

前端领域中的算法 (下)

@被删

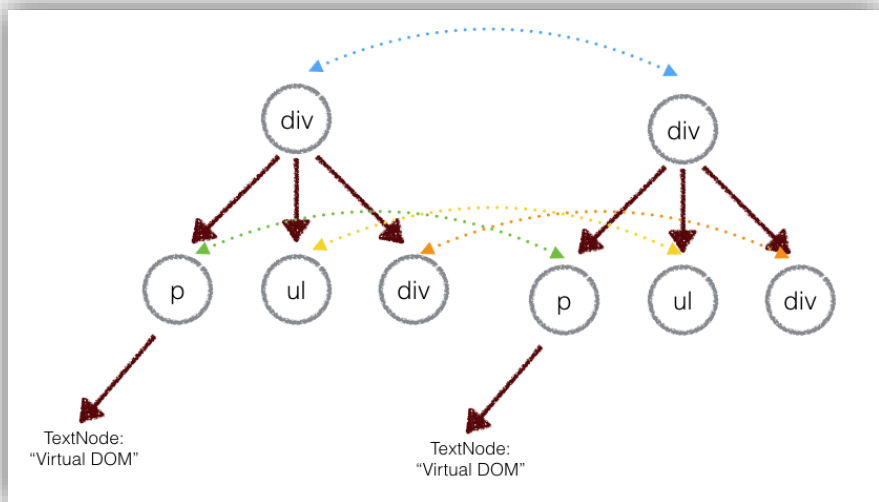
面试造火箭 入职拧螺丝



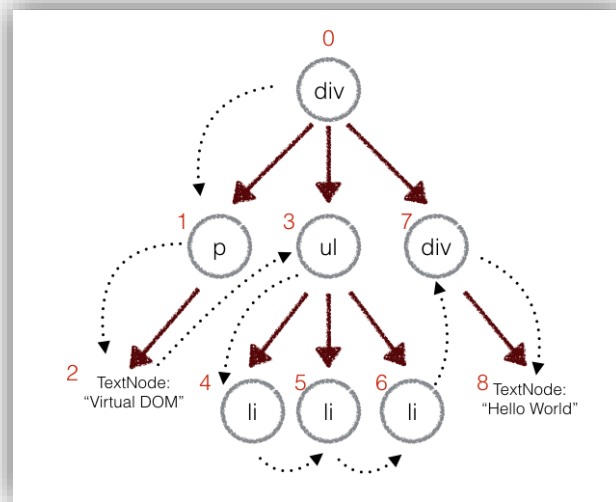
2

前端开发中的 算法和数据结构

React DOM Diff



tree diff: 同层级 DOM 节点对比



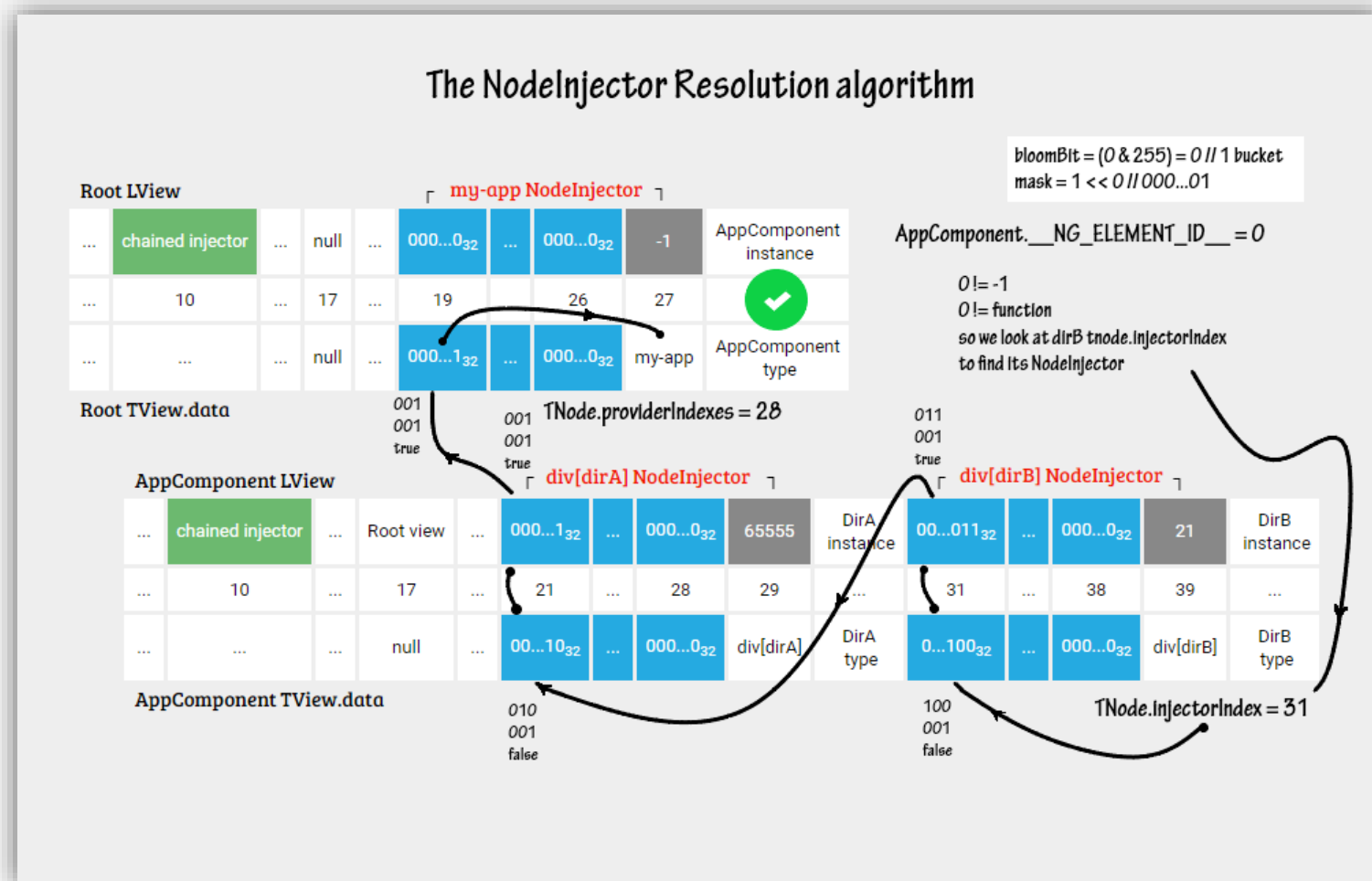
element diff: 深度遍历

component diff

- 保持稳定的 DOM 结构会有助于性能的提升
- 给元素设置唯一 key
- 尽量减少类似将最后一个节点移动到列表首部的操作

Angular 的布隆过滤器

- Angular 通过递增整数值为令牌生成（如果尚未定义）令牌的唯一 ID，并将其置于静态 `__NG_ELEMENT_ID__` 属性中
- 布隆过滤器用于检查给定的 ID 是否在集合中



