Creating an intelligent reality with Cogine

# REALITY WORLD

## 开 源 & 线 下 技 术 分 享 会

官网： https://realityworld.com

Github：https://github.com/Cogine/RealityWorld

秦春林

# 内容

## 未来数字世界会有什么新特征

去中心化

元宇宙

AI

XR

## 对软件构造方法有什么新需求

互操作性/多应用协作

语义抽象/人人可编程

组合性/动态个性化

## 带来什么新的增量价值

进化计算带来更智能的软件

人人参与的数字经济

促进硬件和体系结构变革

对话式计算带来新的交互变革

被动式交互带来更复杂的数字化

# 一个疯狂的梦想　一段疯狂的冒险

行走在无人区，最艰难的不是孤独与挑战，而是不断的自我怀疑

# 开源目的

**构建公共共识：**

为未来的数字世界基础架构的技术发展方向提供一种系统的视角。

**技术学习交流：**

为程序员对相关技术的学习提供一些不同维度的理解。

**可能技术路径：**

为上述的架构体系提供一种可行且简易的技术实现方案。

**挖掘产品形态：**

借助社区的力量去挖掘这种新型技术架构能够支撑的产品形态。

# 起源：User generated content

2D -> 3D -> 程序

**UGC / AIGC**

文字、图片、视频、3D模型等

**河图**

基于真实场景定位信息的内容创建

**Snapchat**

基于场景理解算法相关的程序构建

**Roblox**

构建完整的3D应用程序

**堡垒之夜**

构建更具互操作性的社交游戏程序


* 图片来自Snapchat官网

* 图片来自Epic Games官网

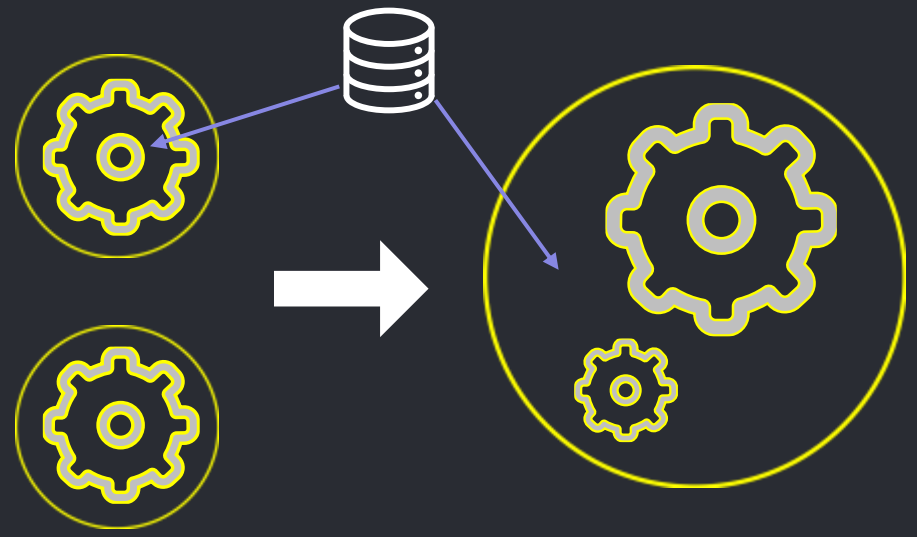
广义具有程序功能的UGC，就是模拟真实世界的运作机制:生产、协作与消费，即真正的元宇宙

# 这使我意识到

软 件 构 造 方 法 层 面 的 技 术 才 是 未 来 的 关 键

# 未来是多应用协作时代

我们当前的整个软件构造方法体系都是基于内存隔离的、中心化的



传统的软件是受内存隔离的，每个应用提供独立的功能

未来的应用是高度互操作的，所有应用相互协作和交互来实现更复杂的数字智能

### Web 3.0

没有互操作，就没有共识算法；数据与代码耦合，就没有去中心化

### 元宇宙

缺乏互操作，应用的形态就无法进化，互操作和协作才能带来真正数字世界的丰富和繁荣

### 大模型

单一智能体能力往往有限，多智能体协作才能带来真正智能

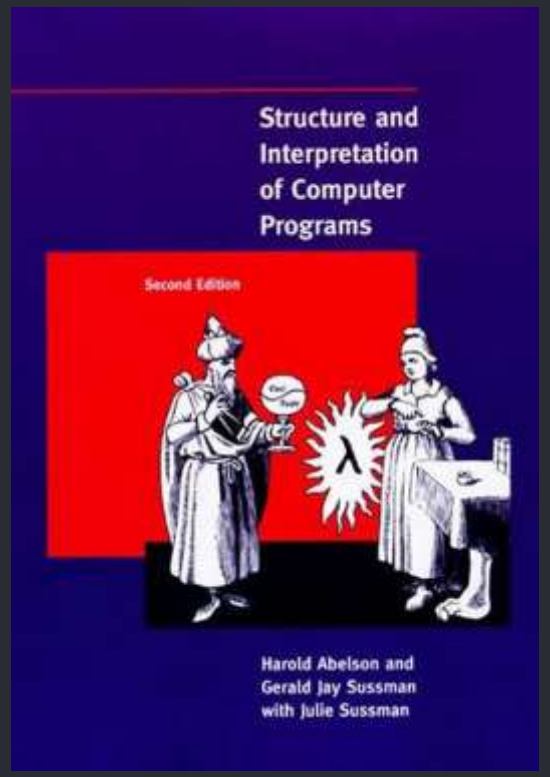
### XR

未来的交互会弱化应用的概念，用户面对的是总体功能而不是多个App，这要求程序数据和功能高度互操作

# Reality World 愿景

构建一个多应用协作的软件构造方法和架构

# 计算机程序的构造和解释




豆瓣评分
9.6

**课程设计目标：**

① 希望建立起一种共识：计算机语言并不仅仅是一种让计算机去执行操作的方式，更重要的是，它是一种表述有关方法学的思想的新颖的形式化媒介。
② 我们相信，在这一层次的课程里，最基本的材料并不是特定程序设计语言的语法，不是高效完成某种功能的巧妙算法，也不是算法的数学分析或者计算本质的基础，而是一些能够控制大型软件系统复杂性的技术。

**通过隐藏细节，创建抽象来控制复杂性：**

① 过程抽象
② 数据抽象

# Reality World的目标

**应用程序**

**Cogine**

语义抽象 进化计算

**编程语言**

过程抽象 数据抽象

**对上述两种目标的升华：**

① 通过将进化看作一种计算，以及构建支持进化计算的架构，使程序成为一种创建和表达智能的形式化媒介。

② 通过构建一个更高层次的语义抽象，进一步更简单地控制大型软件系统的复杂性，并支持如去中心化、元宇宙互操作和大模型应用多智能体相互协作的新需求。

# 过程的抽象

**表达式：**

```
(* 2 3.14159 10)
62.8318
```

解释器直接对表达式求值。

**命名和环境：**

```
(define pi          3.14159
(define radius  10
(define circumference (* 2 pi radius))
circumference
62.8318
```

define定义了一种最简单的抽象，它允许用一个简单的名字去引用一个组合运算的结果。

**过程抽象：**

```
// 过程定义的一般形式
(define (<name> <formal parameters>) <body> )

(define (square  x) (* x x)
(square 21)
441
```

过程是一类抽象，它描述了一些对于数的复合操作，但又并不依赖于特定的对象。它使我们能将一般性的计算方法，用程序设计语言里的元素明确表述。

**高阶函数抽象：**

```
(define (sum-integers a b)
   (if (> a b)
       0
       (+ a (sum-integers (+ a 1) b))))
```

过程限制只能以数作为参数，高阶函数则进一步，它使我们可以将过程作为参数，或者以过程作为返回值，使我们可以显式地用程序设计语言中的要素去描述这些过程抽象。

# 数据的抽象

将一个复合数据的使用，与该数据怎样由更基本的数据对象表示和构造的细节隔离开

**不是指struct定义：**

```
pub struct Message {
    pub sender: String,
    pub receiver: String,

    pub content: String,
}
```

**访问数据的接口：**

```
pub trait MessageTrait {
    fn get_sender(&self) -> &String;
    fn get_receiver(&self) -> &String;
    fn get_context(&self) -> &String;

    fn new(
        sender: &str,
        receiver: &str,
        content: &str);
}
```

**几种不同的形式：**

① 针对不同表示的通用操作
② 针对不同参数种类的通用操作
③ 不同类型数据的组合：强制转化
④ 类型的层次结构：继承

只涉及数据的表示和使用抽象，还不涉及运行时和模块化管理

构造出一些使用复合数据对象的程序，使它们就像是在“抽象数据”上操作一样。
一种“具体”数据表示的定义，也应该与程序中使用数据的方式无关。
这两者之间的界面将是一组过程，称为“选择函数”和“构造函数”。

# 构造模块化的大型系统的策略

将大型系统看成一大批对象，它们的行为随时间的进展而不断变化

**局部状态变量：self/this**

```
impl Message {
    pub fn print_content(&self) {
        let content = self.get_context();
        print!("{}", content);
    }
}
```

与所有的状态都必须显式地操作和传递额外参数的方式相比，通过引进赋值和将状态隐藏在局部变量中的技术，我们能以一种更模块化的方式构造系统。

命令式程序设计：广泛采用赋值的程序设计
函数式程序设计：不采用任何赋值的程序设计

**引用透明性：**

```
impl Message {
    pub fn set_message(&self, msg:&str){
        self.content = msg.to_string();
    }
}
```

赋值打破了引用透明性，其结果是需要在程序中维护真实世界的时间状态，同一个表达式在不同的状态下具有不同的值，形式化推理变得异常困难。

# 不完美的计算模型

当对象之间不共享的状态远远大于它们所共享的状态时，对象模型就特别好用。

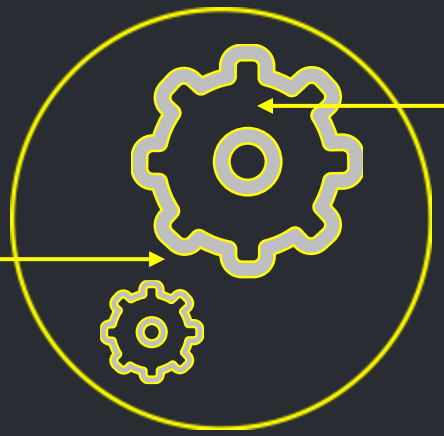
对象模型仅考虑的是模块化，且前提是子系统之间相对比较独立，但实际情况是对象之间的交互往往都很复杂，对象模型没有考虑这种交互特性，仅仅提供一种直接的对象引用访问，你需要管理所有对象的地址和组织。

本章开始时提出了一个目标，那就是构造出一些计算模型，使其结构能够符合我们对于试图去模拟的真实世界的看法。我们可以将这一世界模拟为一集相互分离的、受时间约束的、具有状态的相互交流的对象，或者可以将它模拟为单一的、无时间也无状态的统一体。每种观点都有其强有力的优势，但就其自身而言，又没有一种方式能够完全令人满意。我们还在等待着一个大统一的世界。

ChorSeb 2014-02-04 03:46:49
传世神书啊！Harold Abelson有传教士神性的光芒啊。。第三章结束有种故事讲完的感觉，而且是个悲伤的不完美的故事。

# 一种新的软件构造方法

我们将围绕这两个层面提出一种新的软件构造方法



## 外在需求：进化计算

**去中心化：**
随着Web 3.0应用的兴起，人们会越来越关注应用数据的隐私，需要一种原生的架构能够使得用户数据与程序功能分离。

**互操作性：**
元宇宙尝试去构建一个更统一的、具有互操作性的数字世界，应用之间相互交互才能构建更丰富的体验。

**多智能体协作：**
大模型应用要求多个智能体程序可以共同协作，也即意味着多个匿名的程序需要在一个内存环境共存并且相互交流。

## 内在需求：语义抽象

**结构组织的复杂性：**
OOP以对象而不是逻辑为单元来组织程序，并且继承等复杂机制带来了复杂且难以理解的程序结构。

**数据管理的复杂性：**
程序内存中对象的创建、状态和上下文管理是程序开发最复杂的部分之一，它们大多是围绕编译器和硬件而设计的，与业务无关。

**数据与功能耦合：**
传统的程序天生的将数据和功能耦合在一起，通过内存隔离进行保护，尽管保护机制简单，但是却失去了互操作性。

# 外在需求

# Open metaverse

So I started out by trying to say what the metaverse is. And my first version of this statement was it's a real time 3d social medium, where people can engage in shared experiences together, but that doesn't really work. Does it? Um, doom is the metaverse then Halo's the matter, every multiplayer game's the metaverse. So that's not quite enough. I think that players being able to create and contribute a large amount of the content to the world is a critical part of this new medium. If players can't create their own stuff, then the whole, platform's just going to be limited to this one central company can build.

Foundational principles & technologies for the metaverse
Tim Sweeney






* 图片来自Epic Games官网

It‘s a real time 3d social medium, where people can create and engage in shared experience as equal participants in an economy with societal impact.

# Standard & Interoperability

Now, if we're going to try to build the metaverse as an open platform, then we're going to need a degree of industry standards to specify what it is, how it works, how all the participants interoperate.

HTML, Email, Audio, Video, Image, Dom, Web service, Json, XML……


* 图片来自Epic Games官网

3D元宇宙时代的标准格式会更加复杂，因为3D引擎能够做的事情远远超过Web可以做的事情：

GLTF　USD　MDL　粒子
角色　动画　物理　环境音频
服务器实时通信　自定义交互
社交　自定义物体　……

有些容易标准化，有些则是个性化的

Verse language 可能是这方面真正的商业化行动

我们需要一种原生互操作的能力

So we need first class ways of treating all of that.

# USD

## USD是一个能够定义格式的工具和运行时

```
class ComplexPrim "ComplexPrim" (
    doc = """An example of a untyped IsA schema prim"""
    # Inherits from </SimplePrim> defined in simple.usda.
    inherits = </SimplePrim>
    customData = {
        string className = "Complex"
    }
)
{
    string complexString = "somethingComplex"
}
```

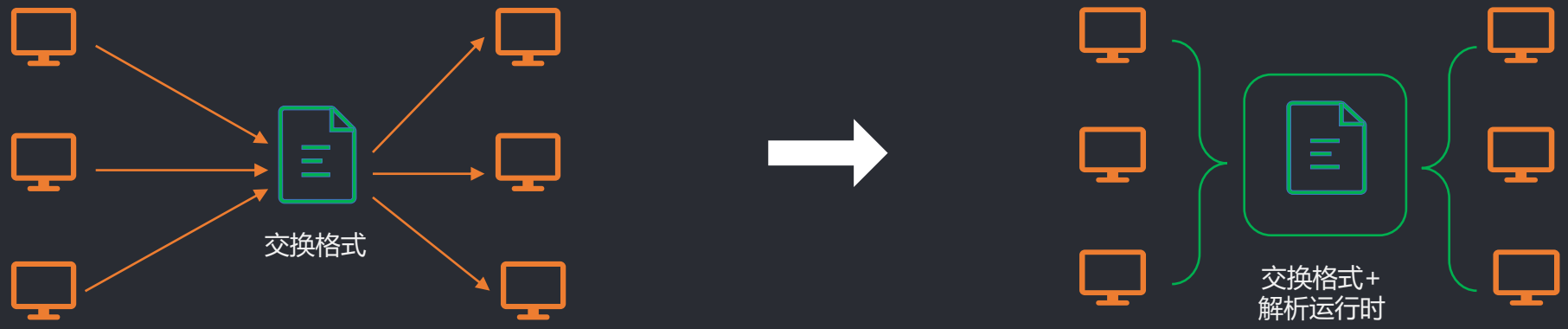
```
$ usdGenSchema schema.usda
Processing schema classes:
SimplePrim, ComplexPrim, ParamsAPI
Loading Templates
Writing Schema Tokens:
    unchanged extras/usd/examples/usdSchemaExamples/tokens.h
    unchanged extras/usd/examples/usdSchemaExamples/tokens.cpp
    unchanged extras/usd/examples/usdSchemaExamples/wrapTokens.cpp
Generating Classes:
    unchanged extras/usd/examples/usdSchemaExamples/simple.h
    unchanged extras/usd/examples/usdSchemaExamples/simple.cpp
    unchanged extras/usd/examples/usdSchemaExamples/wrapSimple.cpp
    unchanged extras/usd/examples/usdSchemaExamples/complex.h
    unchanged extras/usd/examples/usdSchemaExamples/complex.cpp
    unchanged extras/usd/examples/usdSchemaExamples/wrapComplex.cpp
    unchanged extras/usd/examples/usdSchemaExamples/paramsAPI.h
    unchanged extras/usd/examples/usdSchemaExamples/paramsAPI.cpp
    unchanged extras/usd/examples/usdSchemaExamples/wrapParamsAPI.cpp
    unchanged extras/usd/examples/usdSchemaExamples/plugInfo.json
Generating Schematics:
    unchanged extras/usd/examples/usdSchemaExamples/generatedSchema.usda
```

