitBook 用了 302 forward 之类的记录，CDN 不能正确刷新这种类型的记录。

原因

最近，在访问 <https://gub.smallyu.net> 的时候页面稍微卡顿了一下，想起这个页面是没有走 CDN 的，想到了 Cloudflare 上的这条不正常的 DNS 记录。顺手 Google 了一下相关问题，没想到这次找到了有用的信息。之前遇到问题的时候也在网上搜过，搜出来的全是更新缓存之类，这次却找到了不一样的内容。

Cloudflare 有一些 partners，这些 partners 有着控制 Cloudflare DNS 的权力，Cloudflare 的域名在解析到 GitBook 上后，CDN 的 DNS 就受 GitBook 控制了，在 Cloudflare 上的配置优先级低于 GitBook 上的配置。

相关问题的链接：

- [DNS subdomain no longer works nor redirects to anything](#)
- [Subdomain CNAME does not update](#)

我发邮件给 GitBook Support：

S

smallyu

Noven

About domain CNAME problem from Cloudflare ask for help!

To: support@gitbook.com

Hello!

I CNAME a domain from Cloudflare to Gitbook, and I don't use it now. I can't domain to another record if I open Cloudflare's proxied, please help me do this from your service config.

My subdomain is "[gub.smallyu.net](https://gub.smallyu.net)".

Some related discussion this problem:

- <https://community.cloudflare.com/t/subdomain-cname-does-not-updated/240984/7>
- <https://community.cloudflare.com/t/dns-subdomain-no-longer-works/240984/7>

没想到 GitBook Support 一天之内就回复并解决了问题：



GS

GitBook Support

Re: About domain CNAME problem from Cloudflare ask for help!

To: Smallyu

I have deleted [gub.smallyu.net](https://gub.smallyu.net) from our Cloudflare configuration. Please let me know if you still have trouble.

--

Petros

[support@gitbook.com](mailto:support@gitbook.com)

[See More from Smallyu](#)

经过测试，现在一切正常，确实是那样的原因。

教训

谁能想到，Cloudflare 如此广泛使用的服务提供商，会把域名在 CDN 上的解析权限交给 partners。

谁能想到，GitBook 的产品即使用户删除了 Workspace 注销了账户，在系统内的域名解析记录都不会被删除。

这件事情可以带来的启发是，我们应该从普通的用户思维转变为开发者思维。也许在某种观念的影响下，因为所谓“官网”、“权威”的概念，当使用一些平台的时候，我们习惯于首先质疑自己的使用方法和操作错误，却很少质疑平台的问题。即使明确是平台的问题，也不会优先试图联系平台方解决问题。从这个点其实可以发散出很多内容，以后有机会再展开。

## 工具

最后现在用 Rust 团队开发的 mdbook 了。GitBook 稍微有点过时、处于不怎么维护的状态。docsify 也有问题，docsify 更适合项目文档，除了样式不那么凸显文字外，页面和页面之前是没有关联的，没有上一页下一页的跳转链接，不太像是一本书。虽然 mdbook 的样式没有很时尚，但是功能齐全完整、编译速度能感受到的快，好用就行。

# 我不太喜欢的一个人

2021-11-12

其实我不太了解那个人，只有非常少数的一点印象。

我实习的时候，正赶上政策压制，公司裁员，同一批的实习生走了一半，包括和我同部门另一个组的小伙伴。应该是当时的部门 leader 把简历给了其他部门，我就转到后来做区块链的部门了。

我到部门第一个见到的就是那个人，也是 leader。从对话内容看应该是看过我留在简历上的博客地址。之后我的工作就全部由小组长分配了，一开始做做网页，到后来做一些其他事情。

我毕业转正式工作后不久，那个 leader 就离职了，去了一家初创公司当合伙人，那家公司目前在做开源的联盟链。也因此我不太看好那条开源联盟链的技术，甚至一开始感觉能在那个项目上看到一点点，我上家公司区块链产品的影子。也可能是联盟链都是那种样子。

在他离职的欢送仪式上，我记得他说过一些话，我不太喜欢那些话。

“有些人下班到点就走，不加班，怎么提高自己？”

好像还说他虽然走得早，但是回家也仍然在工作什么的。时间太久，记不清具体内容了，大概 是那样的意思。当然不是冲我说，是对所有人说的，不是针对谁。

虽然我当时还年轻，加得动班，但是我完全不认同那样的混蛋逻辑。后来公司有段时间还要求 996，我完全能做到去得最早走得最晚，就差公司没床了。况且我上班的时候也挺认真的，主要是简单，做网页什么的老本行了。而且，你怎么知道下班时间只有你在做有用的事情？

“如果我创业不行了，希望大家能给我留个位置。”

这话我是理解不了，走都走了，还想着后路。不过不幸的事实，“大家”倒的比他的创业事业还快，不知道是不是因为他走了。

“你们都不行！”

饭局上，有人说，“你创业，把我们也带过去吧？”然后他就说了这样的话。终于在离开的时候把心里话说出来了，哈哈。

后来就没什么了，总之感觉自己看到了一个经典的油腻大叔的形象。记得我喝了两杯白酒就开始晕了。

再后来见面是一次部门团建，把他一起叫上了。他和部门 leader 比较熟。一起团建的还有另一个部门的 leader。

同样是在吃饭的时候，另一个部门的 leader 提到了落俗老套的问题，没什么人搭茬。然后话题就到了他创业的公司那边有没有女的，有女的给我们介绍一下之类。

“有女的。（停顿一下）女博士！”

这是和我价值观几乎相悖的一种言论。言外之意就是，博士嘛！你们呢？

在一个匿名的社区上，可以查到有人对他们公司的评价是，既有国企的腐朽文化，又有互联网公司加班、on-call 的优良传统。

有一次他们的产品发大版本直播，简直像是在谈笑风生，气氛松松垮垮，着装和环境都很 low。

在他离职创业的同年，他们公司开始做现在的项目，也可能稍后几个月的时间。反正到现在没多少时候，他们也没多少人，版本倒是发的老勤快了。当时还看到他发朋友圈说要打造国内最牛的区块链平台。

看看他们在其他直播会议上的演讲 PPT，最亮眼的就是列了一大堆指导单位和成员单位的一页。

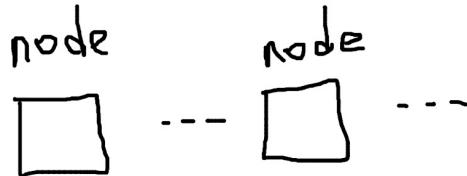
之前偶尔有猎头联系我，说那个公司在招人。

由于这样一些特殊的原因，我可能永远也不会去或者说去不了人家那种高大上的有女博士的公司了。

# 一种区块链节点存储扩容的方式

2021-11-10

区块链是天然支持水平扩容的系统，节点扩展能力首屈一指。

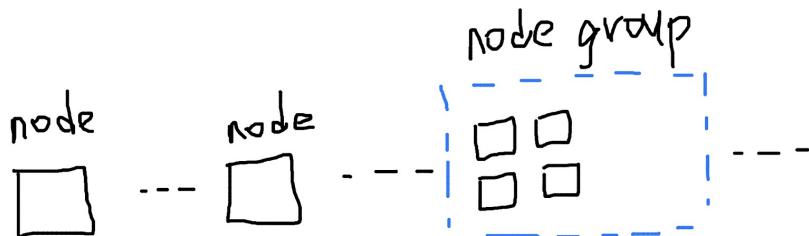


但区块链的垂直扩展能力还是一个经常被讨论的课题。单个节点的硬盘容量总是有限，如果节点拥有全部的数据，对单机性能要求会比较高；如果节点没有全量数据，就不能认为是 P2P 网络的节点之一。

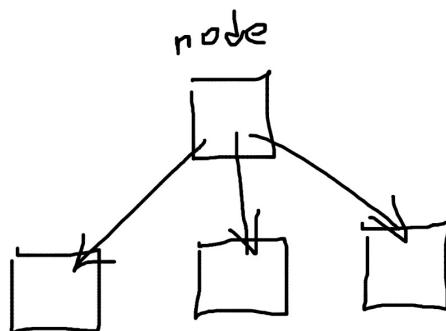
最直接的办法就是用分布式数据库，数据库本身就支持扩容，区块链节点的存储模块就也算是支持扩容了。（如果区块链在立场上和数据库没有冲突的话。）

这里描述一种简单的实现思路的设想。

节点的垂直扩展，是想用多个节点合力代替原有的一个节点的位置，整体形式上一个集群提供了和单个节点一样的输入输出。



节点完全可以将块数据分散储存在不同的子节点上，比如按照数据库分库分表的经典思路，对块号取模，或者随机分发也行。

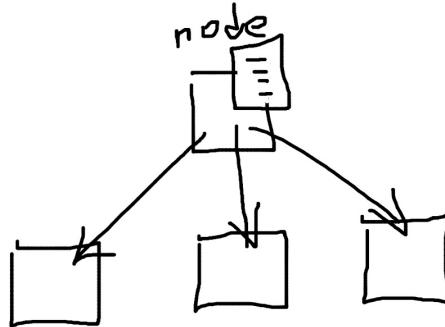


节点可以区分为索引节点和存储节点，索引节点只记录块号和子节点的对应关系，子节点集群就作为索引节点的储存模块。索引节点同时负责发送和接收块等操作。除了网络延迟带来的存取速

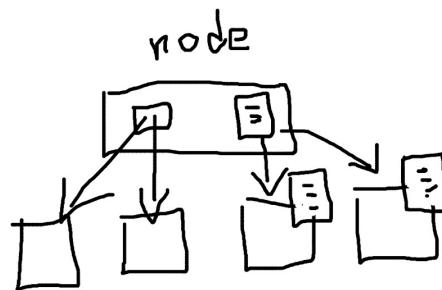
度的降低，似乎没有大碍。

对于节点类型的问题，是必须要有不只一种类型的节点吗？有没有办法实现只要一份源代码、只有一种二进制程序、只用一种类型的节点，就能实现所有的功能？当然，不是说把三种类型的节点打包到一起就行了。由于功能侧重点的不同，尤其是节点“身份”的不同，可能节点不得不区分类型。一个节点对外提供能力和一个集群合作对外提供能力，集群内的节点和单个节点应该是同样的地位吗？

将块数据分散开后，对于“世界状态”一类的数据，可以全部储存在索引节点上。



如果状态数据也想扩容，同样可以只在索引节点上保留索引数据，然后将状态数据也分散到储存节点上。



这样的方案也许过于简单了，万一行之有效呢。

# 一种基于“自我中心主义”的共识机制

2021-10-29

## 简介

“自我中心主义”的含义是，对于每个人来说，世界的大小取决于他能够接触到多大的世界，世界很大，和我无关，世界很小，全和我有关。一个人认识的人、了解的事、接收到的信息，无论是否命中注定，一定是有限的，你不可能认识世界上所有的人，知道世界上所有的事。才学渊博、见多识广，又怎样呢。

在区块链中，共识机制是用来保证数据一致性的关键手段，也给区块链带来了最核心的去中心化的特点。共识机制是强一致性的，或者是在拥有一定容错能力的情况下，达到大多数一致的效果。有没有一种可能，存在一种共识机制，不以数据一致为首要目标呢？

世界本就是复杂的，试图将所有节点同步到仅仅一种数据状态，其实是违反直觉的。而且，无论是不需要授权的大范围共识，还是基于身份授权的小范围共识，最终实现数据一致的方法，都是“多点变单点”，也就是同一时间只有一个节点在处理数据，其他节点可能是在共识后接受满足条件的数据，也可能在确认数据前投票是否同意对数据的操作，总之都需要有一个“英雄”一样的节点，在关键的变更数据的时刻，做一些事情。

有的英雄实力强大，先斩后奏，改过数据后过来跟你说，“我改了数据”，你一开始不满意，但是接触后发现英雄确实厉害，能做出你没有解决的难题，于是你就认可了英雄的行动。

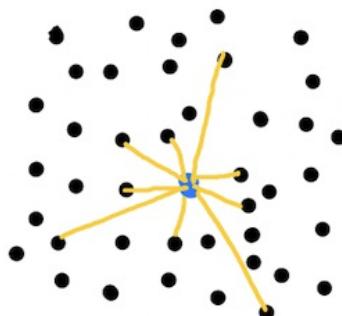
还有的英雄被公众授予权力，行动之前作为代表被选举出来，行动的时候会万分小心，挨个问民众，“改动这里的数据，你同意吗？”如果大多数人同意，英雄就会行动。

当然，每个人都有平等地享有做英雄的机会，虽然有的人天生神力，有的人八面玲珑，但机会总还是有的，题放在那里，你做不出来，怪谁？每个人就可以被选举，别人不选你，怪谁？

所以，为什么我们不能做自己的英雄？为什么我们要屈就于别人的光环之下？每个人都是自己的英雄，在我们的世界里，在大小受限于个人接触范围的世界里。一种共识机制，节点的边界受限于其触及的网络规模。

## 网络概况

在非结构化的点对点网络中，路由表是必不可少的组成部分，节点能够接触到的网络大小，就取决于路由表中存着多少“联系方式”。共识对数据的处理，就以路由表中的节点为依据，路由表中有 10 个节点，就争取和这 10 个节点达成一致，路由表中有 10,000 个节点，就和 10,000 个节点达成一致。也没有必要使用分布式路由表，就普通的数组就可以。在这种情况下，网络中的节点会是这种样子：

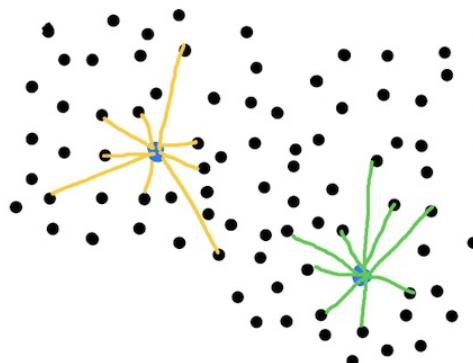


以当前节点为中心，连接到的节点数量可多可少，有的很远，有的很近。弱水三千，只取一瓢。在路由发现的问题上，节点也是需要种子地址的，比如节点启动的时候先解析种子地址的记录：

```
lookup("seek.domain")
-> 127.0.0.1
-> 127.0.0.2
```

然后依次请求解析出来的节点地址，去得到他们路由表中的内容，将其添加到自己的路由表。这其实是常规做法，不过这样有可能引起的后果是，节点会瞬间获取到整个网络的路由信息。这并非不好，只是感觉有点快了，我们认识一个人是需要时间的，和人交谈也是需要时间的，你无法同时和三个人交谈，或者无法同时听三个人说话。即使拿到了很多人的联系方式，也没办法“多线程”联系每个人。我们处理信息的“带宽”有限，节点也一样。我们甚至可以对网络发现的速度稍微做一点限制，比如串行处理路由表新增记录的动作，先与节点建立连接然后再添加信息。

让路由发现慢一点，似乎听起来不太正常，难道是想让网络处于不同步的状态吗？很多共识的瓶颈就在网络带宽上，就在协议交互的复杂上。如果降低节点间的交流成本，共识的容错能力也会随之降低。网络可能会被划分为不同的区域，可能形成大大小小的圈子。

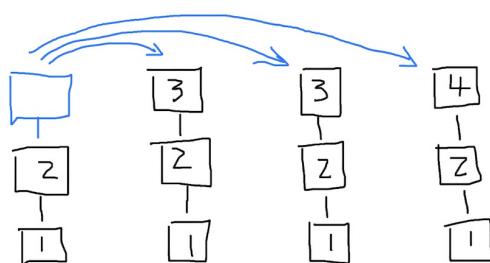


对于共识算法来说，脑裂是要尽可能避免的问题，但其实网络分割是再正常不过的事情，是自然存在的情况。我们人类的思想是分裂的，有可能是对立的，但是经过一些事件后又可能达成一致。所以在一个网络中出现割裂是完全允许的情形，形成的小规模网络可能是互联互通开放的，也可能是保守封闭与外界隔离的。重点是要有一种机制能够“修正”这种分裂，也就是在某种条件下，割裂的两个网络可以相互合并。

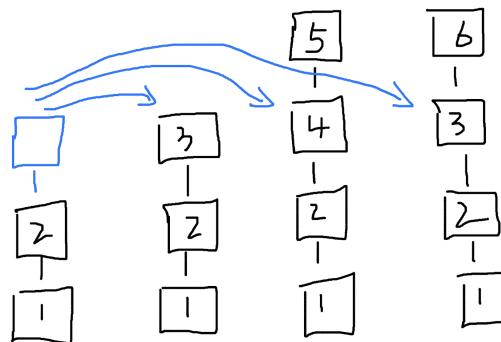
## 同步数据

### 主动

这里保留区块链创世块的概念，所有节点的第一个块内容是相同的。新加入网络的节点，会从创世块开始启动，此时其他节点的块高度已经有很多了。比如当前节点的块高度是 2，想要从网络中同步第 3 个块高度的内容，节点的路由表中有其他节点的地址，其他节点块高度均等于 3。当前节点会发起一个对块高度 3 的请求，依次到其他节点。

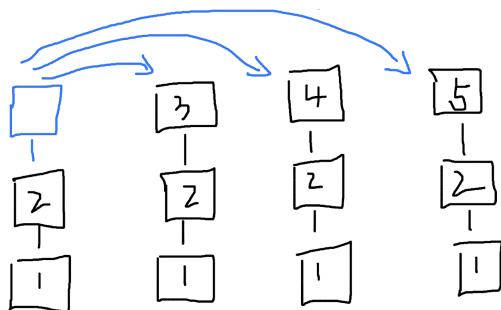


请求过后，发现有 2 个节点块高度为 3 的块一样，块的内容一样、块哈希一样、前块哈希也一样，那当前节点就把这个多数节点都存在的块作为第 3 个块。



如果请求的时候发现有的节点的块高度已经大于 3 了，那么是不是应该块高度最高的优先呢？如果考虑时间尺度的话，会觉得事物发展是需要遵次序的，你不能跳过 3 岁直接过起 4 岁的人生，区块链的数据也应该有先后次序。况且，当前节点需要的块高度是 3，请求的块高度是 3，管你有没有其他高度的块？你的块高度再高，我就要 3 的，你说你多高有什么用？

那么如果请求过所有节点，发现每个节点的块都不一样呢？该信谁？

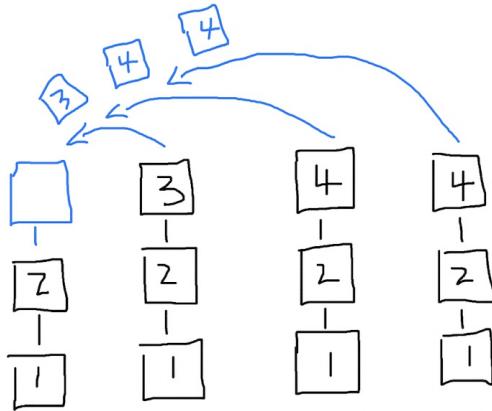


总得挑一个吧。如果不能确定哪个节点或者哪个内容可信的话，就随机选吧。最好是选择最后一个请求的节点，因为错过的节点就已经错过了，此时最后一个节点是距离你最近的节点，并且在此之前你并不能判断，是否存在块内容相同的节点。所以在放弃之前的节点后，最后一个节点就是你不可以放手的选择。

对于主动请求块数据的情况，很关键的地方是，请求一定是按照路由表顺序依次进行的，在得到第 1 个节点的响应之前，绝不向第 2 个节点发起请求，做人不能太三心二意了。如果有节点就刻意加速、同时请求多组数据呢？其实也无伤大雅，毕竟只是同步数据，有的人喜欢快点，有的人喜欢慢点，有的人喜欢快生活，有的人享受慢生活。

## 被动

除了主动请求某一高度的块数据，节点也会收到其他节点的广播消息，比如当前节点的块高度是 2，收到了来自其他节点的内容分别是 3、4、4 的块。



按大多数一致的原则，是不是应该选择内容是 4 的块放在自己的第 3 个块高度上？但是这样存在一个问题，你无法预测自己会收到多少个块，没办法计算块内容的总量和占比。所以对于被动接收的块，可以以第一个收到的块内容为准。

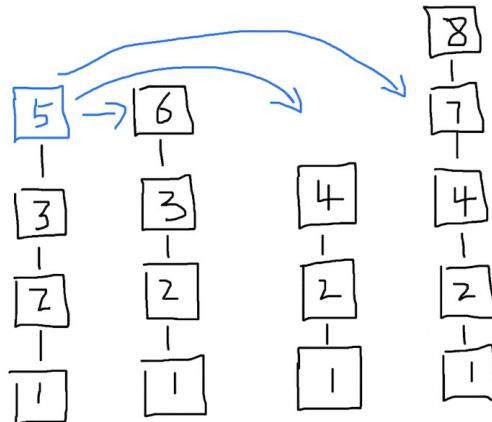
人生的出场顺序很重要，如果正好需要的块高度是 3，接收到广播的块高度也是 3，那就它吧，遇到哪个算哪个。实在不行后面遇到更合适的再换。如果有节点很激进，为了自己的块能够被大范围接受，把同一个块标记为从 1 到很大块高度广播出去，就为了碰运气，让正好缺块的节点接收，那也就随他吧。因为节点在主动同步块的时候，是按照高度获取内容的，这种激进一点的做法并不能带来很好的收益。

## 新增数据

网络中的数据由谁产生？为了解决这个问题，可以先定义为，每个节点都可以产生数据。一种极端的情况是，每个人都只相信自己的数据，各玩各的，整个网络就变成单机版了。所以节点也需要将自己产生的数据散播出去，发送给其他节点。对于其他节点来说，就是“被动同步数据”的情况了。

### 有节点需要

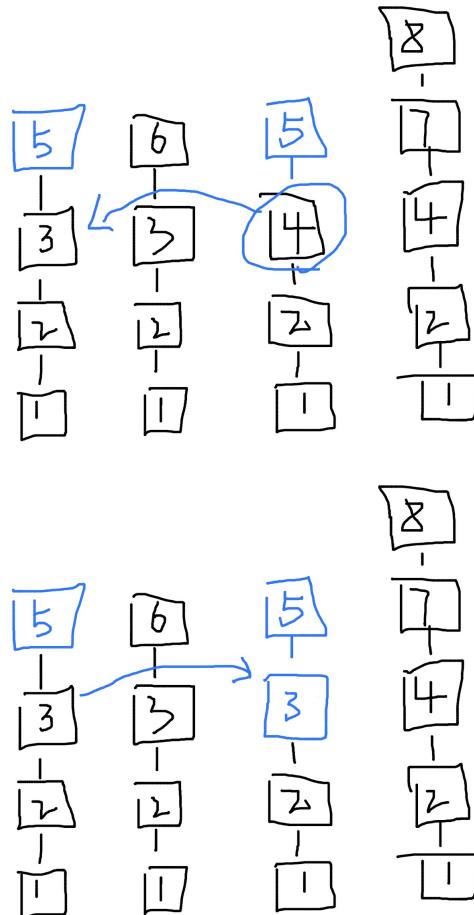
主动广播块数据分两种情况，一种是有节点正好需要块，你正好发送给他了。



当前节点接收到内容的请求，新增了块高度为 4 的块，这时会直接把块持久化到主链上。接着开始对块高度为 4 的块进行广播，广播按照路由表依次进行，在广播结束之前，当前节点不会打包下一个块。广播开始后，有节点块高度为 3，说明你是第一个发送给他块高度 4 的节点，它一定

会接收你的块，同时给你一个响应消息。在收到响应后你就可以知道，当前网络至少有一个节点接收了你的块，你可以继续处理下一个块了。当然，在路由表遍历结束之前，节点即使收到响应消息也不会停止这一轮广播，这是理所当然的，希望有更多节点可以接收块内容。

如果对方节点在收到块后，发现块内容是 5，前块哈希是 3，并不对应它自己的前块哈希 4，对方依然会接收这个块，并且依次替换自己之前的块，直到哈希一致。

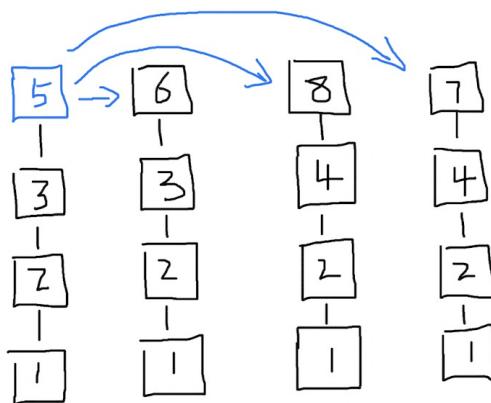


这种机制有可能带来的风险是，接收到一个块，然后把整条链都替换掉了，这是非常严重的不能接受的开销。但确实存在这样的可能，你遇到了一个坏人，这个坏人乘虚而入，他的思想颠覆了你的人生观，让你误入歧途，六亲不认。倒是你需要反思一下，你的路由表里为什么会有这样的坏人。而且这样的坏人多吗？如果一个恶意节点用一个块替换了你的整条链，但是接下来会有很大概率有好人来把你的整条链置换到大多数一致的情况。

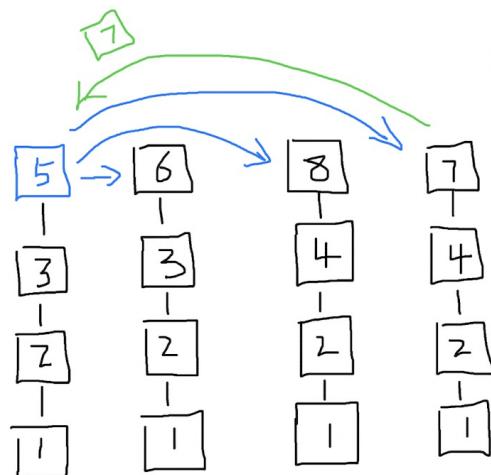
这里暴露出了一点问题，接收到第一个块就认可，是不是太草率了？如果是坏人怎么办？为了增加节点作恶的难度，在接收到块内容为 5 发现前块哈希对不上时候，应该不止向第一条链请求块内容，而是走完整的“主动同步数据”的逻辑，根据块高度把路由表里的所有节点都请求一遍。如果块内容为 5 的块，前块哈希和大多数节点不一致，就直接把 5 抛弃掉。如果一致，就说明它不是有害内容。

没有节点需要

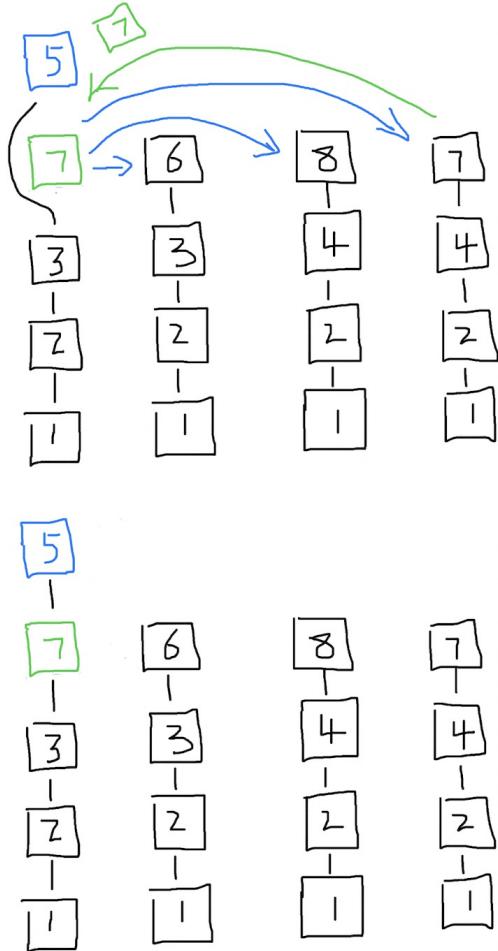
节点新增块后，可能遇到没有节点需要当前块高度的情况，



其他节点的块高度都大于等于广播出去的块。这种时候，当前节点就有必要做一点点妥协，为了让别人接收自己，为了让其他节点接收自己的块数据，只好先从其他节点同步数据，和其他节点保持一致。

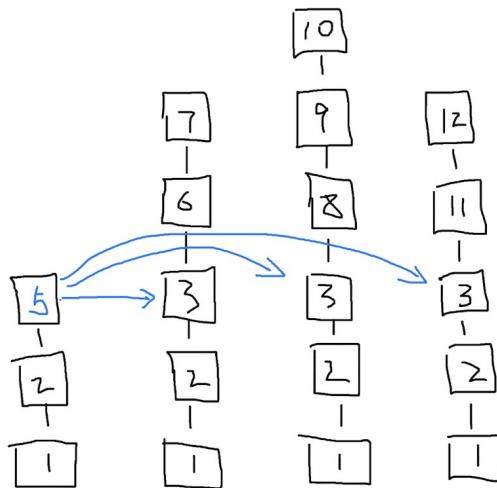


在块高度为 4 的块上，当前节点广播了一圈发现没有节点愿意接收这个块，那当前节点就把最后一个访问的节点的，当前块高度的块，请求过来。为什么是最后一个请求的节点？这里也可以走一遍完整的“主动同步数据”的流程，但为了提高效率，减少网络交互，可以先随意接收一个块内容，再做后续的判断。选择最后一个节点，是因为离得近。在错过了万丈红尘纷纷扰扰之后，恍然回首，发现最后一个节点是你此时最亲近的伙伴。



收到最后一个节点的块内容是 7 的块后，当前节点继续广播块内容为 5 的块。

如果不小心遇到了其他节点的块高度都远高于自己的情况，那说明自己确实落后了，先把其他节点的内容都同步过来再说。想要创新，想要新增内容，至少要先到达某一种顶端，

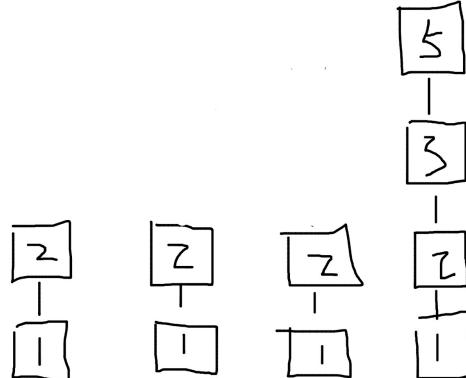


不一定是整个网络的顶端，至少是某种圈子的顶端。

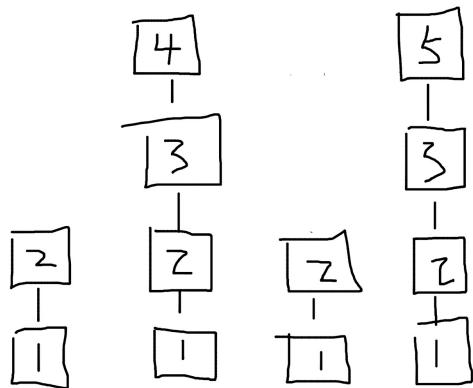
## 交换数据

### 类型

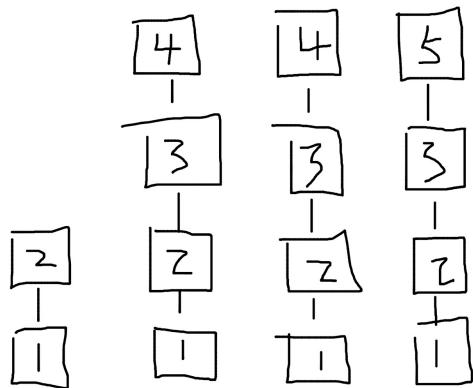
在目前的机制下，网络可能是混乱、不同步的。虽然对数据同步的速度预设是慢的，但如果有的节点就喜欢快呢，用快的计算、大的网络带宽，就是要达到整个网络的最前沿。



也就是大多数节点慢，少数节点快的情况。每个节点都是按照块高度平行更新内容的，也都是按照块高度广播内容的，在一定程度上会缓解这种问题。你想快就快，和我们没有关系，我们慢的自成一派，我们遵循大多数一致的原则，不是谁块高度高就听谁的。你想内卷就尽力去卷，我们不跟你玩。



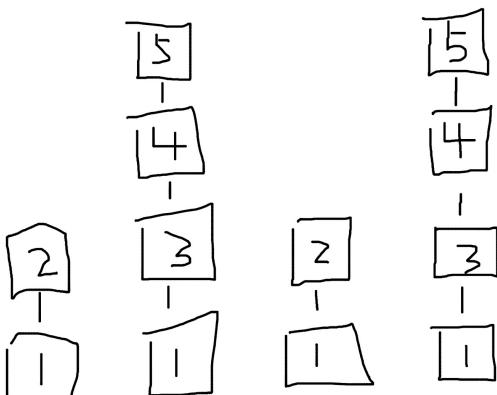
另一种是一半节点慢，一半节点快的情况，也没有什么好担心的，最坏就是形成两个网络，无关痛痒。

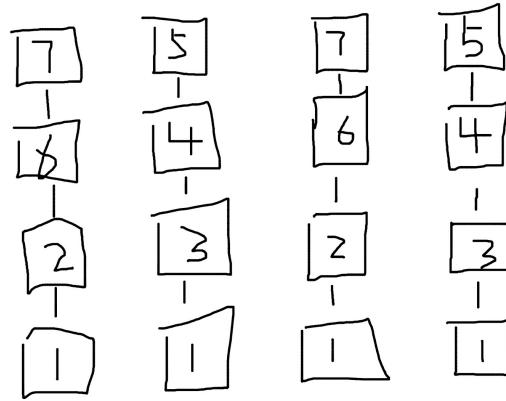


至于少数节点慢，多数节点快，属于最正常的情况了。

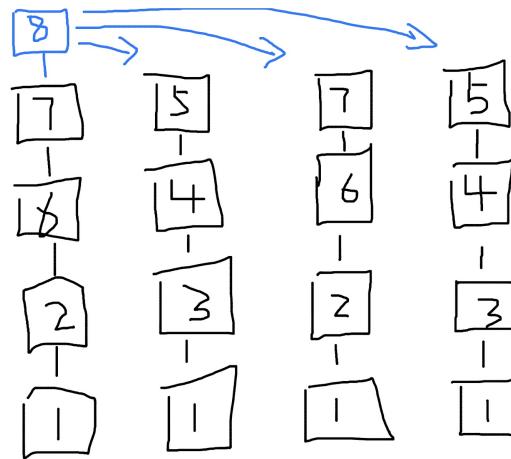
## 融合

在一半节点慢，一个节点快的情况下，很容易造成这样数据对立的情况，即使块高度一致，也是两种数据。

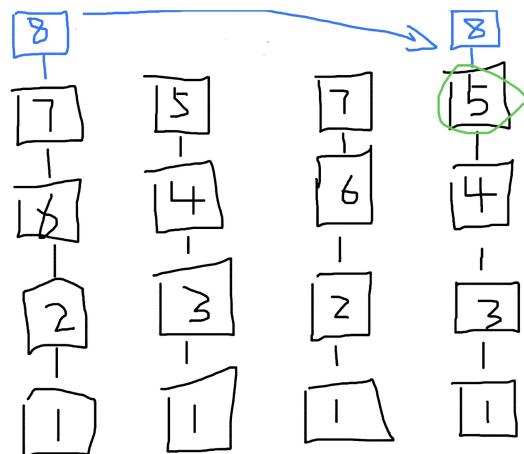




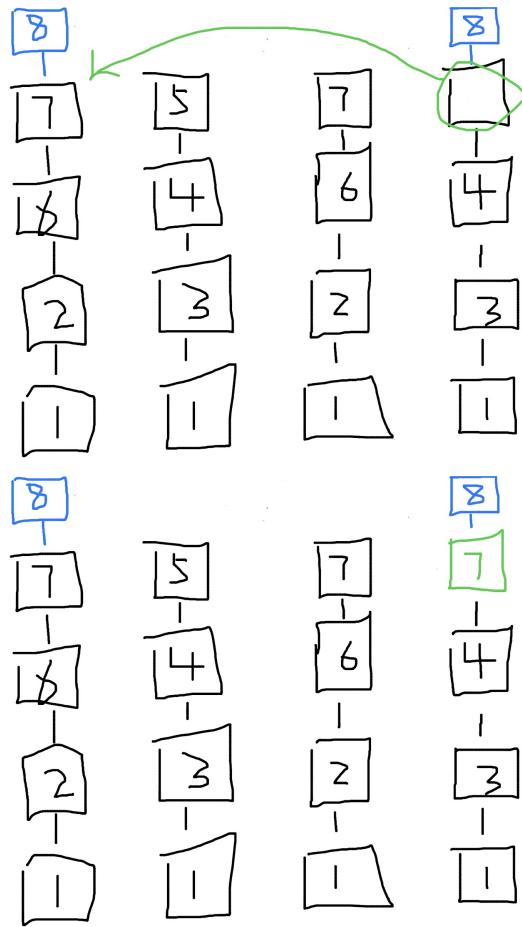
这个时候就不得不有一方妥协了。如果两个网络想要融合，就必须有一方做出一些牺牲。在块高度一致的情况下，假设路由表互通，产生新的块数据后，其实就是“主动广播数据”的过程，当前节点先产生一个块：



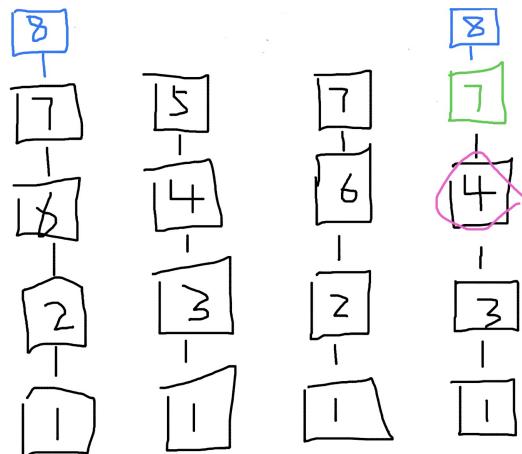
例如最后一个节点在收到块后，发现前块哈希和自己的对不上：



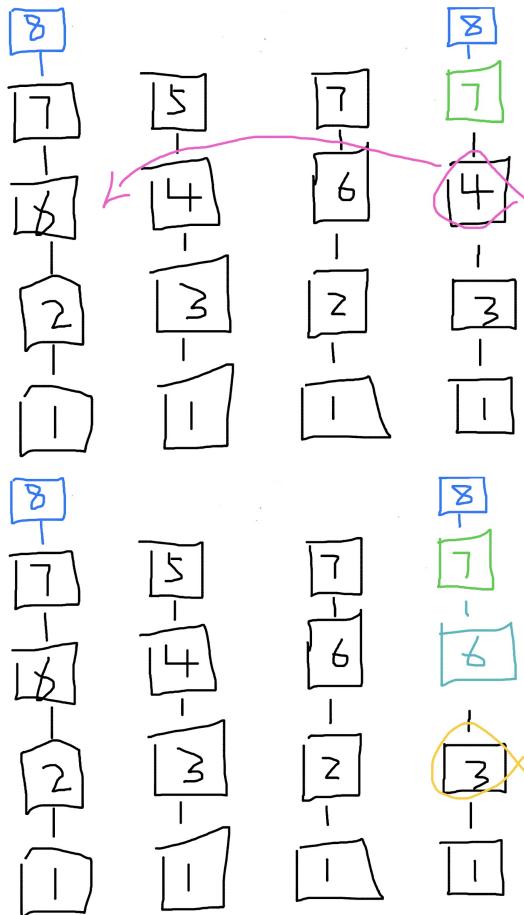
就去其他节点请求上一个块高度的内容：



发现上上个块的哈希对不上:



继续请求其他节点对应块高度的内容;



依次类推，直到整条链完全相同。

被替换掉的块内容可以放到一个缓存队列，作为新块的内容，继续向外广播，减少节点内容的丢失。

## 总结

这是一种没有经过实践考验的、也无法简单用公式来建模的共识机制的设想，共识以自身立场为出发点，关心自己如何应对网络中其他节点的不同行为，而不是从整个网络的角度“上帝式”地设计交互协议。这样的机制会给网络带来不确定性，但也会带来很多可能性。我们只能考虑节点基本的行为规则，就像我们学习生活规则一样，我们很难预测整个网络的走向，就像我们无法预测世界会向什么趋势发展。这种共识机制并不是而且也许不能解决特定的问题，比如建立电子现金系统或者提供图灵完备的运行平台，它关注在更基础一点的层面，提供一种实现数据一致性的方法和思路。

计算机通过网络组成的虚拟世界，一定也很精彩。

# 人生终极目标

2021-10-20

衣食无忧、冬暖夏凉、结婚生子、安享晚年。

这个目标高吗？

# 魔法师

2021-10-09

勇者受伤了。

他来到山脚，发现有怪物挡住了上山的路。

怪物的名字，叫做权威。

他和怪物完成交易，放弃此前所有的勋章和荣誉。

他扔掉了剑，丢掉了盔甲，杀掉了马。

他可以上山了，他没有上山。

他需要新的宝剑。

他去了附近的魔法镇，去了远处的藏宝山。

去了最好的兵器店，去了神秘的海盗湾。

秃鹰啄伤了他的肩膀，刺猬挡住了他的去路。

门卫刁难他衣着朴素，法官质疑他没有天赋。

他累了。

他回到寄居的酒馆，想起了从前。

他需要回家一趟。

他想休息。

甚至想放弃。

他回到新手村，见到了朋友。

他是凯旋的冒险者，他是无畏的勇士。

他有亮丽的盔甲，他有傲人的白马。

他不再是新手，也再没有朋友。

他不可以倒下，也不可以害怕。

虽然他累了。

一间房子，一年住 351 天，另一间房子，一年住 14 天。

哪个是家，哪个是旅店。

一个地方，一年度过 351 天，另一个地方，一年度过 14 天。

哪里是生活，哪里是旅途。

他回到酒馆，一切从新，一切照旧。

他终究没有找到宝剑。

如果你看到他，

他可能在迷茫，

也可能在游荡。

请你告诉他，

他可以继续前进，

也可以稍作停留，

也可以上退几步，

他可以写一本魔法书，

可能掌握高明的法术，

他会成为优秀的魔法师，

而不仅仅是挥剑的战士。

# 你是.....

2021-10-02

你是内战时期的军阀，伪装成屠杀了全村并抢劫了公主的强盗，押着公主到交流情报的酒馆，寻找生活在黑暗里的情报头子摩尔斯帮忙，想要找到军火的藏匿地图。你遇到了幼时有救命之恩的记者，遇到了追求公主至今的特务。他们笑里藏刀，想要抢走你的地位和朋友。最终你输了，死在血泊里，值得欣慰的是，她过的幸福。

你是年轻有钱的渣男，女友背叛你和兄弟约会，你背叛女友和同学约会，同学喜欢你的钱，女友也喜欢你的钱。最后你需要选择，新欢还是旧爱。无论哪种选择，都是凄惨的结局。

你是来自美国的富商，受邀参加一场神秘的宴会。宴会主人曾在十几年前犯下杀人罪行，宴会当天遭到复仇惨死密室。你作为警官调查这起案件，发现凶手精通易容术，来自美国的富商不是富商本人，发现外表纯洁的美女主播暗藏杀机，隐忍多年配合凶手完成报仇。

你是秦国公子扶苏，荆轲来进贡献上地图，胡亥受小人控制想要篡位，你想保住秦国的千秋大业。你的盟友毫不怀疑胡亥伪装的面目，输的理所当然。

你是身负血海深仇的日本警察。在警局办公系统数字化的过程中，很多罪犯利用关系伪造了自己的死亡。世界上没有人知道他们的存在，因为他们没有“身份”。你暗中搜查并一个接一个杀掉“身份干净”的人，他们才是真正的罪犯，因为司法机制的不健全，那帮愚蠢的警察总是无动于衷。你的妻子死于未成年罪犯之手，你怀孕的妻子一尸两命，被罪犯一刀一刀刺进肚子失血而死。你眼睁睁看着未成年的犯罪被释放。这个肮脏的司法系统，这个无能的侦查体系。你亲自动手，替上天惩罚早不该活在世界上的人。

你是龙族公子，你的爷爷是众神族长，由于其他族落的叛乱惨遭杀害。真的是叛乱吗？不是！你想起了前世收养你的爷爷，为了包庇你的军事起义，用脚印引导追捕你的军队走向错误的道路。你成为了雄霸一方势力的首领，才得知你的爷爷舍命为你争取的这个机会。你想起了前世的青梅竹马，由于地位悬殊，神妖殊途，你们的结合引发天谴降临。结婚当日，一半族人为你们的婚礼献祭。你想起了今世，原来族长为了避免悲剧发生，为了保护各族的孩子，走火入魔迷失了心智。三生石，三生路。

你是一味追求大学旧爱的男生，她受到你父母的干预无声息消失，你念念不忘。

你是一只妖怪，在怪谈中寻找自己的故事。

你是反叛军的首领。机器人占领星球，你带领强大的势力奋起反抗。后来发现你不是真正的你，你只是被覆盖了意识的克隆人，你的梦里出现了奇怪的画面，有雪山、有草地、有风、有散落满地的红色眼睛，你唯一的亲人其实是在苦苦寻找你的姐姐。

你是电话诈骗团伙的成员，负责选择合适的语音风格针对不同的目标进行下手。你是公交车祸里的受害者，伙同其他受害者一起，用诈骗的方式让当初事故的始作俑者受到惩罚。

你是小镇的猎魔人，配合小镇唯一的警官一起处理小镇上忽然疯掉的居民。原来你们都生存在计算机的世界里，主网服务器已经停机，你们是仅存的拥有完整建模的主机，病毒很快就会侵入这里。你的好兄弟警官勇敢地牺牲在和病毒的搏斗中。你面临选择，是毁掉这里让一切瞬间消失，还是带着渺茫的希望伙同伙伴们和机器战斗下去？

你是王朝的君主，鸟类神族的公主弃你而去，你征战多年，遇到一个很像公主的女孩，你隐约把她看成了梦中人的样子。多年后，你发现手上的伤口中残留着公主的神识，她没有忘记你，她一直在关心你，但是她不能背叛自己的种族。一边是自己深深想念的深明大义的公主，一边是陪伴自己多年的可爱的女孩。从前车马慢，一生只够爱一人。

你是班里的大哥。一场突如其来的灾难把你们压在倒塌的房屋废墟之下。古木低吟，你们得以重生在另一个世界。新的世界中，校园是黑色的，操场上的老奶奶面目苍白、肚子里的肠子还在对外淌血。教室里有很多带着面具的红头娃娃，睁眼看着你。树林里白色衣服的女人来回飘荡，手

上是阴森的长指甲。你们齐心协力战胜恐怖的幽魂，世界重归于美好，正好是校庆日。生活回到正轨了吗？校庆日当天，你自杀身亡在舞台之上。地震了，只有你意识清醒，你为了让弟弟妹妹们勇敢起来面对今后的生活，许愿让古木将你们带到更加可怕的生活中，让他们一次又一次重复面对不堪的世界。

你是山神的孩子。那一年闹山鬼，你的父亲消失在对抗山鬼的战斗中。长大后，每个孩子都很忙，约定的每年的忌日，也没有人再关心。他只想看你们一眼，今年他买了可以延迟拍照的相机，终于可以把他自己拍进去了。人呢？没有人回来？那年森林火灾，你们的消防员父亲死在火灾之中，他一个人将你们7个孩子抚养长大。你们儿时晚上看到的油鬼，是他脸上绑着绷带面目全非不忍吓到你们的脸。他一直在默默关注和保护你们。

你是经常光顾青楼的公子。你和招牌艺妓青梅竹马、私定终身，却无疾而终。

你是有钱的大学生，有个人为了钱接近你，装作你的好朋友。他亲手杀了你喜欢的女生，拜月教献祭，是她死亡的表象。混杂着地下交易的地下组织，她终究没有逃脱那里。

你是星期一。从星期一到星期日，一人一天。你们是一个人，又不是一个人，你们的性格不同。你们被坏人关到地下室，一个好朋友救了你们。从被救的那天开始，你却察觉到异常，你真的是你吗？你希望七个人永远在一起，可其他人逐渐有了新朋友，只有你孤单一人。你们许下的生日愿望，都被躲藏在黑暗中的神秘人用奇怪的方式完成了。有人想要母亲的怀抱，神秘人就把母亲的尸体藏在沙发里，每次躺到沙发上，都可以感受到母亲的温暖。有人想要重见小时的朋友一面，神秘人就杀掉一个体型相仿的孩子，把照片贴在尸体脸上。对了，这个神秘人，儿时父亲和母亲吵架，他害怕自己被抛弃，在父亲自杀后，亲手用刀子割开了父亲的肚子，钻了进去。这个神秘人，就是把你从地下室解救出来的好朋友。

你是台湾原住民。荷兰军队进驻台湾，想要占领中国大陆。你一面作为明朝的卧底，在荷兰军官中深受赏识，一面想要团结台湾原住民们团结奋战，建立自己的军事政权。她是你唯一想念的人，你曾经落船漂流岛孤岛上，和一个自以为是的野蛮人共同生活十二年，从零开始种植作物、建造房子。他现在是荷兰军队的高级将领了。你不忍和他兵刃相见，可又必须将荷兰人赶出这片土地。没想到，决战当天，清朝军队、西班牙军队、飞翔的荷兰人号，都来了。

你是一个依靠作弊拥有傲人成绩的学生，你是一个商场得意却患有抑郁症的商人，你是一个在马戏团工作的小丑，你是一个商场失意的货车司机。你的父亲也许不理解你，高考当天你离家出走，逃离你熟悉的城市。第二天，你的父亲满脸疲倦，敲开你的房门。走！抑郁症也是病，咱们慢慢治！你以为会迎来一阵毒打，但收到一个充满愧疚的拥抱。

你是北欧之神奥丁诅咒的产物，你的祖先曾刺伤奥丁的一只眼睛。其他人看到你是白色的乌鸦，失心人看到你是正常的男孩。每当有白色乌鸦从蛋壳中诞生，当天会凭空多出不存在的第十三个小时，一切事物都停滞了，来自地狱的怪物会在此刻出现，肆意杀害方圆一公里的人类，夺走他们代表过去和未来的心脏和眼睛。被怪物杀害的人类会进入眼中世界，等待下次代表预言的第十个小时出现，他们可以趁机再次回到人类世界，夺舍还处于存活状态人类的身体。你的姐姐，你唯一想念的人，唯一对你好的人，唯一想放你走的人。你咬断自己的一根手指，和怪物做交易，让他不要杀害你的姐姐。可是由于万一挑一能够记住梦境内容的人出现，害你姐姐穿上了女仆的衣服，被怪物误杀。在这个世界上，你唯一挂念的人也离你而去。你本就是邪恶的产物，你的存在只会带来灾难。下一个诅咒之子，又是谁呢。

你是罪演员的候选人。你曾经是全国最优秀的刑侦警官，一同全来的还有你曾最欣赏的部下。神探仪的出现让刑侦警察显得可有可无，公安系统大规模减员，随之兴起的是配合神探仪定位真正凶手的职业罪演员。当年最优秀的罪演员团队六边形集体身亡，数次愚弄大众的逃脱者到底身在何方。

你是重点大学的毕业生，你是表妹的哥哥，今天替父母参加她的家长会。你的父母重男轻女，对自幼遭遇家庭变故的表妹并不上心。你的女朋友送妹妹一双皮鞋，不知道妹妹有多高兴。可是有一天你发现妹妹弄丢了她的皮鞋，穿着一双破破烂烂的鞋子回家。妹妹怕你生气，专程到你的房间里道歉。你对一切并没有什么感觉，直到收到学校妹妹身亡的消息。妹妹和其他同学一起，尸体被发现在悬崖的下面。你和其他家长一起分析妹妹和其他同学的日记，这么可爱的妹妹，这么可爱的同学们，全部遭遇混蛋老师的猥亵。更加可恶的是，你们没有任何证据指认凶手。你们只有用一生去弥补，自己对罪犯的无能为力。

你是.....

