从BI项目实践思考软件复杂度控制的常见方法

GeorgeZou

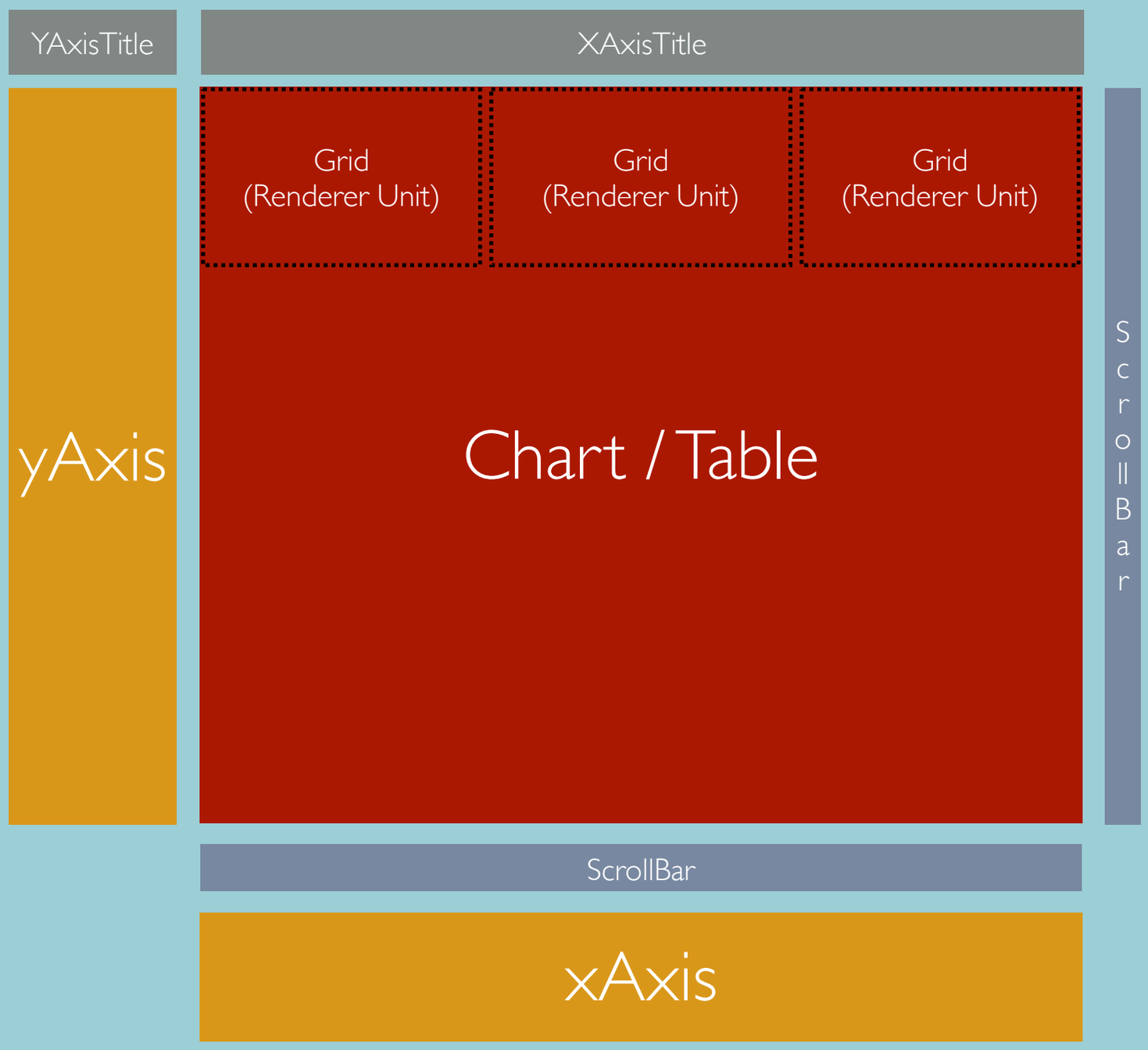
对**复杂问题**进行合理的封装和处理，

是软件工程要解决的核心问题

项目基本信息

BI Chart 组件总览

BI产品本质上是一个数据分析工具，通过将一维表根据不同字段拆分x、y两个维度上并通过X/Y轴分拆、图表分层下钻等方式，实现在二维平面展示多维数据。其核心在使用户可以通过拖拽和简单设置，对数据进行基本的计算、动态生成分析图表并控制展示样式。



Tooltip

Border

DataPreview

xAxisTranslator

yAxisTranslator

Charts

Complexity 复杂度

$$C = \sum_{p} c_p t_p$$

- Cp：功能单元复杂度
- tp: 功能单元开发事件
- C：软件整体复杂度

- 修改扩散，修改时有连锁反应。
- 认知负担，开发人员需要多长时间来理解功能模块。
- 不可知（Unknown Unknowns），开发人员在接到任务时，不知道从哪里入手。