除了前端框架/工具库，
前端业务/应用开发中？？

3

VSCode 与红黑树

VSCode 的文本缓冲区问题

用户无法打开 35 MB 的文件



文件太多行，共 1370 万行

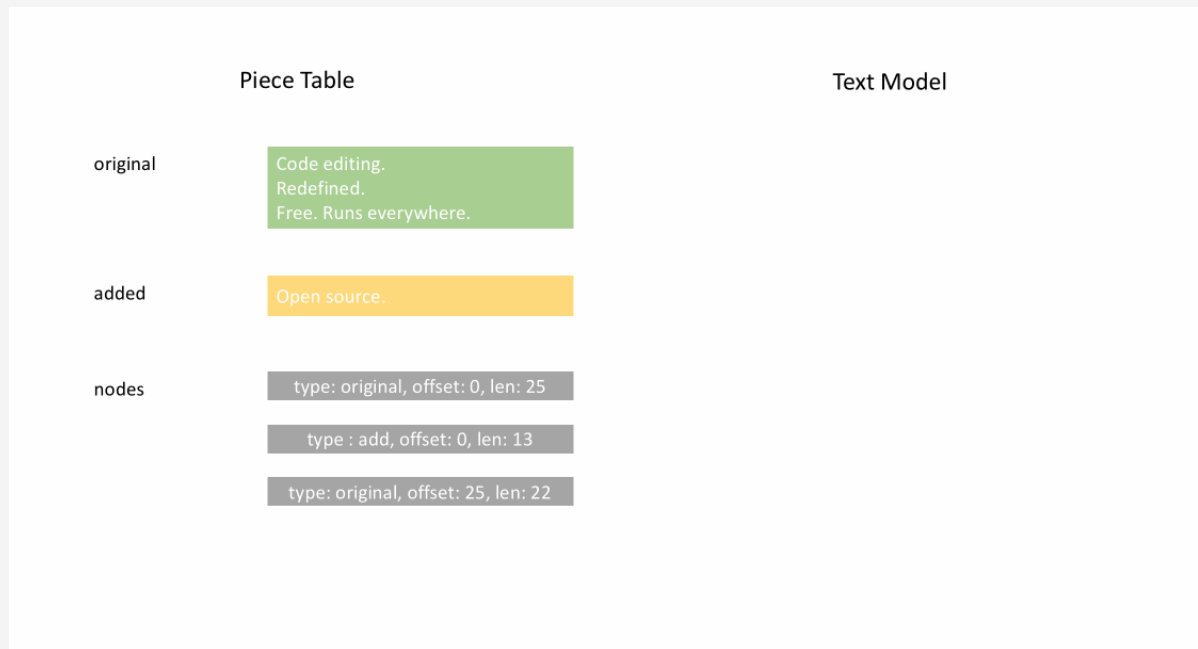


- 每行创建一个对象
- 每个对象使用大约 40-60 字节
- 大约 600MB 的内存来存储文档

是初始文件大小的20倍!!

使用 PieceTable 避免过多的元数据

```
class PieceTable {  
  original: string; // original contents  
  added: string; // user added contents  
  nodes: Node[];  
}  
  
class Node {  
  type: NodeType;  
  start: number;  
  length: number;  
}  
  
enum NodeType {  
  Original,  
  Added  
}
```



1. 最初加载文件，**original** 字段包含整个文件的内容
2. 用户在文件末尾键入内容，将新内容附加到 **added** 字段中，并添加 Node 节点
3. 用户在节点中间编辑，拆分该节点，并根据需要插入一个新节点

- 初始内存使用量接近于文档的大小
- 修改所需的内存与编辑和添加文本的数量成正比

优化 PieceTable – 1. 随机行访问

如果要获取第 1000 行的内容，需要：

1. 遍历从文档开头开始的每个字符
2. 找到前 999 个换行符
3. 读取每个字符直到下一个换行符

存储遇到的每个换行的偏移量


访问文档中的随机行：

1. 从第一个节点开始读取每个节点
2. 直到找到目标换行符为止

```
class PieceTable {
    original: string;
    added: string;
    nodes: Node[];
}

class Node {
    type: NodeType;
    start: number;
    length: number;
    lineStarts: number[];
}

enum NodeType {
    Original,
    Added
}
```



优化 PieceTable – 2. 大文档访问

两个缓冲区:

1. **original**: 用于从磁盘加载的原始内容
2. **added**: 用于用户编辑

```
class PieceTable {  
    buffers: string[];  
    nodes: Node[];  
}  
  
class Node {  
    bufferIndex: number;  
    start: number; // start offset in buffers[bufferIndex]  
    length: number;  
    lineStarts: number[];  
}
```

文件很大 (例如 64 MB) 时



收到 1000 个块



将它们连接成一个大字符串



将其存储在 **original** 计件表的字段中

V8版本中, 最大字符串长度为256MB

从磁盘接收到 64KB 的块时 → 将其直接推送到 **buffers** 数组 → 创建一个指向此缓冲区的节点

可以打开大文档

优化 PieceTable – 3. 行查找优化

```
class PieceTable {  
    buffers: string[];  
    nodes: Node[];  
}  
  
class Node {  
    bufferIndex: number;  
    start: number; // start offset in buffers[bufferIndex]  
    length: number;  
    lineStarts: number[];  
}
```

获得一行的内容:

- 必须遍历所有节点, 直到找到包含该行的节点
- 最坏情况的时间复杂度是 $O(n)$

```
class PieceTable {  
    buffers: string[];  
    rootNode: Node;  
}  
  
class Node {  
    bufferIndex: number;  
    start: number;  
    length: number;  
    lineStarts: number[];  
  
    left_subtree_length: number;  
    left_subtree_lfcnt: number;  
    left: Node;  
    right: Node;  
    parent: Node;  
}
```