你不想玩一场剧本杀，体验一次光怪陆离的奇异人生吗？

# 『Ground-Up Blockchain』前言

2021-10-02

以前有过一个设想，假如一个普通程序员，愿意花费 5 年的时间专心学习和研究某一个技术框架，比如 [Spring Boot](#)，5 年的时间足够了解这个框架的多数细节了，那么他在 5 年之后，可以凭借对 Spring Boot 足够深入的了解，写一本《Spring Boot 从入门到精通》之类的书没有问题吧。写出一本书可以带来什么？至少可以增加一些被认可的依据。“写书”的难度大吗？看看现在中文的技术类书籍，哪个不是电子垃圾？然而事实上，大多数 5 年以上工作经验的程序员都做不到“写书”这件事。即使写的难看，毕竟也是有，聊胜于无啊！为什么没有人写？

有多少书写着写着就变成了技术手册？大而全没有重点，有用的没用的全写进去。也许是本着对观众负责的态度，“我写的内容不一定有见地，至少全啊！”这样的逻辑类似于，消费者在买东西的时候，“这个功能我可以不用，但不能没有！”

我也想制造一些电子垃圾了，就像各种无聊的技术博客文章的集合。现在是 2021 年的国庆节假期，正好有时间可以思考一下这件事情。

书的内容会和 Blog 冲突吗？如果有有意思、值得写的东西，应该优先发到 Blog 上。好像也是，不过 Blog 上的内容更多是描述个人经历、表达态度和观点，一直都无法专注尤其是低质量的技术内容。Blog 的内容往往需要字斟句酌，可能最终看到的只有 100 个字，但实际上也许想了 1000 个字，思考好几天，然后去掉不合适的措辞、精简内容、明确清晰观点，剩下了少数简练但有用的内容。

书的内容会更随意一点，为了节省时间，也尽量避免对内容的反复修正。总得有一些新的事情做，这些事情总需要一个开始，你不能等所有材料都准备好了才下锅。如果以后有一天，我有足够的写出有价值书的能力，可能就不想写了。

书名借鉴了 *Ground-up Computer Science*，我暂时没有更好的主意了。内容会聚焦在 Blockchain 上，这个应该没什么问题，这个方向的水很深，有足够的内容可以写。也不需要太多担心机会成本的问题，其他领域并没有更好的选择，

在用语上，可能会直接用一些简单的单词。因为经常出现的情况是，在阅读其他资料的时候看到了某个词并且留下了印象，然后就直接拿来用了，不希望刻意在头脑里翻译一下。比如，这本书的内容没有 magic，就是一些普通的技术大杂烩。

这件事情的周期可能会有点长，预计 1~2 年左右。希望在 2023 年年底之前，这本书的内容可以初步让自己满意，可以归档 0.9 版本。差的 0.1 用来勘误。

书里具体的内容以思路为主，我们从小就知道“画一条线 10000 美元”的故事，画一条线价值 1 美元，知道线画在哪儿 9999 美元。故事也许不是真的，但故事广为流传，侧面说明故事中的逻辑至少有道理。把代码写出来，远不如知道为什么要写，解决了什么问题，有没有更好的解决办法。

书的目录结构可能杂乱无章，因为不太希望按照结构化知识的方式组织内容，一方面不好操作，有些东西不好分类，另一方面，结构化组织内容的实际效果不一定好，反而会由于一味最求全面而忽略思路和细节。世界上没有“最全”的一个状态。内容可以是主题式、时间线式或者随心所欲的。

按照同样的逻辑，也很容易有 *Ground-Up Golang*、*Ground-Up Programming* 之类。Talk is cheap, just do it first.

预览地址：<https://gub.smallyu.net>

# 联盟链比公有链差在哪儿

2021-09-29

中文语境下的“公有链”和“联盟链”并没有明确标准的定义。2018年，美国国家标准与技术研究院（NIST, National Institute of Standards and Technology）在 [Blockchain Technology Overview](#) 中将区块链分为 Permissionless blockchain 和 Permissioned blockchain，但那样的分类方式并不严格对应公有链和联盟链。也许公有链和联盟链的区别在于节点网络规模的大小，也许区别在于区块链面向的范围是公共互联网还是私有局域网。无论是怎样的定义，我们至少可以大概区分出公有链和联盟链。

公有链和联盟链的好坏，不单纯在于技术或者某些评价指标的比较，也许会有人下意识地认为，公有链面对比联盟链更复杂的网络环境和用户体量，但其实技术上的差距总是有办法弥补，联盟链也有少数好于公有链的技术特性。

有一个段子《[皇帝的金锄头](#)》：

古代有两个老农民畅想皇帝的奢华生活，一个说：“我想皇帝肯定天天吃白面馍吃到饱！”另一个说：“不止不止，我想皇帝肯定下地都用的金锄头！”

联盟链就是在用区块链做传统行业的业务，甚至可以说是打着区块链的幌子到处骗钱。联盟链的问题就在于，格局小了。

# 享受你的生活

2021-09-28

我本可以有很多开心起来的理由，但总是因为一些不必要的目标让自己感到困扰。

我没有任何理由因为工作不相关的人和事情，放弃自己感兴趣的职业和技术方向。

我不想再花费时间关心自己不需要关心的问题，我有很多感兴趣的事情还没有做。

我为什么不优先关注自己的事情？

Maybe today is not an “improve yourself” day. Maybe it can be an “accept yourself and get through” day instead.



# 我的问题是什么

2021-09-27

唉，又要发牢骚了。我希望博客可以多一些纯粹的技术性的内容，似乎很难。最近几天感到一些焦虑和不安，但是不清楚来源是什么。想了几天，发现是一个有点熟悉的、想想就有点心累的话题。

## 说明

入职的时候，直接用已有的 Github 账号用作账号了。到现在也不确定是不是正确的选择。为了避免看起来像是对人有意见或者像是在吵架。后面的内容尽可能只描述客观发生的事情，不掺杂我自己的体会和感受，尽量不以对话形式写什么内容。

求生欲：博客就是写着玩，不要把网络上的言论和现实中的人物对应起来。

## 起因

今天发生了一些事情。我先后在微信上问了 HR、市场、测试一共 3 个人，想了解一下目前项目的进展和情况。后来 HR 知道后说，“市场的直接汇报对象是谁谁谁，你是不是疯了，怎么能直接问人家呢，这不合适”。

后来 HR 把我叫到会议室，问我说，我还有什么问题、有什么不知道的，可以直接问他。

HR 说，你为什么要问测试那些问题，测试的那些东西你也不懂。

HR 说，你就安心敲你的代码就行了，用得着了解那么多吗？

HR 说，公司融资路径清晰、前景很好，要是不好我自己就先走了，你有什么好担心的，人不都是为了那些吗？

## 问题

我认真想了想，我的问题到底是什么。我刚才想明白了，并且能解释我的行为和动机，解释我为什么要做那些事情。

我的问题是：我可以做些什么？

对于这个问题，我为什么没有直接问技术负责人？因为不是我没有事干，想找工作上的安排。

这个问题的另一种表述是：我能力的上限是什么？

这个问题，问是问不出答案的，没有人知道，除了我自己，甚至我自己，也不知道。

所以我为了回答自己这个问题，需要了解整个项目的情况，包括公司的盈利模式、市场计划、项目进度、开发节奏，等等。技术方面，代码放在那儿，我自己去看就可以了，其他的，可能得问人。

为什么要了解那么多？只了解技术方面的东西不够吗？

假设，强调，假设，我的能力上限是可以负责起整个项目，甚至在公司没有项目的时候，考虑公司未来产品的方向，就需要结合市场计划、营收状况等进行判断了。

是不是想多了？这些东西用得着我操心？这些东西轮得着我操心？

所以要假设……

我需要尽可能多地知道，哪些事情我能做，哪些事情我做不了，更重要的是，我做不了的事情，我为什么做不了，差距有多大。这个，也许只有我自己能判断。

## 立场

我面临的这个问题，其实长期存在，只是现在明确出来了。这个问题不会因为外界环境而改变。

## 分析

换工作解决了一些旧的问题，带来了一些新的问题。

重要的是，解决这个问题需要大量时间和经历。

具体的工作其实多么脚踏实地都可以，对于问题本身，目前只是处于了解状况的阶段，短时间内不希望和没办法做出什么动作。

## 回忆

可能因为之前经历的项目规模比较小，我是有信心负责起整个项目的，当然，指技术方面，从前端到后端，从应用层到底层。也可能之前的公司放权限比较多，我们从写标书开始跟进整个项目到交付。也当然，由于工作经验、资历、为人处事，公司不会给我直接的权力和位置。

最近总是时不时想起，当时提离职的时候，当时的领导说，你要是去大厂或者安全一点的公司，我们也只能是祝福，可是……也不是不好，怕你走错路。

当时的领导说，他和现在公司的高管也是有过来往的，你……（想好。（没明说，可能是这个意思））

## 最近

有同学要结婚了。每次都怀疑自己，是不是真的错了。

最近身体不舒服的厉害，十一假期后可能需要去医院检查一下。

## 其他

唉，发现很多东西不敢写出来。

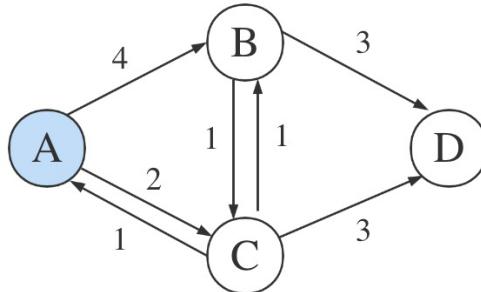
# 在 Dijkstra 算法中保存路径

2021-09-18

区块链的 Layer 2 中有一种 State Channels 的扩容方案，其中会需要搜索距离最近的路由节点。

## Dijkstra 算法思路

Dijkstra 算法能够解决 single-source 的最短路径问题，算法本身只输出一个点到其他点的最短距离。比如在这样一个图中，起点是 A，想知道到 D 点的最短距离是多少：



Dijkstra 算法实质是动态规划的贪心算法的结合，要寻找最短路径，就去遍历所有的点，每到一个点更新最短距离的记录，直到走过所有的点，就可以确信拿到了可靠的最短距离的记录。初始化的状态集合为：

**A B C D**

0 - - -

此时位于 A 点，未出发的状态，到自身的距离为 0，到其余点的距离未知。

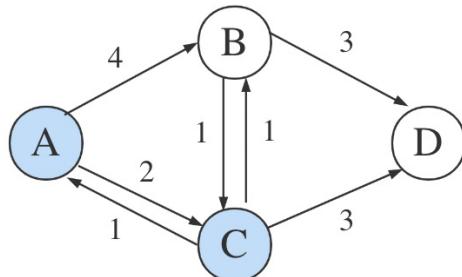
从 A 点出发后，发现 A 点可以到达 B 点和 C 点，距离分别为 4 和 2，那么就更新状态集合为：

**A B C D**

0 - - -

4 [2] -

中括号的含义是在当前这一轮中距离最短的点，哪个距离最短，下一步就到哪个点。到 C 点的距离比到 B 点的距离短，所以下一轮到 C 点：



到 C 点以后，发现 C 点可以到达 A、B、D 三个点，这个时候意识到，其实 A 点已经走过了，

不会再往回走的。于是需要另一个集合记录走到过哪些点，以避免下一步重复。定义 `prev = []`，因为 A 和 C 已经走过了，就把这两个点放到集合里，`prev = [A, C]`。

在这一步的时候，到达 B 点的距离从 4 变成了 3，`A -> C -> B` 的距离小于 `A -> B` 的距离，更新状态集合，同时因为已经能够到 D 点了，更新到 D 点的距离：

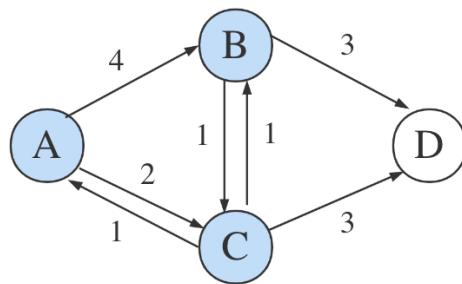
**A B C D**

0 - - -

4 [2] -

[3] 5

这一轮中，到达 B 点的距离小于到达 D 点的距离，中括号选中 3，并且下一步到 B 点：



此时 `prev = [A, C, B]`，状态集合更新为：

**A B C D**

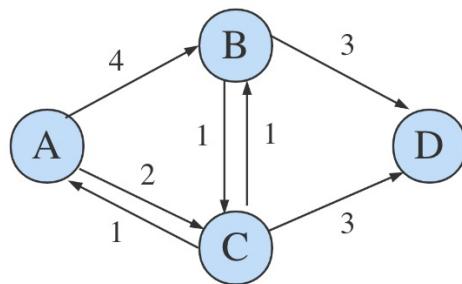
0 - - -

4 [2] -

[3] 5

[5]

中括号只剩一个选择，只有 D 点没去过了：



`prev = [A, C, B, D]`，所有点遍历结束，最终结果为：

**A B C D**

0 3 2 5

现在就可以知道从 A 点到 D 点的最短距离为 5.

## 最短路径跟踪

算法结束后，可以得到从 A 点到其他点的最短距离数据。可是如果不只想要距离值，还想要具体路径，比如从 A 点到 D 点的最短路径，该怎么处理？

### 正向贪心算法

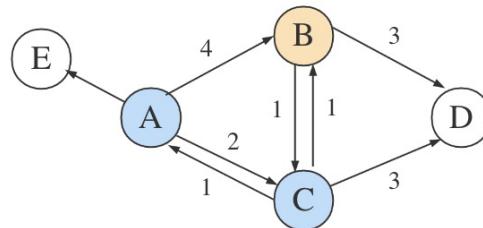
可以判断出，从 A 到 D 的最短路径是  $A \rightarrow C \rightarrow D$ ，而上面的 prev 集合为 A, C, B, D。因为从 C 直接到 D 比  $C \rightarrow B \rightarrow D$  的距离要短，所以在路径中抛弃了 B 点。

按照这样的现象进行对比，是不是只要在 prev 的基础上，在合适时候抛弃某些点，就可以得到正确路径了？比如上面从 B 到 D，存在 4 种情况：

- B 可以到达 D
- B 不可以到达 D
- 通过 B 到达 D 是状态集合中到达 D 距离最短的方案
- 通过 B 到达 D 不是状态集合中到达 D 距离最短的方案

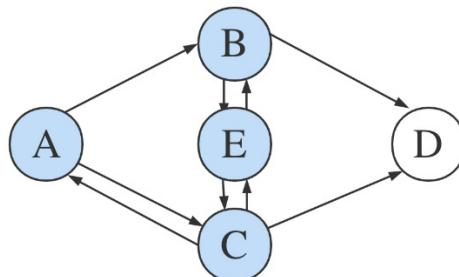
这 4 中情况中，只有 B 可以到达 D 并且 通过 B 到达 D 是状态集合中到达 D 距离最短的方案 的时候，才会保留 B 这个点到路径中。否则就应该去掉 B 点。

中括号每选择到一个点，就把点放到路径中，如果不满足上面的条件，就从路径中去掉这个点，也就是不放到路径里面。这样的话，即使有其他捣乱的点存在，程序也可以应对，比如：

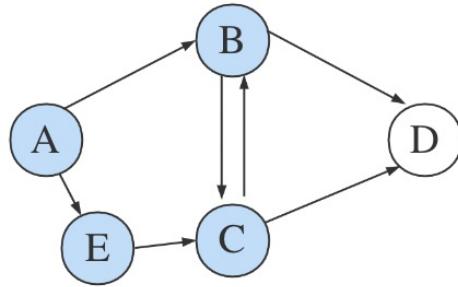


在选中 B 点后，发现 B 点不满足条件，此时路径由  $\text{path} = [A, C, B]$  回退到了  $\text{path} = [A, C]$ 。如果下一轮最小的点选中了 E， $\text{path} = [A, C, E]$ ，但是 E 点不满足条件， $\text{path} = [A, C]$ 。直到最小的点选中目标点 D，整个程序结束。

或者这样的，也可以处理，E 点不会被放到路径中：



那么这样的思路存在问题吗？当然有问题，这样的程序是不能处理这种情况的：



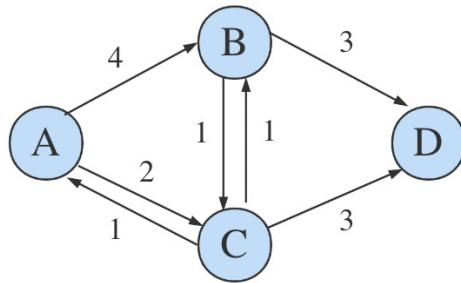
假如最短路径是 [A, E, C, D]，E点是不满足上面被放进路径的条件的，E点无法直接到达D点，但是又必须被包含在路径里。去掉可以直达D点的限制？那上图的E点也会被放到路径里。

也就是说，需不需要能够直接到达目标点，取决于对于最终的路径，被选中的点是不是倒数第二个点。这样的条件在一个未知的图中是无法判断的，谁能知道一个点是最终路径的倒数第几个点？

正向的贪心算法试图每一次都把距离最小并且在最终路径上的点记录下来，但其实很难做到，因为根本无法判断一个点是不是在最终的路径上。

### 反向贪心算法

当D点被中括号选中，作为本轮距离最小的点，就已经能够确定从A点到D点最短距离了。那么只要知道这一步是从哪个点过来的，来源的点就一定是最短路径的倒数第二个点。依次类推，只要层层回推到出发的点，整条路径就出来了。



假如在到达D点后，能够知道是从C点而不是B点过来，在C点的时候，能够知道是从A点而不是B点过来，整个路径就很清晰了。

问题是怎么样在D点的时候，知道是从C点而不是B点过来的？选中最小距离点的顺序可是 [A, C, B, D]，按照最小点的顺序显然是不行的。

这看起来不是一件难事，在DFS或者树的遍历中，经常会前后进入多个路径然后在适当的时候返回以修正路径。换个角度看，其实在DFS中维护最短距离，也可以达到目的。维护了距离状态的DFS == Dijkstra algorithm吗？显然不是。

### 递归 vs 尾递归

Dijkstra适合写成循环的形式：

```
for {
```

```
}
```

更适合写成尾递归的形式：

```
func recursion() {  
    recursion()  
}
```

总之，程序会是单向的循环。适合写成递归的形式吗？

```
func recursion() {  
    for {  
        recursion()  
    }  
}
```

当遇到分支情况的时候，用 for 循环“同时”进入多个路径，寻找最合适的那个。比如到 C 点的时候，for 循环前后进入  $C \rightarrow B \rightarrow D$  和  $C \rightarrow D$  的路径，每次循环将只保留一条路径，找到最合适的直接终止递归就可以。

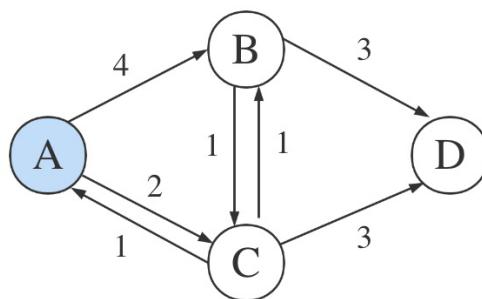
这样的写法存在问题吗？问题在于，怎么确定在哪个节点进行分叉。在 C 点分叉？为什么是 C 点？为什么不是 B 点？如果是 B 点，路径上就会多出 B 点。为什么不是 A 点？如果是 A 点，到了 C 点的时候需不需要继续分叉？是每一个点都需要分叉吗？想象一下那会造成多么大的冗余……为什么树可以同时遍历？因为树的节点不会交叉。

## 第二个动态规划

第一个动态规划是指算法本身距离数据的维护。第二个动态规划可以维护一个路径数据的状态：

```
pathList = {  
    A: [],  
    B: [],  
    C: [],  
    D: []  
}
```

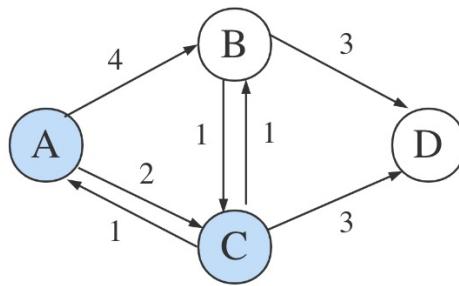
路径状态保存从源点到达每个节点在当前阶段的最短路径，在一开始的时候，因为 A 点已经可以到达 B 和 C：



```
pathList = {  
    A: [A],  
    B: [A, B],  
    C: [A, C],  
    D: []  
}
```

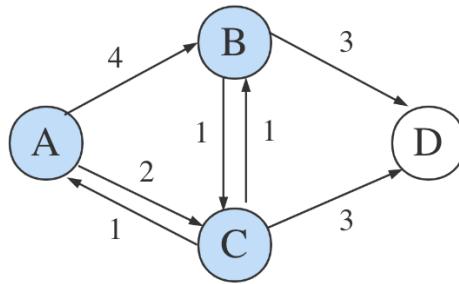
选择并到达 C 点，这个时候因为 C 点可以到达 B 点并且  $A \rightarrow C \rightarrow B$  的距离小于  $A \rightarrow B$ ，所以更新路径状态数据为 `pathList[C].push(B)`。D 点也可以到达了，更新路径状态。（更新路径状

态数据发生在进入下一个点之前，甚至发生在选择下一个节点之前。可以想一想为什么这样做。  
)

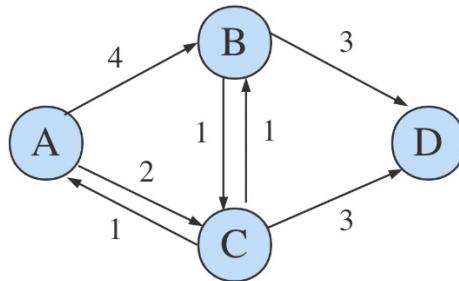


```
pathList = {  
    A: [A],  
    B: [A, C, B],  
    C: [A, C],  
    D: [A, C, D]  
}
```

这一轮在距离的状态数据上，会把 B 点选中为最小距离的节点，判断到达 D 的路径  $A \rightarrow C \rightarrow B \rightarrow D$  大于目前已有的距离记录  $A \rightarrow C \rightarrow D$ ，所以不更新路径状态。（判断距离是否大于已有距离是根据距离的状态数据，也就是表格的数据。）



最终进入目标 D 结束，路径状态不更新。



得到路径  $A \rightarrow C \rightarrow D$ 。

路径的状态数据可以为了节省空间，只维护到达目标点的路径吗？不可以，因为更新下一个点的路径需要依赖当前点的路径，路径的状态必须是全量的。

## 非最短路径跟踪

Dijkstra 算法包含了贪心算法的思维，每一步选出的都是距离最短的点。如果需要保存不是最短路径的路径，Dijkstra 算法也许可以做到，但是就已经不需要 Dijkstra 算法了。DFS/BFS 更合适一点。

# 最近

2021-08-22

## 理想

一开始设想了一些标题《来，我们谈一谈梦想》、《我们谈一谈理想》、《理想修正案》做大奖，现在连小标题都不敢轻易用了。

会想到“梦想”这个话题，是因为我已经不敢用“梦想”这样的词汇。不记得两年前为什么会频繁用“梦想”这个词，也不敢回头看当时写了些什么。不同时间说话的遣词造句都会不一样，比如上一篇的内容前后历时一两个月，会不断修改之前的用语和内容，到后面就变了很多样子。现在也许会说“理想”，不会说“梦想”，可能我已经不再做梦，不知道从什么时候开始。这是一种进步，也是一种退步。

前段时间换了工作，入职第一天和 HR 一起吃饭，谈到他们之前招人的时候，都会问有没有看过某项目源码之类，还问我有没有看过，我说没有。我当时说，我不是很担心（自己）。在后来接触到项目之后，我可以确信确实不用担心，我自认为是没有问题的，我需要考虑更多的是让别人相信我没有问题。怎么让别人相信自己呢，比如用幻灯片做技术分享。具体的内容分三个层次：

1. 写自己看到了项目的哪些东西，让别人知道一下，自己了解到那些了
2. 了解对比同类项目，分析功能和实现上的差异
3. 了解需要解决的问题是什么，目前有哪些做法，项目是怎么做的，其他项目是怎么做的，有没有更好的解决方法

这三个层次是递进的，需要的时间也是递进的。你分享的内容在那个层次，也许会让别人把你定位到哪个层次上，可是越往后面越需要时间，技术分享其实没有一个“准备好了”的时间点，总有更大更难的问题摆在那里，摆在你面前，摆在人类的面前。技术分享是为数不多可选的形式，但不一定是最好的主意。

就在思考这些问题的时候，我意识到一个危险的信号。我竟然在担心自己能不能够胜任工作，能不能够通过试用期，能不能够得到别人的认可。以前从未担心这些问题（光顾着 judge 别人了？/捂脸）。这些问题可以关心，但不应该担心才对。

也是前几天花了一些钱，突然就感觉没钱了，钱真是好东西。喜欢和追求钱没什么不对的，钱是货币符号，是等价交换物，你花费时间和生命换取金钱，消耗劳动力获得生产资料换取金钱，金钱是大多数社会资源的代表，这没什么不好。

我需要反思的是，为什么我在找工作的时候，包括前几个月和去年，都不止一次对不止一个人说，我不在乎钱（工资）。当时确实是那么想的，我简历上的期望薪资就是当时的工资，甚至比当时的工资还要低一点。现在分析原因的话，一方面，在衣食住行得到基本满足的情况下，暂时没有花钱的地方，没有太多欲望。另一方面，我当时是相信自己技术实力的，我相信只要实力到位，工资该涨就会涨，我一开始就只希望得到和自己能力相称的报酬。

我似乎忘了自己一开始的目的，忘了为什么来北京，为什么换工作，如果我只是在担心自己能不能保住市场平均水平薪资的工作，只是在追求稳定安逸的资金来源，那不是我该有的眼界和胸怀，我自己首先会看不起我自己。

在 Hacker News 的推荐列表上出现过一篇文章 [\\*The Top of My Todo List\\*](#)，作者是 Hacker News 创始人之一，在他的 TODO list 里，第一条就是“Don't ignore your dream”，现在我也需要这样提醒自己。文章中还提到另一篇文章 [\\*The Top Five Regrets of the Dying\\*](#)，两篇文章同时出现了意思相近的话，一个是“不要工作太多”，另一个是“不要工作太努力”，我没有明白其中的含义。

## 工作

前段时间的面试也反映出很多问题，我目前没有什么能够证明自己的经历，以至于我都开始怀疑自己。用阿里的话说，就是你做了哪些有难度的事情，为什么难，你是怎么做的，有没有更好的

做法，等等。如果我遇到那样的问题，我会无话可说，因为我遇到最大的难题，就是没有遇到困难。也因此我需要寻找更好的出路，如果有什么能拿得出手的经历，我也不会想走。

能拿得出手的经历实质是在为工程能力做背书，我之前不是很在乎和看重工程能力，毕竟有手就行，但工程能力实际上又是我赖以生存的重要手段，我不得不关心和提高它，因为公司可以相对简单直观地判断你的工程能力，很多公司更愿意为你的工程能力买账。码农嘛。（问：有偏离“梦想”的原意吗？）

前几天看到一个新闻，一开始是在 YouTube 的推荐里看到了这样的标题《[世界上最安全的加密货币、被黑客窃取6亿美元](#)》，没有点进去看，心想应该是人类行为，被诈骗什么的，标题唬人只是噱头。后来在不同的渠道都看到了类似的新闻，比如（现找的）《[被盗6.1亿美金，抽丝剥茧还原黑客攻击Poly Network 与O3 Swap现场](#)》，或者 \*[Poly Network hackers potentially stole \\$610 million: Is Bitcoin still safe?](#)\*，据说是去中心化金融历史上最严重的安全事件。

看到消息后各种不明白，Poly Network 那么有钱？也许只是一种营销行为？但是很多媒体都在说这事了？更多疑问和细节就不追究了。重要的是我忽然明白，我需要了解和掌握的绝不只是手上工作中接触到的项目，我需要了解的，是整个 DeFi (Decentralized Finance) 和 NFT (Non-fungible Token) 的生态和市场环境，那个圈子各种复杂……而且区块链的变化和演进相对互联网要快很多，在这样的时代的洪流中……

公有链才是真正的区块链，联盟链只是套了区块链的形式炒概念，是具有中国特色的区块链，是在用新的技术解决旧的问题。不过我也还没搞明白公有链的现状，暂时没有什么结论。很多时候一些公链项目都在强调全球生态，为什么是全球生态呢，因为国内没有生态。

反思之前，我为什么会有信心认为自己对联盟链有一定了解了？真的有足够了解到已经看不到未来的程度了吗？应该是有三个方面的原因：

- 之前接触的项目，虽然在技术上算不上顶尖，至少在国内能排个名吧
- 我当时对比了网上和能听说的各种联盟链，在技术上真就那么回事
- 当时认真了解了国内厂商广泛参与的某标准测试的规则，通过不算难

如果局限在那个圈子里面，只会安于现状固步自封自欺欺人。

## 哲学

在曾经的某个时候，我对一个人说，“我应该成为一个哲学家”。当然，没有收到任何回复。

为什么是哲学家？因为哲学家在思考的问题，人人都在思考，时刻都有人思考，没有人知道答案。为什么是苏格拉底？因为从传说来看，苏格拉底是一个粗俗的人，仅仅凭借一种执着的精神。为什么要了解耶稣？因为苏格拉底或者哲学家，一定程度上有宗教的意味，有一帮信徒，在传递某种思想。

不过，如果你去了解哲学史，会看到各种流派，各种历史人物，对几乎所有的问题，提出了几乎所有可能的观点。哲学的问题，是已经被定义好的问题。哲学的思想，是已经被穷举的思想。退一步说，哲学到底是什么？有人知道吗？哲学是什么本身不就是一个哲学问题吗？或许我们每个人都应该建立一个小目标：重新定义哲学。

有很多我们从小就熟知的哲学家，很容易被忽略。我们眼里的苏格拉底、柏拉图、亚里士多德、伊壁鸠鲁，可能就相当于西方人眼中的孔子、老子、墨子、荀子，甚至包括王阳明之类。美国在计算机技术上领先世界，可是由于历史短暂，并没有太过知名的哲学家，像德国还有叔本华、尼采之类大名鼎鼎的人物。在这个问题上，我一度充满疑惑。在技术领域上，前沿资料一定是看美国的。但其他领域，可靠的信息源是什么、在哪里？

## “新生代农民工”

前几天，人社部（中华人民共和国人力资源和社会保障部）发布对新生代农民工的 [监测报告](#)，这好像是第一次明确使用“新生代农民工”的称呼。这是一个正常的统计报告，发布报告的部门是“农民工工作司”，他们就是这样地工作，即使有问题也可能是很多人觉得，把之前一些模糊

不清的群体，现在明确划分到了这个部门做统计？问题他们做统计的依据是户口类型，他们一定没错。

看到报告，我第一时间想到，早在很多年前的某人语录，就吐槽国中国的户籍制度。他说的是另一个方面的问题，说我出生在中国，作为中国公民，从中国的一片土地到中国的另一片土地上，居然 TMD 只能暂住？不办证还驱逐？好在这样的现象已经改正了。

有人分析说，现在这个时候国家发布这样的信息，意思是让新生代农民工们认清楚自己的定位，安心回老家生孩子过小日子，别想那些乱七八糟的东西，只盯着光鲜亮丽看。现在不想生孩子的群体特性，一是高学历，二是在一线城市工作。学历越高、越是在一线城市工作，就越会接触到贫富悬殊的状况。他们不想生孩子，一是贫富差距大，二是阶级固化。在一线城市没有竞争力，就不敢生孩子。不敢生孩子，国家劳动力就不够。人口不增长，光是人口数据都不好编了。今年发布的一系列政策，互联网大厂减少加班、不允许孩子补课，都是在减少压力鼓励生育。

央视曾经有过论调，“认命可以，躺平不行”。有人说，自己读了研究生读了博士，结果读成了农民工。还有人说，天天自嘲码农，现在被官方叫一声怎么就破防了？我觉得，这个报告，最可怕的不是给人区分类型贴上了一个“农民工”的标签，而是默认了出生决定论。你是农民，本该务农，现在不从事农业工作进城打工了，我们得好好关注一下你，看看你什么情况。

不过这样真能提高生育率吗？如果你知道你的孩子生下来就是农民工，孩子就要说了，“生而为农民工，我很抱歉”。但是为什么要抱歉呢？如果有人刻意强调什么是重要的，那说明主流意识形态已经是和强调的观点相悖了，才需要强调。

## 听说

有一首歌《[听说](#)》，演唱者是华子。大学的时候听到觉得感人，在宿舍里单曲循环。室友听到说，这么难听的歌你老听它干啥。听过几次之后，我也觉得难听了，曲实在是很差劲。

最近经过一些事情，无意间想起这首歌。这首歌确实难听，因为曲根本不重要，华子不是一个合格的歌手。他是诗人。越是生活有体会，越是能够理解《听说》的含义。我甚至尝试模仿听说的文体写点什么，当然只是即兴的，没有任何推敲，先想起来写，后想起来这首歌。

不过，《听说》在思路上存在一个很大的问题。它只是把一些现象说了出来，这种现象还是结果形式的现象，没有说引发现象的原因是什么，人们面对的问题是什么，解决方案是什么。相比之下，白居易的《卖炭翁》就有很明显的指向。

也可能是言论尺度的问题？可以出现在古诗里的内容，不能出现在当代诗歌里？（还真有可能

## 塔利班

前几天有个刷屏的新闻，塔利班夺取阿富汗政权。有人大胆地开玩笑说，塔利班就是曾经的我们。还有个海外华人的自媒体说，他早在 8 月 1 日就发布预言，塔利班已经取得中国和美国的支持，将会夺取政权，然后 8 月 16 日，塔利班真的行动了，之类。

具体情况也不是一时半会儿能搞明白，我们需要明白的是，历史时刻在发生。经常有人说，在面对历史的大变革大是非的时候，能清楚时事的往往不是哲学家，而是历史学家。历史不可能倒退，但历史总在不断重演。

在关注到一些海外平台的政治不正确的言论后，我唯一了解到的信息，就是真的有坏人，他们可能训练有素、身居要位、执掌权政，不是因为他们想坏，而是因为他们都是人。

同一个国家在不同的历史时期，即使同一个决策者都会做出不同的决定，即使是同一个执政党，内部关系也错综复杂，一些幼稚的言论经常会把问题简单化，比如阿里女员工在食堂拉横幅，很多人直接归责于阿里的制度，把阿里作为一个统一的整体看待，包括很多人把百度作为统一的整体，百度干了什么什么坏事之类。“新生代农民工”也是同样，它只是某部门某些人发布的言论，沧海一粟而已。

## 其他

好难呀。不知道为什么，这次写这些东西感觉磕磕绊绊，经常语句不通用词不当。以前写东西都是带着情绪有感而发一气呵成，现在真的不一样了，不知道发生了什么。

以前写什么还会通读几遍修修改改，我不想改了，我累了。我要出去遛弯了。

# 工作两年的体会

2021-07-23

如果你看到了这些内容，那么我已经离开上一家公司，开始了新一个阶段的职业生涯。

## 最后的稻草

在一次技术分享会议上，我意识到我的技术理念和上级、和公司、甚至有可能和整个行业不同。这无关对错，没有冲突没有矛盾，仅仅只是——不同。

我希望技术用来解决技术问题，更关注技术的发展趋势和形态，我并非不会让技术变得具体，只是不认为“具体”意味着深度，尤其是把“具体”和人类的心理（信任模型）联系起来。具体的技术可以解决特定的问题，有价值，可是格局小了，我不会满足于那样的程度。

比如，我认为，计算机和计算机之间的路由器，就是负责转发网络消息、保证消息准确无误到达目标计算机的，不应该牵扯到消息的具体内容。如果是转账消息，路由器还应该冻结转出方的资金保证交易事务的顺利完成？这太扯淡了。

当然不仅仅是这样的原因，只是因此突然产生了出去看看的想法。想法的产生只需要一瞬间，在上级纠正我的观点时，我绝对承认上级在技术问题上，尤其是在业界，理解上的正确性。只是突然觉得，这一切都太无聊了。那个圈，我跳不出去。

即使没有实现理想的实力，也不妨碍继续拥有理想。

一直只知道远处的山上有恶龙，现在发现，山脚下有怪物挡住了上山的路，山脚下的怪物，名字叫权威。

## 时机

去年上半年也产生过离开的想法，原因是，当时完全在从事没有技术含量的 CRUD 项目，我不能就这样下去。其实今年的原因也类似。

后来没有离开的原因，不是工作经验或者技能情况。当面试官问我一些很无聊的问题时，我突然觉得，我没有兴趣再继续找工作。换个地点上班并不能解决我的问题，不应该把技术能力的提升期望于任职的公司。甚至，我其实自己都没想清楚，对新工作的诉求是什么。外加当时大环境影响，面试不方便，以及当时的上级后来给了我不少发挥的空间，就安静呆着了。

去年下半年，降薪，项目失败，部门人心动荡，离职率高。有同事问我，为什么不跳槽？我倒一直不那么看重薪水，尤其当时在“忙”一些别的事情，自己的事情，没有心思关注工作的问题。包括今年也是，不是想不想，是没有去想，个人问题太多了，顾不上想。

现在是一个好的时机吗？好不好不知道，至少闲下来了。你也许能想象到，一开始看原生的英文视频和电视剧，感觉多么痛苦，听也听不懂，看也看不懂。有一段时间对美剧丧失兴趣，完全不想看，因为 TM 看不懂啊。

在个人能力的问题上，有一个简单的逻辑。如果我的能力拿得出手，我将找到更好的工作机会。如果我的能力拿不出手，那我必须得走，无论成本多高，即使挣不到钱、交不起房租、在这座城市生存不下去，一定要走。如果只有现在的公司认可我，我就会苟活在这里吗？在现在的公司算是“被”核心，但是一两家公司对我认可与否，并不能说明什么。我从来没有期望拿到比自己能力更多的报酬。

这两年来，我没有闲着。虽然没有把时间用在刷题或背基础知识上，但确实没闲着。速度和加速度，我选择了提高加速度。斧子和木头，我选择了打造更好的斧子。我花费时间，让自己成为一个更好的人类，而不是成为一个更好的码农。

在工作时间没有明确工作安排的时候，我在摸鱼吗？是的。但那种鱼摸起来，异常痛苦。我需要

时刻思考，接下来做什么？我能做哪些工作相关的有价值的事情？每天早上醒来、每天晚上下班，想的都是，今天干什么？明天干什么？明明我已经尽力了，可什么事情都做不了。在工作上，我问心无愧（基本上）。不是故意偷懒，是实在想不到可以做的事情，何况连上级都没有。我才是真正想做些事情的人。

关于面试，我不希望刻意准备什么。必要的回顾和梳理经历是必要的，但不应该关注面试题。技术能力不可能一时半会儿提高，那种能短时间学会的东西，在哪儿学不一样呢，无论身处现在的公司还是未来的公司。我需要的，也许是一个试用期。如果能力不够份，我会自觉知趣地主动离开（当然，我知道现实社会不是那样运作）。我希望可以压力缓和的完成这次跳槽，如果在找工作过程中突然启动某个主题的学习计划，手忙脚乱不说，可能连找工作的兴趣都没了。

突然回想起在大学的时候，每次考试的前一天晚上，教室封楼，所有人都在宿舍里专心急迫地看书复习，校园里、操场上空无一人。只有我在外面溜达、在宿舍睡觉。需要会的已经在脑子，不会的临时看也没什么意思。

如果这次找工作遇到了比较大的阻力，那么我将会用亲身经历验证一个浅显的道理：两三年的工作经验远没有两三年的学历证书重要。我自认为学校里充满各种形式化的事情，各种魔幻的心理状态，无法教给我真正有用的知识。

（事实上，面试机会还是不少的，把握不住，是我自己的原因了。）

## 一些事

以前没有在博客里提起，因为这些事情对我没有“成长”上的意义。虽然有接近1年的时间都牵扯在这一系列事情和情绪里面，但确实没有意义，不值一提。现在回忆一下作为记录，防止以后忘得太彻底。

### 第一阶段

部门分两个小组，应用层小组和底层小组，简称A组和B组。我当时从属于A组。其实部门并没有多么明确的权限划分，我也没有搞山头的意思。

去年下半年，一个项目需要我（A组组员）和B组的组长合作，共同完成一件事情。从我当时的立场看，我只是个写代码的，出于在A组的工作经历以及对A组组长一定程度上的信任，心想好歹也是B组的组长，我又不太懂那个项目，他说什么我照做就是了。

实际的合作过程中却被难受了一把。他的思维逻辑没有足够条理，一会儿测场景一，一会儿测场景二，一会儿又要测场景一。然后突然产生个想法，说那就测一下验证一下，场景一和场景二都测。完全想到哪儿做到哪儿。我们测的是分布式系统，有4~6台机器，每台机器上2~3个程序，切换场景需要所有的程序改配置、重启、清日志、记录机器状态、记录结果、分析结果等，还是在远程桌面、网络延迟到能让人爆炸的情况下。其实这个还能忍受，当时也没有多想什么，只是觉得这样的工作有点枯燥繁琐，有点抵触。

重点在于他不信任我。我事后复盘了一下，他在整个过程中、对我指手画脚的这些工作中，得出的结论，全是我早就已经知道的结论。他连带我消耗的那些时间，完全没有对事情起到推进作用。我事前就和他说过，结论是怎样怎样，然后他说，“那这个得跑一下”，“看看结果”之类。我当时对自己没有足够的信心，毕竟他的title在那儿，而且我也期望他可以得到什么不一样的、更有用的结论，就听他的了。结果呢，他让我做的事情，只是把我的结论又验证了一遍。我开始对他的能力产生质疑。

后来又隔了几天，他让我重新验证一遍之前最好的一次结果。我在一些操作后，说直接写结论是多少吧。他坚持让我再操作一遍，把过程和结果拿给他。我就生气了，问“这次测试的目的是什么？是为了把结果给你看一眼吗？”我心想，我工作的内容就是给你打下手？我工作的价值就是给你看结果，让你相信我？在这件事情上，我们是合作的关系，还是从属关系？我凭什么做这样的事情？是你高估了自己，还是低估了我？你要真有能力也行，问题是对你没有。