路径数 = 图表类型 × 样式设置 × 轴组合 × 指标状态 × 数据状态 × 轴设置

- Line
- Bar
- Area
- Scatter
- Table

- 颜色
- 边框
- 透明度
- 标签
- 大小
- 工具提示

- X轴 度量
- Y轴 度量
- 都有度量
- 都没有度量
- 都有1个维度
- 都有多个维度
- 只有X轴有维
- 只有Y轴有维

- 指标聚合
- 指标并列

- 全正数
- 全负数
- 正 - 负数
- 小数

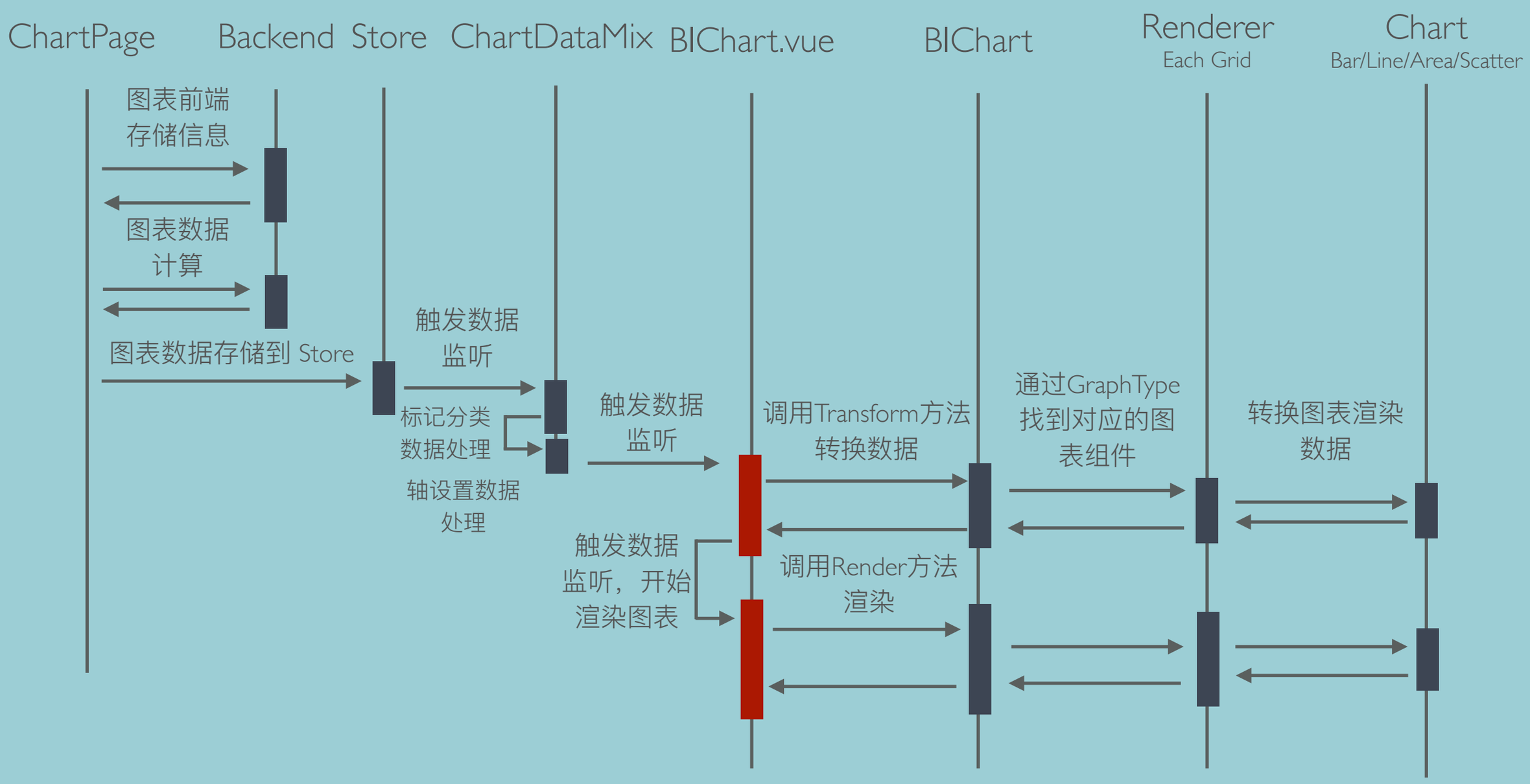
- 统一轴
- 独立轴
- 对数轴
  - 包含零
  - 不包含零
- 普通轴
  - 自动刻度线
  - 手动刻度线

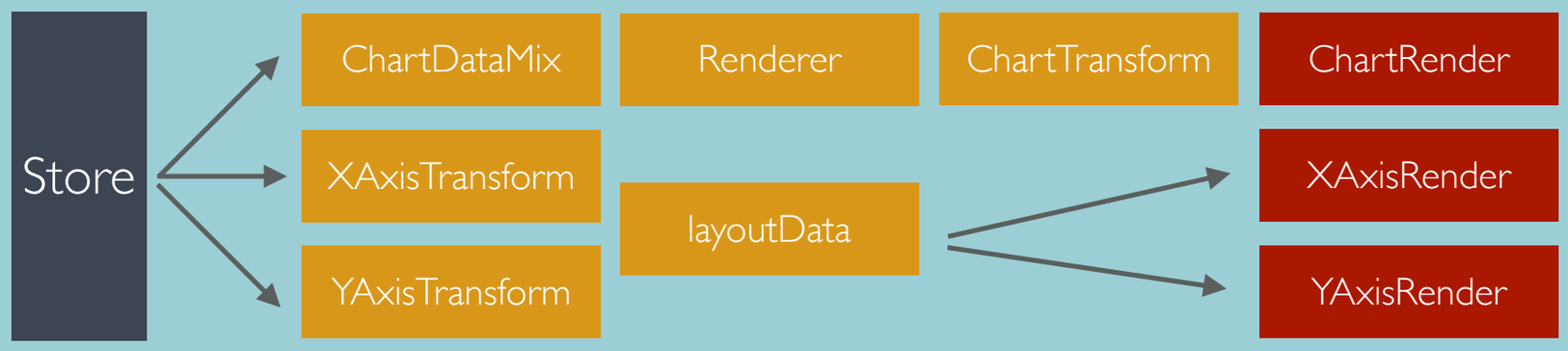
人类大脑生理上存在认知极限
—— George A. Miller (1956) The Magical Number Seven