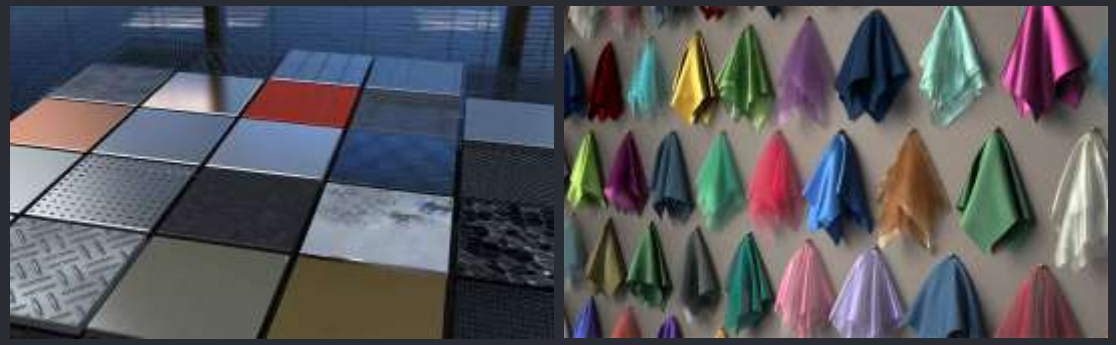
```
#usda 1.0

def ComplexPrim "Complex"
{
    string complexString =
        "a really complex string"
        int intAttr = 10
        add rel target = </Object>
}

def Xform "Object" (
    prepend apiSchemas = ["ParamsAPI"]
)
{
    custom double params:mass = 1.0;
    custom double params:velocity = 10.0;
    custom double params:volume = 4.0;
}
```


* 图片来自openusd.org

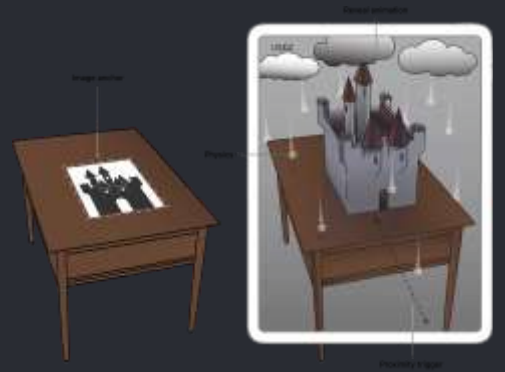
# 交换格式的运行时

统一交换格式的运行时，可以大大减少互操作性的复杂性



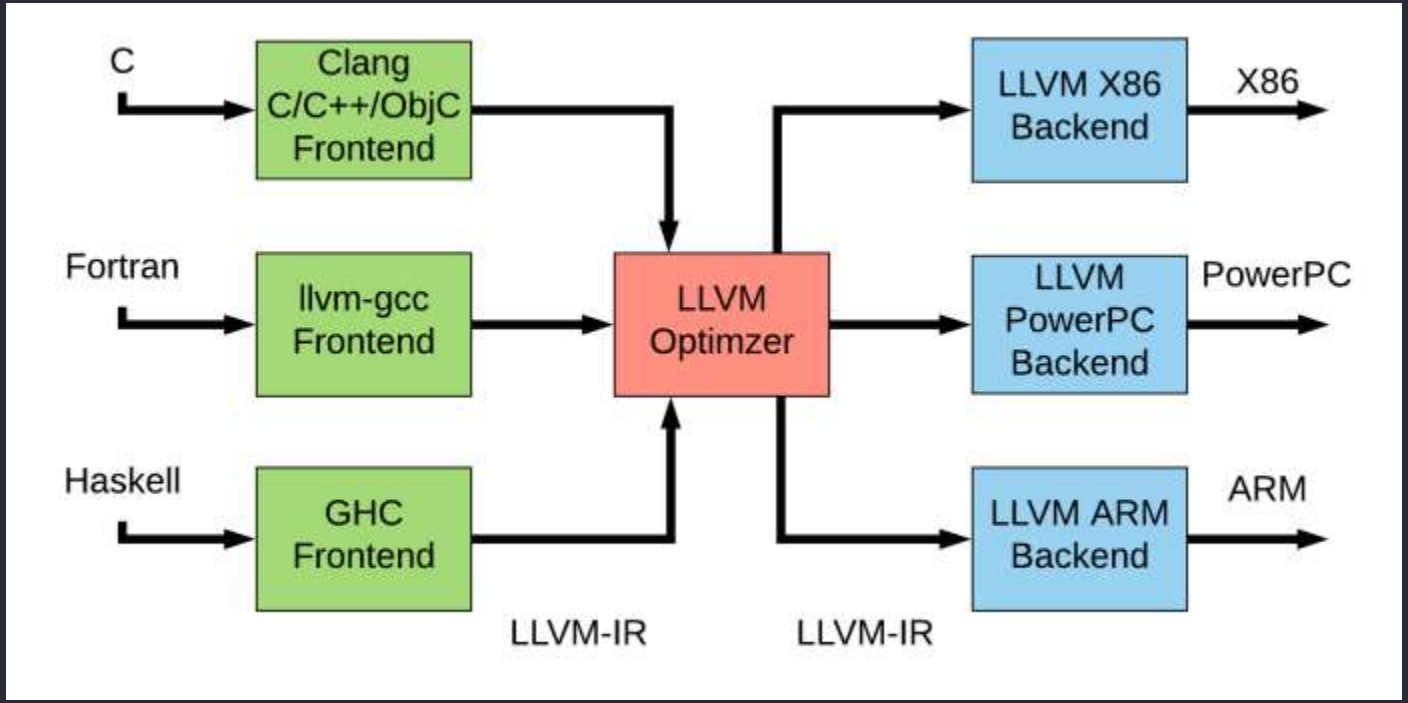
* 图片来自NVidia

* 图片来自Apple

# LLVM：互操作性与模块化

有一个很统一的互操作架构就可以避免碎片化




先定义交互格式，就能促进模块化和互操作

# 互操作性的本质是碎片化

# MLIR

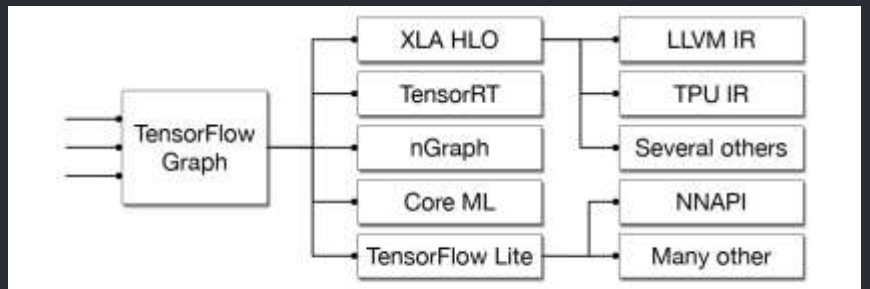


Fig. 1. TensorFlow execution spanning different frameworks.

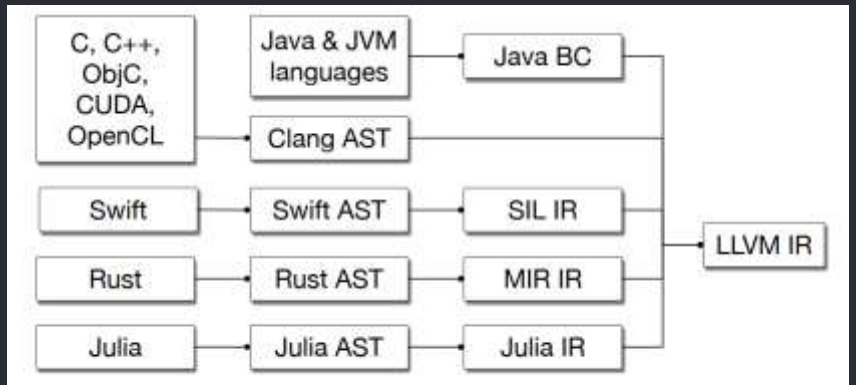


Fig. 2. Compilation pipeline with mid-level language IRs.

```
def Toy_Dialect : Dialect {
   let summary = "Toy IR Dialect"; let description = [{
      This is a much longer description of the
      Toy dialect.
      ...
   }];
  // The namespace of our dialect.
  let name = "toy";
  // The C++ namespace that the dialect class // definition resides in.
  let cppNamespace = "toy";
}
```

---

```
class ToyDialect : public mlir::Dialect {
 public:
  ToyDialect(mlir::MLIRContext *context)
   : mlir::Dialect("toy", context,
       mlir::TypeID::get<ToyDialect>()) {
   initialize();
  }
  static llvm::StringRef getDialectNamespace() {
    return "toy";
  }
  void initialize()
}
```

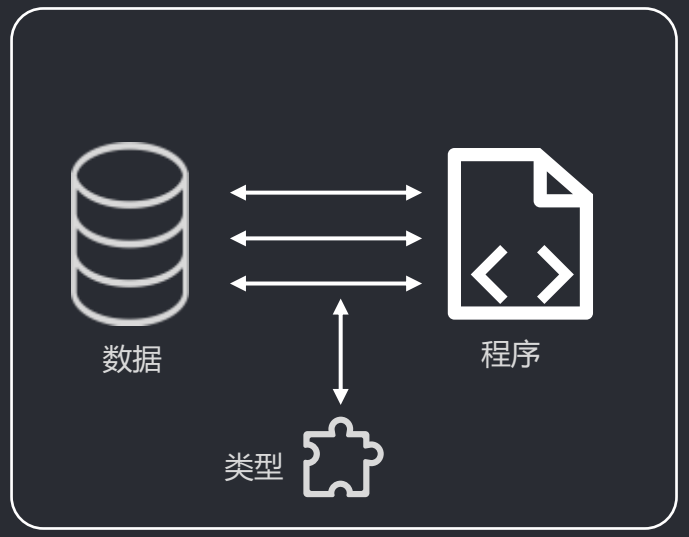
# 启示

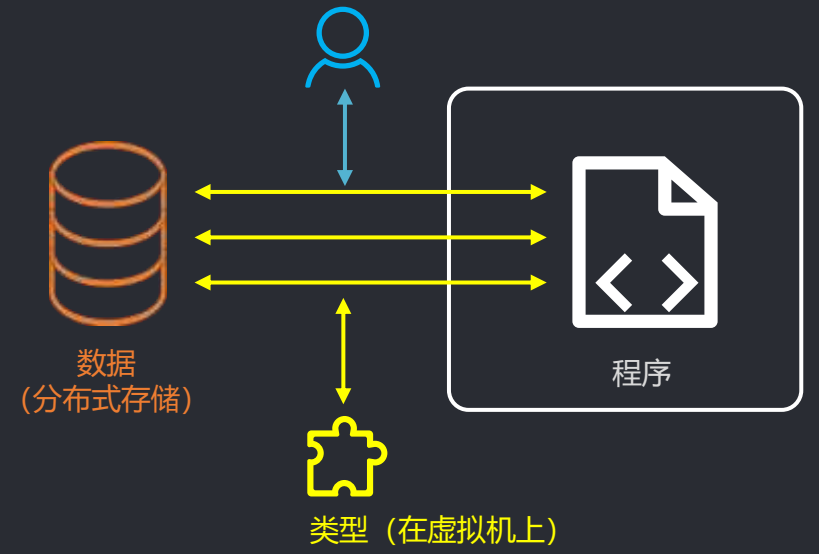
在语言和运行时级别构建一个动态类型系统

Interoperability as first class feature

# 区块链的软件构造视角

首先将数据、程序和类型解释拆分开



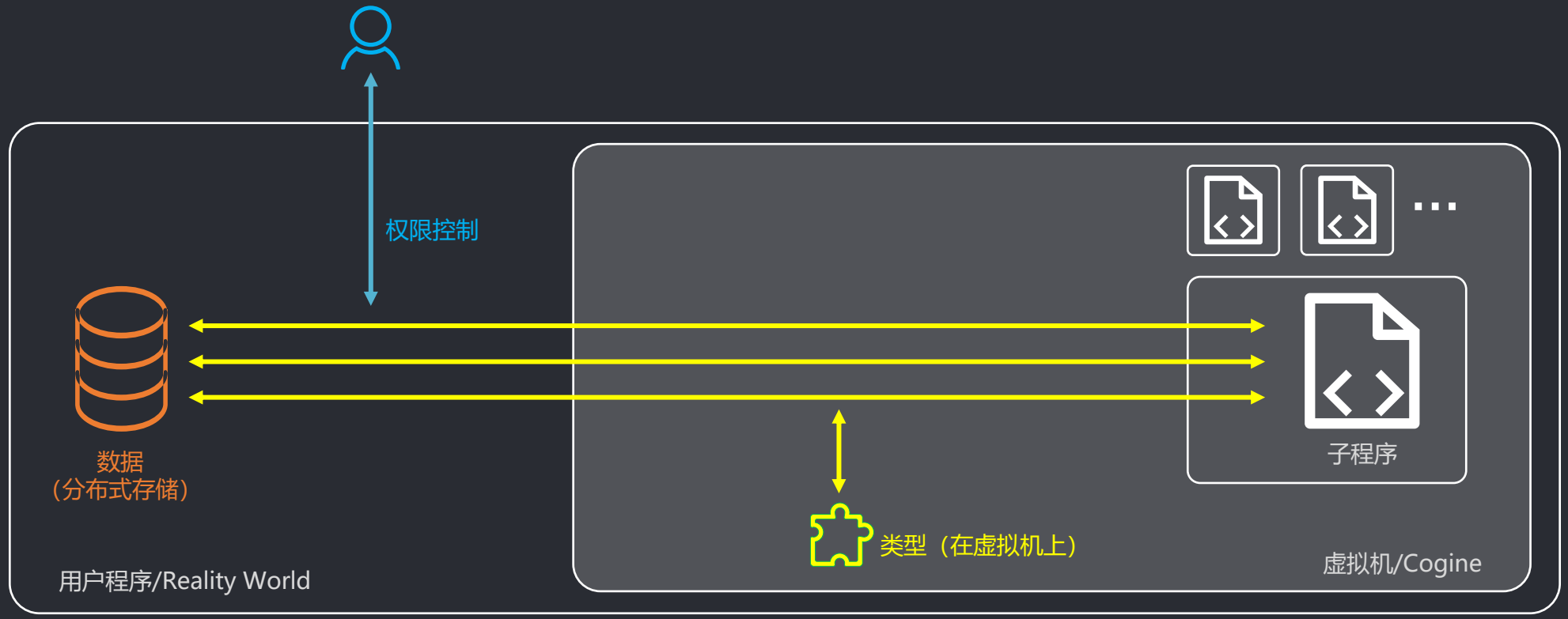

## 传统软件结构

在传统的软件构造方法中，数据、类型和程序是耦合在一起的，程序及其开发者拥有对数据和类型的绝对控制权，且用户几乎不容易管理和访问自己的数据。这种程序结构便利带来的中心化，并且软件之间无法互操作。

## 区块链软件结构

区块链架构将这几者剥离开，数据存储在分布式账本中，程序则对应每个虚拟机上交易的程序，类型则内置于每个虚拟机上，它可以看作是独立于客户交易程序。传统的区块链程序只有一个数据类型。