然后我的解决方法是，我对他说，以后再有事情，找我的领导（组长或部门负责人），让我的领导来协调，而不是直接命令我。意思就是让我的组长帮忙挡一下，拦掉一些没意义的事情。我从

立场上并不能直接拒绝他，说这事我不干。如果我的领导一致认为一件事情是有意义的，从工作的角度，我肯定没有理由拒绝领导安排的工作内容。另外，如果我的领导一致认为我的工作内容就应该是给他打下手，那我当场辞职。

## 第二阶段

经过第一阶段的事情后，我偶尔还瞎想，要是有一天领导要求或安排，我转到他们那个小组了，肯定得搞得天翻地覆，因为我并不信任他（B组组长）的能力，我不会放心地把工作内容交给他。

然后神奇的事情发生了。B组组长离职，B组的某个组员成为组长，我转岗成为B组组员。

这NM搞笑呢？别说B组组员升的组长，就是原组长在的时候我也不服啊？甚至原组长的上一任组长、部门技术负责人（已离职），我也是一开始就怼的。

我确实不服。一个人突然来当我组长、当我领导，有权力安排我接下来要做的工作，这样的人，我不得先试试他的水平吗？我不可能轻易把将来的工作时间，任由没能力的领导瞎指挥瞎浪费啊。

然后我就用比较生硬的方式搞了点事情。^\_^

事实上，各种迹象都表明，他（现任B组组长，即将离职）能力确实不行。不是说技术能力，是指当组长的能力。一般来说，一个内心和实力都强大的人，无论面对别人怎样的质疑，都应该是岿然不动、稳如泰山的。结果他那儿直接就歇菜了，没有反驳，甚至还有点认可和顺从我说的话、我的行为。自己都承认自己不行，那我干嘛听你的？

最后是部门负责人出面协调解决。当我说出我的判断时，部门负责人说，你不应该反抗他，而是帮助他，一起完成该做的事情……（侧面默认了我的观点，至少面对我是这样的说辞）

事后，部门负责人还找我谈话，大概意思是，你最近没有搞事情吧？你不要再说他了，再说他就想走了，我怕他走……

## 第三阶段

或许你会觉得，第一阶段和第二阶段都是我的问题，是我让他们的工作遇到了阻力。要是换个听话一点、服从性强的人，就完全不会发生那样的事情。

换个角度想，也许是他们太弱了，连我都压不住。我不可能主动降低自己的标准，假装服从他们的安排。

我到目前还是相信，无论身份、地位、背景、履历，过去多么辉煌、现在多么有钱、关系网络多么庞大、人际来往多么高端，技术就是技术，逻辑就是逻辑。不会因为一个人是部门负责人、首席架构师、行业知名人士，他说的话、他的技术观点就是绝对正确的。

这是一个能压得住我的人。

所以我换工作了。:P

## 一些坏话

我喜欢《海贼王》的故事。

如果你在一艘几乎停摆的船上。连船长都不知道，船往什么方向开，要到什么地方去。你还敢坐吗？

## 感谢公司

必须要说明的是，上面提到的事情是从非常片面的视角、带有强烈主观个人情绪的描述。公司员工关系和睦、领导管理有方，基本不存在让人不愉快的事情。我只不过是把某条线单独拿出来说了，公司带给我的正面记忆远比写出来的事情多，我的成长离不开每一个遇到过的人。

感谢公司出现在我生命中的这两年半的时间。

## 感谢上级

在交接工作的时间里，我一度担心会被领导卡离职流程，甚至想好了各种对策来应对各种有可能发生的情况。这种担心的主要的来源，不是上级，也不是我的交接对象，而是另一个和我的工作无关也和我的交接无关的人。

他的逻辑基本上是，在我离职后，某些事情（不是我的本职工作，友情支持）他们自己处理起来会有困难，所以期望我在离职前就做好一些一步到位的事情。我推测他的动机有两种可能：一种是他认为自己技术能力不足，无法顺利完成某些事情；另一种可能是，他认为这些本该由他自己处理的事情，比较繁琐复杂，他也懒得做，就借机推卸事情，让我做。

无论是哪一种可能，都应该和我的离职流程无关才对，但也就是稍有点担心，他会以此去影响领导在离职流程上的判断之类，因为在新的组织架构和工作安排上，他是比较重要的承担工作的人，在领导那里可能会有比较高的权重，算是我小人之心了。我相信领导还是拎得清的，事实上也确实没有出现不合理的情况。

比起这种具体的小事，我倒是更在意，其实我现在离职得稍微有点早，还没有做一些什么事情，就走了。主要也不敢继续留在这里，这次换工作，更像是在逃生。我不敢牺牲以后的可能性，在这里做不知道结局的事情。

对以前和现在的上级心怀愧疚中心怀感激。

## 新的工作

在新的工作上，我认为会离“自由”更近一步。

这里的自由也许包含多种含义，但我明面上肯定只会承认技术意义上的自由。可以接触真正的互联网，是无法用社会资源衡量的精神财富。

## 祝好

将军不下马，各自奔前程。

# 剧本杀的剧本为什么需要区分类型

2021-07-20

前段时间，我写下这样一句话：

给剧本打标签、分类其实是非常错误、愚昧的做法，尤其会让习惯于“循规蹈矩”的玩家有先入为主的映象，认为什么什么标签的本就应该怎么怎么玩。

当时玩了一个情感本《春昼短》，游戏结束时，一个玩家吐槽（不是针对我）说，情感本不是这么玩的，重要的是代入人物体会情感，而不是全程都在分析凶杀案的凶手是谁，不应该把重点放在推理上。

我有感而发，一方面，是 DM (Dungeon Mater, 主持人) 说，现在是推凶环节，要找出凶手，然后所有人都在找凶手了。有什么问题？另一方面，情感本就不能关注推理和凶杀案了？剧本本身的情节薄弱无趣，根本不能引人注意让人感动，也不值一提啊。

过了很长时间后的最近，我满怀期待去玩一个备受好评的情感本《古木吟》，却由于其他人全程在关注凶杀案，最后感觉留有遗憾，明明是很好的剧情和故事，却没能把剧本发挥出来，体验到该有的效果。曾经讨厌别人吐槽的我，现在像那人一样（在心里）吐槽别人。

我开始反思，给剧本区分类型、打标签、划分不同玩法的做法，对吗？

答案是，对。

一般来说，剧本作者的水平高不到哪儿去，你不可能指望一个普通的剧本作者，推理部分逻辑缜密，故事部分感人至深，机制部分欢乐有趣，这不合理。

就算平时看刑侦推理电视剧、机制游戏真人秀，花那么多钱、请那么多明星、牵扯那么多资本，好看的有多少？何况是能够“欺骗”观众，让观众误以为真相是某一种，然后留下线索和悬念，最后有精彩反转的剧情，即使在美剧里也屈指可数。

仅仅是能达到好玩程度的剧本，就需要至少影视剧编剧水平的作者了。那样的作家又哪有功夫写剧本杀剧本，剧本杀的成本才多少。剧本杀的剧本，能有一两个亮点其实就不容易了。

所以为什么需要给剧本打类型标签？因为受限于剧本作者的水平，一种类型的剧本还真就只有一种玩法。用硬核本的玩法玩情感本，实际上牛头不对马嘴。

为什么会有“一种剧本不应该只有一种玩法”的错觉？因为很多剧本内容同质化严重，不管什么类型的本都会有凶杀案的桥段。好在这样的状况已经在逐渐改善，比如今年的一个热门剧本《来电》，没有凶杀案但是机制有趣、游戏欢快，最后的反转立意深刻，无论算不算最高水准的剧本，至少在好玩的同时还避开了传统落俗的套路，这是不小的进步。

那么，作为玩家，是不是应该在游戏前查看剧本的背景资料，了解这是一个什么类型的剧本，知道应该用什么样的玩法，明白重点关注哪些环节？

当然不是！

这分明是 DM 的责任。DM 自己首先应该对剧本了然于胸，然后在游戏开场前说明注意事项，让玩家知道侧重点，甚至针对不同水平的玩家在不同的游戏阶段进行单独的指导。DM 不能默认所有玩家都是老手，更不能默认所有玩家都知道某个剧本该怎么玩。

事实上，DM 的控场能力对玩家的游戏体验至关重要。

事实上，DM 的水平十有九差，剧本杀的体验十有九差。

我参与过的为数不多的体验比较好的游戏场次里，要么 DM 专业、全程在线，要么玩家之中有能自发站出来控场的老玩家。体验不好的场次里，几乎全是 DM 的问题，稍差一点的不参与不

引导流程，更差一点的直接误导玩家，给不重要的环节留比较多的时间。还有的 DM 催着玩家加快节奏结束游戏、完成自己的工作任务，丝毫不顾玩家的感受。

不好的体验也算是一种经历了。我本来就在尝试不同的剧本杀店。

回到剧本类型的问题，道理是相似的，如果一部电影在宣传的时候说，这是集爱情古装玄幻动作剧情伦理喜剧历史为一体的青春偶像片，不看都知道是烂片。

如果有一种编程语言，声称同时支持 FP 和 OOP，同时支持 actors 和 pipeline，同时支持 try... catch 和 union type，既是 static typing 又是 dynamic typing 还支持 type inference，适用于熟悉不同编程语言风格的人，那这种编程语言……是 JavaScript 吧？:P

如果说一个人说他既会前端开发又会后端开发，会 APP 开发还会嵌入式开发，懂区块链原理还懂神经网络原理，对大数据处理和数据平台的建设都有深入研究，那这样的人……岁数肯定不小了……

# 最近找工作的经历

2021-07-03

一直以来，自知心比天高，眼高手低，目空一切。也许看不起或者不在意某些东西，但闭门造车绝对不是好的主意。我经常站在高的角度思考问题，但实际上身处底层，这是无法轻易改变的事实。我会参加一些明知道通过不了的面试，以此了解自己和现实的差距，并由此选择接下来的路。这会是一段艰难的时光。

面试前就明白面试通过不了，其实很离谱。主要是早对一些企业的面试难度有所耳闻，这两年我虽然没闲着，但是都把技能点加在别的地方了，前段时间还在思考公司产品的方向、能不能找到亮点之类，几乎没有过记知识点，没有专注技术细节。这两年的经历让我能够应付目前公司目前职务的工作，以及同级别公司同级别程序员的工作，但是向上跳一个台阶，我是没有信心的。

(1)

经历过一些面试，有一点简单的体会。

招聘的公司分三种。第一种是我需要你拥有哪些技能，然后去判断你是否拥有相应技能。第二种是，我想知道你拥有哪些技能。第三种是，我看不起你，你看不起我。

面试的问题分三类，面试官会问这三个方面的问题：编程语言基础；数据结构和算法；项目经历、系统架构、系统组件。

虽然目前面试次数并不多，但一次次的面试失败还是会让人沮丧。理想的工作到底是什么？薪资？办公环境？工作强度？发展前景？同事关系？不可能样样齐全啊？好的机会够不着，不好的机会又没兴趣。

还是坚持一下，怎么也得面 10 家起吧。

(2)

招聘的公司会更加关注技能匹配情况。

其实对于跳槽来说，现在不太是一个好的时机，一般都说金三银四，金九银十，现在 6 月份大热的天，属于招聘淡季。

我突然想提前换换思路。

之前投简历的全是知名公司，也就是做互联网产品的传统行业。这些公司关注的技术能力也往往侧重常见那一套。在最近的面试中，我最大的感受，就是对区块链的了解完全无法发挥。这样的跳槽，是直接换行业了。

之前对区块链是充满失望的，国内的联盟链基本都不靠谱，即使是蚂蚁链、趣链这样的行业头部，过得也并不好。除了币圈和交易所能挣到钱，其他的企业都没找到商业模式。

一方面，目前基本能认为，我没有去互联网头部企业的机会，而且我已经大概知道这些企业的招聘要求。另一方面，不得不承认，区块链是当前为数不多的“新”技术之一。在优先级上，肯定要排在互联网一般企业之前。

所以，我接下来开始投简历到区块链公司，重点寻找有海外业务、海外背景的创业公司，希望从事 Permissionless Blockchain 的开发工作。

(3)

为什么进不了大厂？

去大厂需要回顾基础知识和刷题，这有点不同于我的价值观。所以首先还是想试试，在不刻意准

备的情况下，能不能找到合适的公司。

目前在等待某公司的结果，正好工作上也有些事情，暂时不再参加新的面试，看看后续的情况。

(4)

从开始到现在有接近三周的时间，虽然面试的公司数量少于预期，但……也很累。即使接下来会进行其他的面试，也算是下个找工作周期了。暂时先告一段落。

(5)

最近一周时间，又面试了几家公司，数量不多，质量不高，距离上一阶段时间较短，就直接把内容补充在这里了。

经历过上一阶段的面试，我意识到应该摆脱一些错误的思想。希望可以从容不迫地进行接下来的面试。

也许这一次找工作，结束地比想象中更快一些。

## 百度 - 百度网盘（工程方向）

记得当年找实习工作的时候，惟一电话面试的大厂就是百度，当时当场就挂了 :)

原计划视频面试，遇到点问题。百度用自家的“如流”视频会议软件，页面提示是有网页版本的，但实际上完全不能用，进入页面反复刷新还是提示“会议不存在”。点击下载如流软件的按钮，网页直接跳到了 `about:blank`。我的笔记本是 Linux 系统，如流又没有 Linux 版本，临时改电话面试了。

面试反馈和我的预期基本一致，算法能力弱，语言基础不是很过关。面试过程中能听出面试官模板化的问题，以及敲键盘的声音，猜测应该是有标准化的面试流程，在记录面试结果吧。据说百度有个传统，除非特别差劲，否则一面不会挂人。。。一面的问题确实偏基础，虽然我没有答好。

我明确没有回答出来的问题有两个：

- Golang 怎么实现一个并发安全的 map ?
- 很多个数字里，找出前 N 大的数字

语言方面，我经验确实不多。一直不太关注语言的细节，尤其是具体实现，除非真的用到，即使问 Java 相关的语言问题，我估计也答不上来。这也是我目前的工作很大的特点和弱点，代码量太少了！不过我不太担心这个。

第二个问题是经典的算法题，难度不大，能说上思路，但以前没有写过相关代码，说不出结论，不知道复杂度是多少。后来面试官降低难度，问排序算法有哪些？要的不是思路，是最高最低以及平均复杂度分别是多少。我不知道。

最后提问环节，我问提高技术能力从哪些方面入手？答沟通能力、代码能力，代码写的好技术不会差。

回顾及感受：今年找工作，第一家面试的公司，没有任何准备，基础不过关。面试官人还不错，但面试是有 checklist 的，一直至少目前不太喜欢这样的面试标准和流程。从意愿上，也没太期望一步到位，到所谓的大厂，跨度有点大。

## 旷视（数据平台）

(一面)

视频面，面试官没开摄像头。

比较重视 Go 语言基础，问怎么退出协程、向已关闭的 channel 里写数据会发生什么。

算法题相对简单，反转二叉树（共享屏幕写代码）、判断链表是否有环。判断链表是否有环追加了第二个问题，环的起点在什么位置？有印象做过但不是很清楚，回答快慢指针相遇的位置。

最后提问环节，我问提高技术能力从哪些方面入手？答去模仿、造轮子。关注独立工作的能力。

（二面）

手写 LRU，要求查询和写入时间复杂度都是 O(1)。不会。面试官比较和蔼，全程在提示。仍然不会。

最后提问环节，我问比较看重哪方面的技术能力？答学习的热情，Geek 精神。

（终面）

表现最差的一次。

一开始网络信号差中断了几分钟，直接打乱自己的节奏，变得慌了起来。想说的东西很多，却反而让说话没有条理。

问，在做的工作中，认为最有成长的事情是什么？其实太多了，没能清晰条理地说出来。

问，如果重新做一次之前做过的工作，会有哪些改善的地方？答的比较差。也是第一次到终面，面对高级别的面试官，面对这种类型的问题。

后来写转置矩阵的题目，实在是太简单的题，结果卡在 Golang 语法上好几分钟。

后来问你平时写测试代码吗，我竟然说工作中写的少。

最后提问环节，我问比较看重哪方面的技术能力？答这个问题太宽泛了，人的能力是多角度的。

（HR 面）

大姐姐笑的很欢……

## 中国知网（CNKI）

现场面试。

办公环境优美。

技术栈落后互联网多年。

面试结束时，面试官建议我说，朝着一个技术细节深挖下去。

回顾及感受：属于互相看不顺眼的情况，问了一些很 low 的问题。如果脱离面试场景，那样的面试官是没资格评价我的技术能力的。

## 亚艺网媒（探探）

现场面试。

一面白板写算法，是一个温柔的小姑娘。

二面白板画架构，重视对系统组件的理解、出题设计一个即时聊天系统。表现比较差。第一次遇到，稍微有点懵。也确实没有太多东西可说。

回顾及感受：能力不匹配，对方期望能够独立架构和运维整个后端服务的人，我目前架构经验比

较少。突然想起来二面的过程中，面试官竟然说 WebSocket 只能在局域网通信，不能在公网通信。我当年没出学校的时候，就看着教学视频，做过基于 React.js + Node.js + WebSocket 的手机端的实时聊天 APP。估计面试官的技术能力也挺水的。

## 海南新软（火币）

(部门一)

电话面试。

只问语言基础。全部不会答。

回顾及感受：原来火币也是血汗工厂，至少我面试的部门是。同样属于互相看不顺眼的情况，面试官自己说，除了语言基础，就不知道问什么了。算法呢？项目经历呢？区块链呢？搞笑呢？

(部门二)

这个部门的面试官态度很好，随便问了问，说是火币集团下做联盟链业务的，不会接触到币圈相关的业务，还说我的能力挺符合他们需求的。虽然我确实没兴趣。

## 轻松集团（轻松筹）

现场面试。

面试结束时，面试官建议我说，要关注技术细节。

回顾及感受：面试官性格还可以，技术上不敢恭维。和知网的面试官感觉类似，属于会根据学历给人贴标签、有心理预期的人。面试半小时，当场出结果，小公司模式，就是让去也不敢去啊。

## 主动取消 / 放弃

- 趣拿（去哪儿网）
- 华为 OD
- 滴滴（服务治理方向）
- 数美（HR 非常主动）

去哪儿网是这次找工作，第一家约面试的公司，后来时间上和百度撞了，推迟一周，面试官临时有事，推迟一周，第三周面试官又有事，我说直接取消吧。

主动放弃这些面试机会。流程化的面试，就不再继续参加了。估计也过不了 :P

## 贝斯平（Bespin）

问了一些语言基础和计算机基础。

问面试官，该从哪些方面提高技术能力？答深入语言基础、计算机基础，学习途径看书看博客。

## 恩佩弗尼（Apifiny）

(一面)

小伙子水平不行啊。小公司和大公司的区别一下子体现出来了。

了解到岗位是区块链钱包开发，做对接加密货币的账户和资产管理服务。

(二面)

面试官级别稍高，和一面面试官的反差太大了。

**京东云**

电话简单了解了一下情况。

**知乎**

很快结束，方向完全不匹配，面试官临时看简历，一开始就是放弃的。

# “我”是……（系列）

2021-06-24

“我”是一个有洁癖的人。

住在别墅的富商们都有洁癖，他们的柜子上摆满金银珠宝，他们认真擦拭灰尘、小心翼翼摆放，把一切整理的井井有条错落有致一尘不染。

我看到琼楼玉宇里的达官显贵们都是那么做的。

虽然我面前只有一个垃圾桶，堆满了肮脏的废弃的物品，但是我看到它们不整齐，我要把它们变得干净整洁，因为我有洁癖。

当我看到有人往垃圾桶里随意丢弃垃圾，不屑一顾置之不理，我会大声抱怨甚至斥责他们，虽然他们毫无悔改之心。

为什么他们不懂得干净和整洁，为什么不能像我一样，不能像达官显贵们一样，做一个严于律己有洁癖爱干净整洁的人，

我有洁癖，我比那些不懂干净整洁的人高贵，我比他们都高级。（狗头）

（2021年03月11日）

---

“我”是一个设计者。

我定义了一种新的加法运算，在这种运算规则下， $1+1=3$ ， $2+2=5$ 。

我出了一些题目，比如  $3+3=?$ 、 $4+4=?$ ，如果有人能回答出题目，就说明他们已经掌握了我创造的“知识”。

有的人热衷于学习我创造的知识，奉我为神灵，乐此不疲，以此为荣，甚至看不起没有掌握这些知识的人。

有的人还会为了我争吵，面红耳赤，在讨论是我创造的知识好，还是另一个设计者创造的知识好。

有的人会觉得掌握了我创造的知识，就比掌握了另一个设计者创造的知识的人高级。

我喜欢这些有趣的人，虽然在我眼里，他们都是虫子。（狗头）

（2021年03月23日）

---

“我”是一个建筑师，能够进行普通建筑物设计图纸的绘制。

有一次工头喊我帮忙，工地缺人手，需要把砖头从一个地方搬到另一个地方。我去了。

搬砖的工人不明白，为什么搬同样的砖，从同样的一个地方搬到同样的另一个地方，给我的报酬比给他们的多。

（2021年04月22日）

---

“我”是奥特曼。

人类总是“好心”地劝告我，怪兽特别厉害，怪兽特别强大，怪兽摧毁了很多人类的房屋和土地，怪兽给很多人类战士带来了死亡和恐惧，要我小心怪兽的攻击，要我对怪兽有“敬畏之心”，不要小看了怪兽。

我懒得解释我的手一举起来就可以发射激光。

(2021年04月22日)

---

“我”是一个船长。

我命令我的船员去一个小岛上寻找宝藏。一定有的，宝藏就在那里。

如果你非要问我，我为什么肯定那个小岛上有宝藏，因为别的船都到那里去找，而且找的很热闹，大张旗鼓。

即使有去探索过小岛的船员说，那里没有什么。

即使有去过更大的岛的船员说，那里没有什么。

一定有的，宝藏就在那里。

如果你非要问我，我为什么肯定那个小岛上有宝藏，因为别的船都到那里去找，而且找的很热闹，大张旗鼓。

(2021年04月29日)

---

“我”是一个游戏的资深玩家。

我经常说这个游戏如何好玩，为什么好玩，为什么值得玩，多有前景。

我们经常讨论这个游戏的玩法，阵营，策略，升级，成就，装备，公会，任务，操作，等等。

一个新手玩家来说，这个游戏不好玩。

一个新手玩家来说，这个游戏虽然好玩，但没有另一个游戏好玩，没有另一个游戏有价值，没有另一个游戏有前景，没有另一个游戏有意义。

我怎么可能抛下在这个游戏里的积累，去玩另一个我没玩过的游戏。

我甚至不会承认，新手玩家来说的是对的。

即使新手玩家来说是对的。

(2021年05月07日)

---

“我”是一个地主。

秋天到了，家里需要雇佣工人去田地里收割麦子，长工短工都有。

经常看见工人们相互探讨收割麦子的手法、步法、心法，相互比较谁割麦子速度快、质量高，谁经验多、能够快速收割掉各式各样没长齐的麦子。

越是手艺好、会说话、忠心的工人，我自然越是喜欢。他们也会为此洋洋得意，因为得到了比其他工人更多的认可和报酬。

手艺好的工人是有优越感的，他们可以到任意一家地主家里去收割麦子，因为他们有精湛的收割麦子的技术，大多数地主没理由拒绝这样的工人。这可不是一般工人能享有的待遇，他们已经算是高级的工人了。

高级的工人有时还是给普通工人讲解割麦子的技巧，传授收麦子的心得。地主招收新的工人时，也会让高级工人代为把关，用一些刁钻的技术问题判断，想来的工人到底能不能达到地主家的要求。

(2021年05月26日)

# 不知所言

2021-05-06

## 人生中必须要做的事

探索自己的可能性，探索自己能力的边界。这是值得为之努力的事情。

庙堂之高的人，和江湖之远的人，总结出的人生道理是相似的。

谁比谁高级，谁比谁高雅。

人类几千年没有进化，人性没有变，本能没有变，欲望没有变，故事没有变，结局没有变。

用技术的眼光看待人类是狭隘的，只存在不到一百年的计算机技术也是狭隘的。

世界上没有惊喜，因为太阳底下没有新鲜事。

新鲜事时常发生，因为对于个体，每天都是全新的。

## 价值

文字的价值不以篇幅衡量。

文学的价值不会随时间消失。

诗词寥寥几句，流传百年，古今传颂。

鲁迅留有长篇，也留有短篇。

周杰伦的歌曲，光良的《第一次》，加上 MV，是任何时候都会动容的故事。

《圣经》中对人生和哲理的思考，穿越千年。

历史上少有闪光点，但每一个闪光点都惊为天人。

很多很多。

## 自然规律

生物的行为符合自然规律。

两个细胞进行复制和分化，需要 10 个月变成人类。

大脑接收信息，脑细胞形成特定形状的结构，也需要过程和时间。

不管什么博士教授天才地才。

所以问题从“我要做些什么”变成了“在当前阶段，我能做些什么”。

范围一下子小了。

## 公平

大年初二晚上，凌晨一点，《你好，李焕英》散场。

我在街上散步，看到睡在路边长椅的人。他们是谁的孩子，是谁的父母，是谁的亲人，是谁的朋友

友。

我比他们多拥有什么，他们比我差在哪里。他们从哪里来，到哪里去，为什么坚持活着。  
我从哪里来，到哪里去，为什么要活着。

他们活着开心吗，对生活充满期盼吗，有人关心他们吗，他们的未来有转机和希望吗。

我难道不会沦落至此吗。

虽然近在咫尺。

## 天下熙熙，天下攘攘

一开始，加了一些群聊。

后来兴起，自己组建社群。

一开始，群里来了富二代。随手发三四个 88 的红包。

大家前仆后继。

后来富二代走了，大家全力挽留，争相怀念。

天下熙熙，天下攘攘。

一开始，大家感到新鲜和开心。

后来，没有新的渠道，没有新的人。

我为此感到疲倦。

来了很多人，走了很多人。

我为此感到不安。

大家叫我群主。

我不敢乱发消息，不敢灌水，怕说错话，怕一不小心，说了别人不喜欢的话，怕人走。

我不敢轻易 cue 人，顾忌被误以为利用身份，牟取什么。

群主的身份，总是有人盯着。

感谢“群主”，让我被很多人认识。

我认真地把“群主”转让出去。

一身轻松。

# 我没有在博客上骂人

2021-04-23

我过去没有、将来也不会在博客上骂人。

对于博客的存在，我的态度一直是不主动、不拒绝。我当然不会在意有人从我的 Github 主页看到了博客链接，或者从其他网站账号看到博客网址的字样，但是也不太可能主动跟人说，“这是我的博客，你过来看一眼吧”。

博客就像是把自己写的一本书放到了公共图书馆的书架上，我不会刻意引导别人到第几排的书架第几个格子去找一本什么样子的书看看吧，但如果有人无意间看到了，自然也随意。如果看到的人正好发现，书里的内容和自己的行径有点相似，难道会认为这本书是为了嘲讽自己而写吗？即使指名道姓，也不会有人那么“自恋”吧。

对于现实生活中认识的人，博客里面的内容有比较大的可能提及相关的人或者事，稍微有点担心尤其是当事人会理解出“听者有意”的含义。对于网络世界中不认识的人，smallyu 只是沧海一粟，不值一提，班门弄斧，怕人笑话。而且，如果真有很多人看的话，我会有关联和压力，不敢轻易写“没有含量”的内容，那样的感觉太糟糕太压抑了。

记得很久之前有一次，把博客链接填写在一个小型的博客导航网站上，那种按点赞和点击权重的全是博客的导航网站，当时好像收到了（起码在某个时间段）排名第二的点赞数量。我第一次意识到，博客里的内容可能会有人认可，也是第一次感受到到“被看到”的压力。后来就没有再主动把博客链接发到什么地方了，我不会做类似“在论坛日经贴下面留言博客链接”的事情。

博客具体的内容，肯定不会专门为了针对某一个人写什么，更多的情况是希望描述某一类人，这些人可能会具有一些共同的特征，然后将某人某事作为典型举例。

如果有人非要认为，这就是在写谁谁谁或者什么什么事，你就是对谁有意见有看法，在表述一些不满或者不敢直接说的想法等等。有一个相关的逻辑：

- 问：为什么毛当年可以从一个北大的图书管理员，成为国家主席？
- 答：只是一个将会成为国家主席的人，在成为主席之前，当了几天图书管理员而已。

与之类比，

- 问：你一个过往经历不堪的人，凭什么觉得将来有可能会做一些什么事情？
- 答：只是一个有可能将来会做一些事情的人，有一段比较寻常的过往而已。

博客的立场不会是“谁的朋友”或者“某公司的员工”，而是“认识过拥有某些特征的人的人”或者“有过在某公司任职经历的人”。

所以，如果真的有人感觉“被针对”了，天地良心，我根本没有把那些人放在眼里。/doge

在关于人生的问题上，我比较好奇也有点盲目地崇拜耶稣、苏格拉底这样的人，同时也对西方神学和古希腊哲学充满兴趣。只是目前没有足够空闲的时间，也没有找到合适的门道去深入了解相关内容。

至于为什么会突然提到这个话题，其实也只是作为预防，包括以前存在的和以后有可能出现的内容。随着时间的推移、生活中往来的增多、网络中内容的增长，这个博客有可能会被越来越多次“不经意”地看到。还是同样的态度，我不会刻意掩饰什么，也不会主动宣扬什么。

# 为什么数字货币使用区块链是政治问题

2021-04-18

本来不想再专门提区块链。由于工作相关，接触相关话题比较频繁。

最近，一个高级别的技术管理吐槽我“数字货币不可能用区块链”完全是外行的观点，即使只是把四大行的大额交易记录到区块链上，也是很好的一件事情，而且可以用分层交易的技术架构，顺势解决小额支付在区块链上性能受限的问题……

我的逻辑很简单。

假如区块链在世界上从来没有出现过，没有存在过。在这种情况下，如果“上面”要求各大银行的交易数据必须同步一致可追溯。下面的人能做到吗？

不但能做到，而且可以做得很好。

区块链能解决的问题，不用区块链也能解决。这几乎是众所周知的事情。这也是为什么有人说“区块链没有新技术”的原因。

当然，这里的“数字货币”特指中国的数字货币，“区块链”指——在讨论区块链怎么用之前，是不是应该先把“区块链是什么”搞清楚？奇怪的是，似乎没有人关心这个问题。

在这个方面上，和区块链形成对比的是人工智能。人工智能能做的事情，如果技术跟不上，就确实做不到。

# 嘿，好久不见

2021-04-11

电影《小丑》里有一句似有非有的台词，我也想说这句话，“我记不清曾经发生过什么……”

也许只有像海子那样有奇特经历的人，才能写出“面朝大海，春暖花开”这样充满美好的诗句。不过这种梦想看似简单，那可是海景房……

想到几个关于外语学习的讨论：

1. 某网友在回答我的提问“50岁还能学会日语吗？”时提到，学习一种外语，至少要上千个小时。
2. 一个视频里的女性作者说，她刚到美国的时候，完全听不懂英语，然后就每天看那种谈话节目。三年后的突然有一天，她发现自己可以听懂了。后来她说学英语要看视频节目尤其是谈话节目，不要指望阅读材料更不要查单词，英语单词那么多查能会几个……
3. 罗永浩在讲话中提到，他用一年多的时间，像坐牢一样地学习英语，成为了新东方的 GRE 教师。他后来培训学校的一个老师，用了大概六个月的时候成为 GRE 入门课的教师。
4. 一个演讲视频里提到，演讲者在很多国家进行英语的教育，有一次她遇到一个听众用英语提问题，那位听众的英语水平一听就是 low level 的，但奇怪的是，和这位听众交流起来，甚至这位听众和其他人用英语交流，对话都非常流畅，虽然确实只是用了 low level 的英语。

我比较在意上千小时这个数字。

1. 假如一天看 1 个小时的视频，1000 小时就需要接近 3 年的时间。
2. 假如一天学习 8 个小时，学习 1 年大概是 3000 个小时。当然，一天 8 小时不是一般人能坚持的。
3. 假如一天学习 6 个小时，学习 6 个月大概是 1000 个小时。

数字是巧合的，光看数字可能无法断定什么，但基本上能够认为，技能的掌握情况和付出的学习时长有非线性但很强的关联关系。

虽然我以前不认可“工作经验”这种评价方式（比如中国古代的官僚制度里，满几年才有机会升迁之类），但一种技能确实需要足够的时间才有可能熟练掌握。更折磨人的地方在于，技能的掌握不是断崖式的，不会说昨天还不会，到了某个时间点，今天突然就会了。现实世界根本不存在这种类型的惊喜。

更常见的情况是，昨天一句话里面听懂了 1 个词，十天后一句话里面听懂了 2 个词，直到一百天后一句话里面可以听懂 10 个词了，但一句话总共有 20 个词，听懂 10 个词根本于事无补。两百天后能听懂 20 个词就可以了吗？语句千变万化，语气、语调、语速……更准确一点的比喻，就像一张图片从分辨率很低开始逐渐清晰，在完全清楚之前无法准确地识别内容但又可以稍微获取到一些信息，而且这张图片分辨率变高的位置和速度是不均匀的。

《神探夏洛克》里的夏洛克·福尔摩斯有个人设，就是关于探案所需要的知识技能无所不知无所不会，看一眼鞋底上沾的泥土就能知道一个人在最近三天的行程，知晓各种枪械的型号、子弹的直径以及各种场合对枪械的使用习惯。会把犯人尸体的人头搁在家里的冰箱研究人类死亡后唾液凝固的过程。就是这样的探案鬼才，却不知道一些常人知道的常识，比如月亮是绕着地球转的，地球是围着太阳转的。

猜到我想说什么了吗？也许而且一定是的，并不需要全知全会。无论是知识还是技能。如果非要说明一种技能掌握到什么程度最合适，只能说，当下的情况就是一切刚刚好。这种观点正好和胡适的“怕什么真理无穷”不谋而合 :P

顺便提一下，我对“知识”和“技能”的分类方法是，知识是可以短时间内知晓并理解的规则，而技

能则是很多很多个知识组合起来形成的可以解决一类问题的方式。知识可以用 3 分钟的时间掌握，而技能需要  $3 + 3 + 3 + \dots$  如果用有没有掌握某几个知识来判断一个人的技能情况，是以偏概全的。

生硬地切换一下话题。最近发现一种有趣的桌游叫剧本杀，虽然经常不尽人意但总还会留有兴趣。戏剧演员有两大表演体系，布莱希特体系和斯坦尼斯拉夫斯基体系，也就是体验派和表演派，在游戏里，我倾向于体验……当然，对这种游戏有个感触：我会玩的是游戏，不会玩的是人心。游戏有好有坏，剧本有好有坏，玩家有好有坏，体验有好有坏，心情有好有坏……

太阳底下没有新鲜事，但是太阳总会照常升起。

我也想有一间房子，面朝大海，春暖花开。

明天你好。

# 博客主题共享计划（草稿）

2021-02-11

## 前言

两个月前写下了这篇草稿，直到今天（04.10）想起来，觉得还是发出来。当初不发的原因之一是，我没能真正把项目维护起来。想把一个软件做到人人喜欢、人人适用其实是一件很有难度的事情，而且博客主题这样的东西受限于各种环境依赖的版本，还是不折腾了。借用《妖猫传》里的一句话，“事是假的，情是真的”。祝好。

## 正文

现在用的博客主题，模仿自王垠的博客 [当然我在扯淡](#)。几年前，我把模仿的博客主题开源在 Github 上 ([smallyunet/hexo-theme-yinwang](https://github.com/smallyunet/hexo-theme-yinwang))。虽然我经常更新和优化自己博客的主题，但是开源的仓库已经很久没有更新了。

一方面，不确定模仿主题算不算侵权。另一方面，我一直心怀芥蒂的是，我并没有想要依靠这样的项目来“赚星星”，我甚至不希望这样的项目出现在我的 GitHub 主页上，于是不得不开始自定义主页的项目面板。总的来说，因为这个东西不是我的，就像《夏洛特烦恼》中的夏洛，穿越回1997年后依靠周杰伦的作品火遍西虹市，可他终究发现那些东西不是他的。他只是做了一个很漂亮的梦。

现在，我想继续维护这个将博客主题开源的项目。

捂着不让人用、让人用着不舒服，这显然都不对，愚昧不会阻止真理的传播，好用的东西也应该被更好的利用。我甚至希望，这个主题样式可以成为一种标志，标志受到王垠鼓舞的人们，标志拥有正直和善良品质的人们，标志拥有真知灼见的人们。

这不是宗教情绪，我们不盲目信任和崇拜某人，我们崇尚 [理性的力量](#) (by yinwang)。我们不拉帮结派，不宣扬、不强制别人使用或者不使用什么，不强制别人相信或者不相信什么。我们以某人为“偶像”，自诩为某人的“粉丝”，不是因为我们归属于某种团体的莫名其妙的优越感，而是因为我们从他那里学到了很多可贵而难以言喻的东西。当我们在说“喜欢某人”的时候，其实想表达的不是某人有多么厉害，而是在说，“我们从他那里学到了一些东西，因为会的这些东西，所以我们也很厉害”。

很意外今天突然想到了这样一件事情，有点意外，有点惊喜，有点害怕和担心。今天刚好是农历大年三十，新年快乐。

# 区块链：下一代数字身份认证体系的基石

2020-12-08

如何在互联网的世界中，证明“我是我”？在 A 网站认证过了身份信息，到了 B 网站又需要认证一次？手持身份证拍照上传、人工审核，流程太繁琐？多个账户密码记不清、容易混，管理起来困难？自主主权的数字身份（Self-sovereign identity, SSI）正是可以解决这些问题的理念。

SSI 是数字身份运动中的观念，指只有用户自己拥有全部的、完整的数字身份信息，没有其他管理者和组织参与的数字身份体系。在 SSI 的理念中，用户拥有属于自己的去中心化的唯一身份标识（Decentralized identifiers, DIDs），用户可以完全控制自己的身份信息，可以在任何时候使用、更新或者彻底删除信息。用户可以创建并管理自己的可验证证明（Verifiable Credentials），自主决定在什么时候使用和分享自己的证明信息，而不需要请求其他中心化的机构、通过机构授权来使用自己的个人数据。

使用 SSI 的系统，所有的密钥信息都可以通过数字身份钱包进行管理，使用一个账号就可以登录所有的网站。在钱包终端中，用户可以随时向权威机构申请签发证明，包括身份证、驾驶证、居住证等各种形式的证件，都将以数字证明的形式储存在手机或电脑上。数字证明拥有机器可读、机器可验证的特性，不但可以放心地展示给第三方应用，第三方应用还可以在没有人工干预的情况下，直接验证证明的有效性，不需要签发机构的参与。

得益于区块链技术的不断发展，SSI 理念的实现逐渐成为可能。区块链系统本身就是点对点网络，天然拥有去中心化的特性，结合独特的数据结构设计和密码学技术的应用，加上共识算法在多节点数据同步方面的优秀能力，区块链不但能够保护数据的隐私安全，而且数据一旦写入系统便任何人无法篡改，为数据提供了极高可信度的储存环境。在 SSI 系统的建设中，将区块链作为可验证数据的数据中心（Verifiable data registry）无疑是最好的选择。

SSI 目前已经有诸多先例。2017 年，Sovrin 基金会发布了世界上首个公开的用于自主主权的数字身份的分布式账本网络，整个系统运行在开放标准以及公开源码的 Sovrin 协议之上，由 Linux 基金会的 Hyperledge Indy 项目维护。Sovrin 在 2018 年公布的白皮书中自问自答，“为什么网络世界中没有像物理世界一样可以用来证明身份的证书？直到区块链技术的出现，我们解决了这个问题！”结合 W3C 的 DIDs，Sovrin 提出了完整的数字身份和证明的解决方案。Sovrin 的主意一直都很明确，就是一定要构建和使用公开的、任何人都可以访问的、像比特币和以太坊一样的区块链网络。

eSSIF-Lab（European Self-Sovereign Identity Lab）是另一个案例。欧洲区块链联盟提出的 EBSI（The European Blockchain Services Infrastructure）是一个横跨欧洲的分布式节点网络，提供跨境的公共服务，有 28 个成员国签署了相关声明。eSSIF-Lab 项目是 EBSI 的一部分，由欧盟委员会资助，旨在促进 SSI 成为下一代开放、可信、安全的数字身份解决方案。欧盟曾在 2014 年 7 月 23 日建立了针对欧盟共同市场电子交易的电子身份识别和可信服务的法规 eIDAS（electronic IDentification, Authentication and trust Services），2019 年 5 月，eIDAS 宣布支持基于 W3C 相关规范的自主主权的数字身份。

微软在相关领域也表现活跃，2018 年 10 月，微软发布《去中心化的身份》白皮书，介绍了基于区块链的去中心化数字身份系统建设的技术方案，包括 DIDs 规范、去中心化的数据系统、DID 用户终端、DID 通用解析器、DID 身份中心、DID 认证系统、去中心化的客户端和服务等核心模块，详细说明了各模块组件以及各种角色在系统中的交互流程，为 SSI 系统的建设提供了非常好的模板。目前，微软已经提供公开的服务平台，可以体验相关的产品和能力。

此外，构建在以太坊和 IPFS 网络上的 uPort、使用自研区块链和支持第三方 DApp 的 Blockstack、能够适配比特币网络的 ShoCard 等都是优秀的案例。国内的厂商和机构也在进行相关的工作，如蚂蚁链提供的分布式身份服务 DIS（Decentralized Identity Service）、腾讯云的数字身份标识解决方案、微众银行的基于区块链的分布式多中心的技术解决方案 WeIdentity 等，都利用了区块链去中心化、数据高度可信的技术特点，构建了可靠的数字身份标识和认证体系。

区块链是一项极具潜力的先进技术，具有非常广阔的发展前景和应用空间，无论是国家政策的支持还是实际案例的应用，都体现出区块链未来的无数种可能。我们也在积极探索和推进区块链相

关的技术发展和场景落地，将区块链与同态加密、联邦学习、多方计算、零知识证明等前沿技术结合起来，使用最优秀的技术能力，促进下一个互联网时代的到来。

# 勇者

2020-10-26

孩子们出生在新手村，一起学习，一起玩耍。

后来孩子们逐渐长大，到了该出门的年纪。走出村子，面对更大的世界。

大家都去到了外面的世界。有的离村子很远，有的离村子很近。

大家手里有魔法石，可以彼此联系。

大家会随时联系，谈论最近的八卦，讨论王国的资讯。

到了特定的日期，大家会回到村子团聚，交流近况。

有的在附近的城镇打工，有的在稍远的城镇进修，还有的依然留在村子里。

有一个人，走得很远。

他在学院里修养身性，寻找高明的老师学习剑术。

修炼内功，精进技能。

他在森林里遇到过和蔼的兔子，也遇到过凶猛的野猪。

他披荆斩棘。

他向执政官证明自己的才能，成为冒险者。

他手里有一把剑。

当外敌入侵，他会用这把剑斩杀敌人。

遇到强盗，会用这把剑保护自己。

更重要的，这把剑背负使命。

不只是眼前的苟且，还有山上的恶龙。

这把剑，是他的武器。

恶龙，是他的远方。

他走在路上，看着脚下，想着远方。

有一天，他回家了，回到新手村。

他发现，大家过得都很好，生活和睦。

黄发垂髫。

怡然自乐。

他发现，小伙伴们关系融洽，有来有往。

不亦乐乎。

他想到了自己的日以继夜。

想到了自己的风雨兼程。

想到了自己手里的剑。

惟一仅有的剑。

启程的日子到了。

他回头看，看到了安居乐业

回头看，看到了市井繁荣。

回头，听到了欢声笑语。

他向前看。

满是苍茫。

身后，没有人在等他。

身前，也没有人在等他。

他想起来，曾有人愿意等他。可他拿起剑，独自上路。

出发了。

他看到了亲信好友。

看到了鸟语花香。

看到了小伙伴们。

他视若无睹。

置若罔闻。

某个小伙伴挥了挥手。

什么都没说。

他却动摇了。

出发了。

去哪儿呢？

他想到了曾经的高山流水。

想到了未来的日月星辰。

他喜欢这趟旅程。

也讨厌这趟旅程。

他再三犹豫。

反复斟酌。

辗转反侧。

彻夜难眠。

绞尽脑汁。

想尽办法。

只想问那个小伙伴一句。

“一起走好不好？”

甚至，他可以留下。

刚要开口。却看到小伙伴们开心的打起麻将。

吃着火锅唱着歌。

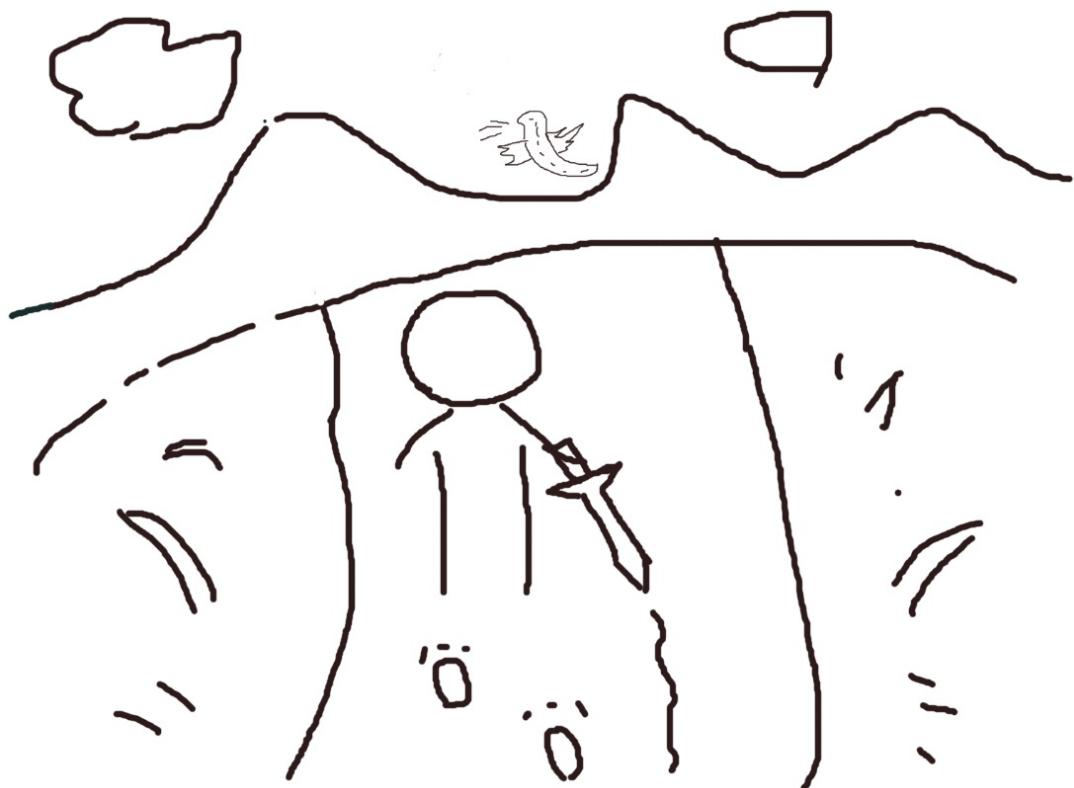
并没有人需要他。

这趟旅程。

没有回头路。

他喜欢手里的剑。

也讨厌手里的剑。



# 网页技术能实现3D建模吗？

2020-09-20

网页技术（HTML5、CSS3、JavaScript）能实现效果炫酷的3D建模甚至是3D动画效果吗？我暂时认为是不可以的。比如期望这样的页面效果：

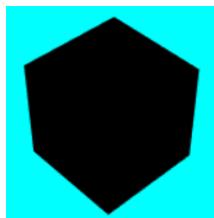


d3.js 是不用考虑的，它仅仅是一个数据可视化的工具，和 3D 建模是两个领域。

three.js 似乎是目前比较流行的 3D 建模库。假如 three.js 可以做到的话，应该怎么做呢？首先的想法是画这样一个正方体出来：



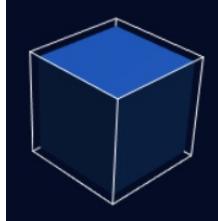
3d 建模里的立方体相当于编程世界的 hello world，很容易就能出来：



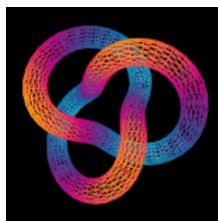
给场景加上灯光，正方体就不是黑漆漆的了。然后给正方体加上颜色，改一下场景的背景色，再把灯光调到正方体的上面，正方体就能像样子一点了：



目标正方体的边缘是发光的，而且是渐变色。怎么给正方体加一个边缘线呢，正方体本身是没有这种属性的，只能用线性材料（three.js 里的 LineBasicMaterial，正方体用的是 MeshPhongMaterial）再画一个正方体出来，套在实体正方体上：



怎么让线性的正方体发光呢？线性材料（LineBasicMaterial）是不能使用渐变色的，只有着色器材料（ShaderMaterial）可以使用渐变色。着色器材料可以实现多彩的效果，比如这样（来自 [StackOverflow](#)），：



但是到这里遇到问题了。在 three.js 里，渲染一个物体需要两个参数，一个是 geometry（几何体），一个是 material（材料），线性材料和着色器材料都是材料的种类。（TorusKnotGeometry 是上图用到的几何形状）

```
线条正方体 = EdgesGeometry + LineBasicMaterial  
渐变曲线条 = TorusKnotGeometry + ShaderMaterial
```

现在想要线性材料和着色器材料（LineBasicMaterial 和 ShaderMaterial）组合是不合逻辑的，我没有找到实现发光的正方体边缘效果的方法。把着色器用在正方体的效果是这样的（颜色从 0x215ec9 到 0x000000）：



所以然后呢？我意识到即使实现了一个好看的效果，离渲染出整张图还差的太多。比如这样的文字效果怎么做？



three.js 的 Texture 本身效果是不错的，可是怎么把文字安安稳稳的放到正方体上，还带透明的黑色背景框？再比如这五彩斑斓的线条，以及准确的箭头指向：

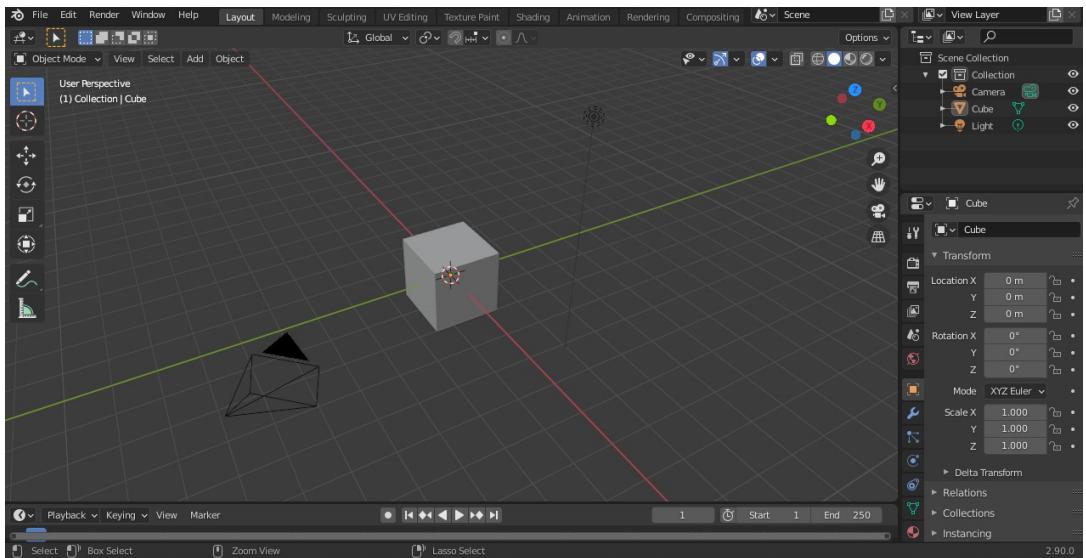


还有整个图上十多种元素的位置布局、动画效果。

我相信 three.js (WebGL) 在技术能力上是可以实现这样效果的，甚至官方的 example 里网页游戏都有，不过假如要实现一个网页游戏，一定会用到图像素材，素材从哪儿来呢？还是得回到 PS、AI 之类的工具上，如果用上了那样的生产力工具，就没有必要用 js 来写布局和动画了。单纯的网页技术似乎很难完全解决 3D 建模的问题。H5 动画也是类似的情况。

单纯写代码来 3D 建模的另一个问题是不直观，代码是违反直觉和视觉的，写个网页、APP 界面似乎还可以（二维的）。如果可以在一个画布上直接放置正方体和线条，然后鼠标拖动改变位置、调整颜色，以及添加各种其他元素，像玩游戏（比如我的世界）一样操作简便，不就比写代码好多了吗……那不就是 Adobe Animate 吗？

可惜 Adobe Animate 没有 Linux 版本，而且 Linux 下的替代品 Blender 有点性能问题。



回到一开始期望的效果图上，图片出自一个大屏 UI 的 [设计演示](#)，其实原效果不是三维的，作者使用的工具是 PS、AI。那么在仅需要二维效果的前提下，网页技术能实现吗？如果要用代码实

现各种图形，就依然还是三维建模的问题。最简单的方式是拿个背景图，把文字贴到上面。背景图从哪儿来呢？

# “做正确的事情，然后等着被解雇”

2020-08-22

“做正确的事情，然后等着被解雇”，这句话出自 Google 工程师 Tan Chade-Meng 的博客文章《[DO THE RIGHT THING, WAIT TO GET FIRED](#)》，一本关于程序员协作的书《[Team Geek: A Software Developer's Guide to Working Well with Others](#)》引用了博客中的内容（P126），然后书里的内容又被 CoolShell 的文章《[我看绩效考核](#)》引用。我在 CoolShell 的文章里第一次看到了这段话。看到之后，念念不忘，现在一定要把它复制过来：

New Google employees (we call “Nooglers”) often ask me what makes me effective at what I do. I tell them only half-jokingly that it’s very simple: I do the Right Thing for Google and the world, and then I sit back and wait to get fired. If I don’t get fired, I’ve done the Right Thing for everyone. If I do get fired, this is the wrong employer to work for in the first place. So, either way, I win. That is my career strategy.