软件实体不仅仅是相同元素重复添加，而必须是不同元素实体的添加。大多数情况下，这些元素以非线形递增的方式交互，因此整个软件复杂度要比非线性增长多得多

—— 《人月神话》

Data Flow 数据处理及渲染流程





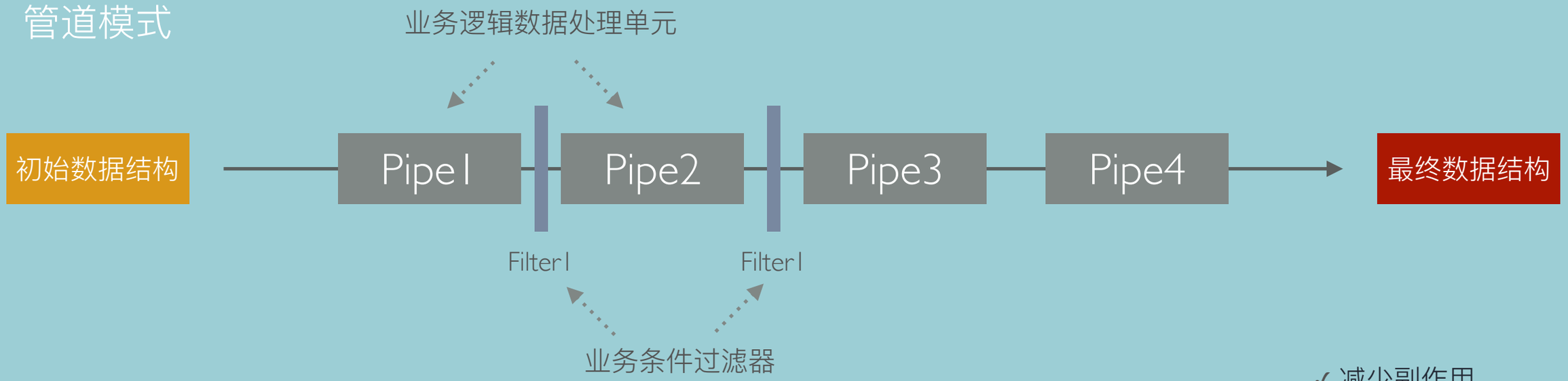
复杂度控制

1. 单一职能
2. 分层和抽象
3. 副作用Effect控制
4. 结构化数据模型
5. Code Review
6. 持续重构

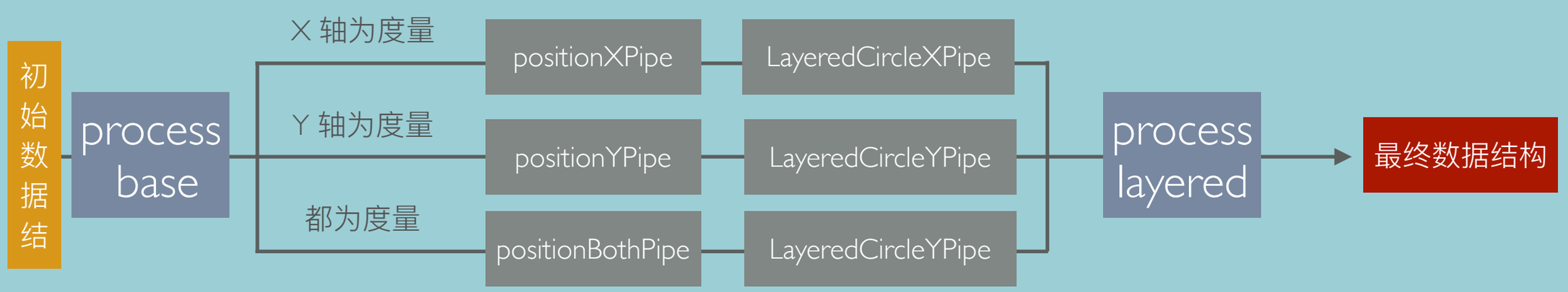
Pipeline 数据处理管道

解决复杂度：单一职能

管道模式



LinePipeline 折线类 图数据处理管道



- ✓ 减少副作用
- ✓ 可测试
- ✓ 信息隐藏

尝试通过一定的设计模式将复杂业务逻辑转化为更为有序、可拓展、易于理解的实现

Isolating complexity in places that are rarely interacted with is roughly equivalent to eliminating complexity.
—— A Philosophy of Software Design
John Ousterhout

```
If (x_value) {
  If (a.a) {
     // 100 line of code
  }
} else if (y_value) {
  // 100 line of code
} else if (both_value) {
  // 100 line of code
}
```

```
If (x_value) {
  this.pipeList.push("positionBothPipe");
  this.pipeList.push("layeredCircleYPipe");
} else if (y_value) {
  this.pipeList.push("positionYPipe");
  this.pipeList.push("layeredCircleYPipe");
} else if (both_value) {
  this.pipeList.push("positionXPipe");
  this.pipeList.push("layeredCircleXPipe");
}

this.invoke();
```