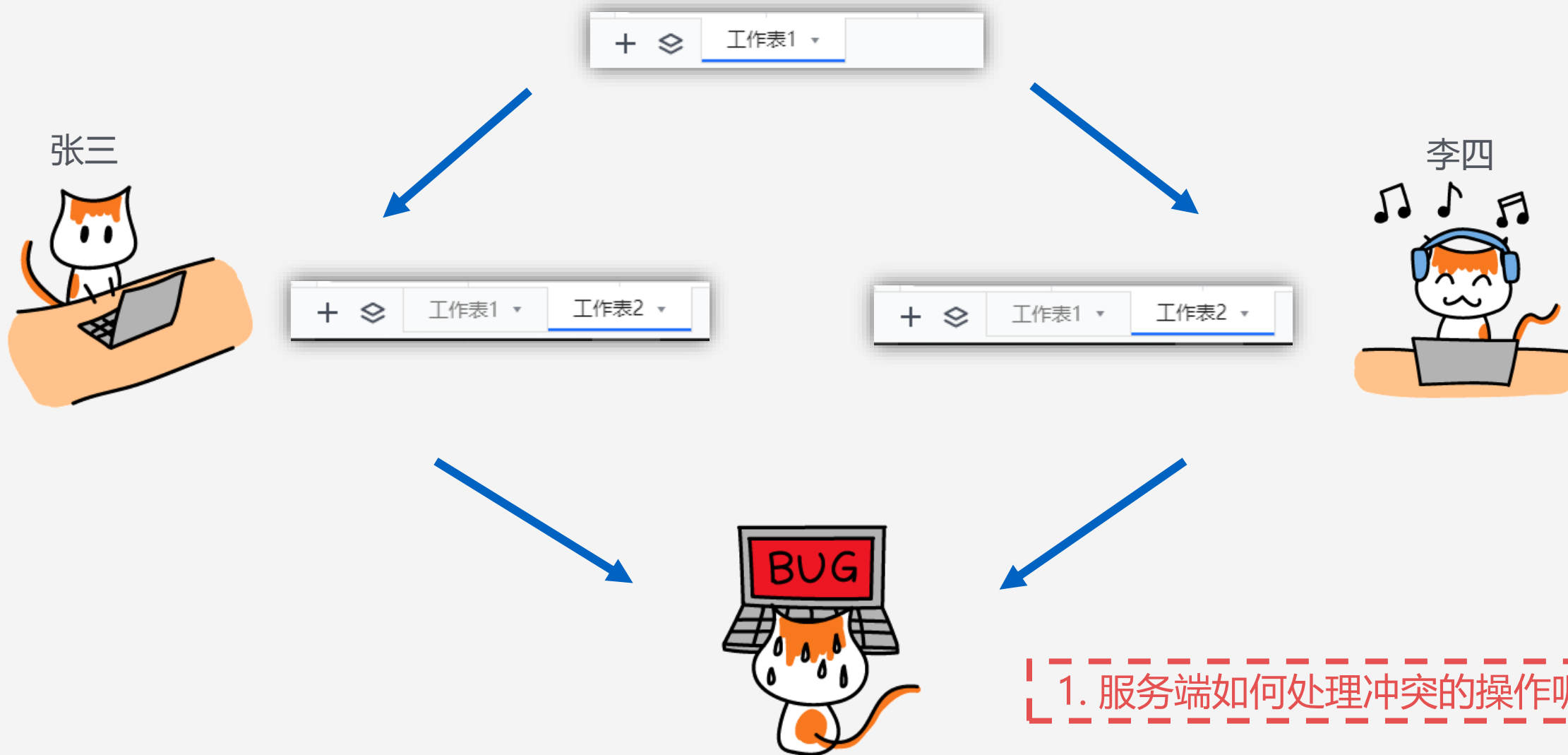
使用平衡二叉树—红黑树:

- 增加存储节点左子树的文本长度和换行符计数
- 最坏情况的时间复杂度是 $O(\log n)$

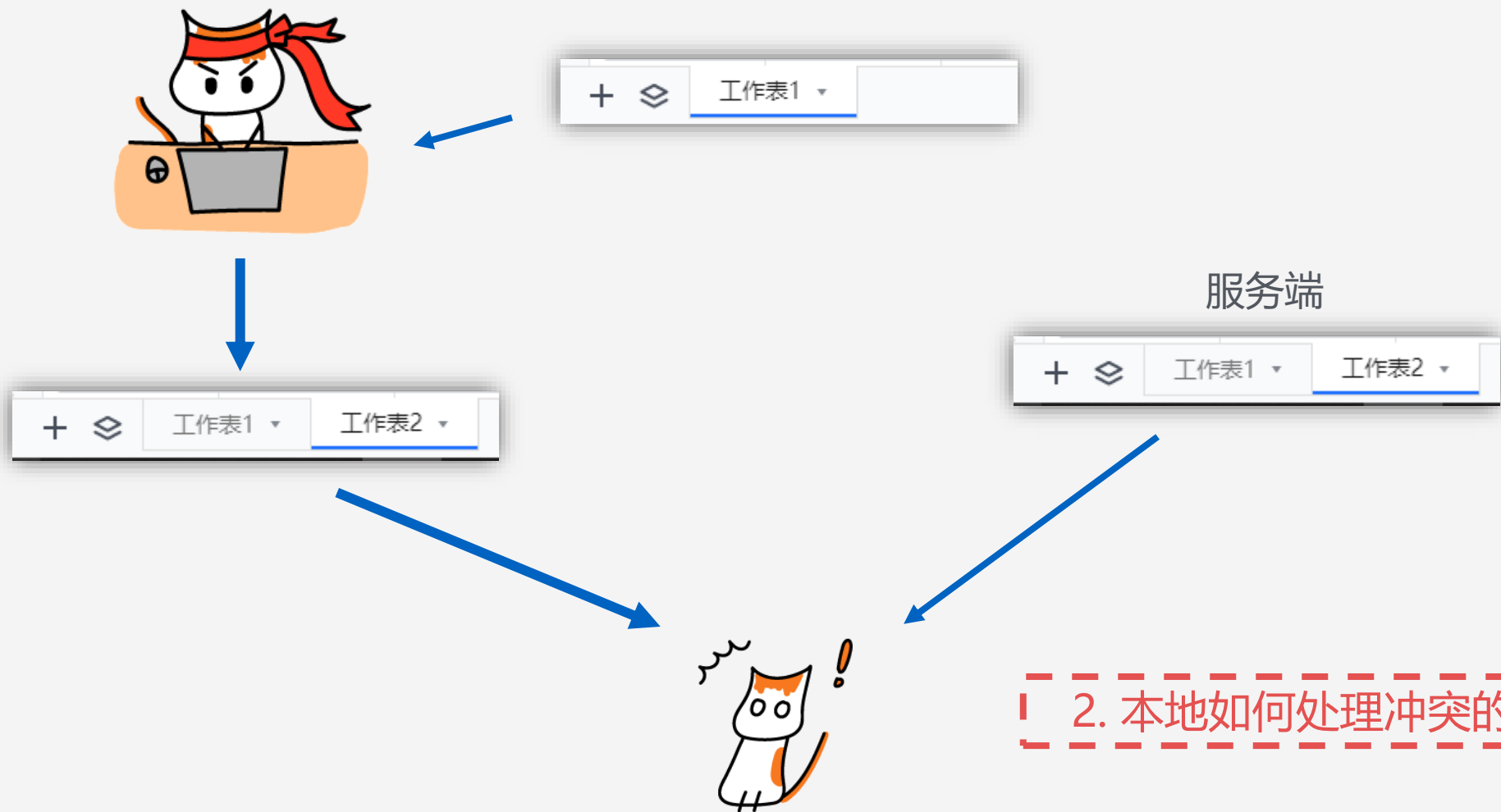
4

冲突处理算法

多人在线协作会发生什么？



多人在线协作会发生什么？



2. 本地如何处理冲突的操作呢？

在线协同冲突处理算法

1. OT 算法
2. CRDT 算法
3. Fluid Framework 冲突处理方案