得多么强大的自信和力量，才能说出这样的话啊！尽管原作者也说，这是半开玩笑的说法，尽管作者已经有了一些影响力，做过了一些“正确”的事情——没有身份和地位确实很难说出这样的话，但这句话背后的精神会一直鼓励和激励着我们——你知道我说的是谁，经久不衰，生生不息。也有不少文章展开讨论这句话的含义，比如《[Do the right thing, Wait to get fired](#)》。

当然，这句话不能反过来说，不是说等着被解雇，就意味着在做正确的事了。这句话也不适用于喜欢混吃等死的人，毕竟努力不一定有结果，但不努力一定很舒服 :-P 我得认真想一下什么是“正确的事”，以及如何做一些“有价值的事”。

# 给区块链一个定义

2020-08-09

一直以来，区块链似乎都没有一个明确的定义，伴随区块链出现的词语经常是去中心化、溯源、不可篡改、以信用为基础、下一代价值互联网之类，这些都是区块链的特性，不是区块链的组成，这些词都在说区块链有什么，没有说区块链为什么会有那些，以及为什么要那些。

感觉上很多东西都没有明确的定义，比如，计算机是什么？都知道是那么一个东西，可以打游戏上网，稍微专业点的会说是基于冯诺依曼体系结构的、有5个组成部分的什么什么机器。其实计算机可以认为是“做计算的机器”，就这么简单。冰箱是什么？“一个放东西的柜子，有一些冷冻的功能”，就可以了。设计模式是什么？一种软件程序设计的范式。微服务是什么？一种软件架构的模式。所以区块链是什么？奇怪的是，区块链（blockchain）这个词不知从何而起，从来没有人明确提出这个概念，比特币白皮书里也只是提到“chain of block”。

我以前对区块链有过一些不成熟的认识，虽然好像也没什么错，但不够清晰，尤其是没搞清楚一个问题，区块链是什么？现在来看，区块链的定义应该是：

区块链是一种数据协同软件，或者说，区块链是一种用来同步数据的软件。

数据协同软件决定用什么样的数据结构通过什么样的通信机制同步哪些数据。区块链不是数据库，区块链不负责储存数据，储存数据的事情会交给真正的数据库来做，区块链并不关心数据是怎么存在磁盘上的，不关心储存结构是否合理，利用率高不高，处理速度快不快。区块链关心数据以什么样的方式同步到其他的机器上，如何及时同步，以及其他机器同步过来的数据有没有问题。可以说，区块链是对数据协同软件的一种实现。

因为是数据协同软件，所以区块链多节点、去中心化，这显而易见。

溯源是指交易可溯源，只要数据之间有关联关系就可以，这是数据模型决定的，比如 UTXO。

链式的数据结构，是为了方便数据协同软件校验数据的完整性，类似用 md5 判断文件是否完整。这种数据结构并不是必要的，数据的全量对比也可以实现目的，只是效率非常低下。所以采用加密算法做摘要然后放到下一部分数据里的做法，相当于保证了一大块数据是完整的，仅此而已。

至于不可篡改，其实是数据协同软件带来的特性。区块链的不可篡改，并不是数据不能修改，而是改了之后其他节点不认可。这是不一样的，数据不能更改是技术问题，比如不提供更新数据的接口，用户就没有修改数据的渠道，通过技术手段可以控制。改了之后其他节点不接受，是一种机制，这种机制问题已经脱离技术领域。

区块链目前的发展受技术限制吗？计算机的计算理论包含两个主要部分，可计算性理论和复杂度理论。可计算性理论判断一个问题能不能用算法解决，复杂度理论意在提高算法的效率。和区块链有关系吗？退一步说，区块链需要计算吗？不需要，没有关系，不受限制。有个有趣的脑洞问题，如果把全世界的人都拉到一个微信群里，会发生什么？起码屏幕上的消息肯定刷不过来了。如果全世界的数据共用一条区块链，会发生什么？所以区块链最终还是机制的问题，不是技术问题。

比特币和区块链是两个概念，比特币是一种使用了区块链做数据同步的交易系统，比特币首先是一个交易系统，其次才需要的数据同步。这也是我以前犯的概念上的错误，把区块链等同于比特币了。很多对区块链概念比较模糊的人，提到区块链也都会往比特币之类的数字货币上想。记得去年参加过一个分享会，主讲人是某知名交易所总监，分享标题是区块链和国家政策什么的，整个会议下来，讲的却全是比特币的趣闻轶事。

区块链是比特币的组成部分。比特币的作者看到了比特币的价值，把软件和白皮书发布出来了。为什么比特币的作者没有把区块链的概念抽离出来，发个通用软件和说明书？是水平不够没有意识到区块链潜在的巨大价值吗？不是。区块链的提出，是因为人们看到了比特币的价值，想要复制比特币的成功，所以把比特币的技术组成提取出来，叫做区块链。可惜比特币是一个设计巧妙

的系统，单独把某些技术特点拿出来难以产生预期的价值，这也是区块链的现状。这是现代版技术圈的东施效颦。

智能合约（Smart Contract）早在 1997 年由一位金融、法律从业者提出，“智能”是指和纸质合同相比，智能合约达到某一条件时就会自动执行某些操作，确实比纸质合同智能了一点，尤其在那个年代，数字化还没有普及，描述这是一种智能并不为过。而且，作者明确说，智能合约没有用到人工智能。

智能合约抽象一下，达到某一条件自动执行一些动作，不就类似编程语言的条件语句吗，事实上现在的智能合约大多是用图灵完备的编程语言实现的。用编程语言来描述合约的致命问题在于，编程语言的表达能力比自然语言弱太多了，如果试图用编程语言来重写保险说明书里的所有条文，“发生什么，就赔偿多少……”，这种改写成本太高了，而且很多时候法律条文需要专业律师、法官解释和判断，现实世界的逻辑远比程序逻辑复杂，编程语言是搞不定的。

一种数据同步软件不应该被推崇，区块链被神化、妖魔化了。也因此，不能说区块链没有价值，因为区块链是且只是一种工具软件。

# 宜怀旧 · 近期微博内容归档

2020-06-26

一些简短的想法好像更适合实时记录在新浪微博（简称微博）这样的微博平台上，而不是直接或者总结后发布到博客里，博客对内容的定位似乎稍微有点“重”。

不过微博的内容质量实在是太差了，我尝试过多次打开微博后随便刷一刷，劣质的信息流让人无法忍受。而且微博有两种罪恶，一是能接受的“尺度”太小，我的网名 smallyu 仅仅因为包含“sm”两个字母，在微博的搜索引擎上就无法通过我的用户名搜索到我发布的内容，这很奇怪，也就是说任何包含“smile”、“smart”之类单词的网名都不能正常显示在搜索页面上，这种问题绝对和技术无关。微博的罪恶之二是用户会受到平台的干涉，崔永元在 Youtube 上开通了账号并时不时发布短视频，有一次提到说他在微博有两个账号，两千多万粉丝，但是忽然就不能发布内容了，没有给出任何说明，就是发不了内容，新浪微博不让发，所以现在开始用 Twitter 和 Youtube 了。

相比之下 Twitter 确实好很多，阮一峰的科技爱好者周刊 111 期里有个故事，Twitter 几乎给了用户最大限度的自由：

一个推特用户做了一个实验，注册了一个帐号，特拉普发什么推特，他就发一样的内容，不是转发，而是原文复制，除此以外不发其他内容。

结果，推特官方三天就注意到了他，关闭帐号12小时，要求他在这段时间内删除违规言论。同样的话，特朗普可以说，你说就不行。媒体报道这件事以后，推特恢复了这个帐号，并说关闭帐号是一个“失误”。

想起来关于工具的选择的另外一个事，deepin 论坛里经常有人推荐360浏览器，我曾经也非常喜欢用360的浏览器，尤其是360极速浏览器，是一个基于 Chromium 内核套壳的浏览器，里面有一个自带的手势操作的功能，按住鼠标右键在浏览器窗口上划拉几下就能便捷的操作页面了，前进、后退、新建标签页、关闭页面、滚屏、滑动到页面顶部，等等，很好玩。直到后来有一天，国产浏览器集体封禁 [996.icu](#)，我决定再也不用国产浏览器了。

我不希望继续使用新浪微博了，Twitter 对内容几乎没有限制，UE 也很好，但是毕竟和我们隔着一堵墙，网络环境总有点差强人意。至于怎么安置内容不是很多的短时间内的想法，得再想想办法了。前段时间本想在微博里“制造”一些看起来“正能量”一点的东西，后来有点跑偏了，不过原则还在，也就是只提“技术无关的话题”。

这是最近一段时间的微博的内容，按照时间正序归档到这里。

5月4日

亲情是一种糟糕的情绪，它会让人有难过的感觉。

5月5日

租个小房子，都不知道天是几点亮的。

5月7日

有时候觉得生活糟透了，会不会以后也这样。

有时候觉得生活糟透了，好像前一段时间也这样。

第一次听到的让人心情激动的歌，再听就不觉得多好听了，第一次吃到的很好吃的麻辣小龙虾干拌方便面，再吃就没有太大感觉了。这是一种惩罚。

5月15日

我们都习惯于在游戏里做主角，在影视剧里做英雄，但是把我们放回盛唐，我们做不了李白，也成不了捉妖精的白居易，我们只会是一介平民，在历史上不配留下姓名。

喊加油的人是比较轻松的，在赛场上拼搏的人压力才最大。所以加个油算什么贡献？

5月16日

王者荣耀删了又下，下了又删。删掉是因为浪费了太多时间，下回来是因为时间浪费在了别的地方，还没有得到好的娱乐体验。

5月17日

现在的人，一旦微信拉黑就是真的绝交了，没有任何再挽回的机会，因为除了微信，没有其他方式可以联系到它了。住址更是无从得知。

5月18日

自得其乐也是一种生活方式，有梦想有追求也是一种生活方式。

5月20日

只有只有你知道，别人不知道的东西，才有价值。

5月26日

“没心没肺的人睡眠质量都高。”

5月27日

听歌的时候，如果突然听到一首好歌，后面会开始逐渐变得不耐烦，期望再遇到一首好歌，切歌变得频繁，但往往再难听到心动的歌。甚至切过很多的歌后，只好回头再去听当初的歌。

5月28日

《像我这样的人》

5月29日

一局不想赢的游戏，好像就变的不是那么好玩了。

流量限速了，早上听歌的时候，QQ音乐停在开屏页面进不去，因为开屏广告一直加载不出来。加载广告已经成了QQ音乐必需的一部分。相反网易云音乐就好一点。

听到一首好歌的另一面是，提高了对烂歌的鉴别能力，只要听一下前奏，或者一开口，就知道不是自己喜欢的类型。

6月3日

百丈怀海有一次陪马祖大师外出，看见一群野鸭飞过。马祖问他：“那是什么？”怀海说：“野鸭子嘛。”马祖又问：“飞哪儿去了？”怀海说：“飞过去了。”马祖闻言上前，使劲拧怀海的鼻子，疼得怀海大叫。马祖说：“叫你还说飞过去了！”怀海当下大悟。

马祖的意思是：野鸭已经飞走了，你还在心里记挂着。

因此想起慧海禅师的故事。有人问慧海，他是如何用功的。慧海说：“很简单，饿了就吃，困了就睡。”那人说：“人人都是一样，这算什么用功。”慧海说：“不一样。”那人问：“怎么不一样？”慧海说：“他们思虑重重，吃饭时不肯吃饭，睡觉时不肯睡觉；我不然，我是该吃饭时吃饭，该睡觉时睡觉，这就是不同。”

——《读者》

6月4日

有时候不好分辨一个人是真的有水平还是只会晃荡，想起以前的一个老师说话，有几个我早就注意到的经典的技巧。

让人觉得自己厉害的话术（一）：

1. 谈起自己和某位“有位置”的人的对话、经历，可能只是很简单的对话、很简单的来往，但是一加上什么主任的头衔，就感觉高大上了一点，什么时候见到什么主任在干什么，然后说了是什么话，计划要干点什么，让人感觉讲话者和主任的位置是一样的，有来往的关系。
2. 谈话的时候引用“有位置”人的话，尽管这些话可能很简单甚至没有道理。“我说的大概就是这个意思，就像什么主任说的，xxxxxx”。一方面体现了讲话者和他人来往密集，有观点的交流，另一方面，引用别人的话不好反驳。
3. “制造”榜样，说自己以前教的某个学生现在如何如何厉害，借用他人能力来抬高自己，让人感觉教出来的学生都那么厉害，这个老师肯定更是厉害多了。
4. .....

买过很多次蓝牙耳机，一开始买的都是几十块钱的劣势的那种，几乎都能勉强用用，能听到声音，但也几乎都会存在各种各样的问题，左右音量大小不一样、有杂音、距离稍微远点就没有声音或者断开连接、容易被周围的磁场干扰等等，耳机的音质也很奇怪。后来因为无法忍受这些问题，就买了一个Redmi的运动蓝牙耳机，虽然也不贵，但那些小问题通通不见了。

用过之后，很难说Redmi的蓝牙耳机有多好，但明显能感觉到的是，同样很难说哪儿不好，起码没有那些糟心的问题了。所以其实好用不一定要有多好的体验，能够避开体验差的地方，本身就是一种好了。

以前用安卓手机，时间长会速度变慢是一方面，各种系统应用无法卸载、系统的操作界面上各种广告、权限无法控制，有的应用随便给你发通知，有的应用想收通知都收不到、屏幕亮度和色调……能够避开这些用户体验不好的地方，就算是一种好了，你可能说不上苹果比安卓好的地方，因为苹果可能确实没有比安卓好多少，只是少了非常多不好的地方。

6月5日

不会有两种，一种是学过了，没学会，另一种是没学过，或者还没来得及学。

解决不了问题也有两种，一种是遇到过，或者正在遇到，解决不了，另一种是还没有遇到过。

对于没学过和还没遇到过问题的情况，很难直接判断一个人的能力，所以就需要拿以往的一些事例来证实一个人有相应的能力，比如学会过什么东西，解决过什么问题。

按照这样的思路，学历是事例的一种，有过学历，说明你学会过一些东西，类比推断，你有能力学会其他的一些东西。

工作经验也是事例的一种，有过解决问题的经验，说明你能够解决一些问题，类比推断，遇到其他问题时，你同样可以解决，所以你能够胜任某些工作。

6月8日

PPT 演讲的主要内容有两种，一种是观点，对待某个问题的看法是怎样，原因是什么，为什么会有这样的观点，然后参会者有什么疑问或者不同想法，交流一下。另一种是案例，就是讲故事，背景、起因、过程、结果、总结，让人听故事，自己的事或别人的事，然后得出一个小结论，过渡到下一个观点或者另一个故事。

第三种内容是在 PPT 上写大段文字、理论、公式、技术方案，然后照着 PPT 读，跟讲课一样，指望能教会别人什么，这种做法显然是不可取的。

meetup 更像是一种技术交流会，大家平等的参与对话、交流技术问题，主讲者负责提出话题，其他人负责讨论。PPT 只是辅助主讲者表述话题的工具，是次要的，真正的价值还在人身上，看有没有观点，有没有案例，引出的话题是不是有足够的讨论性。

6月10日

最近睡眠不好有点头晕，精神恍惚，突然想起来以前的一个大学同学，疯了……起码是有一些精神上的问题的，可能那段时间压力大，也可能那段时间经历了一些失败的事情，也可能那段时间没睡好，成为了精神失常的导火索。在精神状态不好抵抗力薄弱的时候受到精神上的打击，可能很危险。

精神失常是一种很常见的现象，是在某些特定的场景下很容易陷入的迷沼。回想上初高中的不太光彩的日子，其实也是浑浑噩噩不清楚，做出来的事情缺乏理智，像未开化的原始人一样凭借原始的冲动说话做事。

《洛丽塔》、《穆赫兰道》、《Hello！树先生》、《小丑》这种精神分析片，基本上都有一个共同点是把真的和假的混在一起拍，作为观众无法直接区分，但是在有了分析影片的要是之后恍然大悟然后细思极恐。假的事情可以和真的一样，即使没有真的发生，你的意识认为发生了，就是发生了。所以我们是否也是患有精神疾病的人？我们经历的事情是否真的确实发生了？如果我们已经进入了精神世界的迷宫，该如何找到出口？

6月17日

前几天吃饭，想起来关于酒桌文化的问题，中国人喜欢喝酒，越醉越好。

为什么想要别人醉呢，因为想要别人出丑。  
可对方是客户、是朋友，为什么想要朋友出丑?  
因为大家其实都很丑陋。

有的人外表光鲜亮丽，金玉其外，但实际上大家都是人，都会有点破事，都半斤八两。  
喝酒，就是想让人把这丑陋但真实的一面表现出来。

你也表现，我也表现，互相知根知底，关系就变亲密了，生意就好谈了。

6月22日

走在路上听歌，发现一首歌有点好听，但也不是特别好听，歌是好歌，原唱是烟嗓男声，撕裂的感觉，这个翻唱是女声，有点软绵绵的，好像没有原唱那种爆发力了。纠结了很久，从歌一开始直到歌曲快结束，经过一系列痛苦的挣扎和思想斗争，终于决定还是把这首歌添加到收藏列表，点个小红心。打开手机，解锁，进入网易云音乐，点开播放界面，小红心亮闪闪的杵在那里，原来这首歌已经在收藏列表了。

什么样的歌算是喜欢的歌？列表里的歌随意播放，能够引起你注意的应该就是了。可能是某一段音乐、某几句歌词、某一种声音，或者听的时候想要知道这首歌是什么歌，或者……就是在你走神的时候能把你拉回来，让你把注意力放到歌曲本身上面。

所以，知道喜欢什么样的东西之后，就能够确保不会错过喜欢的东西，避免永远忘记这个本可能是你喜欢的东西，避免第二次遇到才明白自己喜欢的悲剧。

包括人。

6月24日

上周末和一个同学吃饭，颇有感慨。以前在博客上用一些篇幅提过这个同学的事情，如今我肯定不会主动拒绝别人的好意。这次见面，总的来说没有太多惊喜，想法还是以前的想法，做的事情基本上是意料之中的事情。有时候也想仔细分析一下他的人物性格，但又觉得不应该占用更多篇幅了。提几个点吧：

1. 感觉没必要过早“成熟”，想着赚钱、买房、结婚……反正我不喜欢。
2. 如果一个人能够经常而不是偶尔带给你启发，那么他的思维能力高过你。

3. 有时候觉得“难”，也许是消费超出了自己的承受能力，重点在于消费的起因，是自己的欲望、外人的压力或者别的什么。

6月26日

最近打游戏发现一件好玩的事情，暂且可以把它称为“神秘的鼓励的力量”。

每次开局后，我都会发一条全局消息：“我们家XXX很厉害，你们小心点”，一开始仅仅是恶作剧，把所谓的队友放在了自己的对立面，感觉自己和对面是一伙的，有一种莫名的快感，卧底、破坏、当坏人。但是后来慢慢发现，也许是错觉，每次我说厉害的人，战绩都不错。

我消息里说到的厉害的人有两种，一种是真厉害的，（当然我经常挑段位高的说，毕竟匹配到的人从青铜到星耀都有），另一种是真的菜的。真厉害那种，要么是自己好好玩了，要么是对面听到这话不服，故意好好玩针对他了。比较菜的，我发出消息后还会解释自己其实才刚刚开始玩游戏，战绩不好的甚至会愧疚，自嘲说自己比美团还能送。

还有刚刚在游戏里发生的一个情况，有人出了三双鞋（演员，不想好好玩的那种），我发全局消息说，“我们家妲己说不买装备也能赢你们，你们不要高兴的太早”（他自然没说这话），本来只是随便调侃一下，结果你猜怎么着？队友们都开始变得团结了，妲己也好好玩了，甚至还道歉了。也许是这话说的太悲壮了？或者……这话里体现出了一种莫名的友善和信任和……？

我有些感同身受，能够理解这种情况下的心理。虽然懒得分析这种行为本身，但是从这里出发想到了关于“鼓励”、“信任”之类的东西，或许有时候鼓励别人、信任别人的能力，要比命令别人、定个严厉的标准规范别人的行为效果好。当然也得分人。当然还有一个词叫“捧杀”。当然有当然……

6月28日

最后补充一条，以前屡次删除王者荣耀还有个原因，就是对人性的反复失望。希望这次卸载后不要再让历史重演。

# 构造函数为什么没有返回值?

2020-05-16

刚才同学问我一个问题， C++里new一个类的成员函数是什么意思？

.....我心想， new成员函数？还有这种操作？

后来我问， 这个成员函数， 名字是不是和类名一样？

他说一样， 就是构造函数。

.....那不就是new一个对象吗？

然后他问了一个深刻的问题， 为什么构造函数没有返回值？

我说new的是类， new后面指的是类名， 不是成员函数。

可是类没有参数啊？

构造函数有参数。

但是构造函数没有返回值啊？

啊？



这是一段简单的代码：

```
#include <iostream>
using namespace std;

class Test
{
public:
    Test(int num);
};

// 构造函数，有参数没有返回值
Test::Test(int num)
{
    cout << num << endl;
}

int main()
{
    // 对象赋值给了变量
    Test* test = new Test(1);
    return 0;
}
```

其实不看代码也能想到这个场景。new一个对象的时候， new后面的变量， 指的的类名， 还是方法名？

如果指的是类名，类本身没有参数，也没有返回值，而且new这个关键字在代码的行为上也会特殊于其他语句。另外，类会有一个默认的无参构造方法，如果new的是类，要默认的构造方法干嘛？

如果指的是函数名，构造函数没有返回值，像上面的代码里明显就把`new Test(1)`赋值给了`Test* test`。

构造函数为什么没有返回值？因为构造函数在执行的时候，告诉编译器在内存上开辟多大的空间，初始化了成员变量，确定了this的地址，然后干了这些事情之后，就不允许用户自定义返回值的类型了？因为构造函数的返回值一定且必须是它自己，所以就不需要对用户透明了（编译器为什么要“擅自”做这样的事情？）？

构造函数为什么没有返回值？我并不知道这个问题的答案，也不大有兴趣知道。但这个问题带给我很大的启发，也带给我很大的震撼，“new成员函数”这种说法真是思路清奇。我们对太多东西习以为常，司空见惯，觉得它应该就是那样，却很少问它为什么是那样，很少认真去思考和认识很多事物。

# 关于人的能力，经验，和.....

2020-05-03

最近好像没什么特别的事情，工作内容比起以前变多了，占据了很多时间，工作之外的空隙时间，似乎没有太多精力去关注学习。

这个博客上的内容，大概就那么两三种，一种是想到和学习到的技术方面的东西，再一种是工作和生活方面的事情，遇到的或者想到的引起了一些感悟的事情，第三种也许就是流水账、幻想之类，单纯做记录用，写下发生了什么事情。设想中博客的内容频率大概两周到一个月左右，时间太短没什么可说的，时间太长就有点长了。如果连续很长一段时间没什么可写或者不想写，就意味着发生了危险的事情，要么是松懈于动脑子了，要么是心理产生了不正常的问题。所以博客也可以作为自己的照妖镜。

最近纠结于的一点事情，问题在于很难得出什么结论。

(1)

前段时间周末的时候一个读研的同学问我一点比较基础的操作问题。一个C++的项目，从Github克隆下来，用VS2012打开后编译报错，说找不到文件。我很少接触C++，不过还是尽快了解了一下。项目叫 [wxCharts](#)，是一个图表的UI库，编译的时候几乎找不到所有的头文件。他的老师告诉他在常规里把外部依赖库加上，他就加啊加还是找不到头文件。

我说把找不到的文件随便挑一个，在磁盘里搜一下，搜不到.....

后来我看了一下wxCharts的官网，知道了wxCharts是依赖于 [wxWidgets](#) 的。wxWidgets是一个GUI库，wxCharts只是这个GUI库的样式组件。然后我本地测试了一下，编译安装wxWidgets后wxCharts就能编译过了。

原因找到了，但是他说他已经安装过wxWidgets了。我当时不太确定window下vs的生成和linux的make install是不是一回事，就说有那么几种方案，一是把wxWidgets下面的wx文件夹复制到wxCharts下（wx文件夹是项目默认的依赖文件夹，类似Go项目的src目录），二是把wxCharts的项目源码放到wxWidgets项目里试试能不能编译过，三是重新安装一下wxWidget，一定要全局安装，环境变量里配置一下。

后来折腾半天总算是能找到头文件了，外部依赖那里配置的路径有问题，用了错误的环境变量作为路径的变量，还重复引用了错误的路径。但是编译还不通过，找不到什么什么文件的，又折腾了一下，就是除了头文件还要引入动态依赖库，而且编译了生产版本的目录位置不一样，后面带的是debug版本.....

其实整个过程面对的问题很简单，就是找不到项目依赖，如果Java项目必定轻车熟路了（话说其实有工作好几年的同事面对项目依赖这种低级问题有时候还难以搞定的）。这是我一个关系不错的同学，本科时我们天天坐在床上打王者荣耀。他的实力我还是稍微有一些了解，可能学术能力比较强，但计算机方面真是比小白还小白。但要说学术能力多强其实本科成绩还没我好，也就是考研的一年多确实实实在在的努力了。

稍有感触的一点是，说起来现在他算是一所还不错的211大学的研究生，我的学历依然是双非本科，学历上差了好几个段位。至于能力的话，只能说曾经有一个考研的机会放在我面前，我没有珍惜，直到失去了也没有一丝丝悔意，如果上天给我一个重新来过的机会，我一定会做出和现在相同的决定，如果非要给这个决定加一个期限，我希望是，有生之年。

(2)

有一个我以前也提到过的让我感觉充满矛盾的同事，实际的动手能力和对工作的负责程度真是让我感觉理解不了。但是后来遇到几次问题，发现她毕竟还是有丰富开发经验的，经验方面真的难以反驳的超过我。

当时面对比较简单的一个递归的问题。我之前也确实一直没搞明白，一个对象传到一个函数里

面，是值还是引用传递？

```
function a() {
    obj = { "a": 1 }
    b(obj)
    console.log(obj)
}
function b(obj) {
    // 方式1
    obj = 1
    // 方式2
    obj = { "a": 2 }
    // 方式3
    obj.a = 2
}
```

必须先搞清楚这个问题，才能使用递归解决问题。方式1直接给对象赋值，方式2给了参数一个新的对象，方式3改变了对象的属性。我当时是没有概念的，一心想着是不是能用原型链解决这个问题。我也知道她不太清楚这个问题，但是她疑疑惑惑的提了一句这个和堆栈是不是有关系，然后又很不确定的没再提了。

我以为没关系，我以为她是随口说的。我错了。

对象是引用传递的，这一点没有问题，但对象的属性有没有改变，取决于改变的是对象的栈空间还是堆空间。给对象赋新值，不管是简单值还是复合值，都会将对象的变量指向新的栈地址，所以对象的属性并不会改变。如果直接用方式3的写法，改变的是对象指向的堆里面的内容，原对象的属性就会改变。

所以其实经验在一定程度上是有用并且难以轻易超越的，经验基本上无法通过捷径获得。

(3)

能力通过经验获得，强大的能力可以更快更有效的获取更高级的经验，然后这些经验会使能力更加强大。

也许能力和经验是相辅相成的，类比所谓的“经验和洞察力”。其实这里的“能力”确实稍微有点“洞察力”的意味。有时候在招聘的时候感觉招聘方一味强调多少多少年经验，好像用工作经验多的人就比用经验少的人占了便宜一样，但是吧，关于能力和经验……

这两天下了一个编曲的软件，简单尝试一下，感觉编曲这个事情，其实还是很难的，和编程虽然只有一字之差，技能要求却是截然不同而且更高的。现在各种培训班、在线课程，编程已经几乎没有门槛，当“全民编程”的时代来临后，我们改怎么面对这个世界呢。未来的编程也许不会把MVC、ORM什么的渗透到生活中，但一套物联网设备、自动化设备，或者iOS系统快捷指令的未来版本，如何更加适合人类个性化的生活习惯，可能多少也会需要点“编程”的逻辑。

PS:



iOS 13的快捷指令已经支持基本的语句逻辑

# 从Erlang开始了解Actor模型

2020-03-31

Actor Model是一个宽泛的概念，早在上个世纪就被提出来，它将Actor视作一个整体，可以是原子变量，也可以是一个实体，也可以代表一个线程，Actor之间相互通信，每个Actor都有自己的状态，在接收到其他Actor的消息后可以改变自己的状态，或者做一些其他事情。一般提到Actor，会用Erlang、Elixir或Akka来举例，它们都在一定程度上实现了Actor模型。

前端的MVVM框架React、Vue等都有各自的数据流管理框架，比如Redux和Vuex，这些数据流管理框架中有几个类似的概念，Action、Reducer、State之类，这些概念有时候会让人感到迷惑。现在前端变得越来越复杂，其中有一些东西可能是借鉴后端的，像TypeScript的类型系统。我好奇这些前端框架里的Action和后端的Actor模型在概念上是否有相似的地方。

其实Action的本质是简单的，甚至代码的原理也是简单的，reducer里面用switch判断不同的操作类型，去调不同的方法。最简化的形式就是一个方法Action改变了全局变量state的值。Redux文档里说它的设计来自Flux架构，Flux架构的来源暂时不得而知，但也不太可能说是受到了Actor模型的启发。

```
let state = null
function action(val) {
  state = val
}
```

Erlang是一门古老的编程语言，也是一门典型的受Actor Model启发的编程语言。单纯去理解概念是空泛的，从具体的、特定的语言入手也许能帮助我们探索这些理论。就像学习FP，选择Haskell要好过Java很多倍。Elixir是基于Erlang虚拟机的一门语言，与Erlang的关系类似Scala和Java的关系，也因此Erlang的语法相对简单和干净一点。

## Erlang

Erlang的代码块以.结尾，代码块可能只有一行，也可以有多行，.的作用类似于}，只是Erlang里没有{。代码块内的语句以.结尾，意味一个语句的结束，相当于一些语言的;。

Erlang将一个程序文件定义为一个模块，在命令行中使用c(test).可以加载模块。模块名称必须和文件名称一致：

```
-module(test).
```

文件头部需要定义程序export的函数，这是模块的出口：

```
-export([start/0, ping/3, pong/0]).
```

这里导出了3个函数，方括号和其他语言一样表示数组，函数名称后面的/0、/3指函数参数的个数。start函数将作为程序的主入口，负责启动整个程序，ping负责发送消息，pong负责接收消息并做出响应。

Erlang里面有个process的概念，它不是线程，也不是指计算机层面的进程，它就是process，或者也能把它当做线程，但是要明白它和线程不一样。我们将启动两个process，一个负责ping，一个负责pong，模拟消息的传输和交互。可以类比启动了两个线程，一个负责生产，一个负责消费。

```
ping(0, Pong_PID, StartTime) ->
  Pong_PID ! {finished, StartTime};
```

这是ping函数的第一部分，是ping函数的一个分支，接收3个参数，如果第一个参数是0，就会执行这个函数中的语句。第二个参数Pong\_PID指包含pong的process，第三个参数指程序启动的时间，用于记录程序的运行时长。函数体内只有一个语句，!是发送消息的意思，意为将数据{finished, StartTime}发送到id为Pong\_PID的process中，其中finished是一个Atom，作为标识

发送到pong那里。Atom是Erlang的数据类型之一，相当于……不需要声明的常量。

```
ping(N, Pong_PID, StartTime) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("~w~n", [N])
    end,
    ping(N - 1, Pong_PID, StartTime).
```

这是ping函数的第二部分，如果函数接收到的第一个参数不等于0，就会执行这个函数内的语句。这一部分函数在接收到请求后，首先会做和分支一同样的事情，就是把数据{ping, self()}发送给pong，区别在于这里的标识为ping而不是finished，pong那里会根据这个标识做不同的操作，至于第二个参数，self()会返回当前process的id，也就是把ping的id传给了pong，用以pong回复消息。pong会选择性的使用第二个参数。

把数据发送到pong之后，有一个receive ... end的代码段，这个代码段会阻塞当前程序的执行，直到当前process接收到数据。代码段里是一个简单的模式匹配，pong是一个Atom类型的变量，如果接收到pong这样的标识，就会执行->后面的语句。io:format是一个简单的格式化输出，把N的值打印到屏幕上。

receive结束之后，马上又调了一下ping自己，递归……直到N为0，也就是说ping和pong的交互会持续N次，io:format那里会把交互次数打印出来。这是ping函数的两个分支，pong函数和ping函数的程序类似：

```
pong() ->
    receive
        {finished, StartTime} ->
            io:format("The End");
            io:format(~w~n, [erlang:timestamp()]);
            io:format(~w~n, [StartTime]);
        {ping, Ping_PID} ->
            Ping_PID ! pong,
            pong()
    end.
```

pong函数在入参层面没有分支，但是receive里有两种匹配，如果接收到了结束标识finished，会把开始时间和结束时间都打印出来，然后程序结束。如果接收到的标识是ping而不是finished，首先给Ping\_PID也就是ping的过程一个pong的响应，然后调了一遍自己，相当于先发了一个消息出去，接着自己等待消息的回复，如果没有收到回复，它就一直等着。

```
start() ->
    Pong_PID = spawn(test, pong, []),
    spawn(test, ping, [10, Pong_PID, erlang:timestamp()]).
```

最后是start函数，程序的入口函数，spawn了两个process，这两个process分别单独地运行。当传入ping的第一个参数为10，ping和pong的交互将持续10次。

## 交互速率

以前听到过一个所谓的“大牛”讲，我们现在想要提高计算机的速率，瓶颈是什么呢，我们应该往哪个方向努力呢，应该是CPU的利用率，Actor是很快的，为什么快呢，因为一个Actor就是一个整体，一个Actor只在一个内核中运行，连CPU内核之间的交互都省了……这种说法的正确性可能有待验证，不过Actor是否真的快呢，我有点好奇，也因此萌生了测试一下Actor速度的想法。

必须要说明的是，我也相当清楚，这种测试方法很不靠谱。

在Erlang程序里启动两个process，两个process之间相互通信，测试不同数量级的通信次数，记录下程序执行所花费的时间。与Erlang作为对比，在Java里启动两个线程，用线程的睡眠和唤醒实现线程间的通信。同样的，在Go语言里用两个协程通信。至于Akka……其实也是Actor的代表。下表是测试之后的结果，次数从1到1亿，时间单位为毫秒。

次数	Erlang	Java	Go	Akka
1	0	0	0	3
10	0	1	0	7
100	3	4	1	17
1,000	26	30	4	83
10,000	610	168	42	225
100,000	2783	1295	404	674
1,000,000	27,085	11,300	4489	3515
10,000,000	273,912	107,673	40,335	29,368
100,000,000	2,851,680	1,092,879	482,196	300,228

本来尝试用Echarts之类渲染一下这些数据，方便对比，后来发现这些数据绘制出来的折线图并不友好。

总的来看，Erlang的速度是最慢的，这可能和Erlang历史悠久有关，也许是因为没有得到足够的优化，相信Elixir的速度会好一些。相较之下，Java的速度胜过Erlang，Go语言的速度胜过Java，这似乎是意料之中的事情。Java的耗时是Erlang的1/3，Go语言的耗时是Java的1/2。

最让人惊讶的在于，Akka的Actor速度竟然比Go语言的协程还要快。在交互1000次之前，Akka的速度比Erlang还要慢，在10K数量级的时候，它的速度超过了Erlang，在100K数量级的时候，速度超过了Java，直到1M数量级的时候，Akka超过了Go语言，并且一直保持领先。这是一个令人难以置信的结果，同样是运行在JVM上，Akka的耗时是Java的1/3，可能Java线程间的交互确实带来了很大的开销。

没有用Elixir做测试是一个遗憾。关于Akka为什快，和Actor模型有没有关系，有多大的关系，还需要进一步探索。

(The End)

## Akka

用来做测试的Akka程序是Akka官方的Hello Wrold程序，能看到明显的Actor模型的影子，尤其是!运算符和receive方法。

```
import akka.actor.typed.ActorRef
import akka.actor.typed.ActorSystem
import akka.actor.typed.Behavior
import akka.actor.typed.scaladsl.Behaviors
import GreeterMain.SayHello
```

这是导入部分，如果使用VS Code之类的编辑器，这段代码还是很重要的。和Erlang的程序类似，有一个发消息的Greeter和一个接收并回复消息的GreeterBot，另外还有一个主方法。

```
object Greeter {
    final case class Greet(whom: String, replyTo: ActorRef[Greeted])
    final case class Greeted(whom: String, from: ActorRef[Greet])

    def apply(): Behavior[Greet] =
        Behaviors.receive { (context, message) =>
            message.replyTo ! Greeted(message.whom, context.self)
            Behaviors.same
        }
}
```

这是发消息的Greeter，当Greeter作为函数被调用，会自动执行apply中的代码。apply方法是一个receive，和Erlang的receive一样会阻塞程序直到Actor接收到消息。replyTo是GreeterBot的”pid”，Greeter接收到消息后会回复消息给GreeterBot。

```

object GreeterBot {
    var startTime = System.currentTimeMillis()

    def apply(max: Int) = {
        bot(0, max)
    }

    private def bot(greetingCounter: Int, max: Int): Behavior[Greeter.Greeted] =
        Behaviors.receive { (context, message) =>
            val n = greetingCounter + 1
            context.log.info("{}", n)
            if (n >= max) {
                context.log.info("The End | {}", System.currentTimeMillis() - startTime)
                Behaviors.stopped
            } else {
                message.from ! Greeter.Greet(message.whom, context.self)
                bot(n, max)
            }
        }
}

```

这是GreeterBot，和Erlang简洁的代码比起来，Scala冗长的类型声明可能显得有些.....烦杂。GreeterBot接收到来自Greeter的消息后，判断n是否为max，如果已经执行够次数了，就停止，否则调用自己进行递归。

```

object GreeterMain {

    final case class SayHello(name: String)

    def apply(): Behavior[SayHello] =
        Behaviors.setup { context =>
            val greeter = context.spawn(Greeter(), "greeter")

            Behaviors.receiveMessage { message =>
                val replyTo = context.spawn(GreeterBot(max = 10), message.name)
                greeter ! Greeter.Greet(message.name, replyTo)
                Behaviors.same
            }
        }
}

object AkkaQuickstart extends App {
    val greeterMain = ActorSystem(GreeterMain(), "AkkaQuickStart")
    greeterMain ! SayHello("Charles")
}

```

最后是主方法，看着可能也有点.....长。继承于App的类是能够运行的主类，向Actor系统中注册了GreeterMain，同时GreeterMain的apply方法被执行了一次。GreeterMain里spawn了两个process，和Erlang的程序行为是类似的。

## Go

Go语言的程序真的要简洁很多，这是程序头部：

```

package main

import (
    "fmt"
    "time"
)

var maxCount = 100000000
var startTime = time.Now().UnixNano() / 1e6

```

定义了两个变量，一个是程序执行次数，一个是程序开始时间。

```

func main() {
    ch := make(chan bool)
    exit := make(chan bool)

    go func() {
        for i := 0; i < maxCount; i++ {
            fmt.Println(i)
            <- ch
            ch <- true
        }
    }()
}

go func() {
    defer func() {
        timeUsed := time.Now().UnixNano() / 1e6 - startTime
        fmt.Println("The End | ", timeUsed)
        close(ch)
        close(exit)
    }()
    for i := 0; i < maxCount; i++ {
        ch <- true
        <- ch
    }
}()

<- exit
}

```

两个协程，从channel中取数据和向channel中写数据交替。Go语言的程序看着清爽太多了，Scala扎眼睛。

## Java

Java的冗长程度不比Scala轻。

```

public class Test{
    public static void main(String[] args) {
        Object lock = new Object();
        Thread sender = new Sender(lock);
        Thread receiver = new Receiver(lock);
        sender.start();
        receiver.start();
    }
}

```

主方法里启动了两个线程，锁是共享资源。

```

class Message {
    static long MAX_COUNT = 100000000;
    static String status = new String("init");
    static long count = 0;
    static long startTime = 0;
    public static void send() {
        System.out.println(count);
        status = "sent";
        count++;
        if (count == 1) {
            startTime = System.currentTimeMillis();
        }
        if (count >= MAX_COUNT) {
            status = "stop";
            long time = System.currentTimeMillis() - startTime;
            System.out.println("The End | " + time);
        }
    }
    public static void receive() {
        status = "received";
    }
}