复杂问题 => 普通问题 普通问题 普通问题 普通问题

Data Process 数据处理

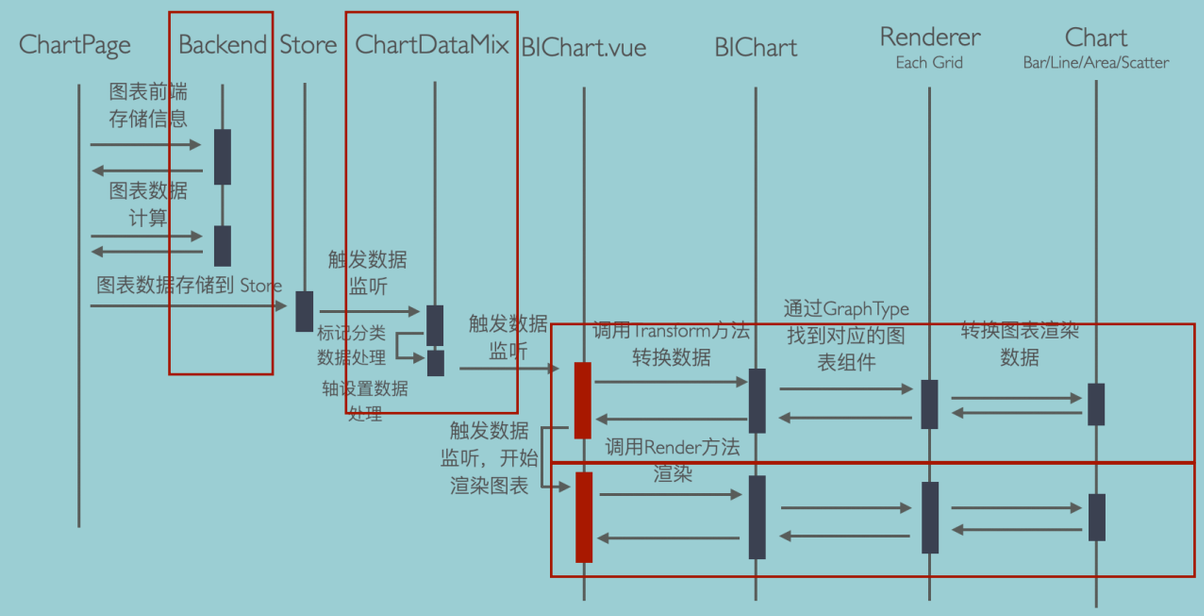
查询及基础数据封装层 API

业务及样式逻辑层 ChartDataMix

展示数据转化层 Transform

图表渲染层 Render

解决复杂度：抽象与分层



Renderer

BaseChart	BasePipeline	Axis
Line	Line	XAxis
Bar	Bar	YAxis
Area	Amount	
Scatter	AxisSetting	
Table		

复杂问题 => 层级1 层级2 层级2

普通问题 普通问题 普通问题

普通问题 普通问题 普通问题

普通问题 普通问题 普通问题

普通问题 普通问题 普通问题

Computer Science https://www.youtube.com/watch?v=1I5ZMmrOfnA

Another Level of **Abstraction**!

Socket -> TCP/IP -> HTTP

二极管 -> 逻辑门 -> ALU -> CPU

BIChart Layout 布局计算

渲染性能优化

✓ 滚动渲染轴元素，防止渲染过多节点导致卡顿

Horizontal size

YAxisTitle

XAxisTitle

yUnitHeight

yAxisHeight

9 - 8 - 7 - 6 - 5 - 4 - 3 -

yAxis

Grid (Renderer Unit)

Grid (Renderer Unit)

Grid (Renderer Unit)

Chart / Table

Scroll Bar

Horizontal size

Full Size

Baffer

yAxis

Render Area

View Port

Baffer

ScrollBar

xAxis

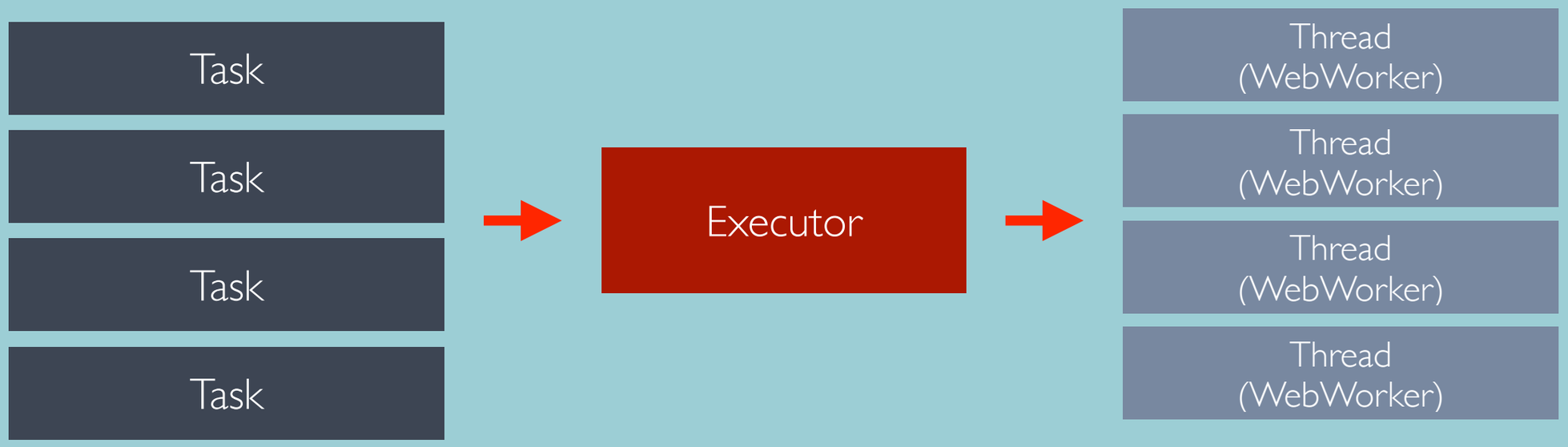
xUnitWidth

xAxisWidth

Heave Compute 大数据量计算

Executor.JS

- ✓ Run Promise MultiThread
- ✓ 减少侵入性
- ✓ 兼容多版本浏览器
- ✓ Promise not Thread!
- ✓ One time job