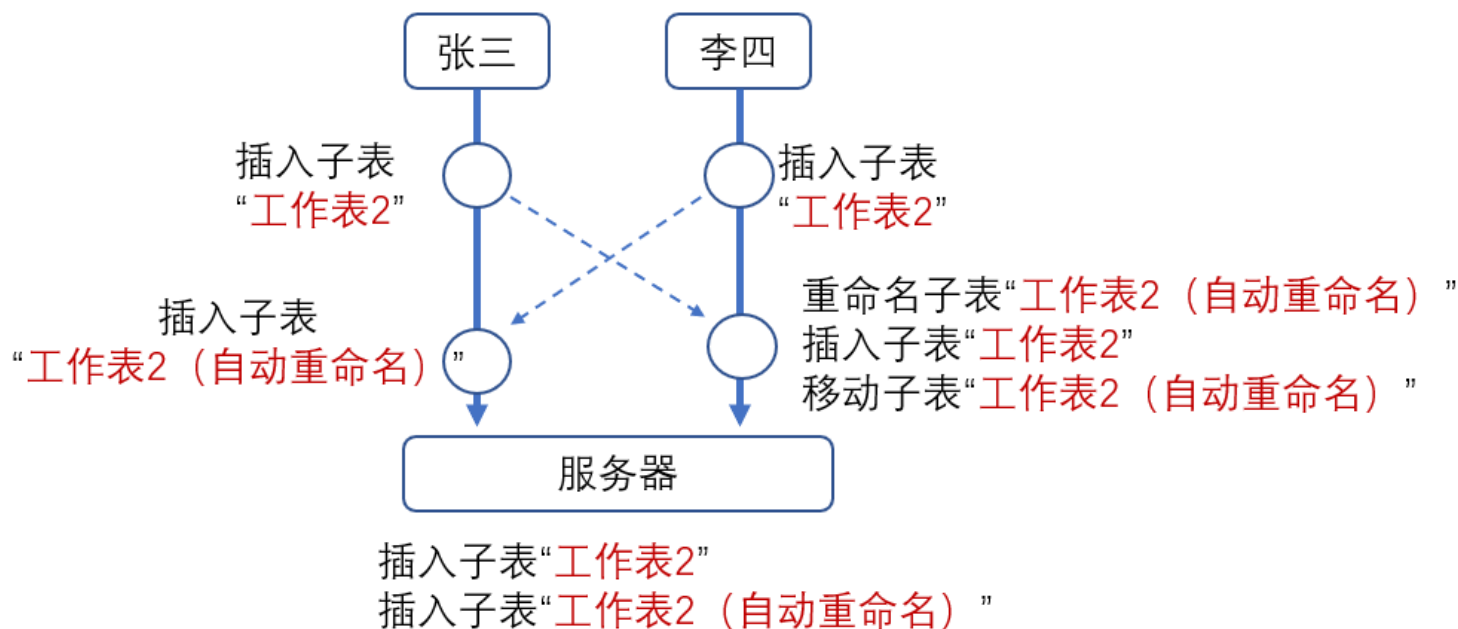
在线协同算法—OT算法 (Google Doc)

- 基于操作转换的协同冲突处理算法
- 核心思想：对操作进行转换后，应用到状态发生冲突的文档，让不同的协同端都能够到达一致的状态

一、操作的拆分

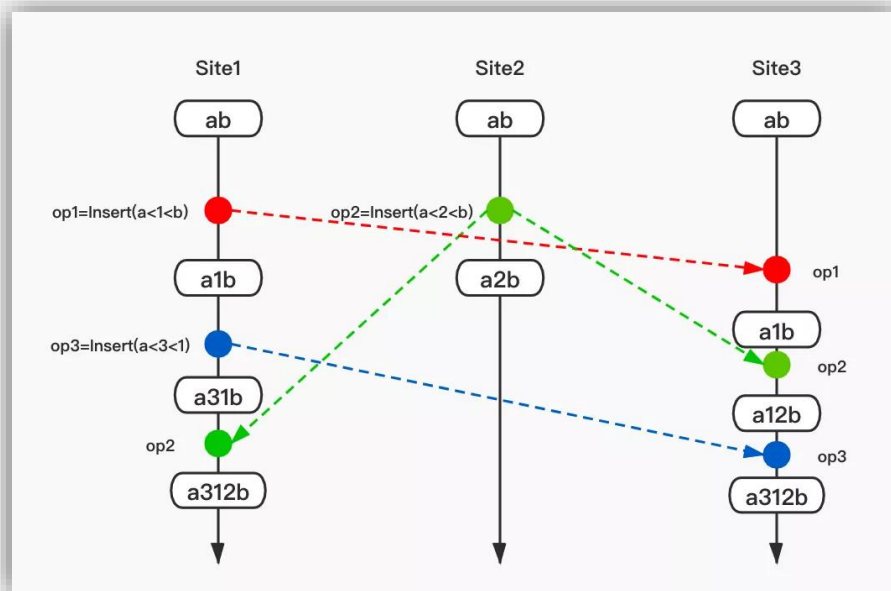
二、操作间的冲突处理

三、最终一致性的实现



在线协同算法—CRDT算法 (Atom-teletype/Yjs)