```

```

    }
    public static String getStatus() {
        return status;
    }
}

Message是临界资源，储存消息的内容。消息内容变更时做了一点其他的事情，把需要的日志打印到屏幕上。

```

```

class Sender extends Thread {
    Object lock = null;
    public Sender(Object lock) {
        this.lock = lock;
    }
    @Override
    public void run() {
        while (!Message.getStatus().equals("stop")) {
            synchronized (lock) {
                if (Message.getStatus().equals("init")
                    || Message.getStatus().equals("received")) {
                    Message.send();
                    lock.notify();
                    try {
                        lock.wait();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}

class Receiver extends Thread {
    Object lock = null;
    public Receiver(Object lock) {
        this.lock = lock;
    }
    @Override
    public void run() {
        while (!Message.getStatus().equals("stop")) {
            synchronized (lock) {
                if (Message.getStatus().equals("sent")) {
                    Message.receive();
                    lock.notify();
                    try {
                        lock.wait();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}

```

Sender和Receiver的程序类似，Sender先发送消息，然后wait，等着接收Receiver的消息，Receiver用while不停地判断有没有收到消息，如果有则回复消息，并且唤醒Sender，通知它该处理消息了，叫醒Sender后自己wait，等着Sender的反馈。

# 一种侧边导航栏的交互方式

2020-03-21

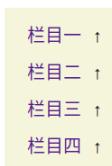
最近看到几个管理系统的演示项目，结合开发过程中不顺手的地方，发现大多数网站的侧边导航栏都是点击展开，点击关闭。



感觉这样的交互方式稍微有点繁琐：

- 在不知道子菜单位置的情况下，需要一个一个点开才能找到需要的页面
- 在知道子菜单位置的情况下，也需要点击一下父级菜单，才能看到想要的子菜单
- 不一个一个点开，就无法知道子菜单有些什么
- 子菜单展开之后，需要一次一次点击父菜单才能收起

后来就想，能不能把点击事件换成悬浮事件呢？只要鼠标放上去，菜单就会自动展开，不用点一下的操作了。但是单纯的悬浮展开，需要考虑菜单长度不一致的问题，如果下一个菜单的长度比当前菜单短，鼠标离开当前菜单，当前菜单收回，鼠标所在的位置会直接越过下一长度较短的菜单。



像图片中这样，栏目二的长度是4，栏目三的长度是2，当鼠标从栏目二向下移动，离开栏目二的瞬间栏目二收回内容，鼠标在没有移动的情况下跳过了栏目三，悬浮在栏目四上，这其实是不合理的，会违背用户的预期。栏目二之后是栏目三，这是最正常的逻辑。

为了应对这一问题，也许可以将交互设计成这样，当鼠标离开栏目二后，栏目二不收回，直到鼠标离开整个导航栏，子菜单才自动折叠。如果子菜单展开时用户点击了某一父菜单，那这个父菜单即使鼠标离开导航栏也不收回。

栏目一 ↑  
栏目二 ↑  
栏目三 ↑  
栏目四 ↑

下面是一个demo页面，通过iframe嵌入到这里，可以对比两种侧边导航栏的交互方式（移动端没有鼠标悬浮事件）。我偏爱灵活一点的交互，第二种方式单击父菜单也可以展开收起列表，相当于在方式一的基础上加入了鼠标悬浮自动展开的能力。



相较于鼠标悬浮自动展开不收回的方式，更进阶一点的做法是，当鼠标从上往下移动时，子菜单

自动展开但不收回，当鼠标从下往上移动时，子菜单自动展开并且自动收回。因为子菜单要不要自动收回取决于对用户接下来的操作有没有影响。不过这样的效果实现起来有些复杂了，对于网页上的一个导航栏来说，需要不断监听鼠标的坐标，开发和维护的成本有点高。

# 新的征程

2020-03-08

我犹豫了很久，要不要把这些东西写下来，应该什么时候写，现在写合不合适……如果我无法在自己的独立博客上写自己想写的东西，我就被生活上的事情束缚了，博客也变的没有意义。我总是担心，如果我的同事、领导偶然看见我在博客上写想要离开公司，或者原来早就产生离开公司的想法，对公司有什么不满之类；或者担心招聘者会偶然看到我对公司的不忠诚，看到我说公司的坏话，会不会对新的公司也产生一大堆意见，毕竟我总是把网址写在简历中。

也可能世上本无事，我杞人忧天了。

## 期望

我期望的工作，可能需要一点实力，也需要一点运气。当HR问我想要跳槽的原因，我一开始的说辞是想要寻找更好的机会，不是现在的公司不好，而是想要更好。具体说就是想要找更好的发展前景，技术上或者业务上能够有更多的成长……直到HR问我，你现在的平台机会不多，前景不好，工作不饱满吗？

我重新思考了这个问题，关于为什么想要跳槽。我没有撒谎，现在的公司确实还好，没什么能说得上不好的地方，当然也说不上多好。我想了想，真正的原因应该是我作为开发人员，作为技术支持这样的一个角色，对于目前公司的情况来说，难堪大用。公司更需要的是能拉客户的销售，能写专利的技术研究员（投标的时候要指标），而不是能把脏活干好的熟练的工人。

我有算不上丰富的开发经验，从还有静态网站和动态网站之分的那个年代，接触到互联网之后对广阔的网络世界充满兴趣。从三极管到模拟电路、数字电路、集成芯片、单片机、汇编语言、树莓派、C语言、Java、JavaScript、MVC、MVVM、FP……除了网站开发，稍微知道一点安卓的玩法、用H5开发原生应用的混合开发、基于Node.js的桌面应用开发……大学读电子类专业并没有阻碍我对计算机的了解，反而知道了一些可能计算机专业不会学的电子电路、通信编码之类。其实我真正懂得东西很少，只是稍微知道一点概念，但内心深处总是充满力量，有一种莫名的热情，让我感觉自以为能够解决更多问题，事实上我也确实希望能有更多机会解决更多的问题，希望在技术上有长足的发展。

有时候会感觉世界的不公，我无法想象不会处理SVN冲突、搞不清二维数组的前端，没用过Docker、不懂RBAC权限模型的后端，不知道String.valueOf()和(String)区别、可以直接往JSONObject里传Map的Java开发，是合格的程序员。我可以解决同事无法解决的技术问题，有超过同事的技术广度，甚至有更多努力工作的意愿，但仅仅因为我的工作年限较短，别人的工作时间较长，我只能拿与我工作经验相当的工资。实际上我接触Web开发的时间可能比某些小作坊的面试官还要长（真的遇到过）。也许我的同事都是幸运者，在互联网还未退潮的时候上了船。

我并不是想说我是自大狂，我很清楚自己没有足够的实力，不懂JVM调优、背不下设计模式、没看过Spring或Redis的源码、不懂操作系统原理（看过两本书，抄过一本，没记住），不懂算法（见medium死），没有好的学历（双非）。我也不知道我会点什么。但是当面试官问我ECharts的用法、“事务的隔离级别”、“死锁产生的条件”、“Map的原理”，虽然有点卡壳……我同样有些犹豫，在百度用中文搜的第一条就能找到答案的问题，需不需要记下来呢，世界上的问题那么多，知识那么多。

## 面向面试学习

对于需不需要记下来的问题，我希望自然而然的解决。就像做开发，踩的坑多了，就成了资深程序员，看的报错日志多了，就能瞬间明白异常原因。我可以多面几个试，在面试过程中了解公司会问哪些问题，期望怎么样的回答，不断积累经验，顺便锻炼一下语言表达能力，在面试结束后弥补不足，补充知识。正好最近的疫情提供了绝佳的机会，几乎所有公司都无法现场面试，可以通过电话、视频面试，我在公司连假都不用请，除了稍微在心态上有点影响工作外，但保证不会落下进度。

到这里我又担心如果新的公司看到这些话，会不会认为我面试是刷经验去了，毕竟我真的会在简历上写下网址，大多数情况不会有人看，但万一呢……

目前对我来说，发展前景比工资重要太多了。所以最最最理想的情况，是进一家能叫得上名字的公司。折中一点的情况，是公司招牌和目前的公司相当，但是工资比现在高一点，底线是两千，试用期加上跳槽离职入职工作交接之类成本，两千应该不夸张。再差一点的情况，再差一点，就是我被公司裁了，不得不找其他工作，又没有合适的机会，随便找了一家保证生存。

## 最后的稻草

我想说说目前的公司。当时在学校还是秋招阶段的时候，HR就跟我说，应届生两年后考虑涨工资。其实我从一开始就明白，我不可能等两年。开始实习后，带我的导师告诉我，他们一年涨一次。

所以在我被迫换部门后（当时公司效益不好，裁了很多人，未入职的应届生走了一半），我想着一定要等过个年，心里还在期望和等待的有两件事，一个是年终奖，一个是涨工资（普调）。倒不是在意能拿到手多少钱，而是想要了解一下公司的运作方式，绩效怎么回事，年终奖怎么发，是否和绩效挂钩，涨工资是怎么谈或者怎么样，还幻想着领导会不会谈个话，聊聊工作情况，表现怎样，未来计划，然后我应该说些什么，怎么表现，让领导看到我的能力。没有经历过那些，就想经历一下。

事实当然是令人失望的。我了解到部门内有的人三四年都没有涨过工资，暂且接受，但是到过年放假前的最后一天，我感到异常生气，因为没有年终奖。往年都有，就今年没有，这一点也能理解，公司效益不好。问题是公司当天把本季度的绩效工资发了下来（我们下季度发本季度绩效），这不是公司良心发现要发当前季度绩效了，如果想发，就应该和当月工资一起发。我当时瞬间想到的是“朝三暮四”这个词，公司把我们当猴养，我们在人格和尊严上都应感受到侮辱，我无法继续忍受这种任人宰割的状况，没有公告，没有制度，没有任何形式的通知，关于工资、普调、年终奖，我希望知道规则，我希望知道该如何努力，该如何表现，问题是根本没有，全部没有，全看领导心情。我无法凭自己的努力改善自己的收入，这就是我在目前公司看不到希望的地方。我甚至都不明白我的直属领导到底是谁，到现在HR系统里还指向一个已经离职的名字，他什么时候来和我谈一谈工资的事情呢？

## 区块链不是风口

最近几天初次知道了“共有知识”和“公共知识”的区别，重新明白了一下《皇帝的新装》里面的道理。提区块链是因为我目前属于区块链团队，公司也把一部分资源押在了区块链上。

我知道区块链没什么用。  
你知道区块链没什么用。  
他知道区块链没什么用。

这是共有知识。

我知道 你和他都知道 区块链没什么用。  
你知道 我和他都知道 区块链没什么用。  
他知道 我和你都知道 区块链没什么用。

这是公共知识。

大家都知道，但不清楚别人知不知道的，叫共有知识。当有人跨越了中间的沟壑，告诉所有人他们的想法是一致的，然后大家都知道了大家的想法是一致的，共有知识就变成了公共知识。目前的区块链就处于共有知识的阶段，因为很多人都说区块链怎样怎样，你无法不认可它的价值，如果说你说区块链没有价值，那就是你不懂，你没有智慧，你见识浅薄，你……连位置最高的管理者都发话了，你能说你看不到区块链的价值吗？

和《皇帝的新装》的故事对比一下，是不是有一些相似之处？

## 新的征程

所以我真的犹豫了很久，要不要写下这些东西，关于当前的公司，关于新的目标，从年前放假的最后一天，直到现在。

我需要新的机会。

# Java项目开发中的3个小问题

2020-02-26

## 1. mysql需要id自增主键吗

有些时候我们会使用具有具体含义的字段作为主键，比如用用户名username而不是无意义的自增id。这样做的好处之一是可以利用主键不可重复的特性，保证username的唯一。如果试图写入重复的数据，数据库会抛出“Duplicate entry \* for key PRIMARY”的错误。

利用这个“好处”的问题是，我们能否依赖以及是否需要依赖主键的默认特性（字段不重复）来实现我们业务上的需求（字段不重复），而事实上“不重复”是惟一索引的特性，并不是主键的特性。主键字段会自动加上惟一索引。

同时，利用这个“好处”的不便之处之一是，数据表缺少一个表示“数据行序列”的字段。oracle还有rownum这样的东西，mysql则没有，得用脚本实现。如果需要分段遍历表的话，没有行的概念就很难操作。

《阿里巴巴Java开发手册》明确要求表必备id字段，他们不一定是对的，但我相信他们的理由一定足够充分。

## 2. Map还是Bean

用mybatis查询数据时，返回值类型一种是List，一种是List。这里的Bean指数据对象（DO）。

使用Bean的好处显而易见，输入对象名再按下一个点，idea会自动弹出bean的getter和setter，表包含的属性一目了然。无论取值还是赋值，都对使用者友好。

使用Bean有可能存在的问题之一是，如果表结构有改动，比如删除一个字段，Bean类中对应删除了一个属性，同时删除了相应的getter和setter，那么只要是程序中用到了这个bean这个字段的地方，都需要逐一修改，如果不改，程序将无法通过编译。

我们经常在开发过程中遇到这种情况，调试A类，改了某个公用的COM类，编译的时候BCDEF...都报错了，这很让人抓狂，我只想要也迫切的只需要确认A类的问题是不是由COM类引起，却由于这种依赖关系需要尝试其他方法。

如果面临系统改造、整合、重构之类的问题，Bean也会暴露出同样的问题。我在最近接触的项目中，为了尽可能多复用原系统，尽可能少做改动，只好用含义不太相同的Bean.username储存实际上内容为Bean.phoneNumber的值。原系统的各种工具类（加解密、本地缓存之类）全都依赖Bean，涉及到Bean的改动之后，要么大批量的全改，要么全不改。

如果使用的是Map，Bean带来的问题可能会得到稍微的缓解。尤其是在合理处理取值操作后，比如`String.valueOf(map.get(""))`可以避免伟大的空指针异常，程序取值为空也可以正常启动，需要变动的地方随心所欲的get、set，不再受Bean类getter、setter方法的限制。

另外，其实Map和Bean都是对属性的一种封装。

## 3. 注解还是xml

Spring的IOC支持注解也支持xml，mybatis的sql同样支持注解也支持xml。

在以前的时代，Java项目往往会打包成war或者jar，放在容器tomcat或者resin中运行。容器在运行的时候会将包自动解压成class文件和资源文件，资源文件就包括xml。使用xml配置的好处之一是，线上环境可以直接修改xml重启项目就完成操作，不需要再把java文件编译成class文件，打包解压替换，走一遍项目上线的流程，拉分支、开发、测试、编译、部署、合并基线，一步步发邮件。

我想现在时代变了，有持续集成，有自动化运维，有properties，有yml，xml似乎显得不是那么重要。

对于mybatis的sql来说，内部有一个xml的解析器，xml相当于一种dsl，通过配置来实现动态sql语句之类的需求。如果使用注解的方式，把sql写在provider里，我们就失去了xml解析器的功能。不过失去能力的同时，我们也会获得自由。面对字符串形式的sql语句，我们完全可以自己封装一个parser。

一个有趣的现象是，国外的开发者似乎更喜欢注解的方式，国内的开发者倾向于使用xml，国外的书籍很多作者使用注解做案例，国内的博客文章则大多使用xml配置做教程。这一点有些类似于前端框架react和vue的状况，技术实力较强的企业、开发者更加青睐react，因为jsx更像是编程语言，有更多的发挥空间，vue则受到技术实力较弱的企业和开发者喜爱，因为上手简单，开发简单。如果你用过vue就应该有体会，用那玩意儿开发不叫编程，它只是软件工业化的产物。

# 程序员的定位

2020-01-11

有一件可怕的事情是，程序员不再关注代码、软件、系统、架构，而是关注产品、运营、销售，展望内容不再是提高代码质量、迭代更好的软件、做更好的工程，而是提高文档输出能力、活跃团队气氛。

## startup与996

公司开始倡导创业心态，创业嘛，大家就要一起努力，一起干大事。BOSS说，有天回公司晚了，看到某某和某某10点多还在……大家也要这样，我们也只能这样做，如果因为身体或者其他原因不能适合我们这种节奏了，可能就不太适合我们这个团队了。大家要一起努力，要10106，我们只能这样做，如果不这样做……公司就会死。

当时听到这儿，哑然失笑。说句不太符合价值观的话，我明知道这样说不对，如果老板看到这样的想法，我明天就得和HR聊聊了，这是没有责任心的体现，这是没有职业道德的体现，这是……恕我直言，公司死掉，于我何干？

画饼啊，平时都听人吐槽，老板给画个大饼，入职后发现遥遥无期，然后上网发帖，不要相信老板的鬼话，不要相信什么福利奖金，拿到手的才是真的。我们这画风好像不太一样啊，如果不努力，公司会死掉？以前看到过一句话，所有人都告诉我们要努力，但是从来没有人告诉我们努力之后可以得到什么。公司死掉了，我们换家公司不也照样干活吗？

年会的问答环节上，大BOSS抓住了“享受红利”这个字眼，大意是红利不是公司给予的，要给公司带来相应的价值。猜想很多时候，公司的想法是，你要先给我干20分的活，我就分你10分的利。员工的想法是，你给我10分的钱，我就干好10分的活，你要再多给我5分，我就再多尽5分力。

996是类似的问题，我们为什么是10106？因为我们9点半上班，坐到工位上打开电脑、开个晨会大概就10点了，所以是1010，而不是99。关于996，一种说法是有些员工挣表现的心理，从小被奖励小红花的培育文化，造就了认为“只要努力就会有收获”的心态。不过很多老板也确实吃这一套。

昨天我听到了另一种说法。我们BOSS虽然说出了10106的话，但是并不会发正式的制度通知出来，因为这违反劳动法。我对此感到疑惑，这就不好办了呀，到底要不要10106？请教稍有经验的员工，得到的答案是，老板喜欢“聪明”的员工。也就是说，老板提到这个事了，具体怎么得自己掂量。

## 加班

有时候项目紧、突发bug、上线、工作失误，偶尔加班很正常，有忙有闲。有的公司要求经常性加班，有干不完的活，也就是所谓996工作制。一般来说965是一个数，996是一个数，对应的薪资水平不同。产生矛盾的点在于，公司希望用965的薪资招996的人，员工希望领996的薪水干965的活。

存在一种情况是老板没有要求加班，但是有人加班。每个人分工不同，任务量不同，完成速度不同，要看具体情况。如果自己效率低于别人，加班无可厚非。比较坑的是别人因为效率不高整天加班，你完成任务没有加班，老板就在工作时间这件事上做文章。你如果和别人一样的工作时间，是不是就能产出更多了？别人没完成的也可以帮帮忙完成一下……相信明智的领导能够分辨出真正的工作能力，也能够按照工作量而不是工作时间判断产出。不过最可怕的是揣着明白装糊涂。

另一种情况是老板要求多呆一会儿。可能没有那么紧急的工作需要加班才能完成，但是别人加班了，为什么你到点就走？所有人都加班，就你特殊？……老板为什么会想要多呆一会儿呢，加班总归要多干点活吧，应该是植根于中国劳动人民群众心中占小便宜的心理。

其实996对工作产出的影响未必是正面的。如果到点大家都下班了，我是不好意思长留加班的（认真），别人都能完成工作，你需要加班完成，是不是你能力不行？所以作品内容会尽量在上班时间紧凑的完成。如果每天需要加班呢，反而会感觉上班稍微拖沓一点也没事，反正要加班，反正到点也走不了，上班时间摸摸鱼，没完成的好好加一把班，还能给老板留个好印象。

## 技术人

公司开始倡导“销售导向”，因为销售是能够直接决定收益的一环，也是老板比较重视的部分。述职会上，我第一次从销售和运营的口中了解到公司的全貌，也是在述职会中，BOSS提到，“像了解机器学习常用的模型和算法是一些很具体的、很普通的事情，我们要……”，那正是我几天前的述职材料中提到的内容。

其实技术不重要，尤其是销售导向的公司，用到了什么技术真的不重要，重要的是把客户谈下来，把客户忽悠住，给公司挣钱。放在整个社会中，技术要足够领先、足够新，才能有立足的可能，像机器学习这样已经谈了十几年的技术，了解常见的用法确实很普通、很平庸。尤其是并不能带来什么收益。

从实行OKR开始，我们的绩效计划有两个特点，一是要有目标有结果，二是这个目标和结果不是作品内容方面的，而是自我提升方面的。自我提升主要指输出，就是写内刊、写专利、分享技术等，尤其是要能够给工作带来帮助的。讽刺的是，我们几乎没有技术氛围，真正写代码的只有几个骨干。从绩效管理能体现出的是，领导不太在意工作方面的内容，比如建立项目规范和开发规范，需求、开发、评审、测试之类的，领导更在意个人方面的东西。

可能很多时候的情况是，公司在意的是“你能给公司带来什么？”员工想的是“公司需要我做些什么？”

述职也是一样，不得不写一些工作之外，但又和工作相关的内容。深度学习本身是一个值得学习和研究的技术领域，即使是常见的模型和算法，从原理到工程至少也要1年的时间才能有一定了解。另外我目前的工作内容是应用层开发，也就是增删改查的业务之类，部门的技术主题是区块链，所以就把机器学习放在了展望一栏。

个人而言，机器学习是很大的一步。它本身有一定门槛，以前也多次尝试，看吴恩达的教程放弃了一次，看斯坦福的图像识别课放弃了一次，……有观点说机器学习是微积分在现代计算机技术背景下的发展，它的基础理论是数学，同时观点也提到，区块链是小骗，AI是大骗……不管骗不骗，总归还是要学的。

## 程序员

和销售或者运营相比，程序员的特点之一可能是不那么依赖外部资源，人脉、市场、监管、业务，技术在形式上千变万化，本质上都是操作计算机进行计算。应用层开发的程序员，本职工作应该是按照进度要求完成开发任务，职业素养是保证代码质量，需要提升的是系统架构能力。熟练的Coder可以高效使用工具链，优秀的Coder可以优化算法、提高程序效率和系统性能。

对不起，I am a Coder，我认为不丢人。（@[不要自称为程序员](#)）

# Rust的ownership是什么？

2019-12-21

Rust是内存安全的。Facebook的Libra使用Rust开发，并推出了新的编程语言Move。Move最大的特性是将数字资产作为资源（Resource）进行管理，资源的含义是只能够移动，无法复制，就像纸币一样，以此来保证数字资产的安全。其实Move的这种思想并不是独创的，Rust早已使用这样的方式来管理内存，因此Rust是内存安全的。Rust中的内存由ownership系统进行管理。

## Java的引用计数

垃圾回收有很多种方式，ownership是其中之一。Java使用的是引用计数，引用计数法有一个广为人知的缺陷，无法回收循环引用涉及到的内存空间。引用计数的基本规则是，每次对内存的引用都会触发计数加一，比如实例化对象，将对象赋值给另一个变量，等。当变量引用被取消，对应的计数就减一，直到引用计数为0，才释放空间。

```
class Test {  
    Test ref = null;  
}  
  
Test a = new Test(); // a的计数加一  
Test b = new Test(); // b的计数加一  
// 此时a的计数是1, b的计数是1  
  
a.ref = b;           // a的计数加一, 因为ref是a的类变量  
b.ref = a;           // b的计数加一, 因为ref是b的类变量  
// 此时a的计数是2, b的计数是2  
  
a = null;            // a的计数减一, 因为a的引用被释放  
b = null;            // b的计数减一, 因为b的引用被释放  
// 此时a的计数是1, b的计数是1
```

因此，在a和b的引用被释放时，它们的计数仍然为1。想要a.ref的计数减一，就要将a.ref指向null，需要手动操作指定为null吗？当然不需要，Java从来没有手动释放内存空间的说法。一般情况下，a.ref执行的对象也就是b的空间被释放（计数为0）时，a.ref的计数也会自动减一，变成0，但此时因为发生了循环引用，b需要a的计数变为0，b的计数才能变成0，可a要想变成0，需要b先变成0。相当于死锁。

这和Rust的ownership有关系吗？当然，没有关系……

## ownership

ownership有三条基本规则：

- 每个值都拥有一个变量owner
- 同一时间只能有一个owner存在
- 当owner离开作用域，值的内存空间会被释放

作用域多数情况由{}界定，和常规的作用域是一样的概念。

```
{           // s还没有声明  
    let s = "hello"; // s是可用的  
}           // s已经离开作用域
```

Rust的变量类型分简单类型和复杂类型，相当于普通变量和引用变量，因为ownership的存在，简单类型发生赋值操作时，值是被复制了一份的，但复杂类型是将引用直接重置到新的引用变量上，原先的变量将不可用。

```
let x = 5;  
let y = x;           // y是5, x还是5
```

```
let s1 = String::from("smallyu");
let s2 = s1; // s2是"smallyu", s1已经不可用
```

赋值过程中，s2的指针先指向string，然后s1的指针被置空，这也就是移动（Move）的理念。如果想要s1仍然可用，需要使用clone复制一份数据到s2，而不是改变指针的指向。

```
let s1 = String::from("smallyu");
let s2 = s1.clone(); // s1仍然可用
```

## 函数

目前提到的有两个概念，一是ownership在离开作用域后会释放内存空间，二是复杂类型的变量以移动的方式在程序中传递。结合这两个特点，会发生这样的情况：

```
fn main() {
    let s = String::from("smallyu");
    takes(s); // s被传递到takes函数
    // takes执行结束后, s已经被释放
    println!("{}", s); // s不可用, 程序报错
}
fn takes(s: String) { // s进入作用域
    println!("{}", s); // s正常输出
} // s离开作用域, 内存空间被释放
```

如果把s赋值为简单类型，比如5，就不会发生这种情况。对于复杂类型的变量，一旦离开作用域空间就会释放，这一点是强制的，因此目前可以使用函数的返回值来处理这种情况：

```
fn main() {
    let s = String::from("smallyu");
    let s2 = takes(s);
    println!("{}", s2);
}
fn takes(s: String) -> String {
    println!("{}", s);
    s
}
```

takes把变量原封不动的返回了，但是需要一个变量接住takes返回的值，这里重新声明一个变量s2的原因是，s是不可变变量。

## 引用变量

引用变量不会触发ownership的drop方法，也就是引用变量在离开作用域后，内存空间不会被回收：

```
fn main() {
    let s = String::from("smallyu");
    takes(&s);

    println!("{}", s);
}

fn takes(s: &String) {
    println!("{}", s);
}
```

## 可变变量

引用变量仅属于可读的状态，在takes中，s可以被访问，但无法修改，比如重新赋值。可变变量可以解决这样的问题：

```
fn main() {
```

```
let mut s = String::from("smallyu");
takes(&mut s);

    println!("{}", s);
}

fn takes(s: &mut String) {
    s.push_str(", aha!");
}
```

可变变量也存在限制，同一个可变变量同一时间只能被一个其他变量引用：

```
let mut s = String::from("smallyu");
let r1 = &mut s;
let r2 = &mut s;
println!("{}, {}", r1, r2);
```

程序会报错，这是容易理解的，为了保证内存安全，一个变量只能存在一个可变的入口。如果r1和r2同时有权力更改s的值，将引起混乱。也因此，如果是r1 = &s而不是r1 = &mut s，程序会没有问题，只能存在一个引用针对的是可变变量的引用变量。

## 返回值

函数的返回值类型不可以是引用类型，这同样和ownership的规则有关，返回普通变量相当于把函数里面的东西扔了出来，如果返回引用变量，引用变量指向的是函数里面的东西，但函数一旦执行结束就会销毁内部的一切，所以引用变量已经无法引用到函数。

```
fn dangle() -> &String {
    let s = String::from("smallyu");
    &s;
} // 到这里s的内容空间已经释放，返回值无法引用到这里
```

?

没有更多内容了。

最近看了一部能够让人振奋的美剧《硅谷》，编剧给主角挖了很多坑，感觉他们倒霉都是自己作的，编剧也给观众留了很多坑，剧情跌宕起伏到想给编剧寄刀片。抛开那些情节，剧中渲染的geek真的很帅，很帅！当然，神仙打架，凡人也参与不了。

# 我亲手写出了难以维护的代码

2019-12-15

前段时间尝试了一些线下活动，加上有几天身体异常痛苦，现在稍微静下来感觉恍如隔世。最近有个项目在做最后的验收，于是搬出了一两个月前的代码，需要手动操作更新一些数据。看到这些不久之前写的代码，同样恍如隔世的感觉，如果这是别人写的，我一定会第一时间认为很垃圾，既然是我亲手写的，客观的说，它确实很垃圾。

那是一个很好记的日子，从那天开始，2019年才真正开始。我曾经接触的是老系统的代码，不得不在resin里启动项目，编译一次至少要两分钟，难以调试，曾经在测试服务器上开发和调试程序，而不是发布代码，曾经写过一些小任务，比如批量执行sql还踩了坑。我曾经看不起垃圾代码，看不起9102年还在用eclipse的程序员。

时至今日，我也变成了写出垃圾代码的程序员。

## 小需求

第一次写出垃圾代码，印象比较深的是打印两次日志：

```
catch (e) {
    System.out.println("捕获到异常");
    logger.error("捕获到异常");
}
```

logger出现了多少次，println就出现了多少次，这显然是丑陋的做法。因为当时的nohup.out和log文件夹不在同一目录，log文件夹专门配置过放在统一的目录，然后希望nohup.out也有日志输出……具体原因有点模糊了，这应该是第一次恶心到我自己的代码。后来好像再版的时候去掉了，恍恍惚惚。

同样是这个需求中，当时的老大的代码里，有一处值得借鉴和赞赏的写法，我当时怀着慷慨激昂的心情认为这是冗余的做法，有点想要挑战权威的意味，事情证明还是年轻了。

代码想不起来了 T\_T

同样还是这个需求，需要实现可以自定义配置sql语句，类似jdbcTemplate中的?，或者mybatis的#{ }，我当时用了非常生硬的做法：

```
// 需求
sql = update set a={a}, b={b}, ...
// 解决
for (i = 0; i < param.num; i++) {
    switch(match.group()) {
        case {a}: break;
        case {b}: break;
        ...
    }
}
```

当被问到参数位置是否可以调换，还是放心的，那参数能多能少吗？……

## 改需求

曾经见过一个程序，一个文件里面有一个类，这是理所当然的，这个类里面只有一个方法，方法有一千多行，是个女程序员写的。

我的任务是原先输出的报表上加几个字段，原先输出了a,b，现在需要a,b,c,d。c,d是我需要实现的部分，大致的程序逻辑和a,b一样，但是来源不同，执行不同的sql查出来的。按理说改个sql不就完了吗，问题是一千多行的方法里面嵌套了三四层循环，包含了从不同银行查数据的业务操

作，循环内外的变量名就是list1、list2，最终的结果还会涉及到对a,b的累加操作。

当时选择的做法是不动原来的程序，复制一份出来，分别执行完后整合一下数据。心高气傲啊，不愿意读垃圾代码。所以来一个类里面有两个一千行代码的方法了。要命的是bug不断，复制出来的代码仍然需要从头到尾看明白，而且对两部分数据的整合也带来了不少麻烦和意想不到的漏洞……

## 完善需求

另一个需求是接手别人做到一半的项目。9102年了在spring boot里面用模板引擎写页面，我不得不不用jquery和基于jquery的移动端样式库完成开发，界面丑先不说了，文档！文档前后对不上！也怪我，小看了需求，应该不明白就问，问清楚了各个环节再动手操作。结果自然好不了，也是反反复复的改……可能不会有愿意再维护那份代码。

## 不断变更的需求

无法在一开始敲定需求，一定会产生烂代码。在理想情况下做好规划、有充裕的时间和高素质的开发人员完成工作，各种设计、模式自然是好的。现实一定不是那样。

vue的前端项目里有一种惯用的写法，是对axios的封装，在一个单独的api.js里写各种export，然后在模块中import。据说是统一管理api的url还是什么，总之在实际的开发过程中，发现这实在是繁琐的写法，带来了很多开发上的不便。维护更容易了吗？并没有。原因之一就是需求在不断变化，如果api里的注释不够清晰也不够准确，那单独抽出来毫无意义。

```
// api.js
export a = () => { get('a') }

// model.js
import {a} from api.js

a().then(res => {})
```

如果api中的a函数和路由a不完全对应，需要修改api.js时就不得不到对应的页面中去找是哪个函数，这和直接把路由写在页面中没有区别。或许可以试着把路由访问的函数在每个模块或者每个页面顶部封装一下，让模块或者页面里的api是自治的。

前端项目里比起api，更让人难过的是随处可见的css！更更让人难过的是项目已经成型了！同样是背景色的配置，index里写一遍，common里写一遍，model里写一遍，甚至内联的写法再来一遍……问题是各个地方的配置内容还不一样。背景色这种样式还好，可怕的是文档流、定位，浮动，边距……css真正的内功是文档流吧，假如完全不懂还撑起了整个项目，这样的代码谁敢动？

## 懒得重写的需求

随着需求的不断变更，会出现的另一个现象是以前的代码能复用就复用，以最快的速度完成现在的需求，反正不知道以后需求会不会变，要是每次都重构出整洁的代码，工作量和工作效率……这绝对是产生烂代码的途径之一。

后端开发中有个我不太喜欢的东西，bean，也叫entity，不知道是从哪个年代流传下来的做法，包括模板化的dao和服务，写个IClass再写个ClassImpl有多大意思呢。bean不利于需求变更，有时候用map反而好一点。回到一开始提到要验收的项目，我写出的烂代码和bean不无关系，大概就是bean的设计包含有很多个字段，后来需求变了，字段也变了，如果要改就要改很多地方，天知道以后需求还会不会变，就强行复用了，导致有些字段含义不明，有些字段完全对不上，仅仅是占用了位置来储存数据。

当然，框架的选择失误也是产生垃圾代码的原因，这要归因于我经验不足，不得不承认架构师在项目管理中起到的核心作用。

## 开发中的困境

一个是起名字的难题，偶尔写错了接口名称，或者复制粘贴相似的逻辑，懒得改变量名，产出的代码一定难以维护，这就可见code review真是企业开发中不可缺少的一步，不过中小公司可能对代码质量的要求也没那么高，除非是要上线的、会涉及到线上业务的代码，review是必须的，搞不好会出人命的，普通的开发可能就没那么严格。

另一个是模块划分的问题，比如a页面有文件上传，b页面有文件下载，那么文件上传下载是不是应该放在同一api路径？都是文件操作，好像挺对的。可如果其他api都是按照页面划分，同一个页面的所有接口都在同一路径下，那单独把文件操作拎出来是不是有点突兀？有时也会按照数据表划分模块，一遇到联表的操作就不好约定了。各种规则混着用多半会引起混乱，让代码难堪。

?

年终之际，还是要给我的2019画上一个圆满的问号。

# Haskell中的Monad是什么?

2019-11-26

第一次听说Monad是在一个Scala Meetup上，后来试着了解Monad的概念，却头疼于Haskell的各种大部头的书和教程。再后来看到阮一峰在2015年发表的《[图解 Monad](#)》，虽然清晰易懂，但是脱离了Haskell，图片的表意和语言中的概念对不上。阮一峰的文章译自《[Functors, Applicatives, And Monads In Pictures](#)》，我阅读了原文。

## 前言

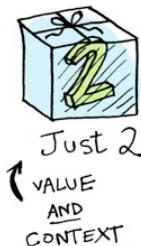
计算机程序用于控制计算机进行运算，程序操作的对象是各种不同类型的值，比如数值。这是一个简单的值2：



用函数对值进行一些处理，可以返回函数执行的结果，比如：



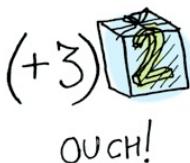
除了简单的数值类型，值也有可能被包含在一些上下文环境中，组成更复杂的值类型。可以把上下文环境想象成盒子，数值放在盒子里面，这个盒子整体作为一个值，描述为Just 2，也就是带盒子的2：



如果对Java有过了解，可以将这个盒子理解为包装类，比如Integer和int，对应带盒子的2和不带盒子的2。

## Functors

面对带盒子的2，我们无法直接把+3的函数作用在它上面：



这时需要一个函数fmap来操作。fmap会先从Just 2中取出数值2，然后和3相加，再把结果5放回盒子里，返回Just 5：



fmap怎么知道该如何解析Just？换一个其他像only之类的类型，还能解析吗？所以就需要Functor（函子）来完成定义的操作。

Functor是一种数据类型：

1. TO MAKE A DATA TYPE  $f$   
A FUNCTOR,

class Functor  $f$  where

$\curvearrowright f\text{map} :: (a \rightarrow b) \rightarrow fa \rightarrow fb$

2. THAT DATA TYPE  
NEEDS TO DEFINE  
HOW  $f\text{map}$  WILL  
WORK WITH IT.

Functor定义了fmap的行为:

$f\text{map} :: (a \rightarrow b) \rightarrow fa \rightarrow fb$

1.  $f\text{map}$  TAKES A FUNCTION (LIKE  $(+3)$ )

2. AND A FUNCTOR (LIKE  $\text{Just } 2$ )

3. AND RETURNS A NEW FUNCTOR (LIKE  $\text{Just } 5$ )

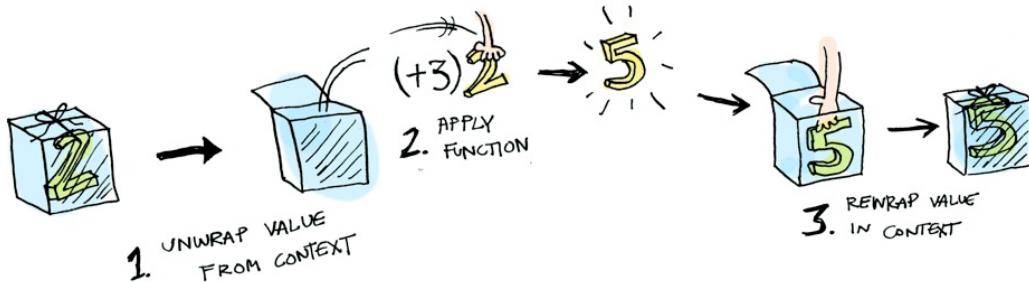
fmap有两个入参和一个出参，入参分别是一个函数和一个带盒子的值，出参是一个带盒子的值，可以这样使用:

```
fmap (+3) (Just 2)
-- Just 5
```

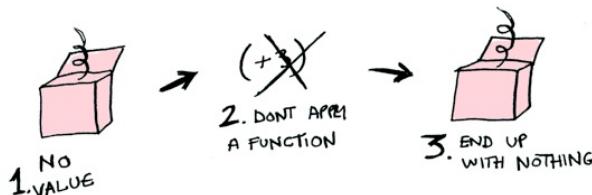
回到Haskell，在Haskell的“系统类库”中有一个Functor的实例Maybe，Maybe中定义了fmap的行为，指定了面对Just类型的入参时对值进行操作:

```
instance Functor Maybe where
  fmap func (Just val) = Just (func val)
  fmap func Nothing = Nothing
```

表达式 $fmap (+3) (\text{Just } 2)$ 的整个过程类似这样:



同理，从Maybe的定义中能看出，如果传入fmap的第二个参数是Nothing，函数将返回Nothing，事实确实如此:



```
fmap (+3) Nothing
-- Nothing
```

现在假设一个Java的场景，用户使用工具类Request发起一个向服务器的请求，请求返回的类型是Response，Response是一个实体类，可能包含所需数据data也可能不包含:

```

Response res = Request.get(url);
if (res.get("data") != null) {
    return res.data;
} else {
    return null;
}

```

使用Haskell中fmap的写法就变成了：

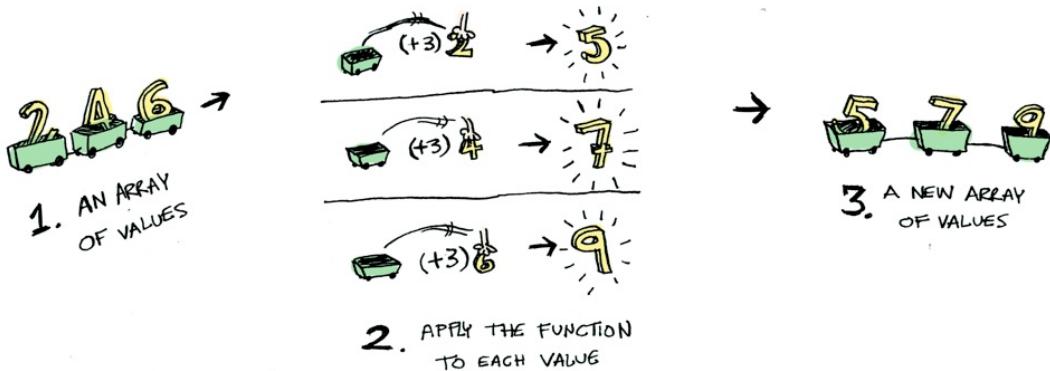
```
fmap (get("data")) (Response res)
```

当然Haskell不存在get("data")这样的写法，可以将由Response获取Response.data的操作封装为函数getData，然后传入fmap作为第一个参数。

Haskell提供了fmap函数的语法糖`<$>`简化fmap的写法：

```
getData <$> (Response res)
```

再来想一个问题，Haskell的函数是如何对列表进行操作的？函数会对列表的每一个元素都进行计算，然后返回列表：



其实列表也是Functions，这是列表的定义：

```
instance Functor [] where
    fmap = map
```

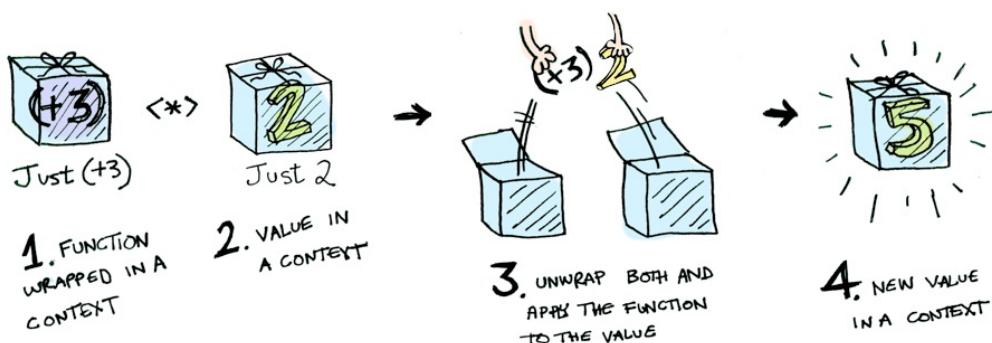
## Applicatives

Applicatives是另一个概念，我们之前说数据被放在盒子里，如果函数也被放在盒子里呢？



Just (+3)

Haskell的系统提供了操作符`<*>`用于处理盒子里的函数：

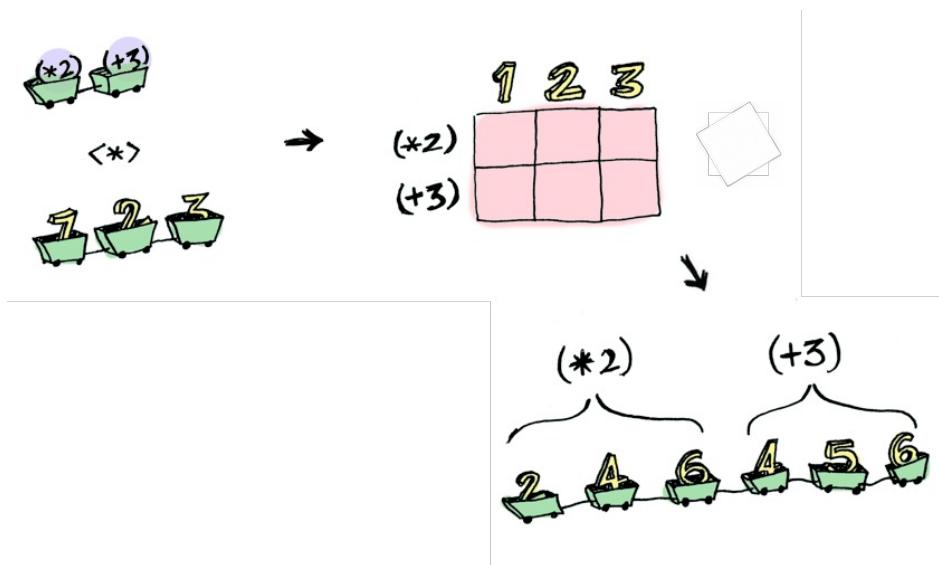


例如：

```
Just (+3) <*> Just 2 == Just 5
```

使用`<*>`还可以完成一些有趣的操作，比如分别让列表中的元素\*2和+3：

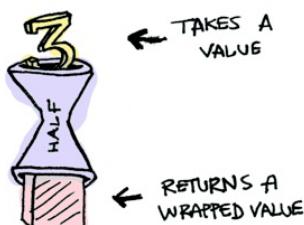
```
[(*2), (+3)] <*> [1, 2, 3]
-- [2, 4, 6, 4, 5, 6]
```



## Monads

函数的执行是使用带入参的函数处理值，涉及到三个角色。Functors是被处理的值放在盒子里，Applicatives是函数放在盒子里，Monads则是将函数的入参放在盒子里。Monads有一个操作符`>>=`来实现Monads的功能。假设现在有一个函数`half`的入参是数值，如果是偶数就除以2，否则返回`Nothing`：

```
half x = if even x
         then Just (x `div` 2)
         else Nothing
```



想要给`half`传一个`Just`类型的值怎么办？



`>>=`可以解决这个问题：

```
Just 3 >>= half
-- Nothing
```

`>>=`操作符把`Just 3`变成了3放在`half`中进行计算。Monad是一个数据类型，定义了`>>=`的行为：

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

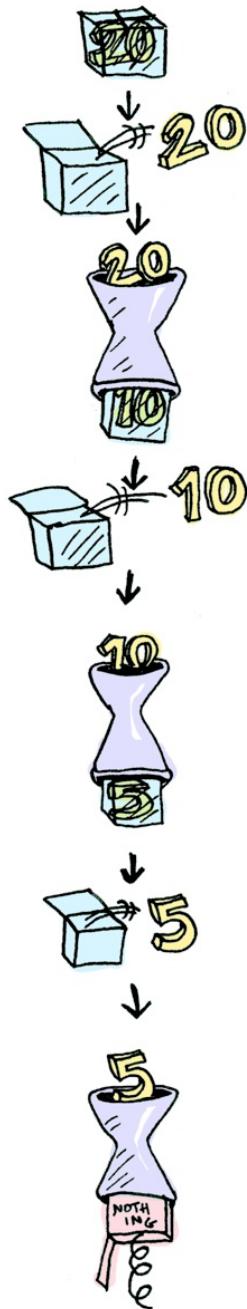
$(\gg=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b$

这里的Maybe是一个Monad (和上文的Maybe同时存在) :

```
instance Monad Maybe where
  Nothing >>= func = Nothing
  Just val >>= func = func val
```