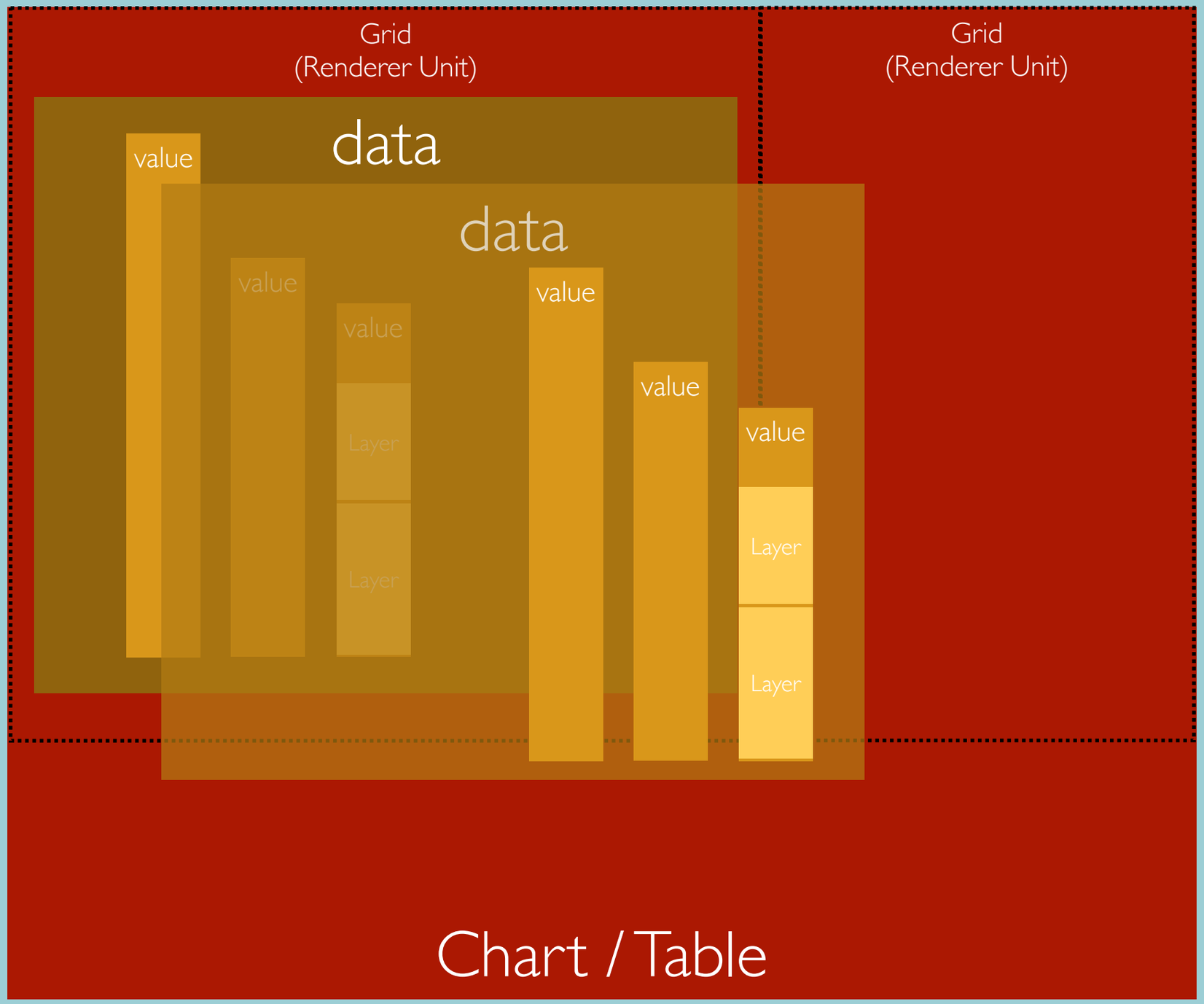
Task

```
const task = TaskGenerator(params => {

    // do heave job

    return result;
});
```

Usages

```
const worker = new Executor(task);
worker.execute(this.fullData).then(res => {
    // after execute
  })
  .finally(() => worker.terminate());
```

Data Structure Layer 图表分层结构



ChartGrid

colIndex: Number
data: Array<ChartData> ' 聚合时按字段拆分
rowIndex: Number
xName: Array<String> ' 要点级别
yName: Array<String> ' x轴名称

ChartData

values: Array<ChartValue>
xName: Array<String> ' 要点级别
yName: Array<String> ' x轴名称

ChartValue

layers: Array<ChartLayer> ' 数据下钻分层后每一层数据
record: Array<Object<key, value>>
style: Style
x: String | Number
y: String | Number

ChartLayer

marks: Array<Object<markKey, markValue>>
record: Array<Object<key, value>>
style: LayerStyle
x: String | Number
y: String | Number