# 如果区块链不是只有一个类型




**几点修改：**

程序：只包含功能代码
类型：程序可以访问任意类型
数据：与程序代码隔离
安全：数据的访问由用户控制

数据存储只是与程序代码隔离，不需要分布式存储，实现了去中心化

# 如果能够控制类型和数据访问

系统就可以定制任意类型，实现任意的互操作和交互

# 不仅如此

互操作带来进化计算，实现智能的一种方法

# 走向智能计算

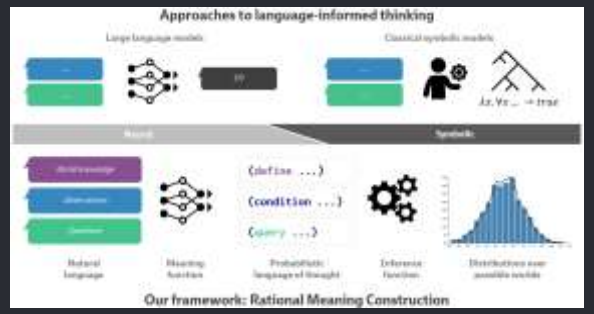
我们的目标绝不仅仅是让每个人都能编写以前那种程序

# 智能是什么

## 符号主义

假设人类思维的基本单元是符号，
基于符号的计算就构成认知过程。

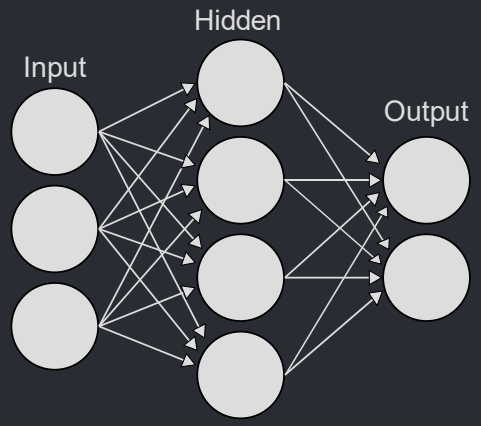
困难：有限的知识、不确定性和噪声




From Word Models to World Models: Translating from Natural Language to the Probabilistic Language of Thought

## 连接主义

心理状态可以描述为网路中神经单元上数字激活值的N维向量，记忆是透过修改神经单元之间的联结强度来创建的。




## 行为主义

基于“感知-行动”的行为智能模拟方法。控制论把神经系统的工作原理与信息力量、逻辑和计算机联系起来。


**The Bitter Lesson**：试图把人类经验教给AI是行不通的，至今为止所有突破都依靠算力提升，继续利用算力的规模效应才是正确道路。

深度学习技术缺乏表示因果关系的方法，并且可能面对在对一些抽象知识的采集，没有执行逻辑推理的明显方法，离整合抽象知识还很远，例如关于什么是对象，它们的用途以及如何使用信息。

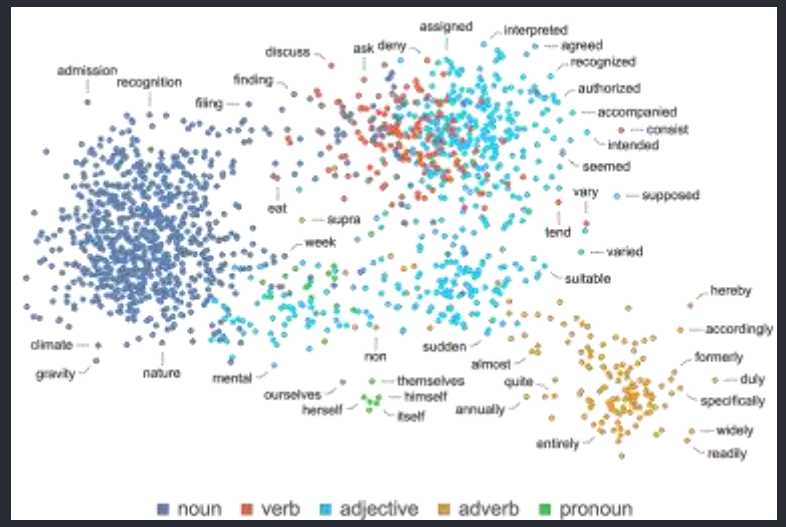
Gary Marcus

AGI架构应该是模块化和分布式的，而不是一个巨大的集中模型。应该是持续的在线学习，而不是预训练之后大部分参数就不再更新。总体目标是模拟一个具有内在动机和持续学习能力的虚拟智能体，在虚拟环境中持续学习。

# 大语言模型

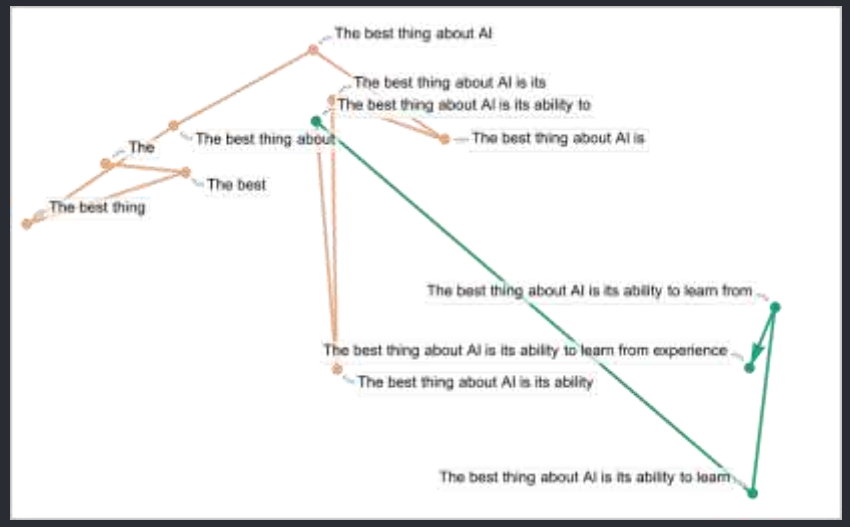
| The best thing about AI is its ability to | | |
|---|---|---|
| | learn | 4.5% |
| | predict | 3.5% |
| | make | 3.2% |
| | understand | 3.1% |
| | do | 2.9% |

图片来自： What Is ChatGPT Doing … and Why Does It Work?



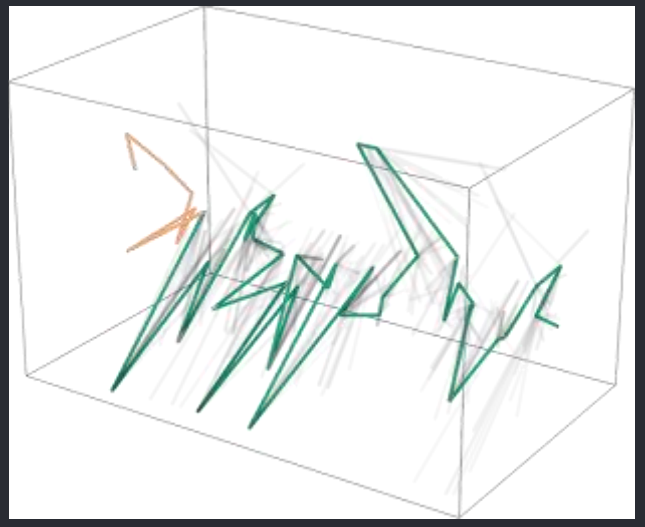

## Linguistic feature space

我们可以将大模型内部的文本表述看作一个语言学的特征空间，其中每个单词都有自己的一个位置，当然实际从计算考虑是以Token的形式存储。




## 预测轨迹

基于Prompt的文本预测补全就可以看作是在这个语言特征空间发射和追踪一个轨迹，最终的轨迹构成预测结果。其中这些轨迹的概率由大模型内部的参数表述。

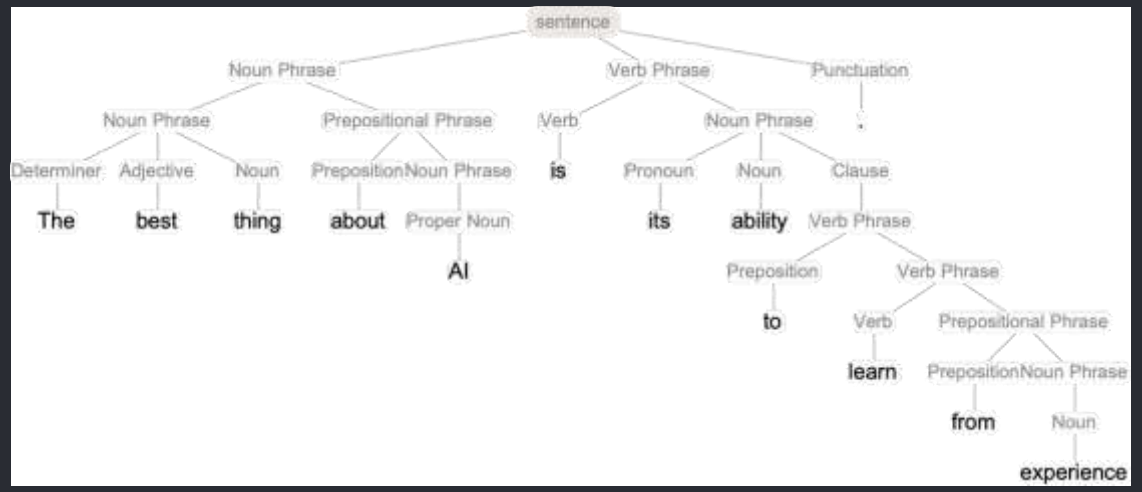


## 预测结果

经过多次迭代形成的完整轨迹。如果temperature参数设为0，每次预测总是取概率最大的值，因此结果是唯一的，大于0的temperature则会得到不同的结果。

# 语言本能

大模型的输出为什么看起来那么流畅和“可信”？


## 语言是人的一种本能

语言能力很大一部分是源自人类遗传基因，因此世界上所有的语言似乎都拥有相同的结构。

语法构造的规则有两个后果：
1）有限域的无限应用
2）独立自主，与认知和意义无关

## 语言的语法规则线性性

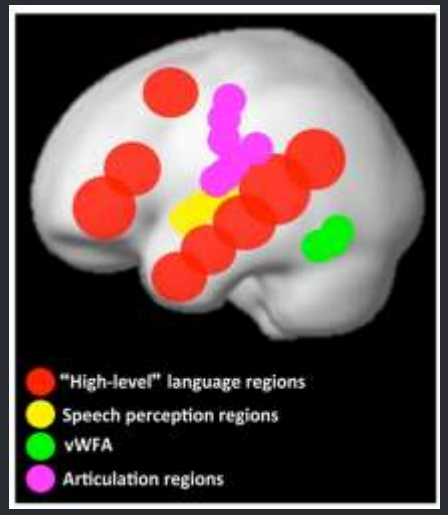
语言的表述是线性的，生成语法的规则限制了语言文字的语法结构，例如我们能够区分平淡无奇的“狗咬人”和堪称新闻的“人咬狗”，因此LLM模型的输出总是看起来语法正确的。

此外，无限的构造语料使得整个问题的空间几乎是无限的，而不是离散的。

# 大模型具有智能吗？

语言（Language）和思维（thought）不是同一件事

Language and thought are not the same thing: evidence from neuroimaging and neurological patients




## 语言和思维是不同的

认知理论长久以来都认为语言和思维虽然有关但却是不同的，思维涉及目标驱动的世界模型，推理，决策，它们构建关于世界信念的心智模型，可以基于观察更新，支持理性推断与策略来实现目标。

Faith and Fate: Limits of Transformers on Compositionality

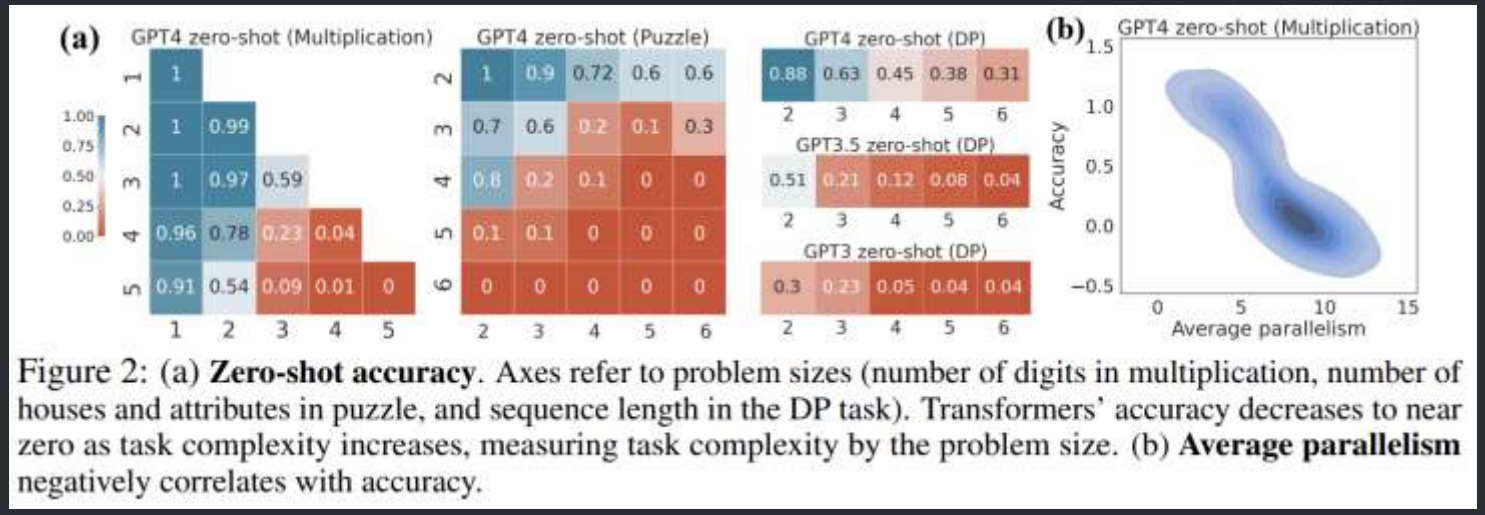



Figure 2: (a) **Zero-shot accuracy**. Axes refer to problem sizes (number of digits in multiplication, number of houses and attributes in puzzle, and sequence length in the DP task). Transformers' accuracy decreases to near zero as task complexity increases, measuring task complexity by the problem size. (b) **Average parallelism** negatively correlates with accuracy.