>>还支持链式操作:

```
Just 20 >>= half >>= half >>= half
-- Nothing
```



## 小结

虽然Haskell的Monad比较有名，但实际上涉及到三个概念Functors、Applicatives和Monads，可能Monad的应用比较广泛一点。在数据处理上，FP并不比OOP高级，逻辑是相似的，只是写法不同。面对同样的问题使用不同的思维方式和表达方式去解决，对应了不同的编程思想和编程范式。世界上有很多精妙的理论等待我们探索。

# 浅析Libra背后的区块链技术

2019-11-20

前段时间国家领导人曾公开表示鼓励区块链技术的研究，称要把区块链作为核心技术自主创新的突破口。Libra的发行计划是区块链发展史上一座重要的里程碑，本文从合约语言、数据库协议、逻辑数据模型、数据结构、共识协议等方面简要介绍Libra区块链的技术方案。

2019年5月，Facebook首次确认推出加密货币的意向，“全球币”、“脸书币”的消息不胫而走。6月18日，Facebook正式宣布将推出名为Libra的加密货币，预计2020年上半年针对性发布。Facebook宣称，Libra建立在安全、可靠、可扩展的区块链上，采用链外资产抵押的模式，锚定一篮子法定货币作为资产担保，由独立的Libra协会治理。Facebook在全球拥有27亿用户，Libra的愿景是建立一套简单、无国界的货币，为数十亿人提供金融服务。

Facebook正式宣布Libra后，同步上线了Libra的官网、白皮书和测试网络等内容。白皮书中提到，Libra网络希望未来以公有链的形式运作，但由于目前没有成熟的技术能在公有链上支撑大规模的交易，Libra将以联盟链的形式起步，计划三到五年内展开由联盟链过渡为公有链的研究。

Libra官网公布了三篇论文详细说明Libra使用的技术方案，这三篇论文分别是《Libra区块链》、《Move：一种具有可编程资源的语言》和《Libra区块链中的状态机复制》。Libra为满足高度安全、足够灵活、吞吐量极高等要求，设计了新的编程语言Move，选择BTF共识机制，采用广泛使用的Merkle Tree作为数据结构。本文简要介绍Libra区块链的相关技术。

## Move

当现实世界的资产进入Libra储备，系统会创建相应的数字资产Libra货币，这些数字资产在不同账户之间流通，当现实中的资产离开Libra储备，对应的数字资产也随之销毁。Facebook设计了新的编程语言Move用于对Libra中的数字资产进行管理，Libra在Move中将数字资产表示为资源(resource)。

Move是带有类型的字节码语言，资源是Move的类型之一，程序在执行前会先经过字节码验证器检验，然后由解释器执行。Move中的资源仅支持copy和move两种操作，可以理解为copy移出资源，move移入资源，也就是说资源和普通的变量类型不同，资源的值可以赋给普通变量，但资源本身只能在地址间移动，不能复制或丢弃。如果在程序中使用了违反规则的copy和move操作，比如copy一次move两次，程序将无法通过字节码验证器，因为在第一次move后原先的资源就已经不可访问了。

使用Move可以编写自定义的交易逻辑和智能合约，相比现有的合约语言要更加强大。比特币脚本提供了简洁优雅的设计用于表达花费比特币的策略，但是比特币脚本不支持自定义数据类型和程序，不是图灵完备的。以太坊虚拟机倒是支持流程控制、自定义数据结构等特性，但太过自由的合约让程序的漏洞随之增多，发生过多起安全事件。Move的静态类型系统为数字资产的安全性提供了保障。

为了配合静态验证工具的验证，Move在设计上采取了一些措施：没有动态调度，让验证工具更容易分析程序；限制可变性，每一次值的变化都要通过引用传递，临时变量必须在单个脚本中创建和销毁，字节码验证器使用类似Rust的“borrow checking”机制保证同一时间变量只有一个可变引用；模块化，验证工具可以从模块层面对程序进行验证而不需要关心具体实现细节，等等。Move的这些特性都使得静态验证工具更加高效可靠。

```
public main(payee: address, amount: u64) {
    let coin: 0x0.Currency.Coin = 0x0.Currency.withdraw_from_sender(copy(amount));
    0x0.Currency.deposit(copy(payee), move(coin));
}
```

这是一段交易脚本的示例程序，是Move语言的中间表示（IR），IR更适合程序员阅读和编写。

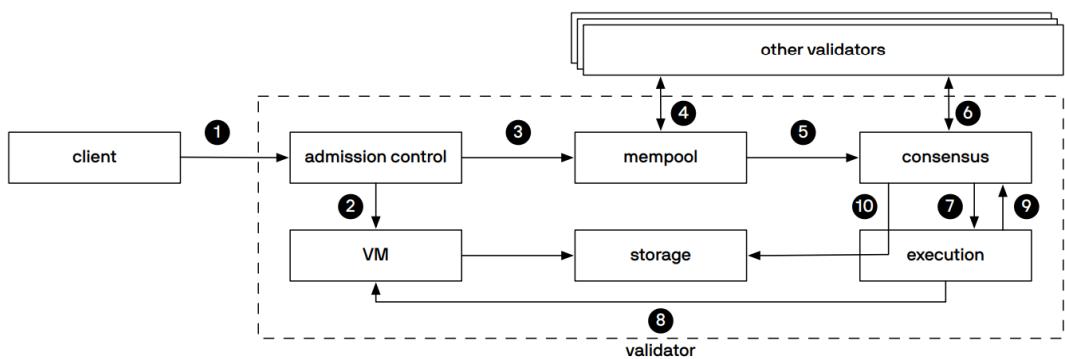
程序实现了一个转移资源的函数，main方法是脚本的入口，包含两个入参：目标地址和金额。程序先从0x0地址Currency模块中移出amount个资源暂存到coin变量，然后将coin的资源移动到payee的地址上。

交易脚本是为Move提供灵活性的一个方面，另一方面来自安全的模块化设计。交易脚本让交易逻辑更加自由，模块化设计则保证了脚本程序的多样化。模块的类型是module，主要包含Move程序，一个模块可以包含任意个资源，也就是声明另一个或多个资源类型的变量，modules/resources/procedures相当于面向对象语言中的classes/object/methods，不同的是Move中并没有self、this之类的概念。

## Libra协议

Libra区块链是一个需要经过密码学认证的分布式数据库，用于储存可编程资源，比如Libra货币就是可编程资源，在Move中表现为资源。Libra协议中有两种实体类型，验证节点（validators）共同参与维护数据库，客户端（client）通常发起向数据库的请求。Libra协议会在执行过程中选举出leader接收客户端的请求，然后leader将请求同步到其他验证节点执行，其他验证节点执行结束后把结果返回给leader，leader再把请求的最终结果返回客户端。

Libra的交易会经过很多步骤，包括验签、运行先导程序、验证交易脚本和模块程序的正确性、发布模块、执行交易脚本、运行结尾程序等。为了使合约交易的计算能力可计量，Libra吸收了以太坊中Gas的概念，消耗Gas作为交易的费用。

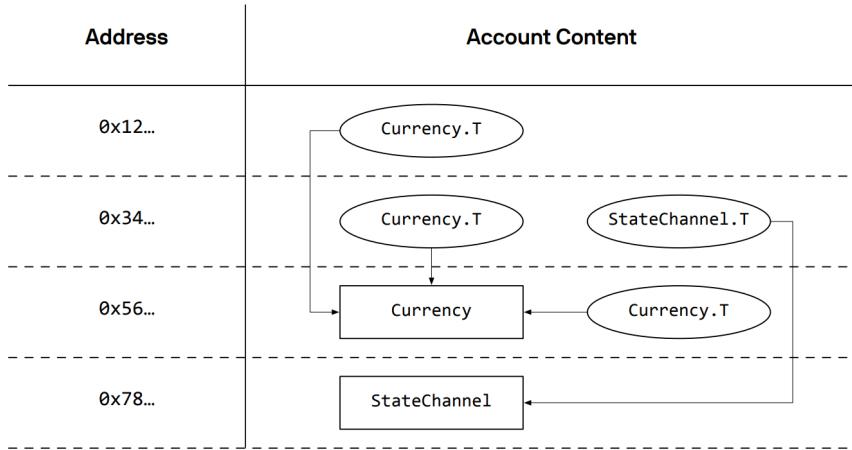


这张图详细展示了交易请求在Libra的网络组件中流转的过程，客户端发起请求到权限控制层，权限验证后将请求数据转给虚拟机进行预处理，同时数据也会进入内存池中，内存池负责将请求同步到其他节点，共识协议在请求同步的过程中发挥作用，节点同步结束后虚拟机执行真正的交易程序，程序执行完毕对结果持久化，基本流程结束。

## 逻辑数据模型

Libra区块链上所有的数据都保存在有版本号标识的数据库中，版本号是64位无符号整数。每个版本的数据库都包含一个元组(T, O, S)，T代表交易，O代表交易的输出，S代表账本的状态。当我们说执行了一个Apply操作，表示为Apply(S, T) -> (O, S)，意思是在S状态下执行了T交易，产生了O输出并且账本的状态变为S。

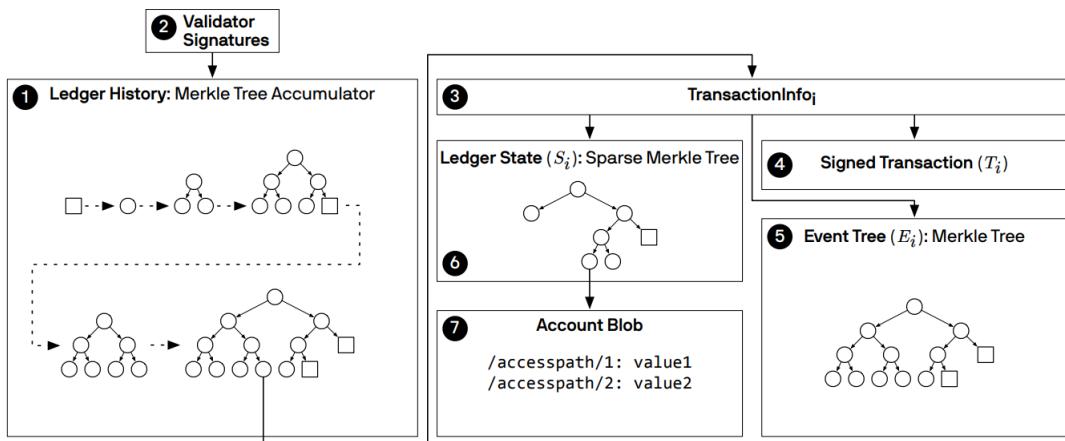
账户是资源的拥有者，可以使用账户内的资源进行交易。账户地址是一个256位的值，创建新账户需要一个验证/签名的键值对(vk, sk)，新的账户地址a由vk经过公钥加密计算得到， $a = H(vk)$ 。具体来说Libra使用SHA3-256实例化哈希函数，使用wards25519椭圆曲线做变量的EdDSA公钥进行数字签名。交易过程中由已经存在的账户调用create\_account(a)指令即可生成新账户。



上图所示有四个以0x为前缀的账户地址，矩形框表示模块，椭圆形表示资源，箭头表示依赖关系。图中0x12账户中的Currency.T在Currency模块中声明，Currency模块的代码储存在0x56地址上。同理，0x34的StatChannel.T声明自0x78的StateChannel模块。当客户端想要访问0x12下的Currency.T，请求资源的路径应写作0x12/resources/0x56.Currency.T。

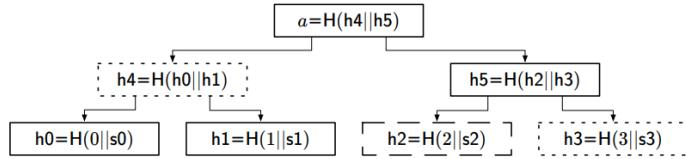
## 数据结构

Libra交易区块中包含各节点签名的数据，交易前会对签名数据进行校验，根据这一集体签名客户端可以相信请求的数据库版本是完整有效的，也因此客户端可以请求任意节点甚至是第三方数据库副本进行查询。Libra协议中的数据结构主要基于默克尔树。



如图所示，账本历史数据的根哈希用来验证系统的完整状态，账本数据由默克尔树累加形成，虚线表示数据累积的过程。账本历史数据的每一个节点都包含交易签名、事件树和账本状态，事件树也是基于默克尔树，账本状态则是基于稀疏默克尔树，账本状态的每个叶子节点都包含有账户数据。

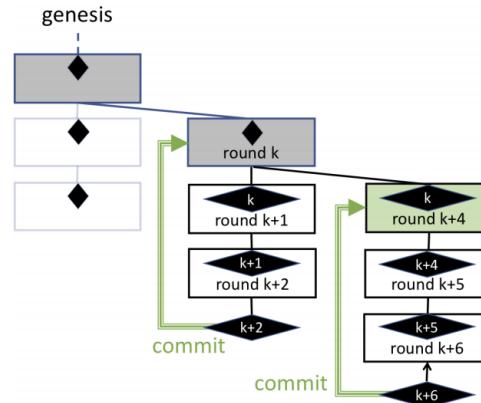
Libra协议中验证节点V会对数据D的根哈希 $\alpha$ 进行校验。例如不受信任的节点在获取到数据D后使用函数 $f$ 对数据进行运算，希望得到结果 $r$ ，同时还需要一个用于验证函数结果正确性的数据 $\pi$ ，协议会要求节点把 $(\alpha, f, r, \pi)$ 都传到验证节点V处进行验证，如果 $f(D) = r$ 则通过验证。



在上图中，数据 $D = \{0:s0, 1:s1, 2:s2, 3:s3\}$ 。假设 $f$ 是获取第三项数据的函数，也就是要获取 $h2$ 的数据，期望结果 $f(D) = h2$ ，此时 $h2$ 就是 $r$ ， $r = h2$ ，用于验证计算结果正确性的数据 $\pi = [h3, h4]$ ，根哈希 $a = H(h4 || h5) = H(h4 \parallel H(H(2 \parallel r) \parallel h3))$ ，验证节点将执行 $\text{Verify}(a, f, r, \pi)$ 对计算结果进行验证。

## 共识协议

Libra选择使用的是拜占庭容错共识，实现了一种HotStuff共识的变体LBFT，简称LBFT。LBFT协议的主要作用是让提交的块在同一个序列上，或者说避免分叉。每三次提交为一轮操作，每一次提交验证节点都会投票选举出下一轮的leader，同时这些投票的集合形成一个法定证书（QC），每一轮的第一个块记为preferred\_round，下一个块写入时对preferred\_round的QC进行验证，也就是preferred\_round后的第一、二、三个块都与preferred\_round校验，第四个块将是第二轮的preferred\_round，依次更迭。



如图所示， $k$ 是preferred\_round，如果在 $k$ 处出现了分叉，并且有 $2f + 1$ 个验证节点投票给了 $k$ ， $k+1$ 将接在 $k$ 的后面， $k+2$ 依次写入， $k$ 左侧的分叉失效。这时假如 $k+3$ 的leader超时了， $k+4$ 成为新的leader并写入在preferred\_round ( $k$ ) 的后面，会引起新的分叉。如果 $k+4$ 获得 $2f+1$ 个投票， $k+5$ 会按照规则写入在 $k+4$ 后面。

在 $k+4$ 分叉后，当超时的 $k+3$ 再次被选为leader并重新提交，验证节点会对 $k+3$ 的preferred\_round ( $k$ ) 的QC进行校验，校验通过， $k+3$ 写入在了 $k+2$ 的后面，这符合规则。再然后，下一个leader ( $k+6$ ) 的preferred\_round实际上是 $k+4$ ，准备提交块到 $k+3$ 后面时发现QC对不上， $k$ 及其后的 $k+1$ 、 $k+2$ 、 $k+3$ 都会被删除， $k+4$ 后的链成为主链。

LBFT基本上是对链式HotStuff的实现，并没有太多创新，Libra团队更多的是在共识协议中做出一种选择，对BFT的选择是好是坏还存在争议，需要时间来验证，LBFT算是Libra从联盟链到公有链过渡前的方案，预计至少支持100个节点，上限大概是1000个左右。和HotStuff一样，LBFT最多容忍三分之一的不诚实节点。

## 小结

Libra协议目前还处于比较早期的阶段，性能上并不算惊艳，支持每秒1000笔交易，每次提交的交易确认时间大概是10秒钟。Libra协议在设计上很多也是兼顾了性能，比如每三次提交进行一

次共识，每一轮操作内不需要等待就可以进行投票，这减少了客户端和验证节点之间的网络延时；考虑到并行和分片的思想，稀疏默克尔树的使用使得账户的身份数据可以跨数据库进行验证，也支持并行更新等等。

Libra选择了众多成熟的技术构建Libra系统，使用内存安全的Rust编写核心程序、使用容易验证的Merkle Tree作数据结构、基于Chained HotStuff实现共识协议等等。Move是Libra在技术上最大的亮点之一，在语言层面保证了数字资产的安全性，Move本身是一种字节码语言，难以阅读，所以提供了Move的中间表示IR，用于编写交易脚本和智能合约。Libra作为一种在金融领域的创新实验，基于区块链提出了世界货币的愿景，其社会意义可能要远大于在技术创新上所带来的意义。Libra协会目前拥有16个成员组织，涵盖支付业、电信业、区块链业、风险投资业等领域，已经在世界范围引起广泛关注。

Libra预计2020年上半年针对性发布，让我们拭目以待！

# 对区块链的理性认识

2019-11-05

曾以为区块链是革命性的、颠覆性的技术，毕竟区块链和人工智能、大数据并列互联网前沿技术。但是，人工智能达到真正的智能暂时还是梦，大数据也实现不了像《复联》里一样的精准分析。前几天国家领导人对区块链的讲话振奋人心，可过去几年的物联网、互联网+，不也都无疾而终了吗。

## 公有链无法脱离货币

有的观点认为，核反应最初目的是建造核弹，而核反应现在也作为电能的来源服务于人民，区块链最初目的是支持比特币的运作，现在我们想要区块链应用于其他方向。也就是说，观点认为核反应的位置和区块链是对等的。

核反应 -> 核弹  
区块链 -> 比特币

可是仔细想想，到底该如何对比这几乎完全不一样的两种事物。如果你看过几本区块链相关的书，会发现讲的东西并不会特别新鲜，观点也完全够不上所谓革命、创新。比特币诞生至今不过10年左右，并没有克服技术上的难题，只是不同机制的组合，它应该和P2P或者某种电子游戏处在同一地位。比特币的价值在于进行电子交易，而不是货币本身。

核反应 -> 核能量 -> 大爆炸  
区块链 -> 比特币 -> 电子交易

因为核反应，才能产出核能量，有了核能量，才能够产生大爆炸。因为区块链，才有了比特币，有了比特币，才能够进行电子交易。所以这样来看，核能量是离不开核反应的，比特币是离不开区块链的。

那么，区块链能够离开比特币单独应用于场景吗？或者说，其实区块链和比特币是一体的，就像核反应产生的核能量，有价值的是能量而不是反应，区块链产出比特币，运作于区块链上的比特币才有所谓匿名、不可篡改、可溯源等特性，有价值的是比特币而不是区块链。

几天前广州市政府发布补贴区块链企业的细则（[实施细则](#)），明确要求“无币”公有链项目。这项政策，一方面是国家不允许发币，另一方面，公有链无币其实是区块链最理想的情况，一般来说，理想是难以实现的。

比特币解决的是交易中的信任问题，解决方式归根结底是数据存在哪儿。两个人进行交易，如果由交易发起方记账，或者交易接受方记账，或者两个人都记账，无论谁记，只要有一方说谎，甚至不说谎，他就是记错了，都会产生争执，不认账怎么办！这时就需要第三方机构介入，通过银行记账，通过律所解决纠纷。要是连银行、政府、法律都不相信，就不用活了。

如果真的不信任中心化的机构，比特币提出办法是，让全世界的人都为你记账，全世界的人都会记住两个人的交易记录，谁给谁转账多少，这样无论如何都不再会有差错，除非全世界一半以上的人都犯了同样的错误。所以问题在于，凭什么让全世界的人为两个人的个人交易记账？人家为什么要记？于是将比特币作为奖励，谁记账了，并且被系统认为记的账是有效的，谁就可以得到奖金。

没有奖励，世界上的人不会主动为你记账，分布式账本还怎么维持运行？

## 比特币并非去中心化

有人认为分布式记账、分布式数据库就已经是去中心化了，但这一定不是去中心化的最终形态。比如，比特币程序的开发、维护和升级？数据确实天下共享，但程序还是要有人制定规则，有人开发，有人发布，出bug了要有人修复，有更好的点子了要迭代升级，分布式的数据全部经由中心化的程序发布中心发布的程序处理。目前解决程序上信任的方式是将程序开源……这一点暂且

可行，但是程序升级带来的困难就要大多了，要么确保向下兼容，要么确保所有人更新程序。

另外，比特币的数据冗余是个极大的问题，每个节点都需要备份全量数据，而且大多数是不相关的历史数据。如果单个节点不保留全部数据，就无法保证分布式数据的可靠性，但如果保留全部数据，又是对资源很大的浪费。中心化系统一份数据就可以解决的问题，为了能够相互信任，就多出来几十亿份数据？就好比我不相信银行，就自己造一个银行，自己管自己？

可以畅想一下，在牺牲去中心化概念的情况下，能够有哪些可能。

一、全球共用一个数据库，数据库只承担储存数据的任务，分布式程序只解决共识问题。数据库非常安全，数据容量非常大，但是写入规则严格，需要全球一半以上的人认可，或者通过其他的共识机制准入。任何人可以随意查询、可溯源，历史数据不能修改。共识程序是必要的，决定了哪些数据可以写入，比如判断余额是否足够，而且是全世界的人一起判断，如果有坏人想要写入非法数据，需要买通全球一半以上的人……这样数据冗余最少。

二、每个节点只保留一半数据，数据拆分为历史的一半和当前的一半。一个人储存最新的一半数据，另一个人储存旧的一半数据，旧数据只需要负责储存，当新数据过多时同步到旧数据这里。新数据负责接收广播、写入数据，功能等同于现在的节点，如果遇到需要查询历史数据的情况，就从旧数据的一半查。相当于两人合作完成一个节点，新旧节点随时随机搭配，节点的新旧由系统平均分配。至于安全性，因为全网的节点随机配对，应该不会低于比特币，最坏的情况是一半的节点全部挂掉。同理，可将两份数据扩展到多份数据的情况。

三、每个节点只保留一半数据或者更多份，数据对半拆分。就是同一条数据，按照一定规则拆分为多个数据包，分别储存在不同的节点，参考HDFS的储存方式，存在一定冗余，但又节省了不少空间。再激进一点，数据可以实现自验证，网络中的每个节点储存的数据大小是随机的，当用户查询某一条数据时，从全网的节点中搜寻可以组成所需数据的节点，然后从中取出数据。也就是说整个节点网络的数据都混杂在一起，难点变成了如何给数据包设计自验证机制。

## 数字货币和区块链没有关系

有的人谈到区块链的应用，会把央行关于数字货币的研究给扯上，甚至某交易所知名总监，以区块链为主题的演讲，却把比特币和Libra的趣闻轶事说了一遍。很多人都在忽略概念上的区别，这无关紧要，也至关重要。央行说有发行数字货币的计划，也说过区块链可以作为技术选择之一，但区块链从不是必须的技术。区块链对于国家的意义，是“以去中心化之名，行中心化之实”，意在一统国内互联网，方便监管。即使没有区块链，国家也有能力实现各种应用，只是借势上了区块链的船而已。

过去的区块链指支撑比特币运行的技术体系，未来的区块链将几乎约等于联盟链。

华为区块链白皮书中的观点很客观，区块链是互联网的补充，它不会脱离传统数据库，离不开TCP，只是在特定场合下发挥独特的作用。对于国家来说，链上的数据清晰可见，没有人能暗箱操作；对于企业来说，可以方便的实现制衡，几家企业合作共享一组数据，区块链则是打开大门的钥匙。如果没有区块链，可能说不上来数据共享是个什么样子，区块链诞生了，并且比特币在世界范围稳定运行了十多年，所以这是可信的、有前途的技术方向，大家都争先恐后创新、落地。

可以预见，未来区块链的开发会分为两类，一类底层开发，一类应用层开发。底层开发的技术要求更高，开发者素质更高，应用层开发则类似于现在的Web开发。会先后出现一些区块链应用提供商，也会相应的出现一些SDK，开发者调用区块链储存数据、进行交易，类似于现在调用数据库提供的API、请求支付机构的接口。

## 所以

区块链会被广泛应用到我们的网络中，但不足以改变世界。（不要笑）

## 更新

无意间发现了分布式网络 [ZeroNet](#)，是一个早在2015年就发布的项目，它几乎满足了所有我对区块链储存系统的想象，而且功能完备，可以基于这个网络搭建博客、论坛、邮箱、共享文件等。当然，我曾想到的、应该存在的问题，ZeroNet也一个都没有解决，算是对我一些想法的验证，唯一不同的是我希望将分布式网络对接到公网，但ZeroNet的做法是建立了一套自治的网络系统，包括.bit域名也只能作为URI的后缀，这无疑限制了该网络无法被更加广泛传播使用。另外，由于点对点文件系统难以监管，GWF将ZeroNet列入名单，这虽然是特殊现象，但ZeroNet和IPFS等网络似乎可以说明，区块链最适合也只能应用于金融领域或者受限制的互联网中。

# 来，我们再谈一谈梦想

2019-10-26

早就想好了标题，可迟迟不知道要写什么，始终无法给内容“定调”。后来一想，其实也没必要强调『基调』，在开头或者结尾说几句概括性的话确实好，少了主题，内容就显得空泛，但这并不影响我“记录”自己的生活，并不影响我表达在不同时期对不同事物的『见解』。

## 工作

最近领导离职了，原因是要去创业。

领导离职带来的直接影响是，我最近的工作任务少得可怜，上一个项目接近尾声，下一个项目进展缓慢，加上领导离职，谁也不知道部门接下来会怎么发展。也因此，我有了比较多的空闲时间，我开始思考该如何提升自己的实力。在职业生涯上，我曾经有两个『追求』，一是不局限于语言，二是不局限于框架。其实这涵盖了太多的内容。

我非常不喜欢『面向面试』的学习，有个同事以前鼓动我，说我们互相分享一下最近的项目经验，有什么收获，这样我们以后去面试就有两个项目可以说了。看我不太感兴趣，他继续说，我看你平时也自己做一些小Demo之类的东西，但是你去面试总不能拿这些去面吧，我可以给你讲讲我们这个项目里的东西，像PDF解析、上传下载，还有关于区块链的一些东西……

对于这件事情，槽点就很多了。

他一直以和他一样没有经验的『应届生』看待我，其实我对Web开发的理解远超过他。从『人』的角度来看，我并不认为他的项目经验能够给我带来帮助，尤其是什么IO流、网络流之类的基础操作。对了，这个同事之前还鼓动我做另一件事情，就是总结项目里可复用的代码，比如PDF解析，网上可能有很多方案，但是自己总结一个能用的放着，以后遇到类似的需求直接CV就行了……他甚至还说，他的目标就是能够熟练的完成工作。

我的目标从来不是成为熟练的工人。就像搬砖的工人，从来不会因为搬的又快又多成为包工头。这也是我现在不想离开目前公司的原因，如果去其他公司996，即使工资涨了，也只是因为付出了更多的劳动力。平时一天搬10块砖头挣10块钱，如果加班多搬5块砖头，一天就可以挣15块钱。我目前还不是那个阶段，还没有必要一心赚钱的。

另外，我相信有能力的面试官，可以分辨出真正有实力的候选人。什么飘来飘去的框架，满口听起来高大上的词汇，微服务还没有过气，中台又炒到天上……另外，规范也不是真理，《代码整洁之道》也不是真理，设定各种条条框框有助于管理者把员工变成机器，但是作为员工如果听信了某种『宗教』，我同样认为不合时宜。

## 创业

领导离职去创业了，原因自然是想赚更多的钱。他说，挣多少钱就可以退休了呢，每年的开销乘以20，也就是说10年以后你的钱就剩一半了，这还不算货币贬值（用钱买基金是不是能够应对货币贬值？）。

创业是一个经常听到的词了，但是我还未见过创业真正的形态。同样是这段时间，上面提到的那个同事，入职三个月的『应届生』，也辞职了，要去创业了。领导创业我还能理解，基于多年的见识和积累，有能力也有经济实力去尝试创业。这位看起来涉世未深的同事要去创业，就有点匪夷所思了。不过也许人家并不是依靠自己的能力，而是有大佬愿意带着起飞，这就很正常了，和咱也没有关系。

在领导的酒席上，印象最深的内容是领导说，以前他作为中层管理，员工入职的时候还会帮着员工给HR说，多给点吧。现在他创业，也出资入股，也要招人，感觉就不一样了。能省下来钱，是给自己省钱。你说你拿着工资整天不干活，到点就走。到点就走，真的就是好吗，真的就好吗……我无法生动形象的转述具体内容了，大意如此。

就这段话来看，我并不赞同。作为中层怎样，作为老板怎样，然后位置不一样有了不一样的观点……然后呢？不同的位置，本就该有不同的观点，但是如果身为老板，想要员工站在老板的角度考虑问题，或者身为员工，想要老板站在员工的角度考虑问题，这不都是要流氓吗，何必呢，矛盾存在就让它存在好了。

另一个问题是“到点就走”。到底对不对呢，这同样存在老板和员工的利益冲突。在知乎上看到关于ownership的说法甚是清晰，ownership的前提是owner，然后才有ship。至于提到加班就不得不提的996，相信只要区分开『奋斗』和『压榨』，情况就可以好一点。

## 武林外传

小时候看武林外传，是在看乐子。最近重温武林外传，却发现自己变了。

以前小郭和秀才组CP，感觉两人挺般配，但现在的想法是，人家小郭是郭巨侠的千金，和四大神捕论兄弟，秀才百无一用，是个没落的知县后辈，年年中不了举。这门不当户不对，好比小郭是北京公安局局长的女儿，秀才是十八线小县城县长的孙子，还没什么功名。现实生活中这可能吗？我们希望并相信生活和爱情的美好，但现实不允许。

然后是掌柜的和老白，开始越看越不明白，老白怎么就看上佟湘玉了呢，和小郭也挺配，初恋展红绫也回来找过他，无双等了他12年零3天……后来想了想，可能是剧情需要吧，再者佟湘玉也是龙门镖局的千金，龙门镖局也是有名望的大集团，老白并不吃亏。

无双是最可怜的，见一个爱一个，却落给了小六。无双又单纯又贤惠，主要是那么好看，老白不要，秀才也不要，后来终究没遇到合适的。难道是因为没有背景孤身一人闯荡江湖吗。尤其是和小郭抢秀才那一段，小郭没了秀才要什么有什么，可无双没了秀才就什么都没有了。无双似乎从来没有融入同福客栈的大家庭。

武林外传和现实生活的另一个差异是，戏里表现出来的都是真性情。生气了，嫉妒了，吃醋了，记仇了，闹别扭了，和好了，真真实实，像我们小时候一样，还没有长大的时候，敢于将真是的情绪表现出来，上演一出出闹剧。对，放在现实生活中叫『作』，不是我说的，是弹幕里满屏的恶意。

现在看个电视，也真是多了很多世俗的眼光。有点感慨。

## 王者荣耀

说到家庭，最近王者荣耀周年庆，变身大作战里有一个夜游峡谷、点亮峡谷的活动，水晶、野区、龙坑，到处都挂着彩灯。看到游戏里的这一幕，忽然想到春节。现在想想，过年什么都没有，新衣服还是好吃的根本不重要，但过年的气氛实在弥足珍贵。除了过年，不会再有什么日子可以让一大家子的人都聚在一起了。

有时候想，一个成年人，如果他生命里乐趣的来源只剩下游戏，那他的人生一定是悲哀的吧。老梁讲到宅男的时候，说你以为宅男宅在家里很无趣，其实宅男的内心是充实的。作为人类，如果在家里没有什么事情可做，那他一定会想着法子和这个搞搞关系，和那个打打招呼。这话不假，也假。常说可怜之人必有可恨之处，相应的，我相信可恨之人，也一定有可怜之处。

## 酒

领导的酒席，自然免不了喝酒。我依然无法融入气氛，甚至没有敬酒的欲望。我和大多数人都没有过“发生关系”，这……不算不合适吧，况且大家都是逢场作戏而已。同事嘛，谈什么感情呢。

关于酒桌上的礼节，我承认我相当生疏。不过我还是认为，人和人合作的根本原因是互利，而不是多喝了几杯酒，混了个脸熟。我们的目的应该是创造价值，而不是求别人给一点施舍。有个老大不小的同事，每天吃饭都尽量和领导一起，那天也是坐到了老板的一桌，后来没说上几句话就回来了。我不知道工作5年以上还是个小兵算不算失败，但我真的不希望失败。

酒那么难喝，为什么自古以来还是那么多人喜欢？可能像人开心了就要哈哈大笑，伤心了就要嚎啕大哭一样，有时候不开心，不伤心，但就是不痛快，生活平淡无奇，反反复复，需要用酒来刺

激自己的舌头和身体，给自己一些痛觉的反馈。“没什么，至少证明我们还活着”。有人说，父母在路上遇到旧识，喜欢聊聊近况，一说起来就没完，小时候觉得烦，长大了才明白，那是父母难得的开心时光。我稍有感悟。

## 食色

随着年龄的增长，越来越喜欢吃饭，发现吃饭也是一件为数不多令人愉悦的事情。小时候视若不见的炒土豆丝、炒白菜、炒鸡蛋、白馒头，尤其武林外传里每次开饭，老白手里的鸡腿、大嘴手里的花生米，垂涎欲滴。为什么以前从未注意这些，我妈喊着让我吃点零食都懒得吃，现在却控制不住自己的食欲。也许是以前有目标吧，想着考试，想着玩游戏，想着找工作，想着未来，想着某人。

现在啊，时过境迁，饮食男女，性也。

# 来，我们谈一谈梦想

2019-09-19

最近租的小隔断被拆了，快国庆节了，查得紧。好像也没什么，找房子，签合同，入住，上班，再习惯。好像一切都很自然，没有什么激动人心的事情，没有什么让人开心的期望。

## 中介

这次找房子花了一笔中介费，美曰其名“服务费”。一开始我问是不是中介，有没有中介费，回答是否定的。不过大哥人还算不错，我也没计较。作为服务行业，我为你提供服务，你为此支付一定费用，这是理所应当的事情。

中介这样的行业存在，本质还是信息不对等。同一城市的中介们会有一个平台，平台里的房源信息是共享的。也就是说，不管房客被哪个中介宰，他们是不介意的。互联网让信息平等了吗？并没有。“互联网的共享精神都是扯淡！”信息共享的前提是，信息提供方愿意将信息分享出来。作为房东，我什么都不想理会，把房子交给你，我每个月就可以收到一大笔钱，何乐而不为？

中介的房子会被查吗？不会的。这种专门折腾用来出租房子，房东、二房东、中介、物业，已经相当于一条产业链。二房东是给物业送过礼的，即使查，也不会查这几个房子，即使查到了，也不会往上报。也许社会就是这样，在哪儿都一样，不管是十八线小县城，还是一线帝都。

## 房子

刚离开之前的房子，还是非常不舍的。毕竟我记得当时是怎么来的，来了之后遇到了不错的人，遇到了不错的房东，性价比来讲之前的房子已经非常好了。其实很久了，已经过了好几个月，大半年了。其实也没多久。

焉知非福呢，只是不会再听到隔壁女生的笑声了。同样的生活节奏时间长了，难免会有一些惯性，看不清楚未来的惯性。偶尔发生一些意料之外的事情也好。现在住的房子临近某个大学，而且大学里的操场没有门禁。以前苦于附近的大学有门禁，公园又太远，运动实在不方便。现在反而手到擒来了。现在每天骑行上班，也算发现了新的天地。以前的生活半径以步行的范围为界限，现在可以将范围扩大一倍了。

焉知非福呢，被举报了的房子，说走就说。我开玩笑的跟同事说，可能今天下班的时候，老板就告诉你明天不用来了，也是说走就走。很正常，不是吗？不过，即使发生那样的事情，焉知非福呢？

## 朋友

前些日子，我主动“放弃”了一个『朋友』。理由很简单，我不想再参与他的任何事情。我绝对不否认今后我们会有互利共赢的可能，但就目前而言，我丝毫不再有兴趣。他可能会是一个成功的商人，有能力赚到很多很多钱。但是能赚钱的人也很多，我知道有的『农民工』攒二三十年的钱，也可以全款买房子。

但是我的『目标』从来不是赚钱。这个朋友，他是做销售的，在朋友圈整天灌鸡汤，说你要努力，要找专业的人办专业的事，我们懂企业管理，知道最完美的薪酬体系，让你学会如何成功……你很难想象一个大学都不能按时毕业的人，跟你谈努力，谈专业。当然，他并不是要卖课程给我。他的另一个特点是，对技术的迷之崇拜。做一个微信公众号，有成就感，开发一个微信小程序，感觉了不起……这种感觉就像我小时候，刚接触互联网的时候，对网络世界的一切都感到新奇，比如注册一个域名，然后全世界的人都可以通过域名访问到我的个人网站，这是多么酷的事情！

一个多月前，他找我说想要开发一个H5网站。我问，具体有些什么功能？大概多少个主要的页面？他说，不太确定，就是有一个大概的想法。然后介于『朋友』的这层关系，我和另外两个人，就马马虎虎开始了。说实话，我答应下来做这个东西，完全是出于私心，因为一起共事的另

外两个人，算是由我负责，给他们安排任务，把控整个项目的进度。哦，忘了说，这个朋友是我同级的校友，但也是一个小公司的老板，在这个项目的过家家游戏里，他仍然扮演『老板』的角色，给我安排的角色是『技术总监』。

本来这算是一个绝妙的机会，我可以趁机了解一些团队管理的东西。然而另外两个人的水平真是让我大跌眼镜，vue的项目，一步一步教也不会启动，git从始至终都不会用，后端……也就只能写几个接口了，毕竟他们都是『后端程序员』。然后就在项目过程中，发现到具体设计操作流程的阶段了，这个项目的应用场景、使用流程、盈利模式，都还完全没有概念。“这样能赚钱吗？”“我也不确定能不能赚钱，咱们总得试试。”我就不奉陪了。

## 金钱的价值

想要认识金钱的『价值』，得了解各种东西的物价，然后侧面对金钱有个认识。以前1块钱就是1块钱，反正学费和生活费都是父母给的，我要多少，他们给多少就是了。最近租的房子没有宽带，我得自己办一个，到这儿就有点犯嘀咕了。

移动100M宽带一年380，是最便宜的套餐。380多吗，不多，平均一个月38。问题是我要吗？好像也不太需要。我的手机套餐¥48/月，一年下来要超过这个宽带的价。我一顿饭就要吃20多，可是上下班骑行一个月才20左右。网吧¥8/小时，38块钱只能上网5个小时不到。然后，我一个月唱歌就要花160，平均¥1/分钟。

买一件衣服50，寿命是一个月，买一双普通的鞋子70，寿命也是一个月。买一双亚瑟士慢跑鞋要380左右，寿命是几个月。一直想要的Redmibook 14是3999，能用3年。一个鼠标20左右，仿机械键盘30左右，真机械键盘100左右，手感有很大差异。两瓶大矿泉水可以喝一周，是10块钱，买一袋怪味胡豆要5块钱，一天就能吃完。一个月的房租是几千块钱，加上押金水电暖气管理服务费，平均下来也不少了。

我一个月的花销是多少呢，算不过来了，懒得算了。

## 编程

最上层的编程语言是脚本语言，往往没有严格的类型系统，基于其他语言实现的解释器编译执行。同样被广泛应用的是领域模型语言，他们是功能更加具体的语言。然后是高级语言，他们具有悠久的历史，优雅的设计，常常用来构建大型软件系统。高级语言会被编译器编译为汇编语言，汇编语言将载入处理器被执行。计算机的处理器根据指令改变引脚的电平，这些指令以电信号的形式与内存进行交互，完成寻址和计算的操作。

程序设计是在编程语言之上的应用，有了编程语言作为工具，就要用编程语言来完成一些作品。面向过程、面向对象和函数式编程是主流的程序设计方案。程序设计框架是开发者为了简化软件开发提供的一系列更加上层的工具。

常见的网站程序设计方案是前端+后端，前端离不开MVVM框架，后端离不开MVC框架，也可能将前端程序嵌入到后端程序中。为了加快任务处理，多线程是广为提倡的一个概念，也是对于新手的一个难点。当网站流量增大，需要保证服务能够稳定运行，高并发就成为了老手的难点，随之而来的是缓存、中间件等方案。当网站的数据量非常大，需要加快数据的处理速度，大数据的概念就应运而生，开源基金会提供了一整套大数据处理的框架。

随着技术的发展，程序员的工作内容逐渐由『开发』变成『组装』。以前是没有这样的东西，需要我们创造一个出来。现在市面上已经有了各种成熟的组件，我们只需要根据业务场景来搭建服务，能够满足业务需求。开源框架的作者们致力于让软件开发变得简单。如果你在开发过程中感觉有不舒服的地方，或许那个地方就差一个好用的框架，你可以试着完成这件事。

前沿技术一般来说有三个方向，大数据、人工智能和区块链。大数据就是通过集群的方式处理海量数据，一台服务器做起来费劲，就用十台服务器一起来做。大数据框架则是在提供给开发者搭建分布式数据处理集群的能力。人工智能是一个听起来比较高级的方向，我有一个同学，大学的时候C语言都搞不明白，考研上了个比较好的学校，现在也开始搞深度学习了……人工智能的核心并不是计算机技术，而是数学理论借助计算机实现了大量计算。至于区块链，它的核心自然也不是计算机技术，而是使用机制。不过话说回来，计算机技术不也是由各种『机制』结合组成的

吗？

## 了解更多

偶尔看看书，世界上其实有很多很多领域，关于群众心理的研究，关于人类语言的研究，关于社会行为的研究，关于简洁设计的研究，关于历史和未来的研究……

一年前在知乎上看到了一段话，感觉非常喜欢。至今，我仍然非常喜欢。所以我想分享一下：

谁让你读了这么多书，又知道了双水村以外还有个大世界……如果从小你就在这个天地里日出而作，日落而息，那你现在就会和众乡亲抱同一理想：经过几年的辛劳，像大哥一样娶个满意的媳妇，生个胖儿子，加上你的体魄，会成为一名出色的庄稼人。不幸的是，你知道的太多了，思考的太多了，因此才有了这种不能为周围人所理解的苦恼。

# Rust基础语法概述

2019-08-19

Rust是复杂度和应用场景都对标C++的语言，一起学习吧！

最近，我开始思考像本文这样类型的内容算什么，编程语言的教程？内容不全面；对语言的评价？够不着；学习笔记？如果是，那绝非我本意。我倾向于认为这是一个探索的过程，无论对于我自己还是对于别人，我希望可以表现出来的是，你看，新的编程语言没什么神秘的，它如此简单！有的程序员终其一生，都将某种语言作为自己职业头衔的前缀，“Java程序员”或是“后端开发”，我们该跳出这种怪圈。

## 语句

Rust必须以;结尾。

## 常量和变量

Rust使用let定义常量，使用let mut定义变量。这样的写法可能稍微有点奇怪：

```
fn main() {
    let x = 1;
    println!("{}", x);

    let mut y = 2;
    println!("{}", y);

    y = 3;
    println!("{}", y);
}
```

不同于其他语言的是，Rust允许在同一作用域中多次声明同一常量。也就是说，Rust里的常量虽然不可以被第二次赋值，但是同一常量名可以被多次定义。我们虽然能在系统层面明白常量和变量的区别，但是写法上稍微有点容易引起混淆。我多次给同一组符号赋值，这个符号不就是变量吗？

```
fn main() {
    let x = 1;
    println!("{}", x);

    let x = 2;
    println!("{}", x);
}
```

另一个有点奇怪的地方是，Rust的变量不允许重复定义。我们无法推测语言设计者的初衷，这明显不是为了允许重复定义而允许。也许，Rust中只存在常量，mut关键字的作用就是给常量一个可以被多次赋值的接口。没有mut，常量就是个常量，有了mut，常量就有了获得新值的“入口”。至于变量重复定义的问题，要啥自行车？

```
fn main() {
    let mut x = 1;
    let mut x = 2;
}
// warning: variable does not need to be mutable
```

## 控制流

Rust的条件部分不需要写小括号，和Go语言一样。谁先谁后呢？

```
fn main() {
    let number = 2;
    if number == 1 {
        println!("1")
    } else if number == 2 {
        println!("2")
    } else {
        println!("3")
    }
}
```

由于if语句本身是一个表达式，所以也可以嵌套进赋值语句中，实现类似其他语言三目运算符的功能。（Rust是强类型的语言，所以赋值类型必须一致。）

```
fn main() {
    let number = if true {
        3
    } else {
        4
    };
    println!("{}", number);
}
```

与Go语言简洁的多功能for循环相比，Rust支持多种类型的循环：

```
fn main() {
    loop {
        // ...
    }

    while true {
        // ...
    }

    let a = [1, 2, 3];
    for item in a.iter() {
        println!("{}", item);
    }
}
```

## 函数与值的传递

Rust似乎不存在值传递与引用传递的区别，因为Rust中全都是引用传递，或者分类为常量的传递与变量的传递。对比Java中字符串的创建，Rust中创建字符串也可以使用“声明对象”的方式：

```
fn main() {
    // 常量传递
    let a = String::from("a");
```

```

testa(a);
// 变量传递
let mut b = String::from("b");
testb(&mut b);
println!("{}", b);
}

fn testa(a: &String) {
    println!("{}", a);
}

fn testb(b: &mut String) {
    b.push_str(" b");
}

```

函数当然也是可以有返回值的，Rust中函数的返回值用->定义类型，默认将函数最后一行的值作为返回值，也可以手动return提前结束函数流程。需要注意的是，在最后一行用来作为返回值的表达式，记得不要加封号……

```

fn main() {
    let mut a = test();
    println!("{}", a);

    a = test2();
    println!("{}", a);
}

fn test() -> u32 {
    1
}

fn test2() -> u32 {
    return 2;
}

```

## 结构体

结构体的基本用法比较常规，没有new关键字，直接“实例化”就可以使用：

```

struct Foo {
    a: String,
    b: i32
}

fn main() {
    let t = Foo {
        a: String::from("a"),
        b: 1,
    };

    println!("{}, {}", t.a, t.b);
}

```

同样可以给结构体添加方法：

```

struct Foo {
    a: String,
    b: i32
}

impl Foo {
    fn test(&self) -> i32 {
        self.b + 1
    }
}

fn main() {
    let t = Foo {
        a: String::from("a"),
        b: 1,
    };

    println!("{}, {}, {}", t.a, t.b, t.test());
}

// a, 1, 2

```

## 列表与模式匹配

下面的例子创建了包含3个元素的向量，然后将第0个元素赋值给常量one。之后使用模式匹配判断列表的第0个元素是否等于one的值，如果相等则输出字符串“one”，否则为“none”。Rust的模式匹配中，Some()和None都是内置的关键字：

```

fn main() {
    let v = vec![1, 2, 3];

    let one = &v[0];
    println!("{}", one);

    match v.get(0) {
        Some(one) => println!("one"),
        Some(2) => println!("two"),
        None => println!("none"),
    }
}

```

## 错误处理

panic函数用于抛出异常：

```

fn main() {
    panic!("new Exception");
}

// thread 'main' panicked at 'new Exception', test.rs:4:3
// note: Run with 'RUST_BACKTRACE=1' environment variable to display a backtrace.