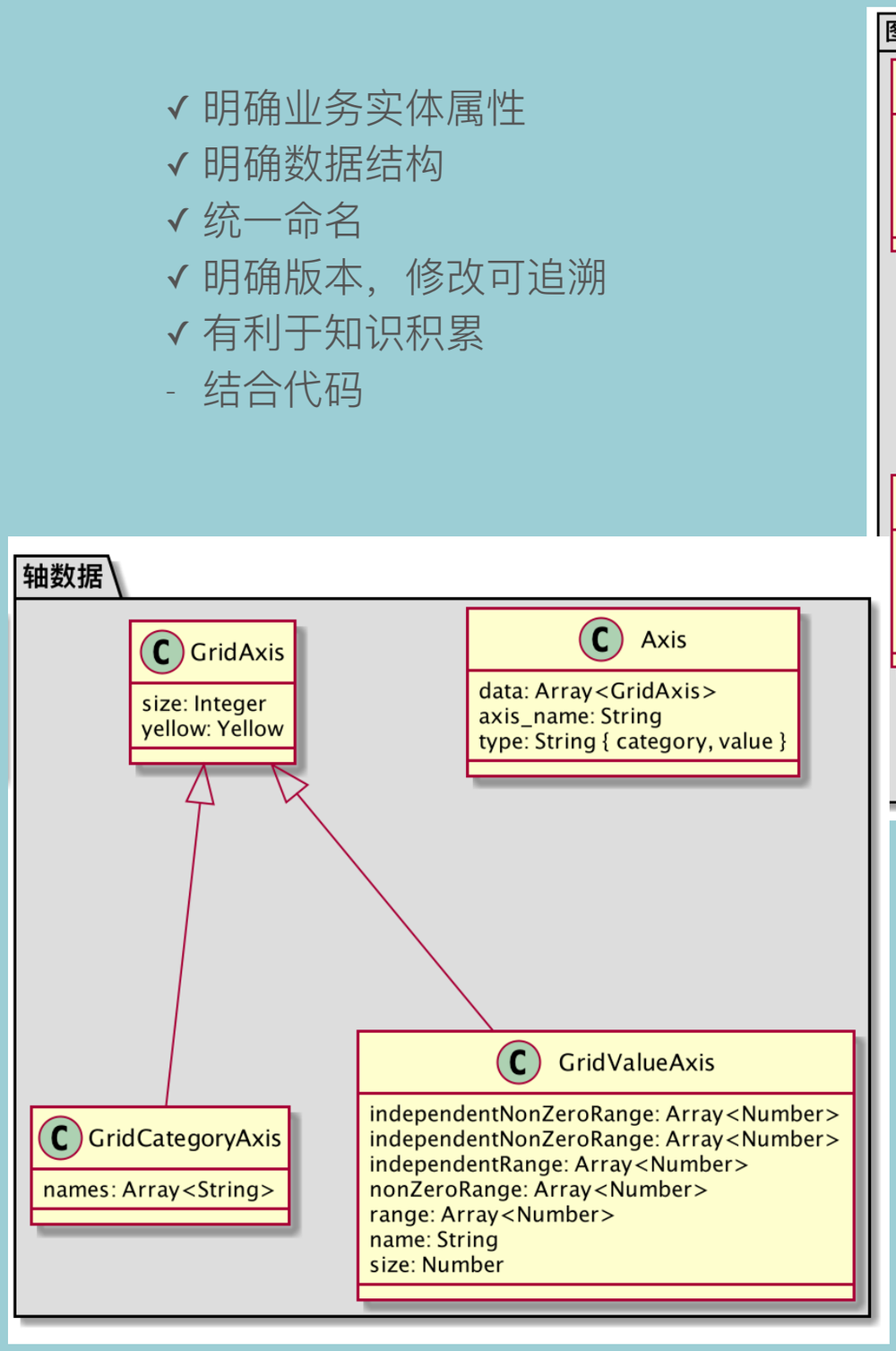
图表部分数据按照不同的业务需要拆分为四层，分别为 Grid -> Data -> Value -> Layer

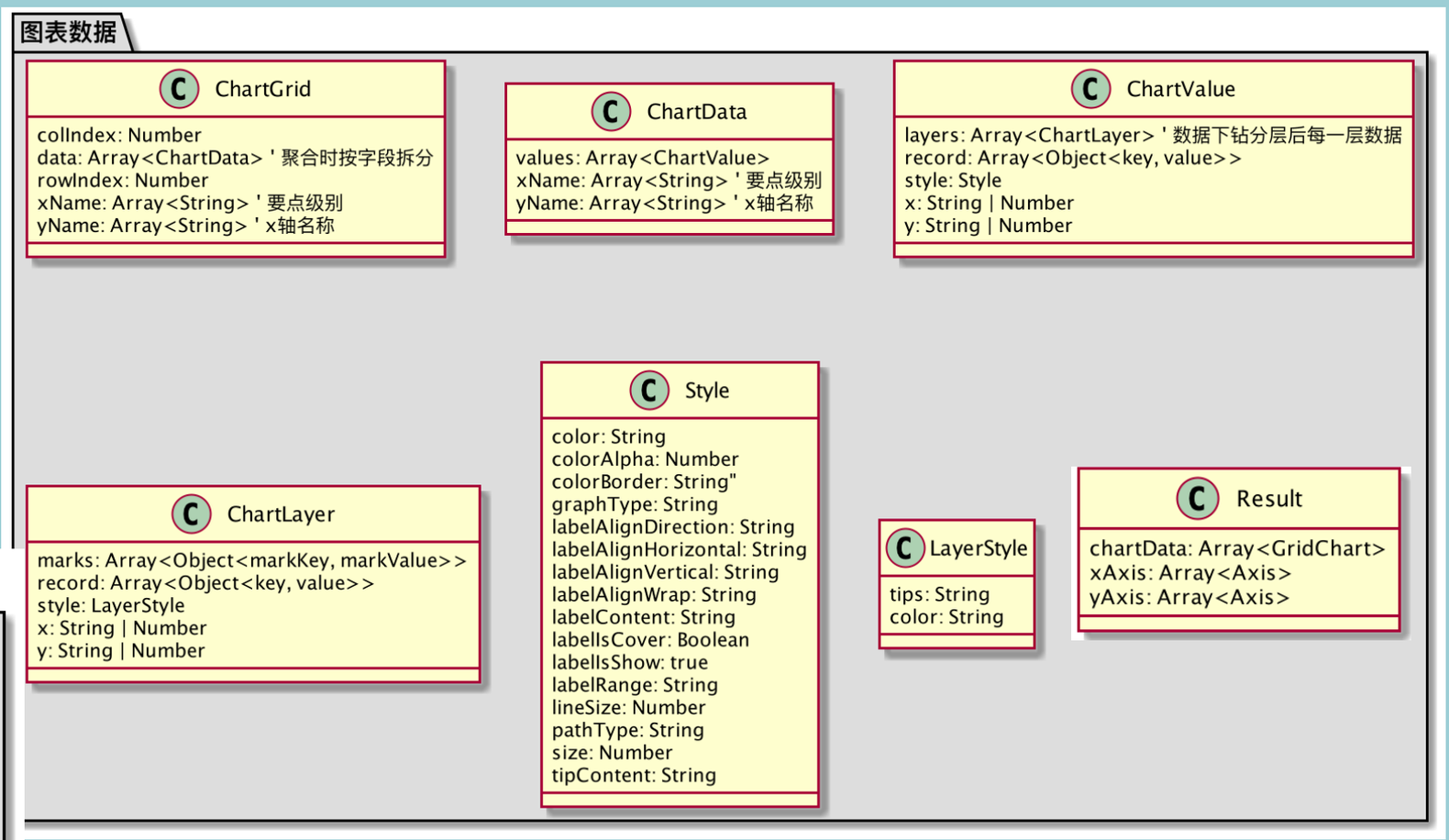
- ChartData(Grid)：单元图数据
- Data：指标聚合时按字段拆分
- Value：单个数据点数据
- Layer：按度量或维度拆分后单个数据点数据