## 组合是非线性的

1）很多常见的任务如编程，数学计算，以及很多逻辑问题都是组合问题，组合是非线性的。
2）人类在构造语言时虽然遵循线性语法规则，但是在理解语言时却需要借助很多其它的神经网络共同完成。

# 智能到底来自哪里？

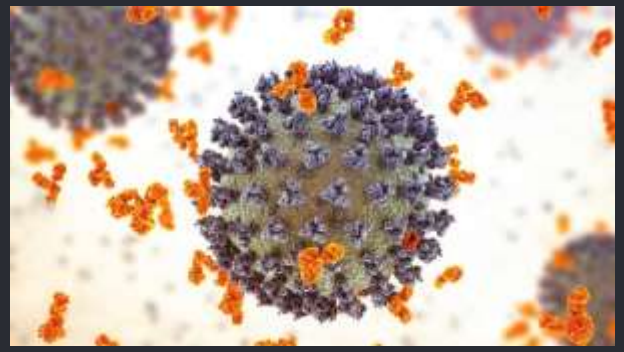
如果说人的智能是自然进化的产物，那么智能的密码应该来自于进化

# 复杂系统

## 还原论无法解释复杂现象

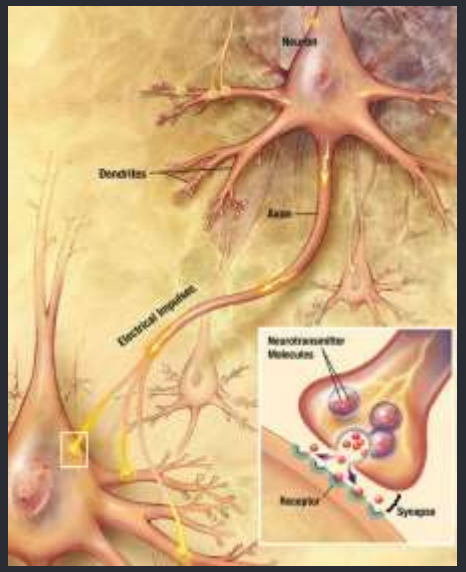

### 复杂系统

复杂系统是由大量组分组成的网络，不存在中央控制，通过简单运作规则产生出复杂的集体行为和复杂的信息处理，并通过学习和进化产生适应性。



### 免疫系统

它们是如何相互协作，从而使系统作为一个整体能够“知道”环境中存在何种威胁，并产生出应对这种威胁的长期免疫力，又如何避免攻击身体？




### 大脑

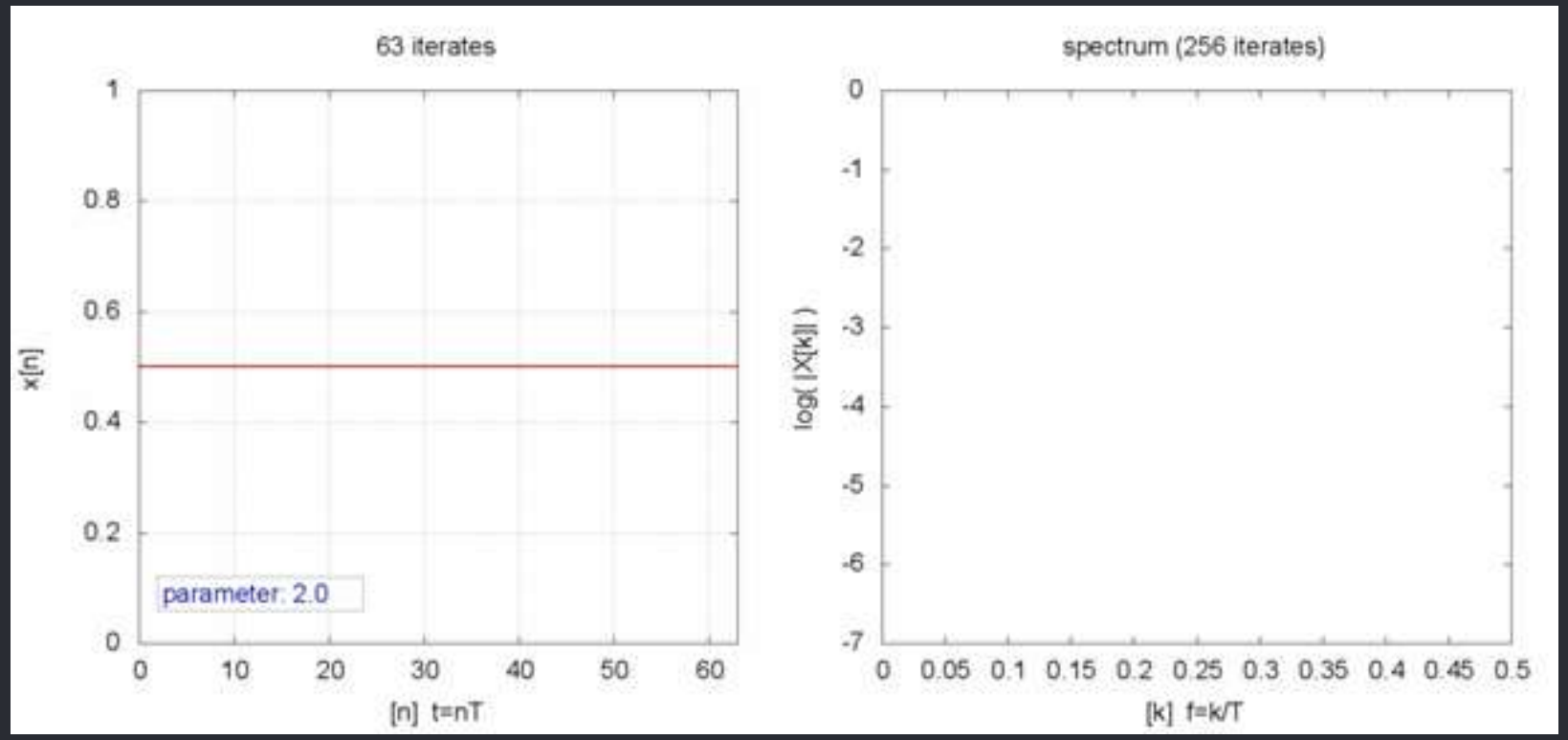
大量神经元如何一起协作产生出整体上的认知行为，它们怎样让大脑能够思考和学习新的事物？

### 蚁群

蚂蚁的个体行为如何形成庞大而复杂的结构，蚂蚁之间如何相互通信，蚁群作为整体如何适应环境变化？

# Logistic map

$$x_{n+1} = r x_n (1 - x_n),$$




图片来自Wikipedia： https://en.wikipedia.org/wiki/Logistic_map

简单的确定性方程能产生类似于随机噪声的确定性轨道，这是一个非常深刻的负面结论，它与量子力学一起，摧毁了19世纪以来的乐观心态：认为牛顿式宇宙就像钟表一样沿着可预测的路径运行。

# 复杂系统研究方法


## 基于数学化的方法

又称为复杂性科学，采用高度数学化的方法，但通常只能在部分特征上有效，没有统一的数学理论。但基于复杂系统的视角为传统的数学、物理、化学等研究方法提供了很多新的结论。



## 基于工程设计的方法

抛开复杂系统的数学理论，但是借助复杂系统的一些思想来解决大型人工系统的设计和过程管理。赫伯特-西蒙认为所有人工参与或制造的系统都适合用这种架构设计视角。

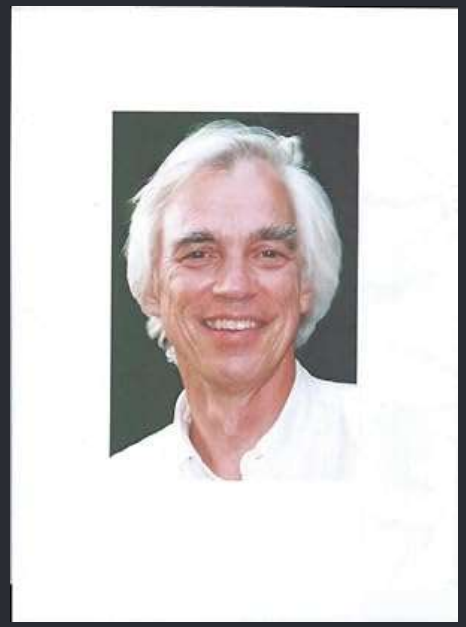
比如，子系统的划分就使内部环境与外部环境无关，这称为体内平衡，这样内部系统和目标之间就保持着不变的关系，而不受外部特征的大多数参数在很大范围内变化的影响。


## 基于进化计算的方法

聚焦于复杂事物是如何形成的，而不是度量其复杂性，将计算的观点引入进化论，认为系统本身携带着能够帮助复制和进化的附加信息，将复制和变异看作一种计算，然后就可以用一个计算的观点来描述整个进化行为，称之为复杂引擎。

# The Engine of Complexity

## Evolution as Computation



图片来自 Amazon.com

**John E. Mayfield**

From an early age John E Mayfield was fascinated with science of all kinds. This broad interest led to a BA in physics, a PhD in biophysics, and an academic career in the area of molecular biology. In the late 1990s he was introduced to evolutionary computer algorithms and became fascinated by the relationships between biological evolution and computer based evolution and more generally in the linkages between computation and biological process. The Engine of Complexity, Evolution as Computation is based on his consequent studies. The book shows how biological evolution is a special case of a more general computational notion of evolution, and how that general view of evolution explains not only how life is possible but also how human technology and most or all complex outcomes of human society are possible. It is his first book.

# 进化系统

进化的本质是从随机过程中累积信息

## 进化系统

所有进化过程的都包含5个要素：

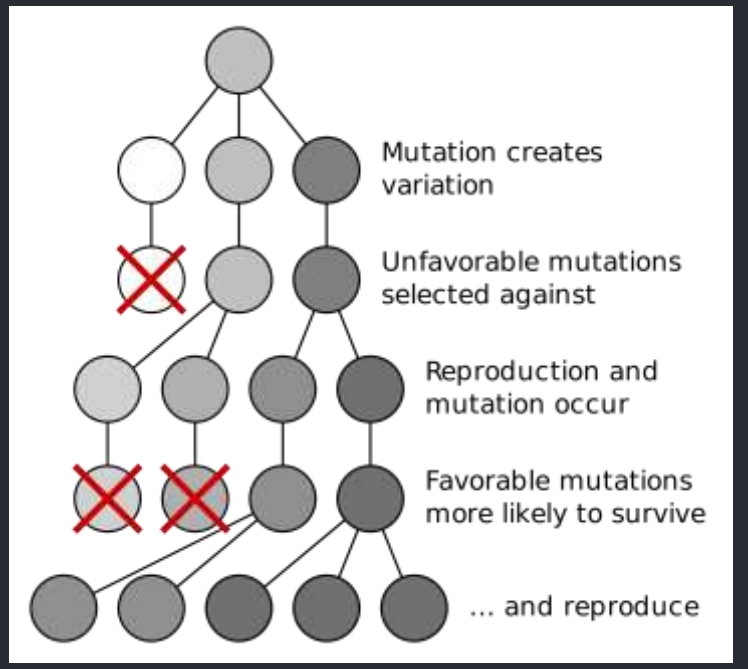
**1）个体：**
一般都是有许多某种类型的单元体。

**2）可遗传的特征：**
个体的描述信息，以某种形式编码为个体本身的一部分（如DNA）。

**3）个体可以繁殖或复制：**
个体可以从父辈或之前的个体拷贝信息

**4）变化机制：**
信息在复制、繁殖或维护过程中必须有机会产生适度的改变。

**5）基于特征的选择：**
繁殖的成功必须部分取决于各个个体编码信息所描述的特征。



# 复杂引擎

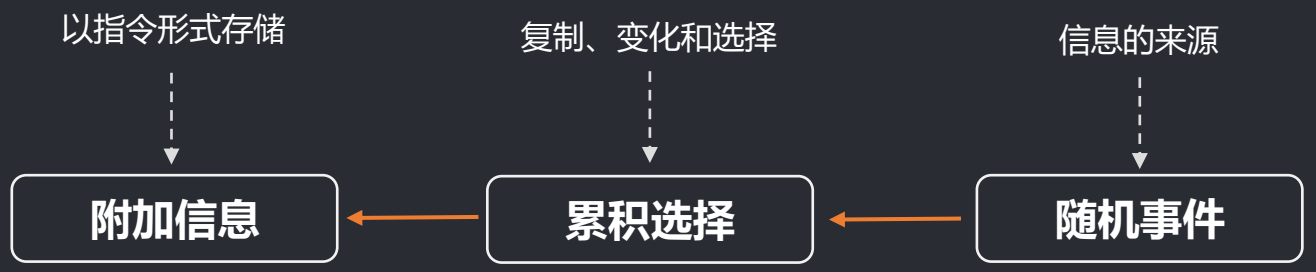
用计算来定义进化，进化本质上是累积信息

## 计算的观点

**数据/信息是什么?**
对于生物，信息是用DNA分子编码的；对于大脑，信息则编码为神经元的通信模式。

**程序/指令来自哪里?**
从随机的变化过程中提取的目的性信息，即进化计算。

**指令的目的性来自哪里?**
所有指令都是有目的性的，选择导致目的性的产生



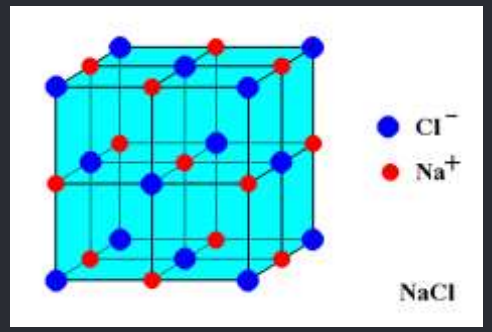

随机事件和随机选择是所有目的性信息的最终来源
进化的本质是累积信息
信息是包含事物形成的规则，或指令
指令含有目的性信息

# 程序的结构

如果物理过程是计算，程序在哪里？

## 免费的结构

在特定的条件下物理定律描述的力相互作用，一些物理定律描述物质聚拢的趋势，另一些则描述原理的趋势，引力和斥力之间会形成平衡点，产生结构，称为免费的结构。



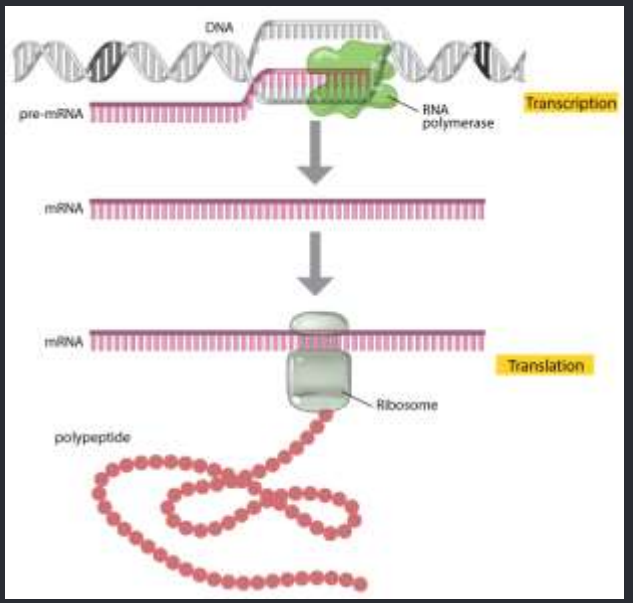

当等量的钠原子和氯原子混合到一起，就会发生自发反应，使得系统转变成新的平衡态，库仑定律使得带正电的钠离子和带负电的氯离子相互吸引，和泡利不相容原理和量子力学一起形成新的盐晶结构。

## 目的性的结构

人体本身也是指令的产物，这些指令以DNA的化学结构存储在我们身体的每个细胞中。在蛋白质的合成过程中，存储在DNA中的信息在细胞中扮演的角色同人类钟表匠执行的指令一样，用DNA写出的指令引导细胞的结构精准组装特定的蛋白质。