```

针对错误处理，Rust提供了两个简写的方法，用于便捷的处理错误信息。unwrap()函数会自动抛出panic，如果不使用unwrap()，程序则会跳过发生panic的代

码。这在某种程度上与Java的异常处理逻辑相反，因为Java如果不对异常进行处理，程序就无法继续运行。而Rust如果使用unwrap()对panic进行处理，程序将不再继续执行，同时打印出错误信息。

```
use std::fs::File;
fn main() {
    let f = File::open("hello.txt");
    println!("a");
    let f2 = File::open("hello.txt").unwrap();
    println!("b");
}
// a
// thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Os { code: 2, kind: NotFound, message: "系统找不到指定的文件。" }', s
// ...
```

另一个简写的方法是expect()，可用于替代unwrap()。它与unwrap()的区别在于，unwrap()使用系统内置的panic信息，而expect()可以传入参数作为panic的错误信息。仅此而已。

```
use std::fs::File;
fn main() {
    let f = File::open("hello.txt").expect("Failed to open hello.txt");
}
// thread 'main' panicked at 'Failed to open hello.txt: ...
// ...'
```

## Lambda表达式

Rust中的Lambda表达式使用|作为入参的界定符，即使用||来代替()。此外Lambda的公用和其它语言是相同的：

```
fn main() {
    let test = |num| {
        num == 1
    };
    println!("{} , {}", test(1), test(2));
}
// true, false
```

## 其他

Rust的语言特性远不止此，尤其是Rust与众不同的内存管理机制，以及让Rust新手得其门不得其道的概念“ownership”，都需要我们不断前行。

# 基于Java的爬虫框架WebCollector

2019-08-10

Long, Long Ago, 网络上出现大量Python爬虫教程, 各种培训班借势宣扬Python, 近几年又将噱头转向人工智能。爬虫是一个可以简单也可以复杂的概念, 就好比建造狗屋和建筑高楼大厦都是在搞工程。

由于工作的缘故, 我需要使用WebCollector爬取一些网页上的数据。其实宏观上, 爬虫无非就是访问页面文件, 把需要的数据提取出来, 然后把数据储存到数据库里。难点往往在于, 一是目标网站的反爬策略, 这是让人比较无奈的斗智斗勇的过程; 二是目标网页数量大、类型多, 如何制定有效的数据爬取和数据分析方案。

## 概述

这是一张简略的概念图, 受屏幕宽度限制, 可能无法看清内容, 请在新标签页打开图片, 或者直接点击[这里](#)。这张图片并不是完美的, 甚至还包含不完全正确的实现方式, 具体内容会在后面阐述。

我将目标网页分为4种类型:

1. 静态的网页文档, curl就可以加载到
2. 需要自定义HTTP请求的页面, 比如由POST请求得到的搜索结果页面, 或者需要使用Cookie进行鉴权的页面
3. 页面中包含由JavaScript生成的数据, 而我们需要的正是这部分数据。由于js是加载后才执行的, 就像CSS加载后由浏览器进行渲染一样, 这样的数据无法直接得到
4. 页面中包含由JavaScript生成的数据, 且需要自定义HTTP请求的页面

## 测试环境

为了便于测试, 在本地使用Node.js启动一个简单的服务器, 用于接收请求, 并返回一个页面作为响应。server.js的内容如下:

```
var http = require('http')
var fs = require('fs')
var server = http.createServer((req,res) => {
  // 返回页面内容
  fs.readFile('./index.html', 'utf-8', (err,data) => {
    res.end(data);
  });
  // 打印请求中的Cookie信息
  console.log(req.headers.cookie)
})
server.listen(9000)
```

index.html的内容更加简单, 只包含一个title和一个p标签:

```
<!DOCTYPE html>
<html>
<head>
  <title>This is a title</title>
</head>
<body>

</body>
</html>
```

## 静态页面

这是一个最简版的爬虫程序，在构造方法中调用父类的有参构造方法，同时添加url到待爬取队列中。visit是消费者，每一个url请求都会进入这个方法被处理。

```
public class StaticDocs extends BreadthCrawler {  
  
    public StaticDocs(String crawlPath, boolean autoParse) {  
        super(crawlPath, autoParse);  
        this.addSeed("http://127.0.0.1:9000/");  
    }  
  
    @Override  
    public void visit(Page page, CrawlDatums next) {  
        System.out.println(page.doc().title());  
        // This is a title  
    }  
  
    public static void main(String[] args) throws Exception {  
        StaticDocs crawler = new StaticDocs("crawl", true);  
        crawler.start(1);  
    }  
}
```

## Cookie鉴权

需要在header中带cookie请求同样简单，在构造方法中添加相应配置就可以，node.js的命令行会打印出cookie的内容：

```
public CookieDocs(String crawlPath) {  
    super(crawlPath, true);  
  
    // 设置请求插件  
    setRequester(new OkHttpRequester() {  
        @Override  
        public Request.Builder createRequestBuilder(CrawlDatum crawlDatum) {  
            return super.createRequestBuilder(crawlDatum)  
                .header("Cookie", "name=smallyu");  
        }  
    });  
  
    this.addSeed("http://127.0.0.1:9000/");  
}  
  
// name=smallyu
```

## JavaScript生成的数据

测试js生成数据的情况需要做一点准备，修改index.html，在body标签中加入这样几行代码：

```
<div id="content">1</div>  
<script>  
    document.getElementById('content').innerHTML = '2'  
</script>
```

可以预见，请求中直接返回的div内容是1，然后js经由浏览器执行，改变div的内容为2。访问静态页面的爬虫程序只能进行到第1步，也就是直接获取请求返回的内容。修改StaticDocs.java的visit方法，打印出div的内容看一下，可以确信是1：

```
System.out.println(page.select("div").text());  
// 1
```

这是一个官方提供的Demo，用于获取js生成的数据。WebCollector依赖于Selenium，使用HtmlUnitDriver运行js：

```
public class JsDocs {  
    public static void main(String[] args) throws Exception {
```

```

        Executor executor = (CrawlDatum datum, CrawlDatums next) -> {
            HtmlUnitDriver driver = new HtmlUnitDriver();
            driver.setJavascriptEnabled(true);

            driver.get(datum.url());

            WebElement divEle = driver.findElement(By.id("content"));
            System.out.println(divEle.getText());
            // 2
        };

        // 创建一个基于伯克利DB的DBManager
        DBManager manager = new RocksDBManager("crawl");
        // 创建一个Crawler需要有DBManager和Executor
        Crawler crawler = new Crawler(manager, executor);
        crawler.addSeed("http://127.0.0.1:9000/");
        crawler.start(1);
    }
}

```

如果你看过WebCollector的主页，一定可以注意到这个Demo和其他Demo的明显不同。在不需要js生成的数据时，新建的类继承自BreadthCrawler，而BreadthCrawler继承自AutoParseCrawler，AutoParseCrawler又继承自Crawler。现在获取js数据的Demo，直接跳过BreadthCrawler和AutoParseCrawler，实例化了Crawler。

为什么要这样做呢？再次强调，这是官方提供的Demo。

## Cookie鉴权后JavaScript生成的数据

根据官方提供的用例，显然是无法设置cookie的，因为Crawler类并没有提供自定义Header的方法。这个自定义Header的方法继承自AutoParseCrawler类。那么如何做到既可以添加Cookie又可以使用HtmlUnitDriver？

其实结果很简单，我在看过WebCollector的代码后发现AutoParseCrawler实现了Executor接口，并且在构造方法中将this赋值给了父类的executor。也就是说，AutoParseCrawler本身就是一个Executor。下面的代码用以表示它们的关系：

```

public class Crawler {
    protected Executor executor;

    public Crawler(DBManager dbManager, Executor executor) {
        // ...
    }
}

public class AutoParseCrawler extends Crawler implements Executor {
    public AutoParseCrawler(boolean autoParse) {
        // 这里的executor指向父类
        this.executor = this;
    }
}

```

new Crawler时传入一个executor，相当于直接new一个AutoParseCrawler。BreadthCrawler继承自AutoParseCrawler，所以BreadthCrawler本身也是个Executor。再看官方关于自定义Cookie的Demo，如何在其中使用HtmlUnitDriver呢？重写Executor的execute方法。

所以，在定义cookie后获取js生成的数据，使用继承BreadthCrawler的类，然后重写execute就可以。这是一个完整的Demo：

```

/**
 * @author smallyu
 * @date 2019.08.11 12:18
 */

```

```

public class JsWithCookieDocs extends BreadthCrawler {

    public JsWithCookieDocs(String crawlPath) {
        super(crawlPath, true);

        // 设置请求插件
        setRequester(new OkHttpRequester() {
            @Override
            public Request.Builder createRequestBuilder(CrawlDatum crawlDatum) {
                return super.createRequestBuilder(crawlDatum)
                    .header("Cookie", "name=smallyu");
            }
        });
    }

    this.addSeed("http://127.0.0.1:9000/");
}

// 直接重写execute即可
@Override
public void execute(CrawlDatum datum, CrawlDatums next) throws Exception {
    super.execute(datum, next);

    HtmlUnitDriver driver = new HtmlUnitDriver();
    driver.setJavascriptEnabled(true);

    driver.get(datum.url());

    WebElement divEle = driver.findElement(By.id("content"));
    System.out.println(divEle.getText());
    // 2
    // 同时，node.js的命令行中打印出cookie内容
}

// 重写execute就不需要visit了
public void visit(Page page, CrawlDatums crawlDatums) {}

public static void main(String[] args) throws Exception {
    JsWithCookieDocs crawler = new JsWithCookieDocs("crawl");
    crawler.start(1);
}
}

```

## 外部代理

也许还没有结束。在一开始概述的图片上，同时定义cookie以及获取js生成的数据，实现方式是内部Selenium + 外部browsermob-proxy。假设没有上述重写execute的方法（官方也确实没有提供类似的Demo），该如何实现想要的效果？一种实践是本地启动一个代理，给代理设置好cookie，然后让Selenium的WebDriver通过代理访问目标页面，就可以在带header的情况下拿到js生成的数据。这是在JsDocs.java的基础上，使用代理的完整实现：

```

public class JsWithProxyDocs {
    public static void main(String[] args) throws Exception {
        Executor executor = (CrawlDatum datum, CrawlDatums next) -> {

            // 启动一个代理
            BrowserMobProxy proxy = new BrowserMobProxyServer();
            proxy.start(0);
            // 添加header
            proxy.addHeader("Cookie" , "name=smallyu");

            // 实例化代理对象
            Proxy seleniumProxy = ClientUtil.createSeleniumProxy(proxy);
            // 由代理对象生成capabilities
            DesiredCapabilities capabilities = new DesiredCapabilities();
            capabilities.setCapability(CapabilityType.PROXY, seleniumProxy);
            // 内置，必须设置
            capabilities.setBrowserName("htmlunit");
        }
    }
}

```

```
// 使用capabilities实例化HtmlUnitDriver
HtmlUnitDriver driver = new HtmlUnitDriver(capabilities);
driver.setJavascriptEnabled(true);

driver.get(datum.url());

WebElement divEle = driver.findElement(By.id("content"));
System.out.println(divEle.getText()); // 2
};

//创建一个Crawler需要有DBManager和Executor
Crawler crawler = new Crawler(new RocksDBManager("crawl"), executor);
crawler.addSeed("http://127.0.0.1:9000/");
crawler.start(1);
}
}
```

## 其他

对于WebCollector我已经没有兴趣了解更多，倒是在注意到框架的包名`cn.edu.hfut`后有种豁然开朗的感觉。凌乱的代码风格，随处可见不知所以的注释，毫无设计美感的代码架构，倒也符合国内不知名大学的开源软件水平，距离工业级的框架，可能还需要N个指数倍东的时间。至于使用过程中遇到`depth`含义不明、线程非法结束、`next.add`失效等问题，就这样吧，也在情理之中，整个框架都像是赶工的结果，或者说是学生们拿来练手的项目。我在WebCollector的Github上RP了关于重写`execute`的问题，从开发者回复的只言片语中，我怀疑开源者自己都没有把里面的东西搞清楚:P

# 工作能力低下的表现

2019-07-25

现在是零点零五分左右，本来是应该睡觉的时间，却突然想写点东西。

- 抱怨

时时刻刻在抱怨，这个不对，那个不好，并且将不满实实在在的挂在脸上，从嘴里说出来，生怕别人不知道有意见 :)

- 无法承受工作压力

抱怨的根源是工作内容，工作量稍微多一些就会开始抱怨，活好多，我好烦，我做不完，我做不过来，我没有时间。其实只不过是正常的工作量而已，其实不用加班，其实摸鱼晒网谁都喜欢 :)

- 情绪化

面对同事提出的问题，常见做法是不予理睬，稍加严重的做法是争锋相对。最让人诧异的是，生气的原因竟然可以是“他不尊重我的劳动成果”。打工而已，劳动哪有成果 :)

- 自我为中心

正常工作中的分工合作感觉像是去政府求人办事，要在适当的时候，方便的时候，有时间的时候，心情好的时候，不然可能会看到不太好的脸色。我曾以为众生平等 :)

- 你的错，我的错

盼天盼地盼月亮盼星星，是你的错，不是我的错，这个问题需要你改，而不是需要我改，有事找你别找我，你能解决的就尽量不要让我动手 :)

- 专业能力不足

- 模块化能力欠缺

功能组件化，模块抽象化。不懂后端架构的前端工程师不是好码农，由于缺乏对整个项目结构的思考，复用能力太差，导致需求变更带来大的工作量，大的工作量会引起第一点 :)

- 编程能力欠缺

会写代码，但不会编程。缺乏基本的计算机常识，遇到功能点显得盲目、无力，由于自身能力的限制导致工作难度增大，工作时间加长，也就是工作量增大，大的工作量会引起第一点 :)

- 对自己认知不足

会做什么，不会做什么？ :)

- 思想上的巨人

思想上的巨人，行动上的矮子。即使知道自己有欠缺，口口声声说甚至有行动作出改变，但是收效甚微。一般来说，造成这种现象的原因是，没动脑子 :)

- 盲目要强

拒绝别人的帮助，希望别人以更加友善、符合自己能力的方式迁就合作。在意“自己”完成

任务，而不是自己“完成”任务。

- 目的不明确

工作是为了什么？既然如此痛苦，为什么不换个行业？:)

- .....

我没有任何恶意，只是描述一些客观现象。我可怜能力不足的人，面对超出自己能力范畴的工作时会感到无助，但我也难受，和这样的人合作也许是噩梦。我会尽量避开那些我不太喜欢的点，告诫自己不要也变成那样。让自己变好，然后拥有讨厌别人的资格。现在一点零五分左右，我要睡觉了。

## 更新

时间过去了将近一个月，我似乎开始理解某些行为和现象。

事情是这样，最近，我试着使用C#开发一个简单的桌面应用程序，我的想法很简单，自定义几个快捷键，然后让快捷键代替重复多次的退格键。比如，全局热键Ctrl-5相当于按下退格键Backspace5次，Ctrl-6则相当于按6次。

这个想法产生于我日常敲键盘的过程，由于双手打字速度较快，类似于多线程的无序执行，经常出现拼音中字母错位的现象，然后手快又按下了空格键，就会有一长串错误的词组上屏。每次都按N次退格键是一件比较让人不爽的事情，尤其是流畅的思路被打断，感觉啪啪的退格键既影响了正常打字的顺畅，又有一种不得不为自己之前犯下的小错误负责任的愤恨。

原先以为只要向系统注册一组全局按键，然后模拟键盘的输入，整个过程就这么简单。然而，在实际的开发过程中，我遇到了很多困难，一开始连WPF中的Window和Form都没有区分清楚，Google到的教程提到了多种注册快捷键的方法，但是真正可用的少之又少，对于按键的模拟也只能依靠搜索引擎，因为官方文档中提到的Keys.send()之类的方法并不起作用（也许是事件等级太低）。

在能够实现基本的注册快捷键和模拟按键功能后，我遇到了新的问题。快捷键是基于Ctrl，在按下快捷键的同时，程序就会执行，也就是模拟Backspace，这个时候Ctrl键还没有松开，系统会认为我按下了Ctrl-Backspace，这显然不是预期的效果。然后我将Ctrl换为Alt，最终确定Shift键和Backspace组合没有冲突。但是新的问题又来了，程序模拟的按键最多触发两次，所以实际效果最多退两格，无论我循环多少次或者改变nInPints参数的值，都没有解决这个问题。

所以，我并没有在表达开发一个那样的东西很难，只是说遇到了一些我不太擅长解决的问题。也是在那个过程中，我体会到了“我好烦”、“这个好难”、“真闹心”、“没那么简单”之类的情绪。

也许，我们应该多一些理解。

# Kotlin：简化版的Scala

2019-07-06

行走江湖的剑客，必然要有一柄趁手的宝剑。好的程序语言就像一把好剑，重量合适，拿着舒服，挥舞起来优雅，杀伤力过关。Kotlin官方对待Kotlin和Scala的关系是，“如果你玩Scala很happy，那你就不需要Kotlin。”

## 脚本化

Scala执行的基本单位和Java一样是类，而Kotlin允许文件中的main方法直接运行，不需要类。Java的入口函数定义在类中：

```
public class Java {  
    public static void main(String[] args) {}  
}
```

Scala的入口函数定义在样本类而不是普通的类中：

```
object Scala {  
    def main(args: Array[String]): Unit = {}  
}
```

Kotlin的入口函数则直接定义在.kt文件中，相应的，Kotlin的类仅相当于一种数据结构，类中无法定义入口函数：

```
fun main(args: Array<String>) {}
```

## 构造函数与单例模式

Kotlin的构造函数同Scala一样写在类定义处，因此也无法像Java的构造函数一样直接写入初始化代码。Kotlin中使用init代码块来执行初始化程序：

```
class Test(arg: String) {  
    init {  
        println("This string is ${arg}")  
    }  
}  
  
fun main(args: Array<String>) {  
    val test = Test("smallyu")  
}  
  
// This string is smallyu
```

如果需要第二个构造函数，就要使用类似ES6的constructor函数，或者类似Scala的辅助构造器。这实在是丑陋的写法，相比之下Java真的友善多了。

```
class Test(arg1: String) {  
    init {  
        println("This string is ${arg1}")  
    }  
    constructor(arg2: Int): this("smallyu2") {  
        println("This int is ${arg2}")  
    }  
}  
  
fun main(args: Array<String>) {  
    val test = Test(1)  
}  
  
// This string is smallyu2
```

```
// This int is 1

Kotlin的构造函数是需要用constructor关键字定义的，默认可以省略，但如果要加权限修饰符自然就不能省了。在Kotlin中实现单例模式的思路与Java相同，让构造器私有，然后通过静态方法暴露实例：

class Test private constructor() {
    companion object Factory {
        fun create(): Test = Test()
    }
}

fun main(args: Array<String>) {
    val test = Test.Factory.create()
}
```

Kotlin中的object定义静态代码块，companion允许在类内部定义静态代码块，因此companion object定义了类外部可以访问的方法create()。

## getter和setter

Kotlin另一个有趣的玩意儿是getter和setter。前端框架React或Vue实现数据双向绑定的原理即使用Object.defineProperty()定义对象的getter和setter，使得对象的变化可以实时同步到页面上。Kotlin提供了对属性getter和setter的支持：

```
var test: Int
    get() {
        println("There is test getter")
        return 2
    }
    set(arg) {
        println("The setter arg is ${arg}")
    }

fun main(args: Array<String>) {
    println(test)
    test = 3
}

// There is test getter
// 2
// The setter arg is 3
```

## 其他

开始对Kotlin感兴趣是因为发现Kotlin竟然支持协程，如果Kotlin真的有语言级别的协程支持，加上运行在Jvm上的特点，以及能够开发多平台应用包括Server Side、Android、JavaScript、Native，那Kotlin无疑是异常强大的编程语言。然而事实上Kotlin的协程只是一个扩展包，甚至还需要使用编译工具来引入，对协程的支持还是Go语言独大。用于JavaScript平台也是个幌子，并没有比TypeScript好用，至于Android和Native本身也是Java的应用场景……

Kotlin提供了许多语法糖，看似可以简化程序员的代码量，但是为了熟练应用Kotlin的特性，使用者又不得不搞清楚类似data class的概念，就像Scala的case class一样。Kotlin的学术性弱于Scala，工程能力又不比Java有很大的优势。Go语言虽然另辟蹊径，语言特性上有广为诟病的地方，但是看着爽，写着也爽。所以Kotlin和Scala一样，并不会有广泛的应用前景。也就是说，它并不会是下一个很流行的编程语言。

# 大学毕业是什么体验

2019-06-30

李诞写的一本书《笑场》的序言里有一句话：“人生确实没有意义，但人生有美”。

## 相同的错误

其实我早就“毕业”了，只是现在才领证书。当我没有挂科记录的时候，当我开始实习的时候，我的学生时代就已经结束了，虽然好像不曾开始过。学业上，可能唯一的遗憾是英语四级（CET 4）没过。也许你会觉得可笑，我也感觉可笑。临近毕业，学校有一个大学四年的综测成绩，我期末考试从没考过第一，但四年下来的综测竟然可以在班里排第一。当然，我们班成绩整体很差，但这不重要，重要的是将近1/2的人可以考过425，我却只有423。

大学刚入学的几天，我曾在笔记本上写过几天日记，整理东西的时候翻开，其中有一句话让我感到吃惊：“我是来学习的，不是来娱乐的”。我自然可以理解当时是什么心情，高三的一年时间里，只有我自己知道我做了什么。高三开学摸底考，我考班里倒数十几，直到后来各种模拟考高考，我可以保持班里前三。不要对这样的排名感到诧异，这就是小县城的真实情况。也正因为这样的经历，我怀着不那么正规的求学的心，开始了大学生活。

就在前段时间，我陷入了同样的思维困境：“我不是来娱乐的，我是来工作的”。因为没有带着游玩的心上大学，所以大学的时光里，我的成绩可能不那么差，技术能力也有所增长，总体而言，是远超部分同学的。但也正因如此，我错过了很多娱乐的机会，甚至我大学所在的城市，都完全不了解，景点、场所等。更重要的是，我错过了非常重要的人，错过了曾经弥足珍贵的机会。现在我已经学会如何爱一个人，可是没有人再需要我去爱了。

所以你可以理解标题“相同的错误”是指什么。我从半个月前开始规划要写的内容，原标题是“关于职业生涯的规划”，后来觉得职业生涯这个范围太小了，无非是认真工作、好好生活、工作和生活平衡。事实上生活本就不需要规划，规划了也没用，我们应该时刻保持清醒，现状如何，未来会怎样，想要的是什么。总的来说，该有的想法是：“我不是来混日子的”。

## 一拳超人

我换了个头像。他说，“我变秃了，也变强了”。琦玉因为感兴趣而当英雄，不是为了被更多人认识和崇拜，也不是为了英雄协会的丰厚待遇，仅仅是因为想当英雄所以当了英雄。可以说，我从事某个职业，也不是为了别人眼中的各种评价标准，仅仅是因为我从初中就开始积累的某些兴趣，让我想要在这条路上继续探索前行。

琦玉的强大是莫名其妙的，即使他把变强的过程全盘托出，你也无法达到他的高度。就好比学霸告诉你他上课如何听讲，作业如何完成，你也无法成为下一个学霸。一拳超人变强大唯一的代价，就是头发变秃了，直接变成了光头。我发现这一点和程序员类似，如果一个程序员因为工作把头发搞秃了，那他往往是大牛，并且即使他把方法告诉你，你也无法轻易超越。不过即便如此，也并不应该阻碍我们对强大能力的向往。

## 毕设和微服务

我的毕设以微服务为标题，但由于我们学校对毕设和论文的要求之低，我并没能完成自己心里的目标，只是应付老师，蒙混过关了。微服务本身是一个很简单的概念，难点往往在于如何在大型系统中实现微服务。在大型系统中进行任何操作都步履维艰，微服务并不特殊，因此，微服务是简单的。关于注册中心、配置中心、负载均衡、断路器什么的，可能在不久的将来，这个博客上就会出现关于它们的文章。

学习应该关注本质，是抽象的，我更希望从高的层面去理解技术，如果只是学习Kafka怎么使用而不关心像Kafka这样的中间件都在解决什么问题，这个世界就太无聊了。结合实际工作的内容，我的下一步计划会聚焦在中间件、区块链和算法上。

## 博客存在的意义

(写到这里，发现自己没有耐心继续写下去。有些内容的输出需要同理心，也就是“当时是什么想的”。这半个月来发生了很多事情，但是现在已经结束了，并且我将面临新的生活和挑战。我的心态回不到过去，也不愿停留在过去。可能剩下的内容将无法完成。)

(这个附件 [关于职业生涯的规划](#) 是之前写的大纲。)

# JavaScript有关联数组吗?

2019-05-18

如果你接触过PHP，那你对关联数组一定不陌生。C或Java中数组下标都是从0开始的数值，而PHP除了数值，还可以用字符串作为数组的下标。用数值做下标的数组叫做索引数组，用字符串做下标的数组叫做关联数组，他们都是合法的数组。

```
<?php  
$arr[0] = 1;           // 索引数组  
$arr["a"] = "b";      // 关联数组  
  
echo $arr[0];         // 1  
echo $arr["a"];       // b
```

在JavaScript中，同样可以使用字符串来作为数组的下标：

```
let arr = []  
arr[0] = 1  
arr['a'] = 'b'
```

昨天，我和漂亮同事在使用JavaScript中用字符串做下标的数组时，遇到了令人困惑的问题。

## 缘起

在Express.js框架的路由处理中，用res.json()返回数组，下标为数值的数组可以正常返回，下标使用字符串的数组却始终返回空。这是一段最简代码，可以用来描述该过程：

```
app.get('/', (req, res) => {  
  let arr = []  
  arr['a'] = 'b'  
  
  console.log(arr) // [a: 'b']  
  res.json(arr)    // []  
})
```

预期返回的数组arr包含1个元素，console.log()直接在命令行打印的文本内容是[a: 'b']，和预期一致，然而如果通过页面请求路由，返回的内容是[]，这是匪夷所思的，也就是说res.json()把数组的内容吞掉了。

## 探寻

为了寻找问题的真实原因，我在框架的中找到res.json()方法的定义：

```
res.json = function json(obj) {  
  var val = obj;  
  // ...  
  var body = stringify(val, replacer, spaces, escape)  
  // ...  
  return this.send(body);  
};
```

返回内容body经过了stringify()方法处理，stringify()方法调用的是JavaScript中JSON标准库的方法JSON.stringify()：

```
function stringify (value, replacer, spaces, escape) {  
  var json = replacer || spaces  
  ? JSON.stringify(value, replacer, spaces)  
  : JSON.stringify(value);  
  // ...  
}
```

那么就说明，`JSON.stringify()`方法的返回值，会忽略用字符串做下标的数组。为了证实这一现象，用简单的Demo测试一下：

```
let arr1 = [], arr2 = []
arr1[0] = 1
arr2['a'] = 'b'

JSON.stringify(arr1)    // "[1]"
JSON.stringify(arr2)    // "[]"
```

所以问题又来了，JavaScript标准库中的`JSON.stringify()`方法，为什么要忽略数组中下标为字符串的元素？是有意为之，官方不赞成使用字符串做下标，还是无奈之举，存在不可抗拒的原因无法实现？为了找到问题的根源，我试着从Chrome解析JavaScript的[V8引擎](#)中寻找`JSON.stringify()`的定义。

V8引擎是用C++写的，关于`JSON.stringify()`的定义应该是这一段代码：

```
// ES6 section 24.3.2 JSON.stringify.
BUILTIN(JsonStringify) {
  HandleScope scope(isolate);
  JsonStringifier stringifier(isolate);
  Handle<Object> object = args.atOrUndefined(isolate, 1);
  Handle<Object> replacer = args.atOrUndefined(isolate, 2);
  Handle<Object> indent = args.atOrUndefined(isolate, 3);
  RETURN_RESULT_OR_FAILURE(isolate,
                           stringifier.Stringify(object, replacer, indent));
}
```

可以推测出，`object`即`JSON.stringify()`处理并返回的内容，返回之前使用`args.atOrUndefined()`方法进行包装。这里`atOrUndefined()`被反复调用，传入两个参数，可以理解为，第一个参数`isolate`保存有完整的参数信息，第二个参数是数据的索引，结合起来便是`atOrUndefined()`方法要处理的完整数据。

然后看`atOrUndefined()`的定义，在下面的代码中，`tOrUndefined()`调用了`at()`方法，`at()`方法又调用了`Arguments::at`方法：

```
Handle<Object> atOrUndefined(Isolate* isolate, int index) {
  if (index >= length()) {
    return isolate->factory()->undefined_value();
  }
  return at<Object>(index);
}

Handle<S> at(int index) {
  DCHECK_LT(index, length());
  return Arguments::at<S>(index);
}
```

`Arguments::at()`方法中，指针`value`获取了待处理参数的内存地址，然后使用`reinterpret_cast`对`value`的值进行类型强转。

```
Handle<S> at(int index) {
  Object** value = &((*this)[index]);
  // This cast checks that the object we're accessing does indeed have the
  // expected type.
  S::cast(*value);
  return Handle<S>(reinterpret_cast<S**>(value));
}
```

到这里值就返回了，但是并没能解释为什么使用字符串做下标的数组内容会被忽略。只要是同一个数组，它的值就会保存在一段连续的地址空间中，即使`reinterpret_cast`处理的是指针变量，也应该无论多少都照常输出才是。

## 真相

最后，通过Google找到了一个关于数组使用字符串做下标的问题和答案（[String index in js array](#)），我才明白为什么字符串做下标的数组如此特殊，因为JavaScript里压根就没有关联数组！

```
let arr1 = [], arr2 = []
arr1[0] = 1
arr2['a'] = 'b'

arr1.length      // 1
arr2.length      // 0
```

给一个数组使用字符串作为下标赋值后，数组的长度不会改变，赋的值并没有作为数组元素储存到数组里。使用字符串作为下标能够正常对数组取值赋值的原因是，JavaScript将字符串作为数组的属性进行了储存。

```
let arr = []
arr['a'] = 'b'

arr.hasOwnProperty('a')    // true
```

因此，JSON.stringify()处理的是数组的内容，reinterpret\_cast也只是基于指针对数组内容进行类型转换，属性什么的，当然不会有输出！

## 后续

1. 为什么console.log()可以将数组的属性也输出？对于要输出的内容，它是怎么定义的？
2. 为什么JavaScript中`typeof []`的值是"object"，也就是数组的类型是对象，但对象的属性会被处理，而数组不会？

# 主流编程语言的异常处理机制

2019-04-24

学习编程语言应该从语言特性入手，而不是编程语言本身。这里尝试对各种编程语言的异常和错误处理机制做一个横向的、简单的了解。涉及到的编程语言包括C、C++、Go、Java、Scala、Kotlin、Ruby、Rust、JavaScript、PHP、Python、Lisp。

## C

C语言没有异常捕获机制。程序在发生错误时会设置一个错误代码errno，该变量是全局变量。C语言提供了 perror() 和 strerror() 函数来显示与 errno 相关的描述信息。perror() 函数可以直接调用，入参是一个字符串，输出入参：错误文本。strerror() 函数入参是一个数字（错误码），返回一个指针，指针指向错误码对应的文本。

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

void main ()
{
    // 打开一个不存在的文件，会发生错误
    fopen ("unexist.txt", "rb");

    // 2
    printf("%d\n", errno);

    // No such file or directory
    perror("");
}

// No such file or directory
printf("%s\n", strerror(errno));
```

## C++

C++ 支持异常捕获机制。C++ 可以抛出或捕获两种内容，一种是 int 或 char\* 之类的内容，程序可以捕获并抛出，这一点和 Java 相比有差异，因为 Java 并不支持直接抛出基本类型的异常：

```
#include <iostream>
#include <exception>
using namespace std;
int main () {
    try
    {
        throw "error";
    }
    catch(const char* msg)
    {
        cout << msg << endl;
    }
}

// error
```

另一种内容就是类，可以是内置的标准异常类，或是自定义的异常类：

```
#include <iostream>
#include <exception>
using namespace std;
int main () {
    try
    {
```

```

        throw exception();
    }
catch(std::exception& e)
{
    cout << e.what() << endl;
}
}

// std::exception

```

## Go

Go语言作为非OOP派系的编程语言，并不支持try-catch的语法，但仍然具有类似抛出和捕获的特性。Go语言有3个错误相关的关键字，panic()、recover()和defer。可以理解为，panic()函数抛出异常，recover()函数捕获异常，defer关键字定义最后也就是finally执行的内容：

```

package main
import "fmt"

func main() {
    defer func() {
        err := recover()
        fmt.Println(err)
    }()
    panic("error")
}

// error

```

## Java

Java是纯粹的OOP语言，仅支持对象的抛出和捕获：

```

public class ErrorTest {
    public static void main(String[] args) {
        try {
            throw new Exception();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

// java.lang.Exception

```

## Scala

Scala和Java是一个流派，同样仅支持对象的抛出和捕获，除了语法上和Java稍有差异，概念上基本是一样的：

```

object ErrorTest {
    def main(args: Array[String]): Unit = {
        try {
            throw new Exception()
        } catch {
            case e: Exception => print(e)
        }
    }
}

// java.lang.Exception

```

另外，Scala抛出的是Java的异常，也许Scala不能算作是独立的编程语言，而是依附于Java、为Java提供语法糖的编程语言。这一点值得深入思考和探究。

## Kotlin

Kotlin和Scala是一种性质的语言， 默认抛出的同样是Java的异常：

```
fun main(args: Array<String>) {
    try {
        throw Exception()
    } catch (e: Exception) {
        print(e)
    }
}

// java.lang.Exception
```

## Ruby

Ruby使用关键字raise和rescue代替try和catch来实现异常的抛出和捕获。Ruby同样支持try-catch关键字，这里暂不讨论，因为我没搞清楚它的用法。

```
begin
  raise "error"
  rescue Exception => e
    puts e
end

// error
```

## Rust

Rust没有try-catch的语法，也没有类似Go的错误处理函数，而是用对错误处理进行过包装的Option<T>或Option的加强版Result<T, E>进行错误处理。Rust的模式匹配和Scala类似：

```
fn main() {
    match find() {
        None => println!("none"),
        Some(i) => println!("{}", i),
    }
}

fn find() -> Option<usize> {
    if 1 == 1 {
        return Some(1);
    }
    None
}

// 1
```

## JavaScript

脚本语言在变量类型上不做强制约束，捕获时也就不能按照异常类型来做区分。抛出错误的内容还是相对自由的：

```
try {
    throw 1
} catch (e) {
    console.log(e)
}

// 1

try {
    throw new Error('')
} catch (e) {
```

```
    console.log(e)
}

// Error
```

## PHP

PHP的try-catch和Java类似，并没有特殊之处：

```
<?php
try {
    throw new Exception("error");
} catch (Exception $e) {
    echo $e->getMessage();
}
```

## Python

Python在语法上能找到Ruby的影子，raise触发异常，except捕获异常：

```
try:
    raise
except:
    print("error")
```

## Lisp

Lisp整体较复杂，Lisp捕获处理异常的内容暂时留坑。以下是Common Lisp触发错误的情形之一，declare会声明函数入参类型，传入错误参数将引发错误：

```
(defun df (a b)
  (declare (double-float a b))
  (* a b))

(df "1" 3)

// *** - *: "1" is not a number
```

## 后续

原先想梳理这些语言的大部分异常和错误处理相关概念，然而真正开始后发现比较困难，并且之前我没能区分“exception”和“checked exception”，以致从立意到标题到内容可能都有偏差。这次就先提及“exception”，之后讨论关于“checked exception”的内容。

# 实习3.5个月之际

2019-04-21

想用《卧虎藏龙》里的台词来描述心情：“被一种寂灭的悲哀环绕”。

## 工作

最近实际的工作产出很少，大多时间用在了熟悉业务上。不出所料，工作中用到的技术并不会超出想象，因此可以安心地认为目前还在及格线上。倒是业务比较复杂，对没有经验的人显得不太友好。线上的交易系统，生产上的数据库字段，真真切切对应具体的业务流程和操作。毫不知情地开发一时爽，等到上线就火葬场。

有经验的程序员做事细致谨慎，也确实是给我上了一课。在学校曾有人问我一些问题，我觉得很简单，然后直截了当的解决问题。现在到工作岗位上，我就是被解决问题的那个人。我之前思考过，我的同学技术水平如何？我的技术能力是否真的高过他们？如果有好过他们的地方，是什么？有核心竞争力吗？这些问题的答案，基本上都聚焦在“经验”上。经验是无法快速获得的，我的同学不行，我也不行。

我的经验来自爱好和习惯，也许是碰巧接触互联网较早，仅此而已。王垠提到过[经验和洞察力](#)的问题，观点精辟，值得深思。我没有所谓的洞察力，并且洞察力类似于学习能力、理解能力，是一个人综合素养的体现，无法轻易获得。没有足够的阅历，怎么能透过现象看到本质？郭德纲说的很好，让人成长的不是时间，是经历。可以理解为，时间助长经验，经历助长洞察力。

## 人际

有件事情带给我一些思考。我请假几天，但有工单提测了没有测完。我说要离开几天时，测试同事问怎么要走云云，我说过几天就回来了，他说，“我不关心这个，我关心的是我工单怎么测……”倒不是他说这句话会让人伤心，我们本就毫无交集，认识没几天，同事之间往往只有工作关系，我也是清楚的。我在意的是，他竟然可以直截了当直击要害地说出自己的需求，这是一种需要能力也需要胆量的做法。

对话发生在几秒钟的时间里，能够快速组织语言描述清楚自己的想法，是我现在不具备的能力。需要胆量的说法是指，这样的做法稍微有点不符合国人的行事风格，客套话、顾面子、话里有话、笑里藏刀，都是中华民族的传统品德。

## 比金钱更重要的东西

看到朋友圈晒工作晒起薪，各种酸。但是就目前而言，我没有丝毫寻找新机会的冲动。融洽的工作氛围，压力并不大的工作内容，能让人有成长的业务需求，现在的工作足够让我满意，除了工资。我曾经的老师个性签名是，成长比成就重要。这句话并不绝对，但有道理。对于刚参加工作的情况来说，有多少挣钱的能力比能挣到多少钱重要，能力可以持续转化为金钱，一时的金钱却无法保证能力的持续增长。

996是一种选择。罗辑思维有过一种观点，就是大脑处于空闲状态的时候，会不自觉的考虑各种人际关系，因此一些现象可以这样解释：聪明人往往易怒、情商低，是因为他们把大量时间用在了目的性较强的问题，留给大脑的空闲时间少了，也就没有太多关于人际关系的思考；某个一线互联网公司内流行冥想活动，就是坐那儿什么也不干，此时虽然没有明确的想法，但大脑会不自主的东想西想，谁对我怎么样，我对谁怎么样，等等。这也正好是修福报之余衍生出来的、可能人类本身就需要的活动。

我开玩笑的和同学说，我的公司是996，同学反问，真的？你们公司是996？我能在言语之前感受到他开心的情绪。因为他在某N线城市找了一份工作，工资不高，但除去开销，能剩下的和我差不多。抛开五险一金的额度不谈，生活质量、圈子、眼界，以及上面提到的成长空间，都是我所在乎的。我突然对单纯的薪资比较感到厌恶，不要再用某种“小气”的思维方式推测我的处境了，我们需求不同，理想不同。我对某个世界充满兴趣，会亲身去探索，麻雀怎么会知道凤凰的志向

呢？

## 获取有效信息的途径

N年前，很多人努力读书上更好的学校，为的是更好的教育资源，甚至是结识更多“高端人才”。如今，因为互联网的存在，某些各种维度的距离已经不再遥远。你想要获取的很多信息，都可以通过网络得到，而且不只是书本书面知识，还包括在某些领域长期的经验和对一些问题独特的见解。

所以我能更加信任自己的决策而不是父母或者其他长辈，因为我和他们相比，虽然有年龄和人生经历上的不足，但由于我可以利用互联网来不断获取各种信息，我对某些事情的决策可能比他们更加正确。

当然，也并非所有会上网的人，都可以有效利用互联网来获取信息。从广泛的信息网络中搜查、筛选、分析、推断出自己想要的东西，很重要。

# 电梯楼层显示屏的设计失误

2019-03-27

每次上楼的时候都感到困惑，有一个问题我注意很久了。

## 电梯

其实没什么大不了的，目前某座楼里的电梯是这样的情况。电梯从楼上到1楼的过程中，楼层显示屏会显示向下的箭头↓，当电梯到达1楼，电梯门已经开了，箭头仍然向下。这时不少人会认为，电梯将要向下运行，到达地下室。

这种想法很正常，因为电梯存在另一个设计失误。当有人等待电梯想要从10楼到达1楼，同时正好有人乘坐电梯从1楼到达20楼，电梯会在10楼停下并开门。此时无论想到1楼的人是否上电梯，电梯都会继续向20楼行驶。在10楼会开门单纯是因为，在10楼想要到达1楼的人按下了电梯的按钮。

正因为电梯向上行驶的这种行为，很容易让人推测，电梯向下行驶也会存在这样的行为。在1楼看到箭头向下，以为电梯本是要到地下室，因为按了电梯的按钮，或正好有人在这一层出电梯，门就开了。然而事实是，箭头只是保留了之前电梯运行的状态（向下），至于接下来是向上还是向下，要看第一个进入电梯的人按了哪个楼层。

并非（绝不是）所有的电梯都这样。正确的做法应该是，当电梯没有运行任务时，显示屏不显示箭头，或者显示置空（-）之类的内容。

## 程序

当一个任务执行结束（电梯运行结束），应该把任务的标志位重置到初始状态（箭头置空）。这样的逻辑在程序里很常见，不管是UI还是内存，用户不可信，GC亦不可信，手动回收大法好。今天就踩了一个坑：(

坑很简单，一句话就能表达：循环里执行SQL语句，Statement对象没关闭。

坑虽然简单，但在生产环境上影响了正常的业务操作。DB2的默认限制为1400-，之后就会报错。开发的时候要小心，自测的时候也应该注意大量数据的情况。此类错误，希望以后不要再犯！

# Go语言基本语法

2019-03-15

Go语言虽然在语言设计上不被王垠看好，但它如此简洁的代码结构确实让人着迷。

## 语句

Go语言语句结尾不需要;。

## 变量和常量

使用var声明变量。当变量需要初始化时，可以使用赋值符号:=代替=以省略var关键字。

```
var a int
var b string

var c int = 10
var d = "golang" // 编译器自动推断类型
d := 10
```

与C语言或Java不同，Go语言的类型声明在变量右侧。需要注意的是，如果程序中声明的变量未经使用，程序将无法通过编译。Go语言是一种工程化的语言，因此它的一些特性让人感觉不可理喻，但又会在实际工程中提高效益。

Go语言的变量赋值支持一些炫酷的写法，比如要交换变量x和y的值，可以使用这种违反直觉的写法：

```
x, y = y, x
```

Go语言中使用const定义常量，true、false和iota是预定义常量。其中iota稍显特殊，iota会在每一个const关键字出现时重置为0，然后在下一次const出现前，每出现一次iota，iota的值加1。

```
const a = iota // 0
const b = iota // 0
const (
    c = iota // 0
    d = iota // 1
)
```

## 数组和切片

声明一个元素个数为3的数组，并初始化：

```
array := [3]int{0, 1, 2}
array[0] = 3
fmt.Println(array)
```

和其他语言一样，Go语言在声明数组后并不能改变数组的大小。所以Go语言提供了像Python一样的切片。切片可以从数组中产生，也可以使用make()函数新建。

```
array := [3]int{0, 1, 2}
slice1 := array[:2] // 从数组中创建

slice2 := make([]int, 3) // 直接创建

fmt.Println(slice1) // [0 1]
fmt.Println(slice2) // [0 0 0]
```

除切片外，映射也是使用make函数创建，映射的类型全称是var myMap map[string] int，意为

声明变量myMap，key为string，value为int。

## 流程控制

Go语言允许if-else语句的条件表达式不加小括号，当然加上也无妨。

```
a := 1
if a == 1 {
    print(1)
} else if (a == 2) {
    print(2)
} else {
    print(3)
}
```

选择语句的条件表达式同样不需要小括号，另外也不需要break，其他匹配项并不会执行，这一点和Scala相同。对选择语句的优化貌似已经是不约而同的做法。

```
i := 0
switch i {
case 0:
    print(0)
case 1:
    print(1)
}
```

循环结构的条件表达式依然不需要小括号。Go语言只支持for循环。同时对无限循环的场景也做了优化，不再需要for(;;)的写法。

```
for {
    print(1)
}
```

## 函数

Go语言诞生自C语言的派系，因此Go语言从一开始就不是OOP或FP的语言，没有类、对象等概念。函数是程序中的一等公民。和C语言相同，（main包下的）main函数是整个程序的入口。

```
func add(a int, b int) (int, int) {
    return a + b, a - b
}

func main() {
    x, y := add(1, 2)
    print(x, y)
}
```

Go语言的语句简洁高效，函数名后的第一个括号为入参，第二个括号是出参。函数支持多返回值。如果参数类型相同，可以将类型声明合并到一起，如(a, b int)。

## 结构体

刚才提到Go语言没有类、对象等概念，但是Go语言有类似C语言的结构体，并且能力强大。这里定义一个Person结构体，包含两个属性name和age，并为Person添加一个方法getInfo，用于输出Person对象的信息：

```
type Person struct {
    name string
    age int
}

func (p Person) getInfo() {
    print(p.name, p.age)
```

```

}

func main() {
    smallyu := new(Person)
    smallyu.name = "smallyu"
    smallyu.age = 1
    smallyu.getInfo()
}

```

用OOP的思想理解这样的程序并不违和。除了结构体，Go语言还保留有指针的概念。Java程序员对指针可能稍感陌生，关于指针在结构体方法中的应用，可以通过一个简单的例子来了解：

```

type Person struct {
    name string
}

func (p Person) setName() {
    p.name = "set name"
}

func (p *Person) setName2() {
    p.name = "set name"
}

func main() {
    smallyu := &Person{"smallyu"}
    smallyu.setName()
    fmt.Println(smallyu)           // &{smallyu}

    bigyu := &Person{"bigyu"}
    bigyu.setName2()
    fmt.Println(bigyu)           // &{set name}
}

```

使用值类型定义的结构体方法，入参为形参；使用引用类型定义的结构体方法，入参为实参。`&()`是初始化对象的方法之一，等同于`new()`。

## 匿名结合

Go语言中匿名结合的概念，相当于OOP语言的继承。一个结构体可以继承另一个结构体的属性和方法，大致是这样。

```

type Father struct {
    name string
}

func (f Father) getName() {
    print(f.name)
}

type Son struct {
    Father
}

func main() {
    smallyu := &Son{}
    smallyu.name = "smallyu"
    smallyu.getName()          // smallyu
}

```

Son并没有定义name属性，也没有定义getName()方法，它们均继承自Father。

## 接口

Go语言的接口是非侵入式的，结构体只要实现了接口中的所有方法，程序就会认为结构体实现

了该接口。