} 非具象化

Data Structure 图表数据结构

- ✓ 明确业务实体属性
- ✓ 明确数据结构
- ✓ 统一命名
- ✓ 明确版本，修改可追溯
- ✓ 有利于知识积累
- 结合代码





数据结构　　业务实体

- 模型是现实的一种简化，他是对现实的解释
- 把与解决问题先关的部分抽象出来，忽略无关细节
- 防止在无限的细节中迷失方向

PlantUML https://en.wikipedia.org/wiki/PlantUML　　TPlant https://github.com/bafolts/tplant

Data Structure 持续重构

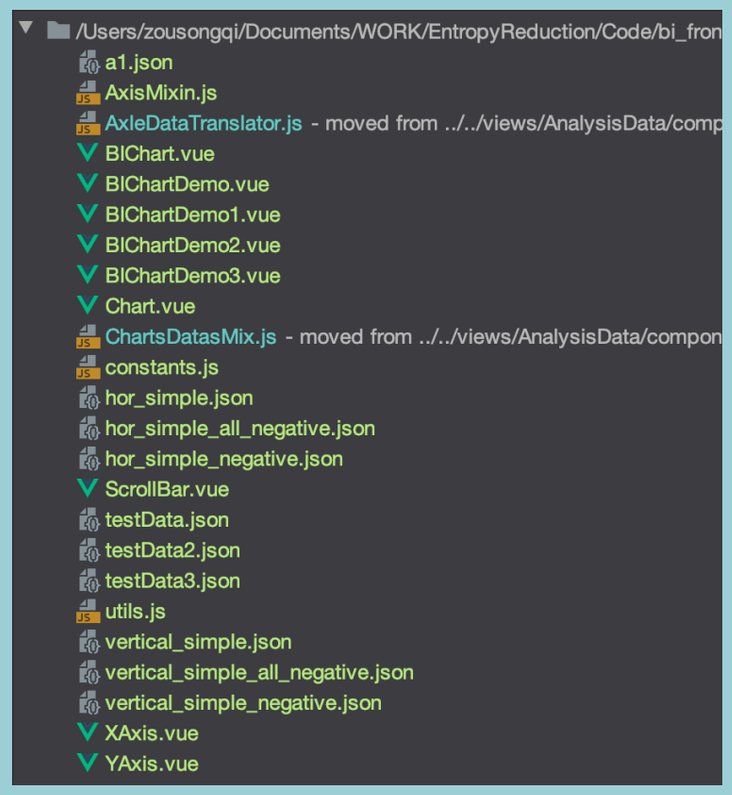
拥抱变化

持续重构

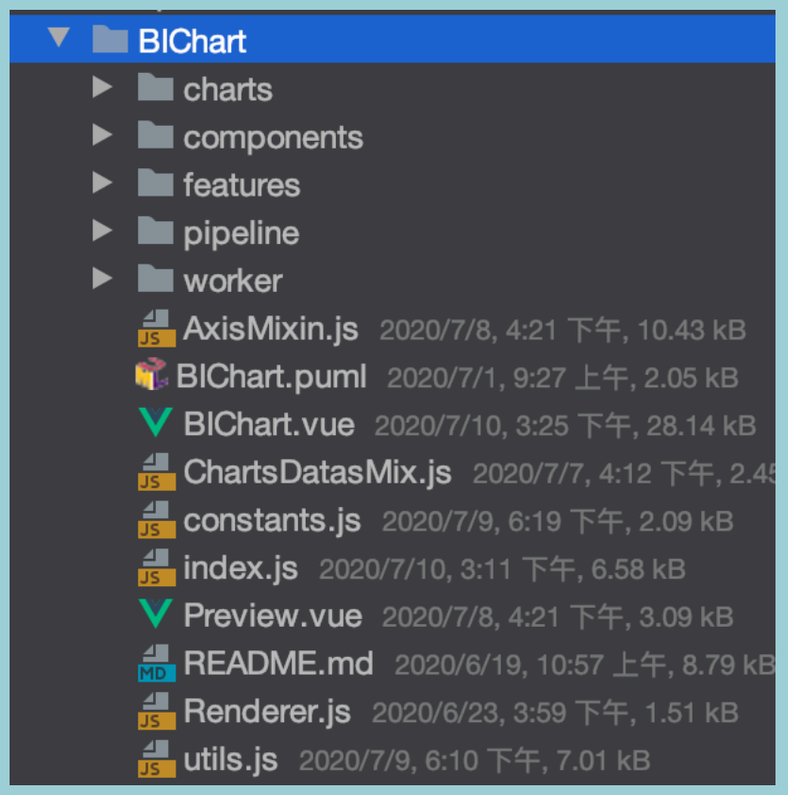
- 有价值的模型不是立即就会出现的，它们需要对领域的深入理解。这种理解是一步一步得到的
- 基于最初的（可能是不成熟的）模型实现一个初始设计，再反复改进这个设计。
- 每次团队对领域有了新的理解之后，都需要对模型进行改进，使模型反映出更丰富的知识，而且必须对代码进行重构，以便反映出更深刻的模型。