合成蛋白质的第一步是复制DNA单链上特定部位合成RNA，RNA的化学结构类似DNA，但使用的核苷酸有些不一样。

从这个意义上说，细胞就是活的计算机，细胞中几乎所有的结构最终都依赖于用DNA语言编码的指令，DNA决定蛋白质和RNA，它们的相互作用以及它们的形成。




图片来自nature.com

# 这对传统程序意味着什么？

全新的视角：将程序看作一个DNA

以指令形式存储

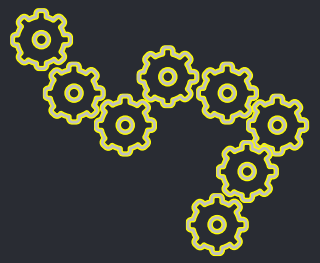
复制、变化和选择

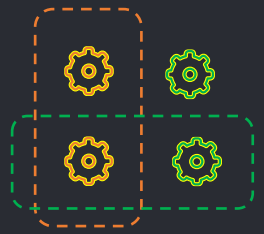
信息的来源

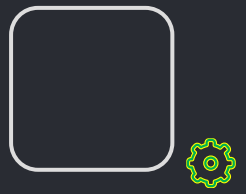
**附加信息**

**累积选择**

**随机事件**







### 程序定义为组合体

程序定义为一个组合体，而不是一个固定结构和功能的单一程序。

### 个性化

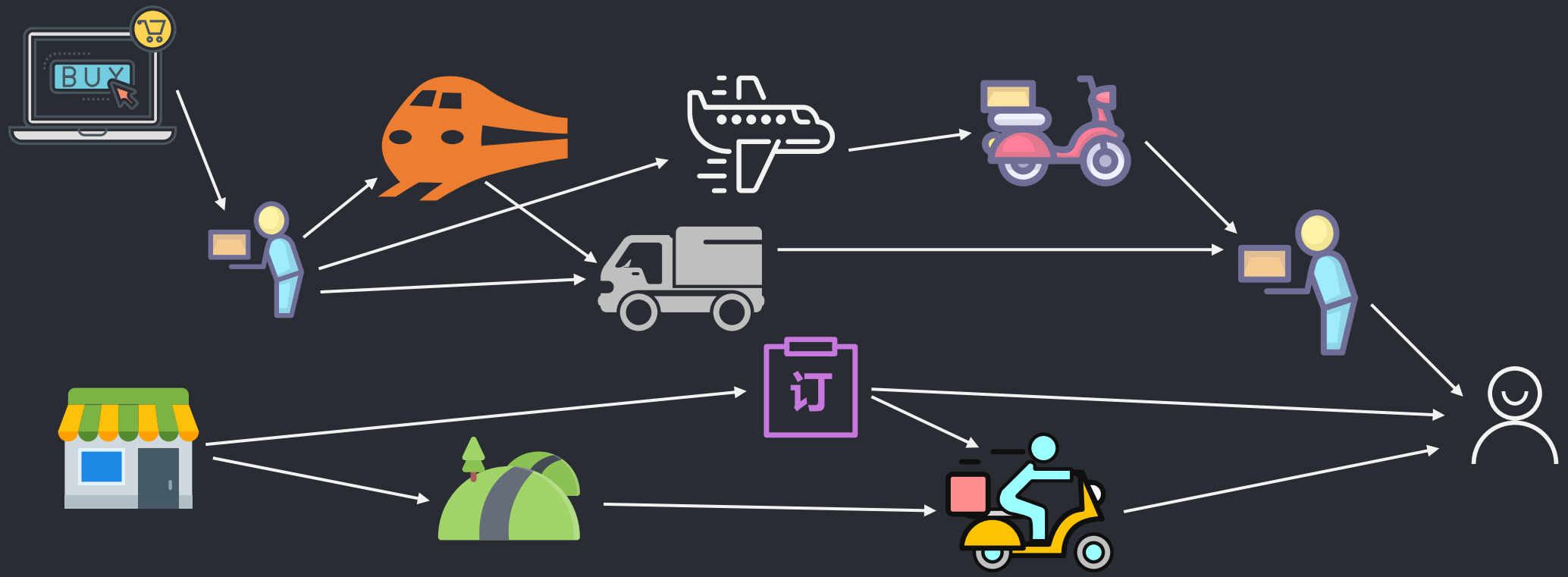
用户可以个性化的选择自己的组合，形成选择，那些优秀的程序将会被保留下来

### 互操作

你不是去修改已有程序，你通过互操作添加新的行为和特性

# 自主性

道路修好了自然有车行走




没有中央控制
整体功能远超单一功能预期：涌现机制
子系统可以自我更新

# 这种结构对软件构造提出了全新的挑战

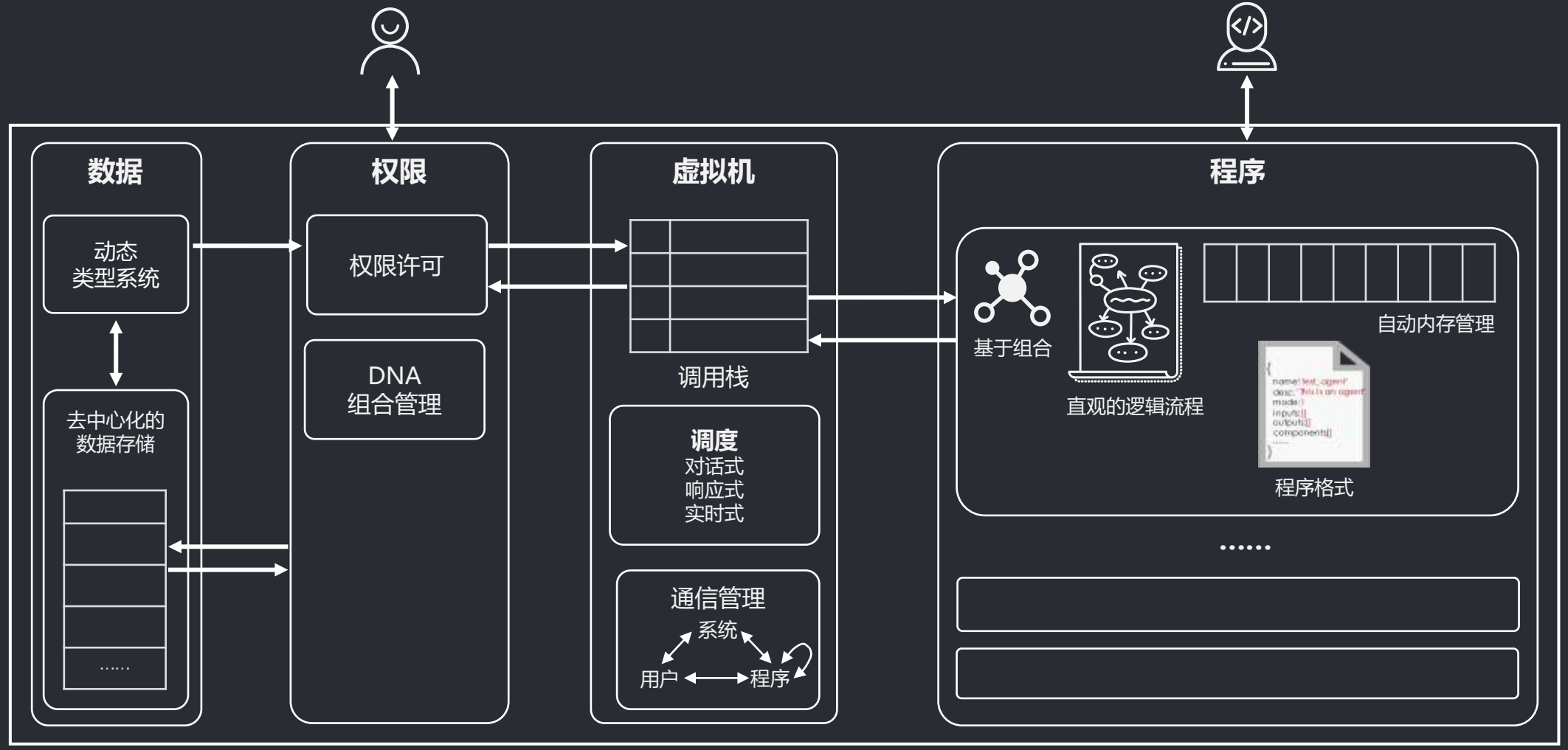
甚至触及传统软件构造方法的核心基础：基于内存隔离的机制

# Cogine

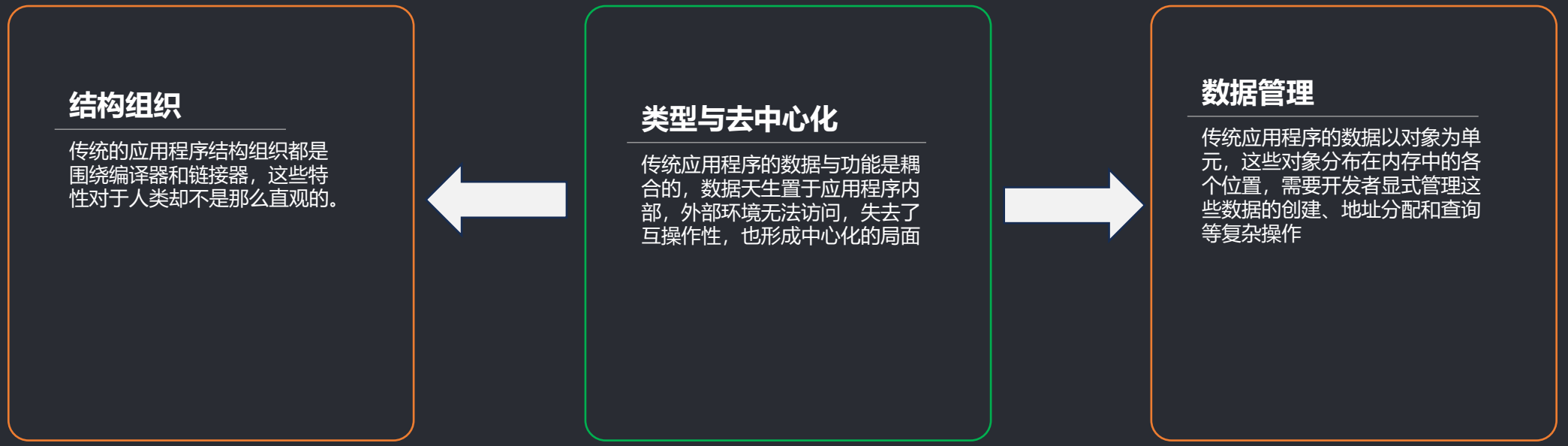
A computing engine of complexity

# 内在需求

# Cogine系统架构


数据
动态
类型系统
去中心化的
数据存储
……
权限
权限许可
DNA
组合管理
虚拟机
调用栈
调度
对话式
响应式
实时式
通信管理
系统
用户
程序
程序
基于组合
直观的逻辑流程
自动内存管理
程序格式
……

# 内在需求


结构组织
传统的应用程序结构组织都是围绕编译器和链接器，这些特性对于人类却不是那么直观的。
类型与去中心化
传统应用程序的数据与功能是耦合的，数据天生置于应用程序内部，外部环境无法访问，失去了互操作性，也形成中心化的局面
数据管理
传统应用程序的数据以对象为单元，这些对象分布在内存中的各个位置，需要开发者显式管理这些数据的创建、地址分配和查询等复杂操作

# 类型与去中心化

# 数据类型在传统程序中的意义

数据类型本身在传统程序中并没有被当作语义使用

## 用于存储

用于告诉编译器怎样为数据对象分配内存空间和布局；


```rust
pub struct Point {
    pub x: f32,
    pub y: f32,
    pub z: f32,
}
```

## 用于寻址

根据上面的内存布局，用于告诉编译器怎么为每个变量的符号计算内存地址

```rust
let p = Point::default();
let z = &p.z;
```


## 不是数据抽象

数据抽象是指对数据表述细节的隐藏，通过过程来定义，而数据类型则是定义数据表述

## 不是对象的意义

对象的意义更多是通过继承或则组合机制实现，单个数据定义通常不能称为对象


## 也不是对象的语义

在传统程序结构种，通常通过对象的实例来表征一个对象的意义，类型只是一种定义，不具有程序语境下的语义信息

# 数据类型的编译与耦合

动态解释+互操作协议机制

```rust
#[derive(Default)]
pub struct Point {
    pub x: f32,
    pub y: f32,
    pub z: f32,
}

fn f() {
    let p = Point::default();
    let z = &p.z;
    print!("point.z is :{}", z);
}
```

编译器

```
{
    "name": "Point",
    "fields": [
        {"name":x, "type": 3},
        {"name":y, "type": 3},
        {"name":z, "type": 3}
    ]
}
```

运行时解释器

```rust
fn f() {
    let p = Point::default();
    let z = &p.z;
    print!("point.z is :{}", z);
}
```

交换格式+
解析运行时

# 实现

## 1. 文本类型定义

可以使用Json或任意文本格式定义任意类型，
运行时要确保类型的定义与使用格式一致

```
{
    "name": "Message"
    "fields": [
        {"name": "receiver", "type": 5},
        {"name": "content",  "type": 5},
        {"name": "sender",   "type": 5},
        {"name": "done", "type": 4}
    ],
}
```

## 2. 动态类型解释

运行时根据类型定义进行正确动态解释并获取数据

```
#[derive(Clone)]
pub struct StandardDef {
    pub name: String,
    pub type_id: u64,
    pub fields: HashMap<String, StandardField>,
    pub description: String,
}
```

## 3. 虚拟机寻址计算

虚拟机根据相关定义将数据填充到程序对象上

```
input = nil
input["email"] = { name = "Elvis", content = "Hello" }

output = { message = nil }
```

## 4. 程序类型定义

Lua程序在构建的时候根据选择类型动态创建和引用程序内部数据类型表述，
在运行时通过虚拟机的调用栈将正确的数据解析成对应的程序内类型

```
function updating()
    output["message"]["content"] = string.format("I have sent a message to %s with the following content: %s",
        input["email"]["name"],
        input["email"]["content"])
    output["message"]["receiver"] = "user"
    output["message"]["done"] = true
end
```

# 带来的问题

这 种 新 的 结 构 调 整 怎 样 跟 现 有 体 系 兼 容 ?

### 去中心化的数据存储

数据可以与程序功能代码进行隔离，实现去中心化。

### 类型安全

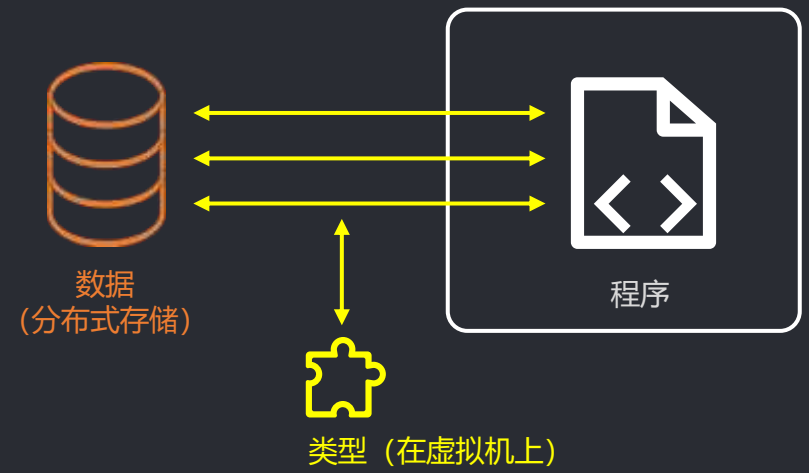
动态语言（如Lua）的对象不含指针，动态语言本身总是能够保证类型安全；Cogine的虚拟机保证程序获得的数据都是经过拷贝到调用栈的。

### 数据安全

由于程序之间具有传递任意信息的能力，因此需要运行时严格的类型检查；数据不能由程序直接发送给另一个程序，数据只能来源于用户合法的数据

### 基于类型的权限管理

显式声明对数据类型的使用，用户就可以基于类型对数据进行管理；即使程序定义了对数据的访问，但是根本就不会执行；就像区块链程序可以随意访问用户账本数据，但是只有获得授权的程序才能执行真正的操作。




```
{
    "std_name": "Message",
    "agents": [
        "hello_cogine",
        "sending_email"
    ]
}
```

# Cogine设计哲学一：互操作性

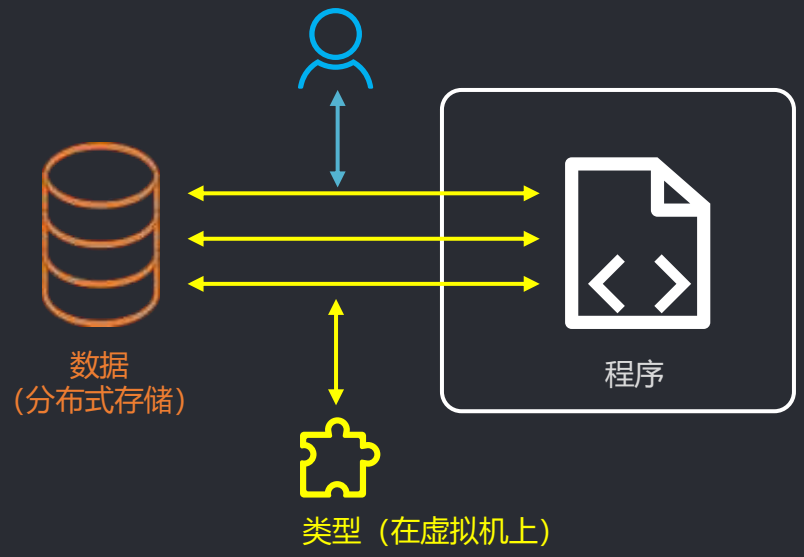
从区块链得到启示，你必须完全放开数据才能促进和实现互操作性；通过对数据的访问授权可以弥补数据安全。