- 一种能够在网络中多个主机间**并行复制**的数据结构
- 各个主机的副本可以**独立**、且**并行**地更新，而无需对各个副本进行协同



- 对于富文本编辑等更高级的结构，**OT 用复杂性**换来了对用户预期的实现，而 **CRDT 则更加关注数据结构**。
- 随着数据结构的复杂度上升，算法的时间和空间复杂度也会呈指数上升的，会带来性能上的挑战。因此，如今大多数实时协同编辑都基于 OT 算法来实现。

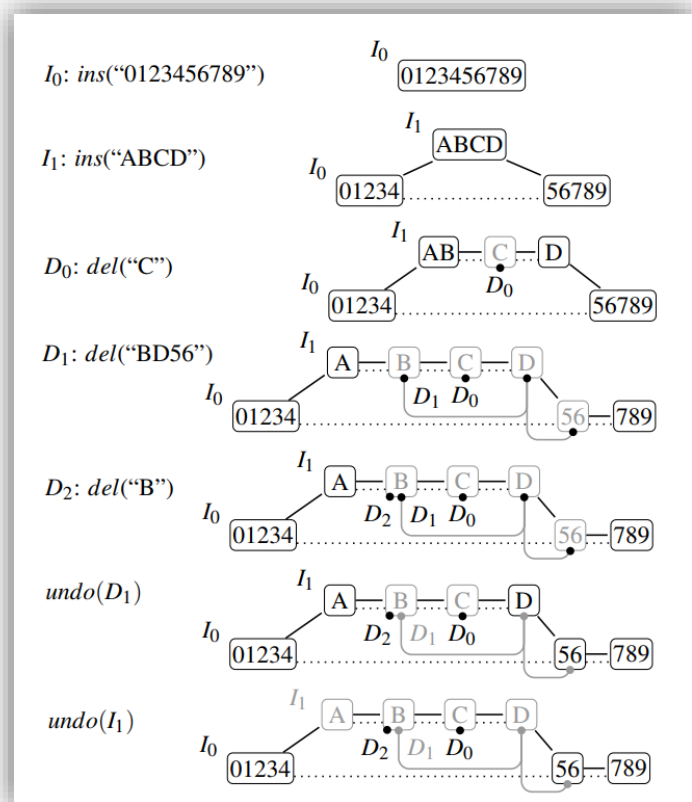
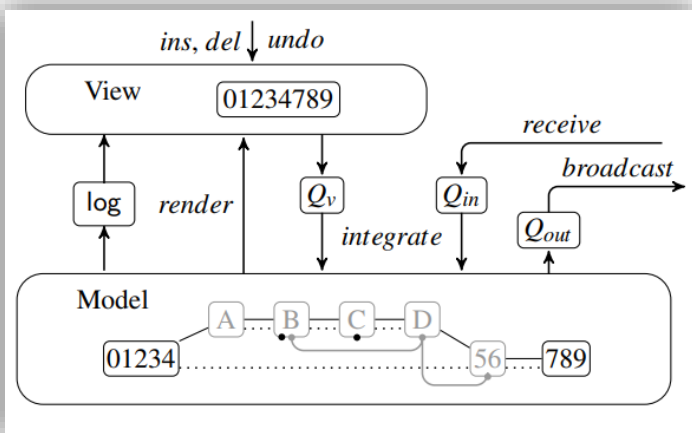
在线协同算法—CRDT算法 (Atom-teletype/Yjs)

1. 基于操作 (复制交换数据类型) : 需要服务端中间件的保证操作的传播
2. 基于状态 (复制收敛数据类型) : 更易于设计和实现, 缺点是每个 CRDT 副本的整个状态, 最终都必须传输到其他副本, 服务端开销很大

- Yjs: <https://github.com/yjs/yjs>

- Atom-teletype:

<https://github.com/atom/teletype-crdt>



CRDT 不仅可用于协同编辑领域, 还被用于在线聊天系统、NoSQL 分布式数据库 (Redis) 等

在线协同算法—Fluid Framework（微软开源）

服务端只负责排序、广播和存储：

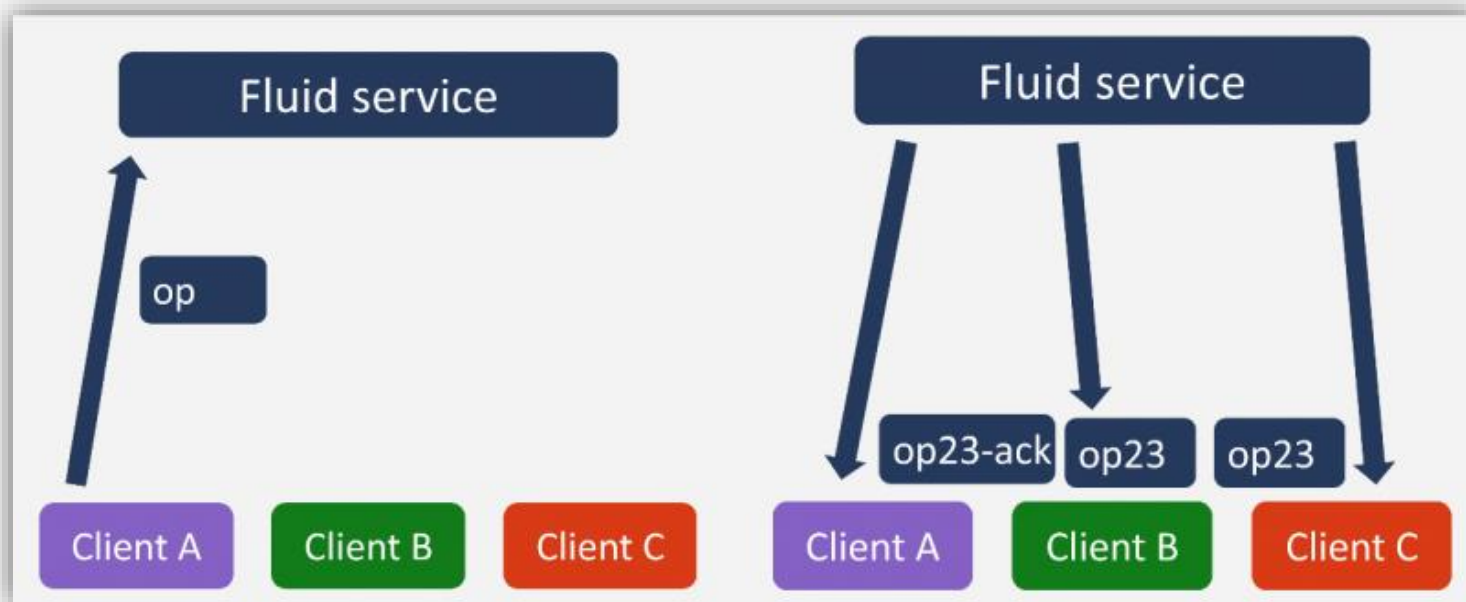
- 对所有传入的操作进行排序
- 将其广播给所有客户
- 将操作保存到持久数据存储中

借助通用的分布式数据结构（DDS）进行多用户协同

DDS主要分为两大类：

1. key-value类型
2. sequence类型

因为操作是有序的，并且每个客户端都运行相同的代码，所以每个客户端中的DDS最终都处于相同的状态。



是否一定要使用到很牛逼
的算法和数据结构？？

<https://github.com/facebook/react/issues/10703>

From our research, the vast majority of the time spent by applications built with React-like libraries lies in the application code. We *wish* optimizing the diffing algorithm was more fruitful for anything other than winning benchmark, but that does not match our experience. The library code was a small fraction of the executed code in real apps, at least in the scenarios we were looking at. Most of the code was spent in the components: rendering and lifecycles. Abstraction has a cost. This is why we are spending our efforts in two areas:

- Reducing the cost of component abstraction with experiments in compilation techniques.
- Better prioritization and interleaving IO with rendering to improve perceived performance.

All of this is work in progress, but we'll share if we find something that works well.

That said if you can summarize your findings and what's novel about your approach, we'd love to hear it too.



35



1



3

使用类似 React 的库构建的应用程序，**花费的大部分时间都在应用程序代码中**，大部分代码都花在了组件上：**渲染和生命周期**。

抽象是有代价的，这就是为什么我们将精力花在两个方面：

- 通过编译技术中的实验来**降低组件抽象的成本**
- 更好地确定优先级，并将IO与渲染进行交错处理，以改善感知性能

算法 => 编程

[解决项目中遇到的问题]



谢谢



Github: godbasin
@被删