```
type IPerson interface {
    getName()
}

type Person struct {
    name string
}

func (p Person) getName() {
    print(p.name)
}

func main() {
    var smallyu IPerson = &Person{"smallyu"}
    smallyu.getName()
}
```

## 协程

使用协程的关键字是go，从命名就能看出协程对于Go语言的重要性、协程是轻量级的线程，启动一个协程非常简单：

```
func f(msg string) {
    println(msg)
}

func main() {
    f("直接调用方法")
    go f("协程调用方法")
}
```

运行程序，你会发现程序只打印出”直接调用方法”几个字。这种情况是不是似曾相识？go启用了另一个”线程”来打印消息，而main线程早已结束。在程序末尾加上`fmt.Scanln()`阻止main线程的结束，就能看到全部的打印内容。

## 通道

通道即协程之间相互通信的通道。

```
func main() {
    message := make(chan string)

    go func() {
        message <- "ping"
    }()
}

msg := <-message
println(msg)
}
```

`make`函数返回一个`chan string`类型的通道，在匿名函数中将字符串”ping”传入通道，之后将通道中的数据输出到变量`msg`，最后打印出`msg`的值为”ping”。

## 错误处理

Go语言在错误处理部分有两个函数较为常用，`panic`函数和`defer`函数。`panic`函数会打印错误消息，并终止整个程序的执行，类似Java的Throw Exception；`defer`函数会在当前上下文环境执行结束前再执行，类似try catch后的finally；`panic`函数虽然会终止整个程序，但不会终止`defer`函数的执行，可以将`defer`函数用于打印日志。这是一个简单的例子：

```
func main() {
```

```
    println("beginning")
    defer func() {
        println("defer")
    }()
    println("middle")
    panic("panic")
    println("ending")
}
```

来分析一下程序的执行结果。首先beginning被打印；然后遇到defer，暂不打印；middle在defer之前被打印；遇到panic，程序将终止，打印defer和panic。

这里要注意，defer是在程序结束前执行，而不是在其他语句结束后执行，这是有区别的。就像这里，panic函数引起了当前程序的结束，所以defer会在panic函数前执行，而不是panic后。程序的执行结果如下：

```
beginning
middle
defer
panic: panic

goroutine 1 [running]:
main.main()
D:/go/src/awesomeProject/main.go:12 +0x7f
```

## 其他

除此之外Go语言还有很多语言特性，也提供了非常多实用的工具包。Go语言是一种值得我们尝试去使用的语言。关于协程和通道，后续会单独探讨这一重要特性。

## 参考

- 《Go语言编程》
- [Go by Example](#)

# 我为什么想要“行万里路”

2019-03-14

## 微信朋友圈

首先这不是一个正确的想法，是典型的做什么决定想什么，而不是想什么就去做什么。

我在朋友圈发过一些景点的照片，这件事情会持续下去，大概1年左右的时间。按1个月3个景点算，1年大概30个景点。帝都稍微知名的、免费的景点也就差不多了。这些景点信息收集自网络，是网友关注并推荐的地点。1年后收入有增长时，会考虑收费（¥19.9以上）的景点。

在某个过渡期间，我的三观开始崩坏，也需要重塑。其实“观”有很多，除了世界观、人生观、价值观，还可以有事业观、婚姻观、饮食观、出行观、打游戏观等。我想见到更多的人，看到更多的事。看看国内顶尖的建筑，看看上万人趋之若鹜的风景。想想身边的人为何而来，会怀着怎样的心情离去。我还记得在太原市图书馆见到的志愿者队伍，都是老大不小的工人，虽然穿着制服，但看得出来他们的出身，猜得到他们的工作，因为那真是熟悉的身影，在宽敞明亮的大厅里不知所措、唯唯诺诺、格格不入，如果你在农村生活过，就会明白。我也记得在道馆里认真参拜神仙的香客，烧香、磕头、许愿、捐钱，让我不得不思考宗教信仰的力量，因为我想许愿，却发现没有愿望可许。我还发现外国大叔见到你真的会点头微笑。

我曾在QQ空间发过一些观点鲜明的内容，不知道有什么好的坏的影响，也无所谓知不知道。一方面不想再维护QQ空间的内容，二是我赞同“谦虚不是美德”的观点，所以有些内容可能不够谦虚，但都是实话 :p

我认为作为接受过高等教育的90后，应该有并且能够容忍甚至接纳新的、奇怪的、特立独行的想法。我们只是独立的、渺小的个体，有可能下一秒钟就会发生意外，也有可能我们会做出在别人眼里是异类的行为，或者我们原本就是别人眼中的异类。几百年前一只跳蚤或一滴水就可以传播瘟疫，贸易船只上每个奴隶身上都可能携带致命的细菌，人们对此类疾病一无所知，求助于僧侣和医生，洗冷水澡、用柏油擦拭身体或是将黑甲虫碾碎涂抹在伤口，却毫无功效。疾病从个体感染到整个家族，蔓延到整座城市，成千上万尸体腐烂在街头，人们将原因归于神灵发怒、恶魔作祟，寄希望于天使或仙女，政府对疾病束手无策，只能组织大规模祈祷和游行。死去或幸存的人，他们也都是有思想的、独立的个体，和我们一样。

微信好友的群体与QQ不同，也曾想过目前朋友圈的内容是不是很low，于是在虚荣心和真实之间犹豫。显然现在选择了后者，并且这里的内容也将作为外链第一次被分享到朋友圈。

## 博客

我的博客经历过多次建立和销毁，上一次的数据还有备份，但是主机因为科学上网被墙了。每次都想写一些值得写的东西，后来发现要写出不让人后悔的东西太难了，就像看到QQ空间以前的说说，会感到脸红想要立刻删掉一样。有些比较无聊但又想写写的东西，像“在XX平台运行一个XXX”、“XX框架/工具的Hello World级demo”之类，会发到另一个平台[语雀](#)上。

博客主题仿自王垠的博客[当然我在扯淡](#)，我在[生而为人，却想成神？](#)中有提到过对王垠的敬佩，除了极高的学术修养外，他对很多事情都有独到、精辟的见解，是真正有独立思考能力的人，光是这一点就远超常人，那也是我努力追求的可贵品质。

从[北京，北京](#)开始，似乎很多技术无关、也没有明确观点的内容。当时多次提到独居的概念，独居并不意味着孤独，而是你可以变成任何你想变成的样子，做任何想做的事情，是一种机会。技术相关的问题会连同其他问题一起重新被审视。技术能力的好坏取决于你有没有用过React或Spring Cloud吗？绝对不是。那又是什么呢？

如果现在一心关注技术，就像在学校临近毕业的一年，技术一定可以有增长，但那样就没时间也没心情做其他事情了。就像当时我如果懂得其他事情，而不是只关心技术的学习，某些事情一定可以处理的更好，但就不会有超过同学的技术能力，可能现在还在这互联网寒冬之际跑金三银四

的招聘会。二者不可兼得，不同的选择会面临不同的人生，这也正是我在 [要是能重来，你会怎么活？](#) 想要探讨却没有具体阐述的主题。那么，现在该怎么选择？

## 最后

这里是我的博客，网址是 [www.smallyu.net](http://www.smallyu.net)，会更新一些和任何人无关的内容（如果你认为和你有关，请屏蔽我）。smallyu可以理解为中英混拼，也可以将YU理解为专有名词，在域名中小写，所以有了smallyu的英文拼写。其他文章页面并没有返回首页的链接，但是这次你可以点击[这里](#) [返回首页](#)。

# 要是能重来，你会怎么活

2019-03-03

如果能够带着现有的记忆重新活一次，我的人生将……

人生道路上，有些事情可以改变，有些事情受生长环境制约。我们面临过很多次选择，有的选择由长辈代劳，有的选择由自己的意愿左右。如果当时做出了不同的选择，我们的人生会发生哪些变化？

夏洛特烦恼算是重新“活过”的故事，武林外传也有一集，讲各自满足自己的幻想，重新“活过”的情景。这两部影视作品中关于重新“活过”的桥段，结局是相似的，认为在经历了种种之后，还是现状比较好。也不知如此的结局，是真实的，还是两位作者都不约而同向生活妥协，本就无法改变，倒不如让自己接受的痛快，活的自在。

回想自己以前做的事情，在最优的选择下，能够获得些什么，便可以知道自己荒废了多少时光；整理现在自己做的事情，在完成度最高的情况下，可以达到怎样的高度，就能知道自己以后的时光该如何努力。

像单机RPG游戏，每升一级就有新的技能点，但技能树上的节点繁多，满级也只能走到某一个分支的尽头。我们必须在低等级，不了解游戏的情况下，选择一个分支，然后等分支到头了，我们就知道加哪些技能好玩了。不过可惜仅此一次，因为游戏已经玩过了。

时隔（仅仅）两个月回到学校，却有了物是人非、恍若隔世的感觉。变了吧，也都没变。遗憾呐，确实有遗憾。

# 生而为人，却想成神

2019-02-12

从哪儿说起呢？

## 生而为人

打开各类新式社交APP，匿名聊天、剧情推理、角色扮演，花样很多，如果投入进去，也能找到些许乐趣。然而这些应用的使用者，大多是00后。想起当年90后还未长大，飞信还活着的年代，各种WAP网站，资讯、论坛、自助、文字页游，社区里还充斥着所谓家族的群体，大家热衷于讨论在线浏览器、免流方法。那个时代已经过去了。

赞叹《忠犬八公》里狗的演技，《一条狗的使命》也另辟蹊径，从狗的视角阐述它一生的经历。有人提到，影片中狗的四次生命，分别对应人生的四个阶段。一开始单纯懵懂，快快乐乐；之后进入工作，生活枯燥，像是失去了一些东西，空空落落；然后是无言的陪伴，生活平淡，身边是家人、朋友；最后生活进入绝境，变得绝望，返璞归真，开始寻找最初的……

《洛丽塔》是一部讲述少女与大叔禁忌之恋的影片。大叔租了一所公寓，公寓中只有单身母女，大叔喜欢上了未成年的女儿，母亲则喜欢上了大叔。母亲是房东，威胁大叔，要么和自己结婚，要么离开公寓。大叔为了能再见到女儿，和母亲结婚了。名义上，大叔已经是女儿的父亲。然而大叔和儿女发生了各种不可描述的关系……

修养有限，在今年的贺岁电影《疯狂的外星人》里没有看出某些深刻含义，本来就是荒诞喜剧。但却在观影过程中，诧异于两个屌丝主角没能把外星人的能力变现，他们明知头环拥有力量，却没想过自己占有那份力量。可能是我太自私，只想到自己。不过从结局来看，经历过一系列波折，起起落落，外星人走了，开着飞船，把一仓库的酒都带走了。然后呢？卖酒的卖酒，耍猴的耍猴，生活还是要继续。

## 何必成神

王垠是我钦佩的程序员。前段时间，他在博客中提到，他正试着做一些改变，比起“垠神”，现在更乐意别人叫他“老王”。很庆幸可以通过网络，了解到这样一位有思想的技术专家。

一个表哥在帝都生活工作多年，最近打算回老家发展。在亲戚朋友眼中，他算混的不错的一类人。早已成家，女儿很可爱。几次接触来看，情商很高，因为是销售出身，会来事，但某些专业能力稍有欠缺。也因为身后庞大的负担，一些小事上表现的不是很慷慨。“北漂的结局有哪些？”也许有上限，没下限。

一个表姐嫁到某二线城市。第一次到家里做客，是在较偏远的旧式楼房里，没有电梯。表姐的家里人，在十八线县城租房子住着。也许是便宜，没我目前租的房子好……

平庸的人始终平庸。正因为能看到未来，所以才惧怕未来。知道努力也改变不了什么，但仍然需要努力。

或许若干年后，我可以向某个人讲述我的所有经历，好的坏的，我希望的，我绝望的，我讨厌我的生活，我也喜欢我的生活。

## 你若盛开

为什么我不再迷信技术？因为我发现，技术无法成就一个“完整”的人。

关注生活，学会生活，因为活着，就是生活，可以精彩，可以失败。

最近脑海里闪现“自我救赎”四个字。生活可能有惊喜，但不会有奇迹。让人难过的是，惊喜也往往是人为制造的。

很久以前有的一些情绪，现在已忘记很多.....

# 实习0.5个月复盘

2019-01-25

0.5个月，2个星期，共经手2个工单。

## 第1个工单

第1个工单内容较简单，需求为原先某字段为空时不允许记录重复写入，变更为允许。解决方式也简单，判断字段为空时不做重复校验，一律通过就行。刚开始主要还是熟悉开发流程。

1. 确认需求，OA出工单；
2. 新建分支，进行开发；
3. 自测；
4. 代码评审；
5. 提测；
6. 收到测试报告，申请上线；
7. 上线完成，代码合并基线。

事实上沟通有一定成本，其他较复杂的需求可能沟通成本会更高。

## 第2个工单

第2个工单需求为修复平台漏洞，1个SQL注入的漏洞，1个反射型XSS攻击的漏洞。解决思路也是常规。

代码评审后改正的第一个问题，也是带给我的第一个教训是，程序里if-else不可以嵌套过多。修复SQL注入漏洞需要对多个参数做校验，我一开始的做法是：

```
if (param1) {  
} else if (param2) {  
} else {  
}
```

这样的做法太糟糕了！而且明显是规范里不推荐的写法。经导师提醒，将代码改为扩展性更好的这种写法：

```
Map<String, String> paramMap = new ConcurrentHashMap<>();  
map.put(param1);  
map.put(param2);  
if (map) {}
```

乍一看没有问题，然而这段代码有bug潜伏，第2个给我带来教训的正是这里。

ConcurrentHashMap不允许传入null！和HashMap相比，有非常大的NPE风险，因为这里传入的参数来自页面（或者参数传入时就应该做非空校验？）。生产环境的代码不允许差错。

第3个教训关于配置文件。程序的配置文件，应该达到的效果是，没有这几行配置，程序就像没上线的版本一样，所更改添加的功能都依赖这个配置文件。当然同时也就要求，程序在没有添加配置时做出相应的默认处理。我自然是没能提前考虑到，配置不存在，直接NPE。

第4个教训是，画蛇添足。程序前一天晚上上线，第二天早上8点接到异常报告，上班之后马上回滚。产生异常的原因是，我改了需求之外部分的代码。上下两个一模一样的方法，看到上面的方法加上参数校验了，虽然需求里面没写，也索性给下面那个方法加上参数校验吧。然后NPE。因为在需求之外，自测没测出来，代码评审没看出来，提测没测出来，上线就出来了。

程序回滚的损失大概就是，线上系统一晚上的异常，涉及到的各部分人员再走一次工单流程，去

修复bug。就两行代码。

## 总结

1. 要对代码负责，从需求到上线。
2. 积极处理其他事情，反应要快，沟通要顺畅，注意力要集中。

# 北京，北京

2019-01-06

初到北京，有很多新奇体验。这也意味着，真正的独立生活，开始了。

## 租房

半个多月前着手看房子，当时对这汪深水知之甚少。市面上的房子大概分三类：中介、公寓和房东直租。

多数人建议租房子不要找中介，并且各大网站如58同城大量充斥的信息源均来自中介。亲身确认像蘑菇租房、贝壳找房之类的APP确实中介居多。中介除了要一个或半个月房租的中介费外，在退房时也会有各种理由克扣押金。中介说了，不扣点押金回去是要挨骂的。中介在租房过程中扮演介绍人的角色，最后合同还是要和房东签。所以在签合同的环节也有机可趁，要注意房东是不是真房东，有可能是二房东或中介。

公寓和中介类似，不过算是“正规”中介，像8090公寓、自如、蛋壳，没有中介费，有年租8~10%的服务费（相当一个月房租），算下来也不比小中介花的少。不过品牌公寓的房子大多统一装修，不算好，但能保证在合格线上。就收费来说，中介多收一个月房租的中介费或服务费，加上押金也大概率是退不到手的，三个月就相当于交五个月房租，假设2000/月，则实际是3333/月，已经远超预期。

房东直租推荐合租找室友小程序，豆瓣小组也有，听说住多多APP也可以，房源不算多，但真实。

运气比较好，联系到一个很nice的央财研究生小姐姐，前两个月房子算是转租给我，没有押金，没有其他费用，房东是北师大的老师，可信度较高，不会乱收一些杂七杂八的费用，合租的其他人，有北师大的研究生，之前还有北大的……总之环境不错，性价比高（感激小姐姐

住进来后注意到缺一些生活必需品，才开始有独自一人生活的感觉，好在附近的超市非常方便，这是北京啊。

## 饮食

北京的物价没有想象中可怕，楼下的小饭馆大概一顿饭¥15~20，已经不算贵，毕竟我们十八线小县城的盖饭还要¥10~12，在太原也要¥12~16。

## 出行

任何地铁站都可以办理市政交通一卡通。太原的交通卡办理点，每天6点30准时下班，一分不多呆，办理交通卡需要身份证件，只能用银行卡刷POS机，整个办理过程在10分钟以上；北京的一卡通就容易多了，7点30左右去的时候当然还是有人在的，小姐姐说话也客气，我没带太多现金，把一堆零钱递过去，不到1分钟就把卡拿出来了，没有各种繁琐的步骤。

刚拿到卡，好奇地铁是怎么坐的，该什么时候刷卡交钱，跟着别人从入口刷卡进去了。看了半天，乘坐地铁的人直接上车，也没有刷卡，下车的人也就下来了。怎么回事呢（哈）然后出站时刷卡发现少了3块钱。

每次出门都要提醒自己钥匙、眼镜、钱必带，也再次加深了独自生活的感觉。

## 其他

刚来的第一天住100/天的宾馆，环境那个差。租房时也看过便宜一点的合住房，环境没比100块

的宾馆好多少，而且合住的麻烦之处在于，如果其中一个人突然不住了，该怎么办呢。

正好有同学在帝都培训几天，通了电话，可能没有机会出去，不过也挺好的，不至于特别落寞。像我这样的人啊，一个人挺好的。

## 2019计划

年初计划里把某件事情放到了重要位置。当时想到大概三个途径，显然其中一个几乎已经失败了，这就是现实。不过在变动中不断调整规划生活，不也正是生活的乐趣之一么。

### 次日

入职前一天还是出去了，先到王府井落脚看了看小吃街，然后在天安门附近逛逛，各个景点周一闭馆，无奈去中山公园转了一圈，3块钱的门票（笑哭

公园出来再次回到王府井，灯光下的小吃街算是很美。

在北京这个陌生的环境，能机缘巧合下有这样的机会，让我有些特殊的心情，或许和人一起就开心，或许看到认识的人就不慌，恍惚间感觉自己不是孤身一人，生活还有很多希望，它给我的2019开了个好头。该感谢上天呢，还是该感谢……

# Scala语法基础

2018-12-17

Scala语法较复杂，参考软件的增量开发，学习一门编程语言也应先找到一种能够驾驭的表达方式，之后再逐步添枝加叶。Scala同时支持面向对象和函数式编程，是其语法复杂的原因之一。一些教程非常全面，但也因为全面，导致难以抽丝剥茧，抓住主干。

以下内容关注最简单的基础语法，希望根据这些内容，可以尝试编写面向对象风格的Scala代码。

## 语句

Scala允许语句结尾不加;，这一点类似JavaScript。

## 变量定义

val定义不可变变量（常量），var定义可变变量：

```
val msg1 = "Hello World"  
var msg2 = "Hello Wrold"  
  
val msg3: String = "Hello World"
```

定义变量时，类型声明在变量右侧，而且是可选的，可以不声明，编译器会自动推断。Scala中的基本类型包括：

Byte、Short、Int、Long、Char、String、Float、Double、Boolean

## 函数定义

函数即方法，下面是定义函数的例子：

```
def max(x: Int, y: Int): Int = {  
    if (x > y) {  
        return x  
    } else {  
        return y  
    }  
}
```

与Java中方法定义的显著区别有三处：一是使用def关键字定义函数；二是类型声明在变量右侧，上文已提及；三是函数声明和函数体中间使用=连接。

注意函数声明的参数必须明确定义类型，编译器无法自动推断入参类型。返回类型则是可选的，除非函数使用了递归。另外，return关键字也是可选的，如果没有显式的返回语句，程序会将最后一次运算结果作为返回。

当然if后是单个语句也可以不使用大括号，因此该函数还可以这样描述：

```
def max2(x: Int, y: Int) = if (x > y) x else y
```

## 选择结构

上面的示例已经用到了if语句，Scala的if语句并无特殊之处，不过与其他语言相比，Scala用模式匹配的概念代替传统的switch结构：

```
val a = 1
```

```
a match {
  case 1 => println(1)
  case 2 => println(2)
  case _ =>
}
```

\_通配符匹配所有值，用于捕获默认情况。匹配表达式中，备选项永远不会掉到下一个case，因此不需要break或return。（如果将\_放到首句，程序不会继续向下执行）。但是要小心，如果程序没有匹配到选项，会抛出MatchError。

## 循环结构

while循环并不是Scala推荐的代码风格：

```
var i = 0
while (i < 5) {
  println(i)
  i += 1
}
```

似乎并没有难以理解的地方，这就是典型的while循环。与指令式语言相比，Scala没有++运算符，只能使用i += 1这样的语句。

提起while，就一定会想到for。Scala中的for循环与指令式语言有一些差异，简单的示例如下，程序会从0打印直到5（不包括5）。

```
for (i <- 0 until 5) {
  println(i)
}
```

Scala不推荐while循环，而更倾向于函数式的编程风格，用于遍历的foreach方法就是其一：

```
"abc".foreach(c => println(c))
```

程序会依次换行打印出a b c三个字符。如果函数体只有一行语句并只有一个参数，这行代码还可以更简洁：

```
"abc".foreach(println)
```

## 数组

Scala的数组并不在语言层面实现，可以实例化Array类来使用。相应的，数组下标使用小括号（也就是方法参数）表示：

```
val greet = new Array[String](3)

greet(0) = "a"
greet(1) = "b"
greet(2) = "c"

greet.foreach(println)
```

实例化对象时，也可以直接传入默认参数。Array确实只是一个普通的类，下面的书写方式并没有黑魔法，只是用到了样本类。关于样本类，后文有提及。

```
val greet2 = Array("a", "b", "c")
greet2.foreach(println)
```

## 类

类使用class关键字定义，类中也包含字段和方法，即典型的面向对象。与Python不同，Scala仍然支持权限控制：

```
class Accumulator {  
    private var sum = 0  
    def add(b: Byte): Unit = {  
        sum += b  
        println(sum)  
    }  
}
```

## 单例对象

单例对象（Singleton对象）相当于Java中的静态类，使用object替代class关键字定义。单例对象由程序共享，可直接调用。单例对象可以作为程序入口，即将main方法定义在单例对象中。下面的程序从上面定义的Accumulator类中实例化出对象c，并调用其add方法，最终程序打印1：

```
object Run {  
    def main(args: Array[String]): Unit = {  
        val a = new Accumulator  
        a.add(1)  
    }  
}
```

在同一源文件中，当单例对象和类同名时，称单例对象为类的伴生对象，类为单例对象的伴生类。类可以访问其伴生对象的私有属性和方法。

## 构造方法

Scala中构造方法的规则比Java要严格。Scala通过类参数的概念来实现构造方法：

```
class Accumulator(a: Int, b: Int)
```

如果类没有主体，大括号是可以省略的。实例化这个类时，就需要传入参数。在Java中的构造方法重载，对应Scala中的辅助构造器，它看起来像这样：

```
class Accumulator(a: Int, b: Int) {  
    def this(c: Int) = this(c, 1)  
}
```

这时类拥有两个构造方法：

```
val a1 = new Accumulator(1)  
val a2 = new Accumulator(1, 2)
```

Scala构造器的严格之处就在于，第二个构造器只能借助第一个或超类的构造器。

## 继承与重写

Scala的继承与Java没有明显差异，只是方法重写必须要使用override关键字：

```
class A(a: Int) {  
    def test = println("a")  
}  
  
class B(b: Int) extends A(b) {  
    override def test = println("b")  
}
```

## 特质

特质（trait）和单例对象相像，除了定义时使用的关键字不同，其余和普通的类一样，可以包含字段和方法。特质的意义在于，支持混入（Mixins），并且允许混入多个特质。这一特性经常和多重继承进行对比。

```
trait A {
    def aMethod = println("A")
}

trait B {
    def bMethod = println("B")
}

class C extends A with B
```

这样C的实例就可以调用aMethod和bMethod:

```
val c = new C
c.aMethod
c.bMethod
```

## 样本类

样本类的定义要在class前加case关键字，即类在定义时用case修饰。这种修饰可以让Scala编译器自动为类添加一些便捷设定：1. 实例化可以省略new关键字；2. 自动将参数作为类字段；3. 自动为类添加toString、hashCode和equals：

```
case class A(a: Int) {
    def aMethod = println(a)
}

object Run {
    def main(args: Array[String]): Unit = {
        val a = A(1)
        a.aMethod      // 1
        println(a)     // A(1)
        println(a.a)   // 1
    }
}
```

## 其他

与Java相比，Scala支持抽象类，但不支持接口，抽象类使用abstract定义，接口则由特质代替。Scala同样支持泛型、注解等语法。

## 后续

以上内容并不全面，也许并不够用。使用一种编程语言，除了掌握它的基本语法外，还要熟悉它的惯用写法，尤其像Scala这种多范式的编程语言。之后会持续修改完善此篇内容，也将继续讨论Scala的其他语言特性。

# 用Scala改写Java浅度实践

2018-12-14

## 起源

想要用Java实现Markdown解析器，目前只完成了多级标题的解析。其实也就是正则匹配之后替换掉相应内容，程序暂时比较简单，大致流程如下：



## 改写

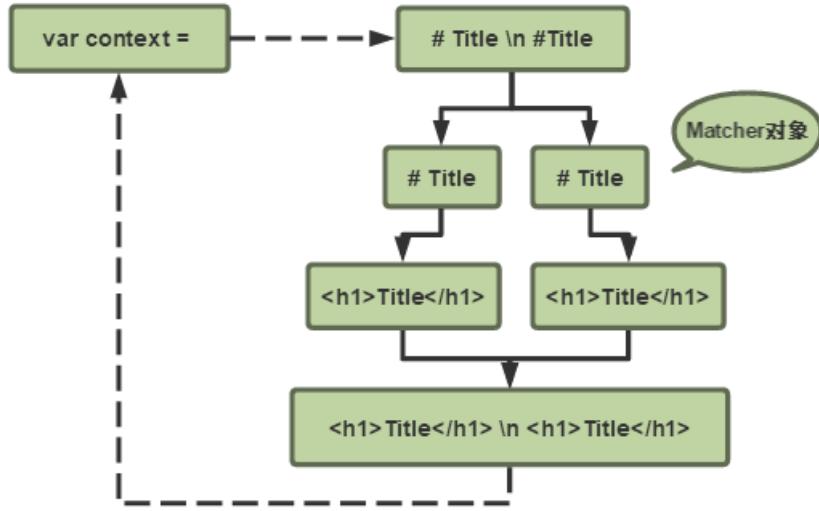
按照同样的流程，用Scala来实现该功能，之后也将使用Scala继续完成开发。首先读取文件内容，IO操作参考《Scala Cookbook》，只需一行代码即可：

```
var srcLines = Source.fromFile(srcFile).getLines().toList
```

与冗长的Java相比，Scala确实精简了不少。这是之前使用Java读取文件封装的方法：

```
/**  
 * 读取文件内容  
 *  
 * @param src 读取文件路径  
 * @return 读取文件内容  
 */  
private static String readFile(String src) throws IOException {  
    StringBuffer content = new StringBuffer();  
    InputStream is = null;  
    BufferedReader reader = null;  
  
    is = new FileInputStream(src);  
    reader = new BufferedReader(new InputStreamReader(is));  
  
    String line = reader.readLine();  
    while (line != null) {  
        content.append(line);  
        content.append("\n");  
        line = reader.readLine();  
    }  
  
    if (reader != null) {  
        reader.close();  
    }  
    if (is != null) {  
        is.close();  
    }  
    return content.toString();  
}
```

至于Scala版本将字符串改为列表操作的原因在于，Scala和Java在使用正则匹配替换的API上有差异。Java使用Matcher对象进行迭代，Matcher对象拥有查找、替换等方法：



而Scala的Regex对象虽然拥有`findAllMatchIn`、`replaceAllIn`等方法，但在`find`中的对象仅用于查找，`replace`方法中又无法定位匹配项的内容。因此在Scala中，将文件读入列表，使用如下方式带索引遍历文本内容：

```
List.range(0, srcLines.size).foreach(index => {
    srcLines = srcLines.updated(index, regexReplace)
})
```

无论是否含有匹配项，循环内都对列表执行一次`updated`，更新原内容为正则替换后的内容。这样做可能稍微欠妥，关于性能问题将持续关注并整改。可以看到的是，Scala的程序思想与Java典型的OOP确实存在些许差异。

最后关于文件写入，SDK中没有提供专门的操作对象，可使用JDK中的`PrintWriter`：

```
val pw = new PrintWriter(new File(outFile))
pw.write(outString)
pw.close()
```

## 后续

“Scala是一门会伴随开发者成长的语言”，我将用它完成我的毕业设计。

# 吹牛和装逼的区别

2018-12-12

吹牛是天马行空的、不切实际的想象，而装逼则类似于，先定个小目标。

区别就在于讲话者的实力，是打肿脸充胖子，还是能让听众都变成柠檬怪>\_<

# PDL的概念

2018-12-04

类比TDD（Test-driven development，测试驱动开发），这里适当提出提出PDL（Problem-driven learning，问题驱动学习）的概念。

这一概念似乎容易理解，我暂时无法将其完整阐述，会在之后逐步完善。这种学习方式对有一定基础的人有效，还不能确定是否可以推广到没有任何基础的人群。

# Python获取海贼王更新信息

2018-12-02

12月2日，晴，海贼王停更。

## 问题

作为一个合格的肥宅，海贼王和妖精的尾巴每周必追。这两部动漫播放源都在爱奇艺，都是VIP内容。每周末看动漫，都要在YouTube或其它网站上找资源。问题是，资源网站的更新往往不及时，常常需要Google“动漫名称 + 最新集数”，比如“海贼王 864”。

每个星期都精确的记住一部动漫应该更新的最新一集集数是多少，恐怕不是正常肥宅会做的事情，况且两部。这样，每次搜索资源前，都需要进入爱奇艺，搜索海贼王，看到最新的一集集数，关闭页面，进入Google搜索。妖精的尾巴也要同样的操作来一次。

而且，在爱奇艺里看到最新一集集数的瞬间，无法判断它是否停更，还需要在复杂的PC页面中找到“更新时间”这一标签，看更新状态是否正常，才可以做出判断。至于手机页面或APP，更是没有途径可以查看动漫的更新状态。

另一个获取动漫最新集数的方式是，百度直接搜索动漫名称，首页就倒序显示最新几集的列表（这一点百度好于谷歌），但也无法判断更新是否正常。再者，浏览器默认为Google，百度搜索需要先输入baidu，按TAB切换至百度搜索引擎，再输入要搜索的内容敲回车，步骤同样繁琐。

## 解决

写一段简单的Python脚本，从爱奇艺页面上抓取信息，自己直接访问程序便能知晓动漫的更新情况。引入工具包：

```
import urllib.request
from bs4 import BeautifulSoup
from wsgiref.simple_server import make_server

# 海贼王页面链接
url = "http://www.iqiyi.com/a_19rrhb3xvl.html?vfm=2008_aldbd"
```

urllib用于发送http请求，并接收页面数据；bs4用于解析页面，更轻易获取内容；wsgiref用于建立http服务器，提供网络服务。url是全局变量，储存海贼王页面的链接地址。

```
# 从页面获取数据
def reciveData(url):
    # 获取页面内容
    response = urllib.request.urlopen(url)
    html = response.read()
    # 解析器
    soup = BeautifulSoup(html, "html.parser", from_encoding="utf-8")
    # 更新时间
    p = soup.find('p', class_="episodeIntro-update")
    # 最新集数
    i = soup.find('i', class_="title-update-num")
    return p, i
```

这几行代码发送了请求，并从页面中获取信息。这里更新时间和最新一集集数的信息就已经拿到了。接着要创建一个http服务器，让程序输出内容到页面：

```
# 服务器环境的处理函数
def application(environ, start_response):
    # 获取数据
    p, i = reciveData(url)
    # 拼接出页面内容
    start_response('200 OK', [('Content-Type', 'text/html')])
```

```
content = ('<h3>海贼王</h3>'  
+ 'msg: ' + p.contents[2].get_text().strip()  
+ '<br>'  
+ 'num: ' + i.get_text())  
return [bytes(content, encoding = "utf-8")]
```

最后启动一个本地服务器，访问8010端口即可看到页面。可将程序部署到服务器，之后直接访问服务器：

```
# 启动服务器  
httpd = make_server('', 8010, application)  
httpd.serve_forever()
```

运行结果如图：

---

**海贼王**

msg: 本周停播12月9日恢复播出  
num: 863

**妖精的尾巴**

msg: 每周日07:30  
num: 286

## 扩展

从这一想法出发，可以扩展程序。一种是从各大网站获取全面的动漫更新信息，主动提供服务；再一种是根据用户的输入，提供自定义的动漫更新信息；或者将两者结合，提供一种大而全的、可收藏、可定制的服务。虽然这种想法毫无意义。

## 更正

之前的代码犯了一个低级错误，程序只会在首次运行时发起网络请求，之后由于网络服务一直处于启动状态，返回网页的内容始终都是初始数据。解决这个问题也很容易，将请求网络的操作封装到一个函数中，再到application函数中调用该函数即可。（代码已更正，为保证简洁，去掉了妖尾部分的代码和控制台的日志输出）

# 为什么习惯说“是”而不是“对”

2018-11-28

原因在于，“对”含有较多评判性的意味，“是”相对弱一些。我们从小接受的教育，对错太鲜明，对即1，错即0，全对即100，全错即0。当对话的两者没有明显的上下级关系、师从关系，可能将“对”用作口头禅会引起别人的小小不满，尽管这种不满不会表现出来，更不会说出口。“你为什么理所当然评判我的对错？”心里想想还是有的，虽然也许时间非常短暂，一瞬而过，之后便忘记了。

另一原因是，回答“对”往往发生在“是……吗？”的疑问句或“是……吧？”的反问句中。“Java中的int是基本类型吗？”“对”“Java中的int是基本类型吧？”“对”好像是再正常不过的对话，但是不是的问句为什么要用“对”来回答？更可笑的是，如果答案是否定的，回答的内容往往就变成了“不是”，而不是“不对”。有的人说话，连自己都毫无逻辑。

或许支持说“对”的观点在于，“对”可以给人带来微小的认可感，听到“对”的回答，也能感觉自己被认可。事实上，优秀的人不需要这种认可。

同理，常见口语中还有“好”和“行”之间的微小差异，等等。

# PHP 7，让代码更优雅（译）

2018-11-01

PHP 7已发布很久，它可以让代码更加简洁，让我们一睹其风采。

## 标量类型声明

标量指string、int、float和bool。PHP 7之前，如果要验证一个函数的参数类型，需要手动检测并抛出异常：

```
<?php
function add($num1, $num2) {
    if (!is_int($num1)) {
        throw new Exception("$num1 is not an integer");
    }
    if (!is_int($num2)) {
        throw new Exception("$num2 is not an integer");
    }

    return ($num1 + $num2);
}

echo add(2, 4);      // 6
echo add(1.5, 4);   // Fatal error: Uncaught Exception
```

现在，可以直接声明参数类型：

```
<?php
function add(int $num1, int $num2) {
    return ($num1 + $num2);
}
echo add(2, 4);      // 6
echo add("2", 4);   // 6
echo add("sonething", 4); // Fatal error: Uncaught TypeError
```

由于PHP默认运行在coercive模式，所以”2”被成功解析为2。可以使用declare函数启用严格模式：

```
<?php
declare(strict_types=1);

function add(int $num1, int $num2) {
    return ($num1 + $num2);
}
echo add(2, 4);      // 6
echo add("2", 4);   // Fatal error: Uncaught TypeError
```

## 返回类型声明

像参数一样，现在返回值也可以指定类型：

```
<?php
function add($num1, $num2):int {
    return ($num1 + $num2);
}

echo add(2, 4);      // 6
echo add(2.5, 4);   // 6
```

2.5 + 4返回了int类型的6，这是隐式类型转换。如果要避免隐式转换，可以使用严格模式来抛出异常：

```
<?php
```

```
declare(strict_types=1);

function add($num1, $num2):int{
    return ($num1 + $num2);
}

echo add(2, 4); //6
echo add(2.5, 4); //Fatal error: Uncaught TypeError
```

## 空合并运算符

在PHP5中，检测一个变量，如果未定义则为其赋初值，实现起来需要冗长的代码：

```
$username = isset($_GET['username']) ? $_GET['username'] : '';
```

在PHP 7中，可以使用新增的“??”运算符：

```
$username = $_GET['username'] ?? '';
```

这虽然仅仅是一个语法糖，但能让我们的代码简洁不少。

## 太空船运算符

也叫组合运算符，用于比较两表达式的大小。当\$a小于、等于、大于\$b时，分别返回-1、0、1。

```
echo 1 <=> 1; // 0
echo 1 <=> 2; // -1
echo 2 <=> 1; // 1
```

## 批量导入声明

在相同命名空间下的类、函数、常量，现在可以使用一个use表达式一次导入：

```
<?php
// PHP 7之前
use net\smallyu\ClassA;
use net\smallyu\ClassB;
use net\smallyu\ClassC as C;

use function net\smallyu\funA;
use function net\smallyu\funB;
use function net\smallyu\funC;

use const net\smallyu\ConstA;
use const net\smallyu\ConstB;
use const net\smallyu\ConstC;

// PHP 7
use net\smallyu\{ClassA, ClassB, ClassC};
use function net\smallyu\{funA, funB, funC};
use const net\smallyu\{ConstA, ConstB, ConstC};
```

## 生成器相关特性

PHP中Generator函数和普通函数的形式相同。生成器使用在foreach的迭代中，比数组占用内存更少，效率更高。这是一个生成器的例子：

```
<?php
// 返回一个生成器
function getValues($max) {
    for ($i = 0; $i < $max; $i++) {
        yield $i * 2;
    }
}
```

```

}

// 使用生成器
foreach(getValues(99999) as $value) {
    echo "Values: $value \n";
}

```

代码中出现了yield表达式，它就像return一样，在函数中返回一个值，每次只执行一次，并且会从上一次停止的位置开始执行。

PHP 7之前不允许生成器函数使用return返回值，现在允许了，return不会影响yield的正常迭代，return的值也可以使用\$gen->getReturn()来获取：

```

<?php
$gen = (function() {
    yield "First Yield";
    yield "Second Yield";

    return "return Value";
})();

foreach ($gen as $val) {
    echo $val, PHP_EOL;
}

echo $gen->getReturn();

```

PHP 7还支持生成器委派，可以在一个生成器函数中调用另一个生成器：

```

<?php
function gen() {
    yield "yield 1 from gen1";
    yield "yield 2 from gen1";
    yield from gen2();
}

function gen2() {
    yield "yield 3 from gen2";
    yield "yield 4 from gen2";
}

foreach (gen() as $val) {
    echo $val, PHP_EOL;
}

```

## 匿名类

PHP 7也有匿名类啦。

## 闭包

PHP 7对闭包的支持更加友好：

```

<?php
class A { private $x = 1; }

// PHP 7之前
$getAFun = function() { return $this->x; };
$getA = $getAFun->bindTo(new A, 'A'); // 中间层闭包
echo $getA();

// PHP 7之后
$getA = function() { return $this->x; };
echo $getA->call(new A);

```

## 可为空类型

Nullable types是PHP 7.1的新特性之一，在参数类型声明前加上一个问号，约定该参数只能是指定类型或者NULL。可以用在返回类型上：

```
<?php

function testReturn(): ?string {
    return 'testing';
}
var_dump(testReturn());      // string(7) "testing"

function testReturn2(): ?string {
    return null;
}
var_dump(testReturn2());     //NULL
```

也可以用在参数类型上：

```
<?php
function test(?string $name) {
    var_dump($name);
}

test('testing');      // string(7) "testing"
test(null);          // NULL
test();               // Fatal error: Uncaught ArgumentCountError
```

## 数组解构

list()函数的简化写法：

```
<?php

$records = [
    [1, 'smallyu'],
    [2, 'bigyu'],
];

// list() 风格
list($firstId, $firstName) = $records[0];

// [] 风格, PHP 7.1
[$firstId, $firstName] = $records[0];

var_dump($firstId);      // int(1)
var_dump($firstName);    // string(7) "smallyu"
```

另一个新特性是list()和[]都支持keys了：

```
<?php

$records = [
    ["id" => 1, "name" => 'smallyu'],
    ["id" => 2, "name" => 'bigyu'],
];

// list() 风格
list("id" => $firstId, "name" => $firstName) = $records[0];

// [] 风格, PHP 7.1
["id" => $firstId, "name" => $firstName] = $records[0];

var_dump($firstId);      // int(1)
var_dump($firstName);    // string(7) "smallyu"
```

## 参考

- «Building REATful Web Services with PHP 7» Chapter 2: PHP 7, To Code It Better
- [PHP: 新特性 - Manual](#)

# Java 11 教程 (译)

2018-10-31

Java 11已经发布，很多人还在使用Java 8。这篇教程讲述一些重要的语言特性和API。

## 局部变量类型推断

局部变量指在方法体内声明的变量。Java10就已经引进一个新的关键字var，用于代替在声明局部变量时候的类型声明。

在Java 10之前，你必须这样声明一个变量：

```
String text = "Hello Java 9";
```

现在你可以使用var代替String。编译器会自动从变量的赋值推断出正确的类型。如文本的类型为String：

```
var text = "Hello Java 10";
```

使用var声明的变量仍然是静态变量，不可以在声明后赋值为其它类型：

```
var text = "Hello Java11";
text = 23; // 编译错误，不兼容的类型
```

同样可以使用final声明变量为常量：

```
final var text = "Banana";
text = "Joe"; // 编译错误
```

当然，在编译器无法推断出类型的场景下，不可以使用var，比如这些情况：

```
var a;
var nothing = null;
var lamdba = () -> System.out.println("Pity!");
var method = this::someMethod;
```

当变量声明包含泛型时，var的优势尤为突出，下面的示例就用var来代替冗长的Map<String, List>：

```
var myList = new ArrayList<Map<String, List<Integer>>>();

for (var current : myList) {
    // current 的类型会被推断为 Map<String, List<Integer>>
    System.out.println(current);
}
```

Java 11的var关键字同样支持在lamdba表达式的参数中使用，并且支持为这些参数添加注解：

```
Predicate<String> predicate = (@Nullable var a) -> true;
```

小技巧：在IntelliJ IDEA中按住CTRL键可以查看变量的推断类型。

## HTTP Client

从Java 9开始引进试用新的API HttpClient，用于处理HTTP请求。现在Java 11将其标准化，我们可以从模块java.net中获取使用。

新的HttpClient在同步和异步场景下都可以使用。同步请求会阻塞线程，直到获取到响应。BodyHandlers定义了响应数据的类型（如String、Byte[]、File）。

```

var request = HttpRequest.newBuilder()
    .uri(URI.create("https://blog.smallyu.net"))
    .GET()
    .build();
var client = HttpClient.newHttpClient();
var response = client.send(request, HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());

// 记得在module-info.java中导入java.net.http模块

```

同样可以使用异步的方式实现请求，调用sendAsync方法并不会阻塞当前线程，它会构建异步操作流，在接收到响应后执行相应操作：

```

var request = HttpRequest.newBuilder()
    .uri(URI.create("https://blog.smallyu.net"))
    .build();
var client = HttpClient.newHttpClient();
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);

// 线程睡眠，防止在返回响应前当前线程就结束
Thread.sleep(3000);

```

.GET()方法会作为默认的请求方式。

下一个示例通过POST方式发送请求到指定URL。与BodyHandlers相似，使用BodyPublishers定义要发送的数据类型：

```

var request = HttpRequest.newBuilder()
    .uri(URI.create("https://postman-echo.com/post"))
    .header("Content-Type", "text/plain")
    .POST(HttpRequest.BodyPublishers.ofString("Hi there!"))
    .build();
var client = HttpClient.newHttpClient();
var response = client.send(request, HttpResponse.BodyHandlers.ofString());
System.out.println(response.statusCode()); // 200

```

最后一个示例演示了如何使用BASIC-AUTH执行权限验证：

```

var request = HttpRequest.newBuilder()
    .uri(URI.create("https://postman-echo.com/basic-auth"))
    .build();
var client = HttpClient.newBuilder()
    .authenticator(new Authenticator() {
        @Override
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication("postman", "password".toCharArray());
        }
    })
    .build();
var response = client.send(request, HttpResponse.BodyHandlers.ofString());
System.out.println(response.statusCode()); // 200

```

## 集合框架

集合框架如List、Set和Map都增加了新的方法。List.of方法根据给定参数创建一个不可变列表，List.copyOf创建一个已存在列表的副本。

```

var list = List.of("A", "B", "C");
var copy = List.copyOf(list);
System.out.println(list == copy); // true

```

因为列表已经不可变，所以拷贝出的列表和原列表是同一实例。如果拷贝了一个可变列表，拷贝出的列表会是一个新的实例，不会对原列表产生副作用：

```
var list = new ArrayList<String>();
var copy = List.copyOf(list);
System.out.println(list == copy); // false
```

创建不可变映射不必自己创建映射实体，只需要将key和value交替传入作为参数：

```
var map = Map.of("A", 1, "B", 2);
System.out.println(map); // {B=2, A=1}
```

Java 11中的不可变列表和旧版本的列表使用相同的接口，但是如果你对不可变列表进行修改，如添加或移除元素，程序会抛出java.lang.UnsupportedOperationException异常。幸运的是，当你试图修改不可变列表，IntelliJ IDEA会检查并给出警告。

## Streams

Streams从Java 8开始引进，现在新增了三个方法。Stream.ofNullable从单个元素构建流：

```
Stream.ofNullable(null)
    .count() // 0
```

dropWhile和takeWhile方法都是用于放弃流中的一些元素：

```
Stream.of(1, 2, 3, 2, 1)
    .dropWhile(n -> n < 3)
    .collect(Collectors.toList()); // [3, 2, 1]

Stream.of(1, 2, 3, 2, 1)
    .takeWhile(n -> n < 3)
    .collect(Collectors.toList()); // [1, 2]
```

## 字符串

String类也新增了一些方法：

```
" ".isBlank(); // true
" Foo Bar ".strip(); // "Foo Bar"
" Foo Bar ".stripTrailing(); // " Foo Bar"
" Foo Bar ".stripLeading(); // "Foo Bar "
"Java".repeat(3); // "JavaJavaJava"
"A\nB\nC".lines().count(); // 3
```

## 其他JVM特性

Java 11包含许多新特性，以上只提及冰山一角，权作抛砖引玉，更多内容等待你探索……

## 参考

- [Java 11 Tutorial](#)