### 基于本地存储的去中心化

大部分用户数据都是自有数据，与其它账户没有共识的需求，所以数据不必存储在分布式账本上，减少计算量的同时，也保护用户数据隐私。

### 基于类型和语义的数据保护

由于数据没有在分布式环境，只需要通过对应用进行授权即可以控制权限，基于类型的权限控制使用户对数据有更好的理解和保护。




这种新的架构从根本上改变了传统软件构造方法数据与功能耦合的局面

# 所有数据都以这种方式存储吗？

通过内建的机制分离多种功能性数据，更方便开发者和用户管理与理解

## 用户数据

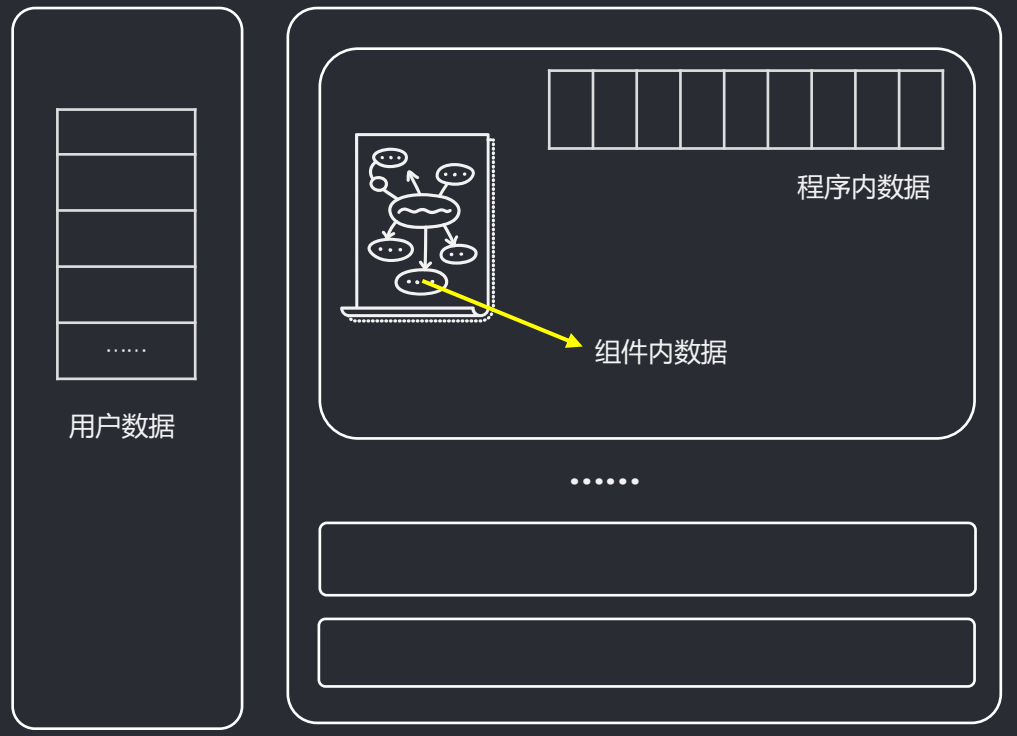
只有需要互操作的数据才设置为用户数据，这些数据通常对用户来说是有意义且易于理解的，用户需要对这些数据进行授权管理。有点类似手机的隐私设置。

## 程序内数据

和用户数据一样的方式工作，但是对用户不可见，主要用于程序内组件之间的互操作。这部分数据是方便开发者管理和理解逻辑的，因此通常是跟业务有关的数据。

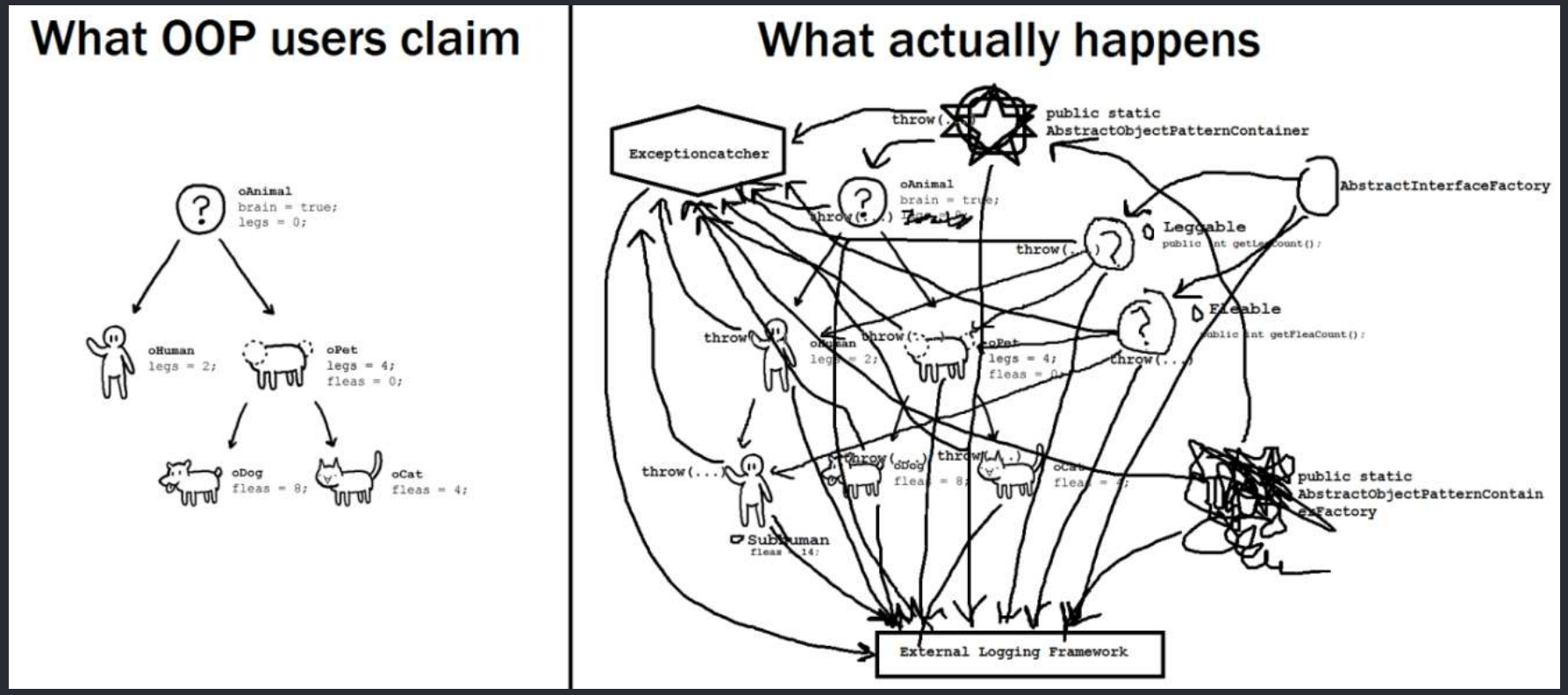
## 组件内数据

是一个组件对象内部的状态数据，只对该组件可见，比如某个组件负责维护一个状态树，该状态树的具体结构和计算对其它组件没有意义，只将状态结果作为程序内数据传递给其它组件即可。



# 数据管理

# Object-oriented programming




What we learn in school vs what happens in industry

# OOP带来的复杂性

## 继承增加语义复杂性

尽管以对象为单元进行模块化的程序组织，看起来能够更好地模拟真实世界；但是主流OOP实现的继承机制却大大增加了对这些概念语义管理的复杂性，真实世界模型并没有那么复杂的继承机制。

## 以对象为单元的上下文管理

在我们日常的事务中，语义会优先于对象实例，我们总是思考对某种类型的事情执行操作，而实际执行的时候具体的实例总是很自然的事情。

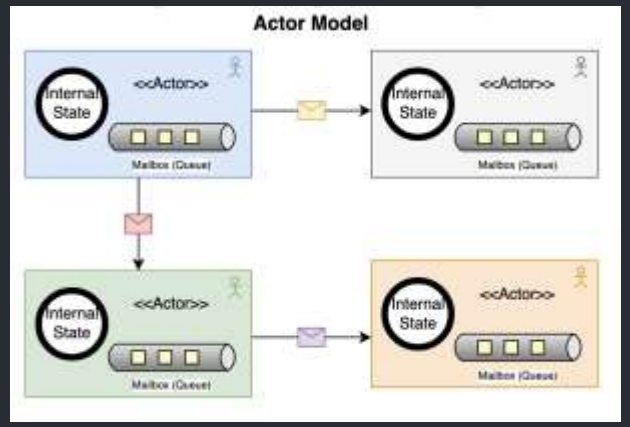
而计算机程序中我们总是考虑实例，实例怎么创建，存储在那里，怎么获取（引用），怎么修改等等，这些概念大部分都是为编译器和硬件规则而涉及的概念。因为程序必须关注数据在内存中的地址。

## 缺乏隔离性

尽管对象封装的目的是将对象的内部细节隐藏起来，形成以对象为单元的抽象，但这种模块化并没有考虑对象之间的交互。为了使对象之间可以进行交互，OOP粗暴地将对象的状态暴露给任何其它对象进行访问，这种特性解决了对象之间交互的需求，但是缺乏隔离性使得程序的组织更加复杂。

# OOP的本质是隔离（isolation）

隔离才能带来真正的对象抽象，避免耦合，降低软件复杂性

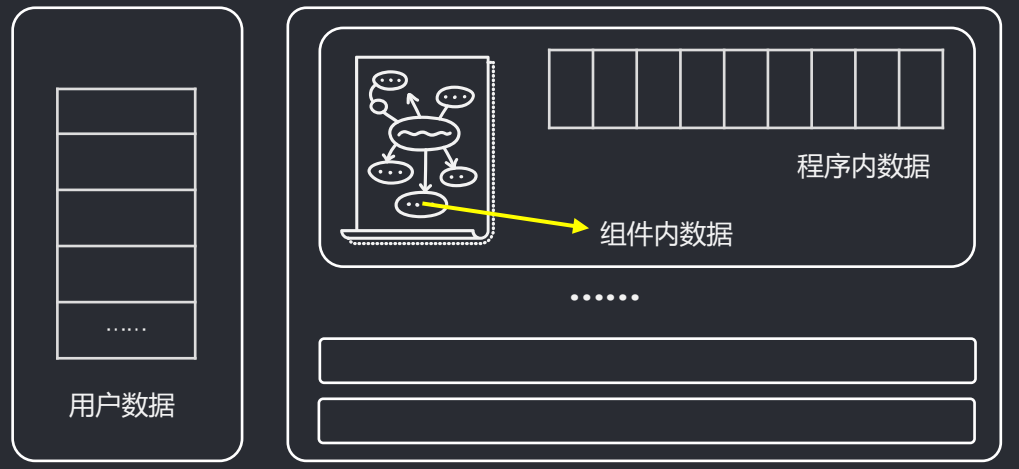



## Erlang actor model

Alan Kay：“面向对象编程的概念完全被误解了，它不是关于对象和类，它完全是关于消息传递”，他认为人们过于强调类和方法，而没有强调消息的重要性，Smalltalk中总是讨论对象，以及总是讨论向对象传递消息和等待返回消息后进行响应。

Erlang具有所有特性：隔离, 多态性和消息通信，可能Erlang是唯一真正面向对象的编程语言。

Ralph Johnson, Joe Armstrong on the State of OOP




## 与Erlang区别

1）线程以Agent而不是一个函数为单位，Agent内部串行执行
2）Agent之间的通信采用强类型数据对象

# 数据管理的复杂性: Context

## Context in programming

A programming context can be defined as all the relevant information that a developer needs to complete a task. Context comprises information from different sources and programmers interpret the same information differently based on their programming goal.

## 分散在各个部分的上下文

这些对象分布在不同的模块、以不同的形式，不同的存储空间，程序之间的关系错综复杂，运行时状态难以追踪。

同一个需求，100个程序员可能写出100种不同的数据组织

```rust
fn add_node(&mut self, node_id: NodeId, editor_state: &mut EditorState) {
    let node = self.state.graph.nodes.get_mut(node_id).unwrap();
    //change index
    let mut index = 2;
    while let Some(s) = self.user_state.agent_def.components.get(&index) {
        index += 1;
    }

    // insert the new component into agent_def
    node.user_data.index = index;
    if let NodeData::Component(c) = &mut node.user_data.data {
        self.user_state.agent_def.components.insert(
            index,
            AgentComponent {
                name: c.name.to_string(),
                outgoing: Vec::new(),
                incoming: Vec::new(),
            },
        );

        let pos = self.state.node_positions.get(node_id).unwrap();
        self.user_state
            .agent_def
            ._node_position_in_ui
            .insert(index, (pos.x, pos.y));

        let def = editor_state.components.get(&c.name).unwrap();
        *c = def.clone()
    }

    self.check_standards(editor_state);
}
```

# 组合优于继承

组合必然是面向数据的设计，它没有对象的概念，必须查表

## Data-oriented design

所谓组合，即没有一个整体“对象”的概念，所以想要对一个“对象”的数据进行操作，必然是通过查表，所以数据的存储大都是基于表的，这是DOD的核心。

| | A | B | C | D |
|---|---|---|---|---|
| entity1 | | | | |
| entity2 | | | | |
| entity3 | | | | |

archtype1

| | E | F | G |
|---|---|---|---|
| entity4 | | | |
| entity5 | | | |

archtype1


## 从继承到组合，要解决什么问题：self

没有整体“对象”的概念，就没有self的概念，因此DOD的主要逻辑就是设计对表格的数据查找来替代self的功能，使得可以以对象的方式对数据进行操作。

A
B
C
D entity1

```
let a = self.A;
```

## 动态的“对象”

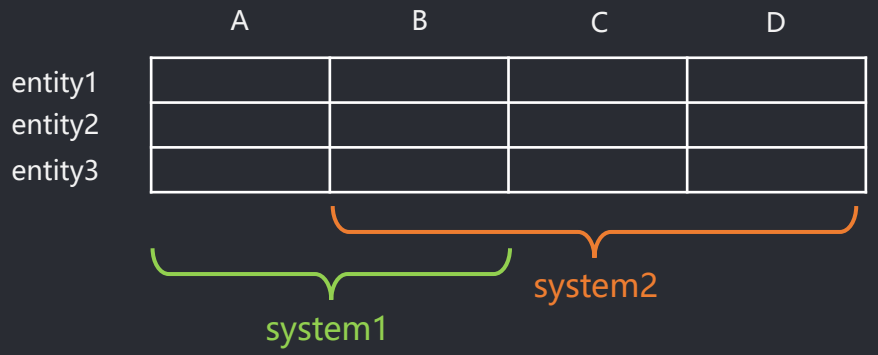
由于对象不再是编译为静态的地址，而是可以动态拼装，所以对象是“动态”的，这种动态性代替固定的继承结构，使得逻辑表达可以更灵活，清晰。

# 组合的精髓是交互

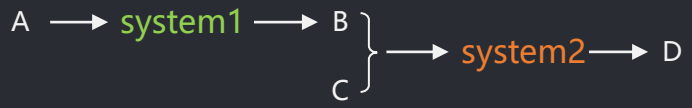
组合的精髓不是组合，而是对象之间的交互

## 提供对象间的交互

OOP仅考虑以对象为单元进行模块化，但是缺乏对象间交互的友好机制，因此直接提供对象直接访问机制，因此带来耦合以及交互的复杂性。




通过弱化对象的概念，以及提供动态“对象”的能力，间接提供了非常强大的交互能力，且避免了耦合。

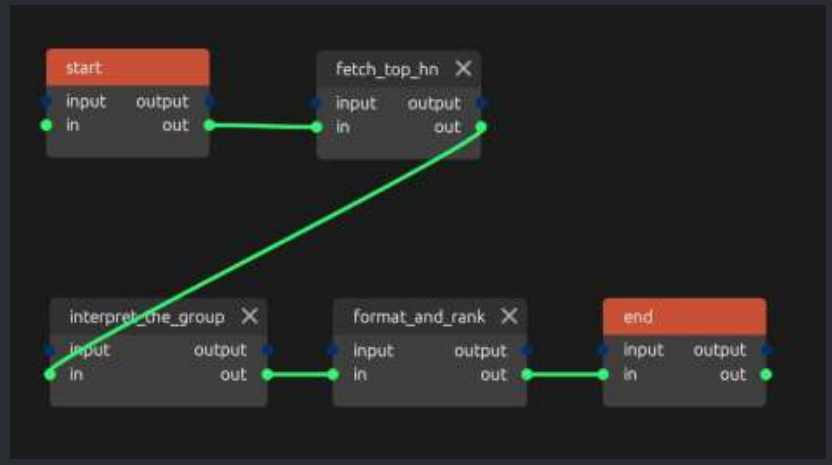



## 显式声明逻辑顺序

由于不是直接将函数调用顺序硬编码到程序代码中，进而编译为固定的地址，因此需要显式声明逻辑顺序。即system1必须在system2之前执行；但是这种显式声明也带来了动态性。