——《领域驱动设计》

v1.3 接手之前



两次需求迭代之后



重构原则

1.测试优先原则

TDD:测试驱动开发

2.OCP原则

Open For Extension:开放封闭原则

3.小步快跑原则

大布局、小迭代

进化式设计和增量开发

避免过度设计，对于未来的变化，既不要考虑的太多，也不能一点都不考虑

代码满足当前需求，并留有可扩展余地

4.数据一致性原则

分库分表（横向、纵向）

字段合并、冗余

索引优化、数据缓存

There is never enough time to do something right, but there is always enough time to do it over

「时间再多一件事情也不可能做的完美，但总有时间做完一件事情」—— 康威第二定律

关于业务的思考

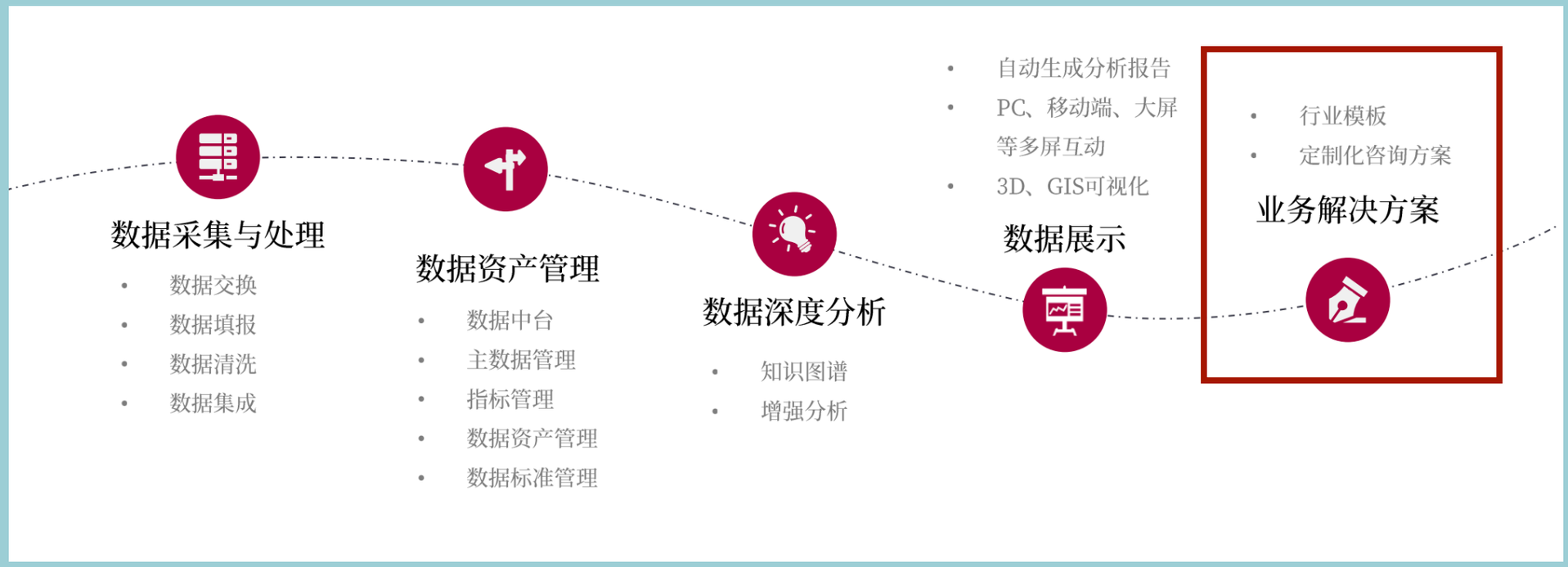
Intelligence or Insight 智能 or 洞察

工具产品：
分析 -> 洞察 -> 业务创新

只是一种洞察（Insight）

业务方案：
分析 -> Run Business Intelligently

真正的商业智能（Intelligence）



产品质量与业务场景沉淀同样重要

甲子光年 大数据时代的商业智能 https://mp.weixin.qq.com/s/lwQU2O3Lj6j_38kFm7oV1A

THANKs

Resources

1. 《人月神话》—— Frederick P. Brooks, Jr.
2. 《领域驱动设计》—— 埃里克·埃文斯
3. 为什么要持续重构 https://cloud.tencent.com/developer/article/1155098
4. 软件开发的核心-控制复杂度 https://zhuanlan.zhihu.com/p/21885300
5. 降低软件复杂性一般原则和方法 https://tech.meituan.com/2019/09/19/common-method-of-reduce-complexity.html
6. The Magical Number Seven —— George A. Miller http://psychclassics.yorku.ca/Miller/
7. Philosophy of Software Design —— John Ousterhout https://www.youtube.com/watch?v=bmSAYlu0NcY
8. Software complexity https://www.sciencedirect.com/topics/computer-science/software-complexity
9. Conways Law http://www.melconway.com/Home/Conways_Law.html
10. Computer Science https://www.youtube.com/watch?v=I15ZMmrOfnA

Github / Blog

@georgezouq

公众号

@跳动的神经元