system1 → system2

这种显式声明也往往使得函数更具有语义属性，而不是单纯是一段代码的名字，因为你需要基于这种语义属性去维护顺序。

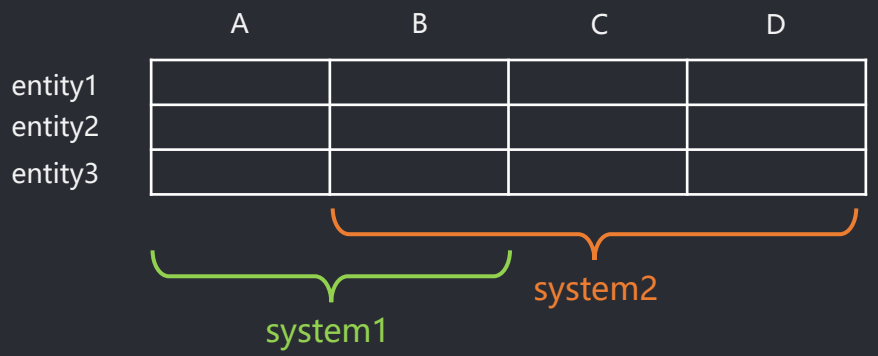


# 复杂的表管理

组合的精髓不是组合，而是对象之间的交互

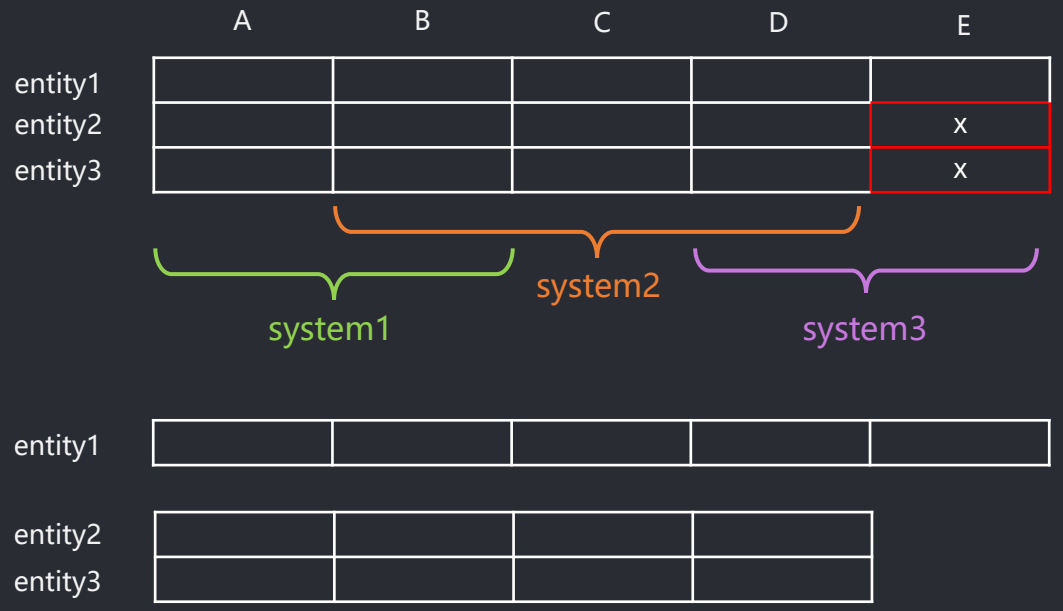
## 内存布局与交互的矛盾

为了使两个system之间可以进行交互，它们所使用的数据必须在一张内存表中；如果所有system之间都是两两相互交互，则所有数据只能存在一张大表中，失去了性能优势。




## 多组碎片的表

由于不是直接将函数调用顺序硬编码到程序代码中，进而编译为固定的地址，因此需要显式声明逻辑顺序。即system1必须在system2之前执行；但是这种显式声明也带来了动态性。



# 基于语义抽象的上下文管理

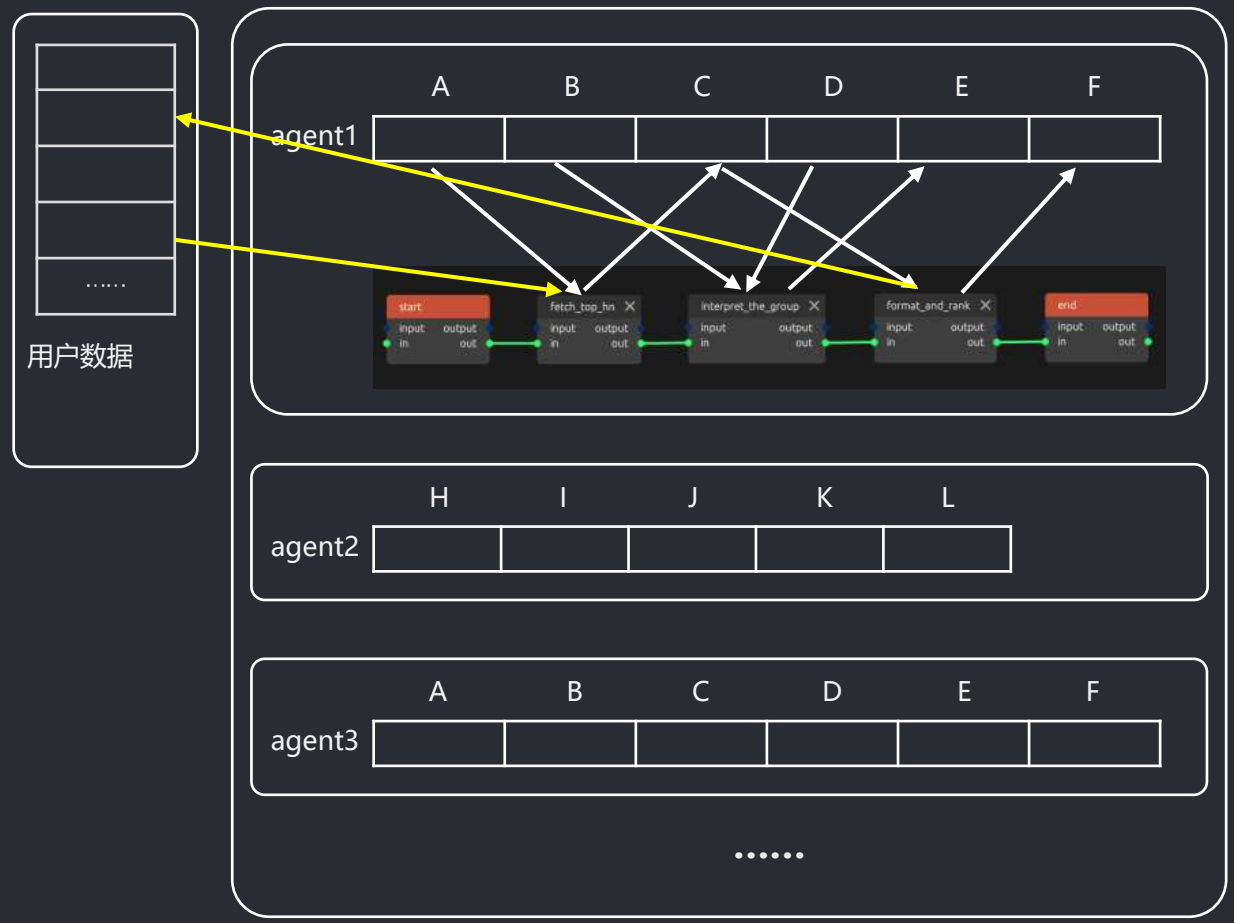
## 大多数应用都不具备多实例对象

1）游戏中有大量相似的对象，但传统应用程序中很少有大量实例对象。
2）实例对象的前提是system的语义化，在多应用共存的环境下，跨应用之间即使是相同名字的system也很难在语义上一致。

## 语义抽象

将所有的数据对象都语义化，结合类型系统，通过类型而不是实例地址对数据进行访问。

通过语义抽象：
1）提供了跨对象交互的能力，简化了逻辑流程表达
2）避免对对象地址进行管理，使用户仅针对语义进行上下文数据管理



# 数据访问抽象

基于类型构建一层数据抽象，使得内存的细节被隐藏

## 基于类型的数据访问抽象

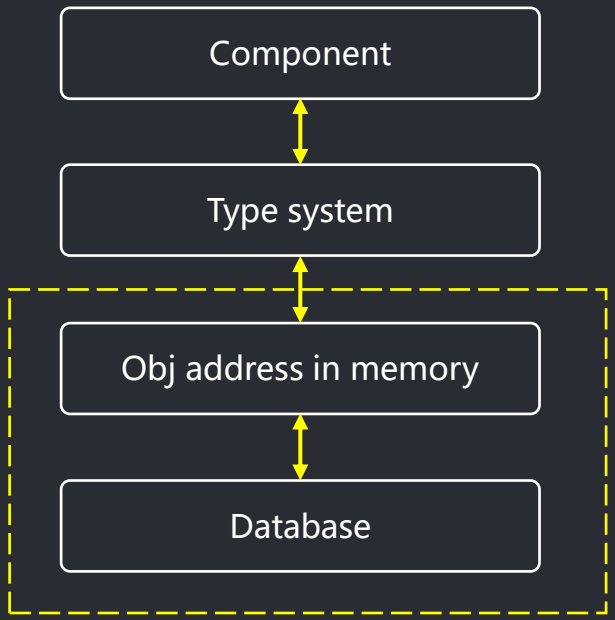
传统编程语言提供的数据访问抽象只是针对数据的表述的隐藏，开发者仍然需要关心对象实例数据在内存中的地址；通过建立以类型为基础的数据操作，开发者只需要关心数据的语义即可，类型成为语义的名字。

## 基于类型的数据操作

以类型为基础，不光是可以以类型名字作为数据，查询对应的单例实例对象，还可以通过类型名称或字段的元数据进行增、删、改的操作。

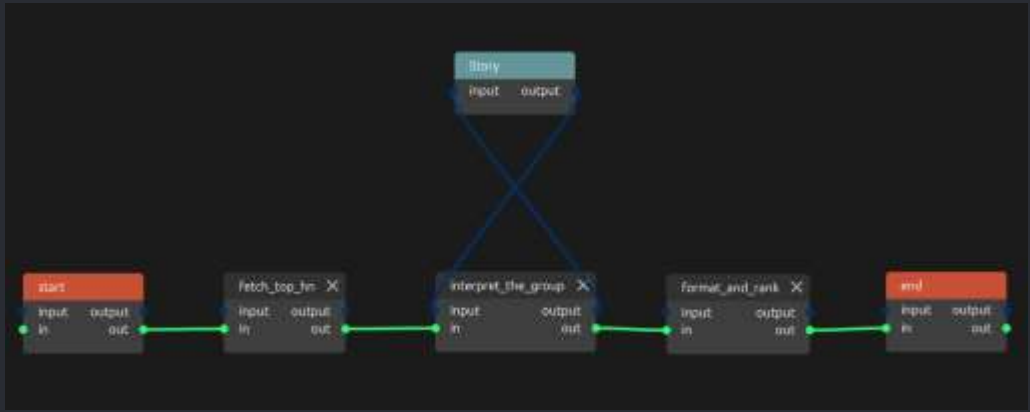
```
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 5.6430448
    }
  }
}
```

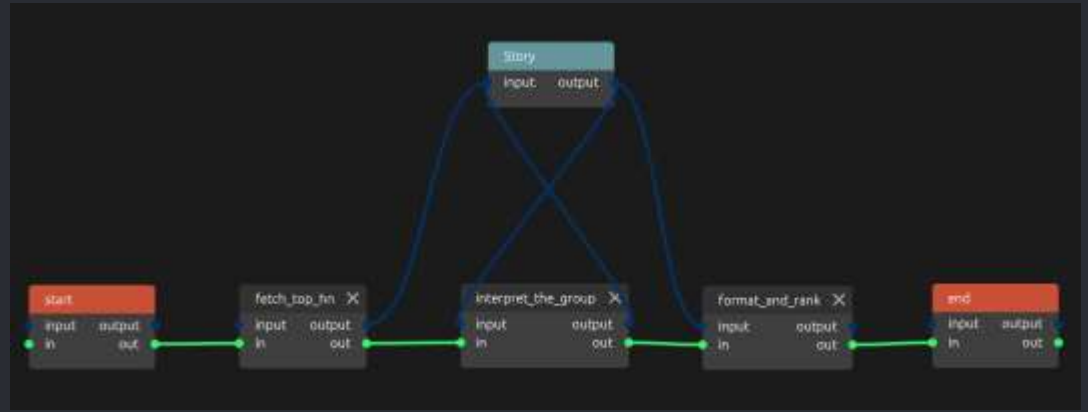


# Cogine设计哲学二：语义抽象

构建一种程序组织方法，使得程序员完全不需要管理数据的内存关系




可以查看任意组件的输入输出，因此也可以方便跟踪和调试




可以查看任意一个数据在全局范围内被哪些组件使用

## 无需关心数据从哪里来

Lua程序在构建的时候根据选择类型动态创建和引用程序内部数据类型表述，在运行时通过虚拟机的调用栈将正确的数据解析成对应的程序内类型

```
function updating()
    output["message"]["content"] = string.format("I have sent a message to %s with the following content: %s",
        input["email"]["name"],
        input["email"]["content"])
    output["message"]["receiver"] = "user"
    output["message"]["done"] = true
end
```

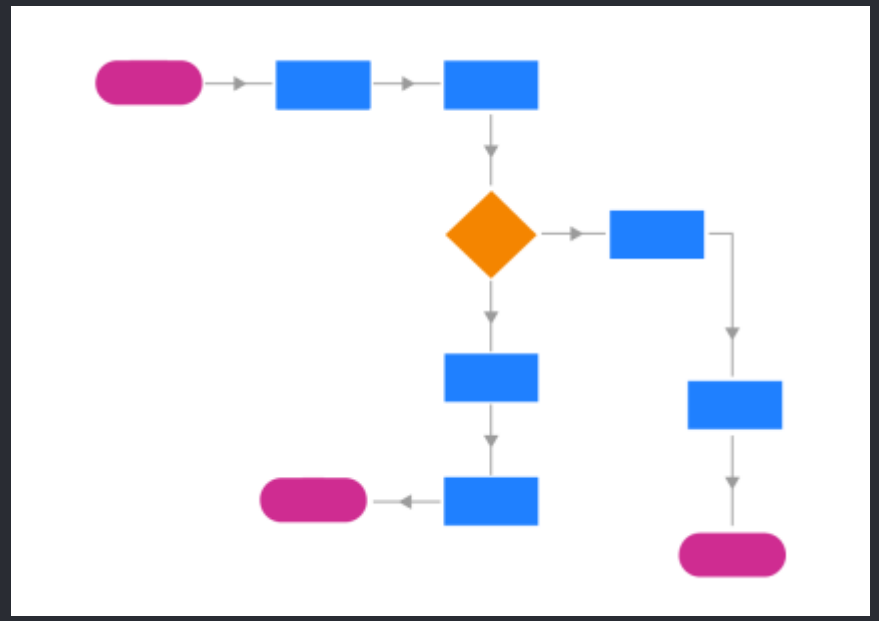
# 结构组织

# 传统软件的流程在哪里？

我们大脑中思考的流程和程序中的流程是完全不一样的

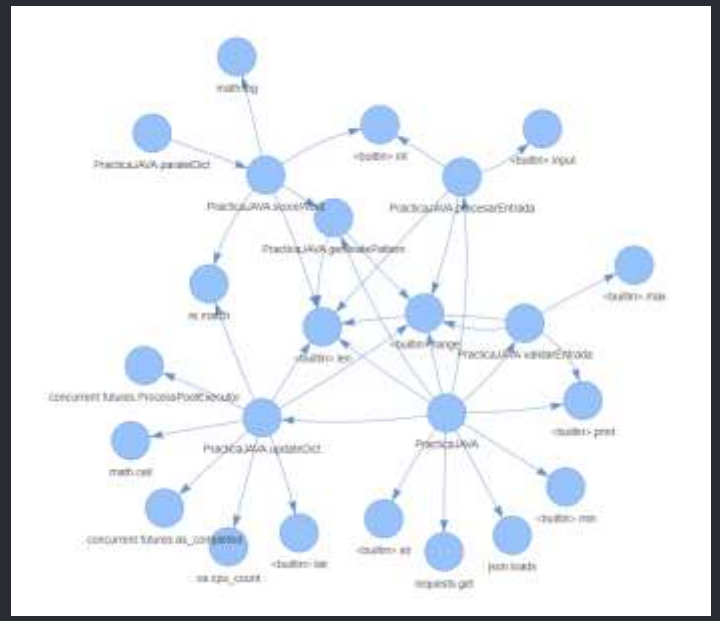
## 大脑中的流程

1）可交流的，两个人之间可以就流程进行交流和理解
2）线性有顺序的，只要保证逻辑顺序，实际顺序通常是可以调整的
3）聚焦流程的意义，而不是细节（例如每个节点怎样获取数据）



## 程序中的流程

1）不可交流，流程隐藏在代码中，你必须看代码才能理清调用流程
2）连线不代表逻辑流程，表示的是函数调用，也无法反推函数调用顺序
3）流程主要靠硬编码的函数调用来决定，函数的语义显得不重要

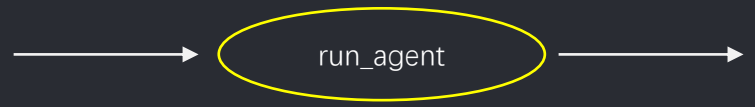


# 为什么

传统软件构造方法没有显式表达流程的机制

```rust
pub fn run_agent(&mut self, agent: Agent, message: &Option<Message>) -> Option<Message> {
    let agent_info = self.agents.agents.get(&agent).unwrap();
    let agent_def = agent_info.agent_def.clone();

    // start component infomation
    let start_index = 0; //computing_agent.start_component;
    let start_component = agent_info.agent_def.components.get(&start_index).unwrap();

    // this runtime history should be cleared before every execution of a component
    // so we can always run begin from start even some component has been paused
    self.computing_agents
        .entry(agent)
        .and_modify(|f| f.init_contitions(&agent_def, message));

    self.run_component(
        agent,
        start_index,
        &agent_def,
        &start_component.name.to_string(),
    )
}
```

1）只有输入和输出才能表示执行流程，且是明确唯一的




2）内部的函数无法单从连线看出逻辑顺序

# 怎样流程化?

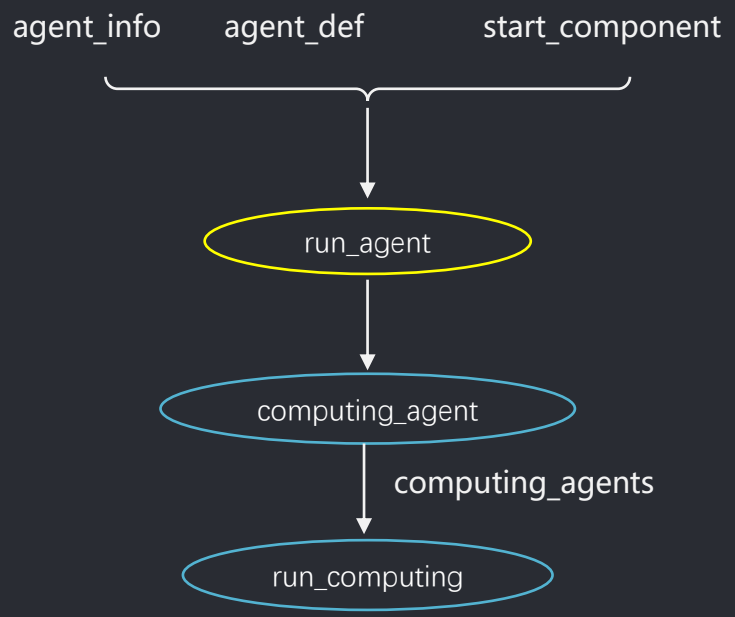
**将所有需要的数据表达为输入参数**

将有意义，且会被反复调用的函数计算过程缓存为数据，即计算数据化

**让函数没有其它流程调用**

每个函数都退化为纯函数，它没有内部流程调用，本身是一个独立流程，它的计算结果均要表征为数据

```rust
pub fn run_agent(&mut self, agent: Agent, message: &Option<Message>) -> Option<Message> {
    let agent_info = self.agents.agents.get(&agent).unwrap();
    let agent_def = agent_info.agent_def.clone();

    // start component infomation
    let start_index = 0; //computing_agent.start_component;
    let start_component = agent_info.agent_def.components.get(&start_index).unwrap();

    // this runtime history should be cleared before every execution of a component
    // so we can always run begin from start even some component has been paused
    self.computing_agents
        .entry(agent)
        .and_modify(|f| f.init_contitions(&agent_def, message));

    self.run_component(
        agent,
        start_index,
        &agent_def,
        &start_component.name.to_string(),
    )
}
```



# 从主动到被动

从编译器手中接管程序控制的流程

## Calling Convention

编译器使用的函数调用约定规则，将整个程序的流程串联起来，并借助链接过程将其地址硬编码。这虽然减少了程序员维护程序流程的工作，但是带来了流程管理的复杂性。

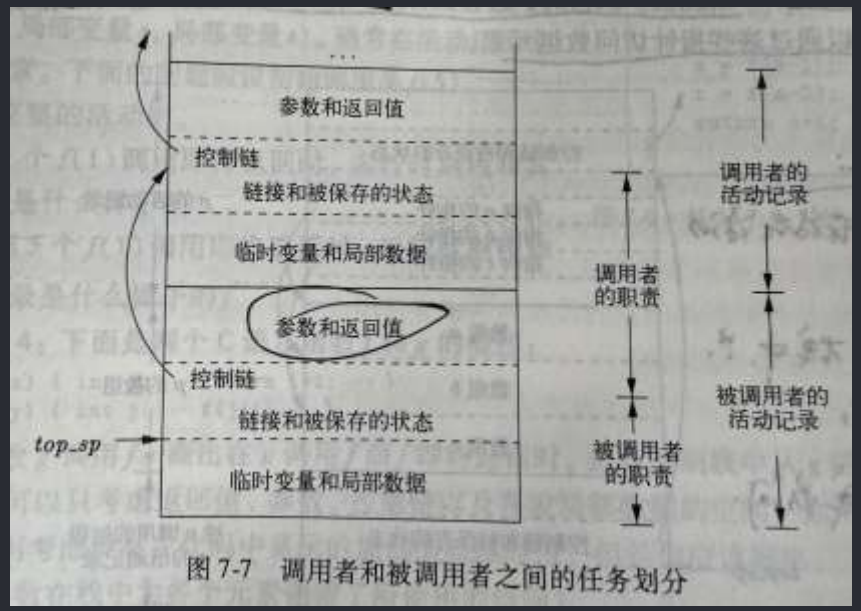
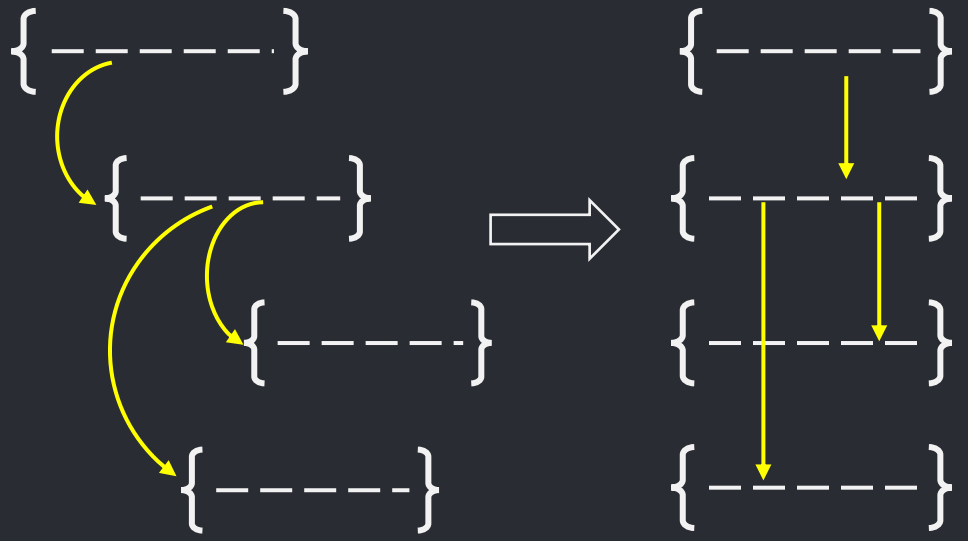


图 7-7　调用者和被调用者之间的任务划分

## 被动式调用

没有了编译器的函数调用及链接过程，就需要虚拟机手动执行显式的函数调用，此时也将函数的主动式调用转变为被动式调用，即函数不能自己发起执行和调用，而是按照定义的流程有虚拟机运行时来调度。



# 程序的编译与链接

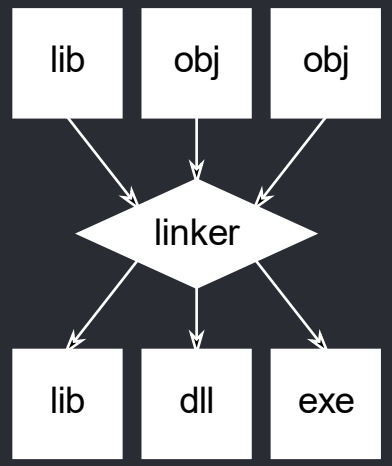
程序链接的过程本质上就是寻址计算

```
$ wasm2wat test.wasm
```

```
(module
  (import "js" "memory" (memory 1))
  (import "js" "table" (table 1 funcref))
  (type $return_i32 (func (result i32)))
  (table 2 funcref)
  (global $g (import "js" "global") (mut i32))
  (data (i32.const 0) "Hi")

  (func (export "callByIndex") (param $i i32) (result i32)
    local.get $i
    call_indirect (type $return_i32)
  )

  (func $add (param $lhs i32) (param $rhs i32) (result i32)
    local.get $lhs
    local.get $rhs
    i32.add
  )
  (export "add" (func $add))
)
```

## Linker & linking

大型程序中充满着复杂的相互引用，这些最终都需要转换为内存中的实际地址，这称为链接的过程。编译器首先将每个源文件编译为单独的目标文件或者模块，并附带各种类型符号信息，然后链接器基于这些符号信息将所有目标文件或者模块链接为一个程序整体。




# 从隐式/静态到显式/动态

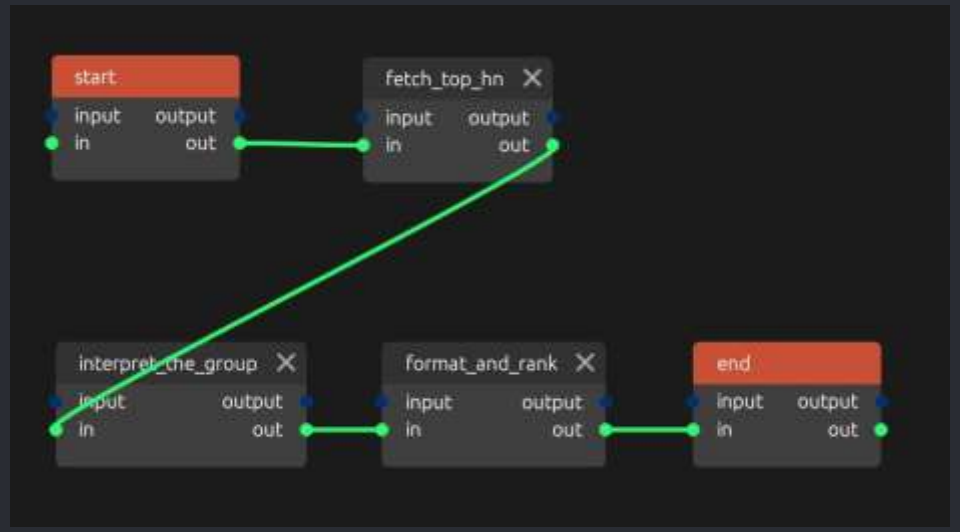
模拟程序的链接过程，将静态的寻址计算调整为运行时链接

## Agent 格式

```
{
    "magic": 1000,
    "version": "1.0.0",
    "model": 1,
    "name": "top_hacker_news",
    "desc": "an example of an agent",
    "display_name": "Top Hacker News",
    "standards": [
        "Story"
    ],
    "inputs": [],
    "outputs": [],
    "components": {
        "0": "start",
        "1": "end",
        "2": "fetch_top_hn",
        "3": "interpret_the_group",
        "4": "format_and_rank"
    },
    "edges": [4,1,0,2,2,3,3,4]
}
```

## 编辑而不是编译流程

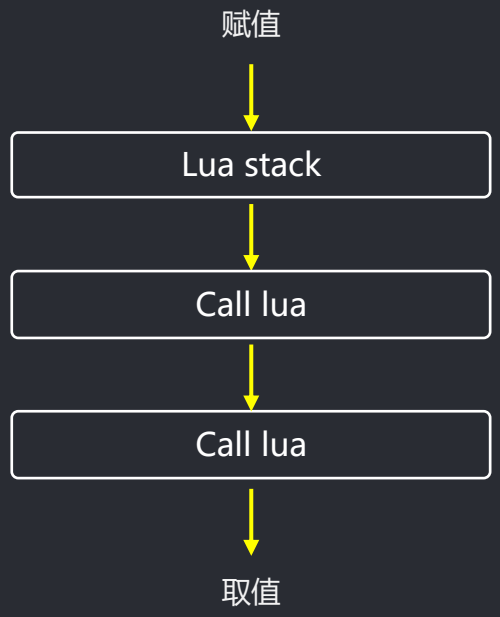
传统的逻辑流程是通过程序员用代码表达为函数调用，然后编译器和链接器使用函数调用管理将源代码转换为静态的可执行程序文件；通过模拟这种机制，将编译和链接过程需要的一些信息调整到运行时，再加上前面的语义抽象，我们就可以通过动态寻址计算，使得流程可以被直观的编辑。



# 参数传递

## 虚拟栈

模拟函数调用的过程，构造一个虚拟栈，在函数调用之前将参数传递至Lua Stack，然后执行组件函数，最后从Lua Stack获取计算结果




## 编辑而不是编译流程

在lua函数中就可以直接获取到虚拟机传递过来的数据，就像链接器正确计算了数据的内存地址，并将内存地址指向栈帧上正确的位置。

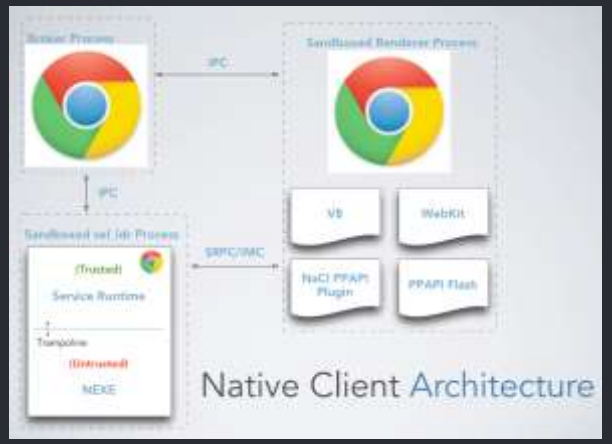
```
function updating()
    output["message"]["content"] = string.format("I have sent a message to %s with the following content: %s",
        input["email"]["name"],
        input["email"]["content"])
    output["message"]["receiver"] = "user"
    output["message"]["done"] = true
end
```

# 软件实现的沙盒保护

Software-based fault isolation, SFI

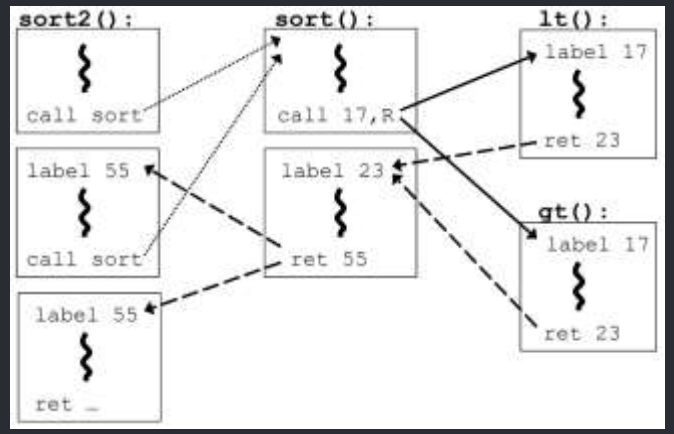
## Native Client SFI system, NaCl

用软件模拟硬件的内存保护机制，涉及较多的状态复制，开销较大。




## Control-Flow Integrity, CFI

Wasm通过在编译期利用CFI来检查程序对外部资源的访问，更高效。




## Roblox Luau

通过删除Lua中不安全的特性来实现沙盒技术，例如对全局数据的访问和代码执行。

**Luau**

Luau (lowercase *u*, /ˈlu.aʊ/) is a fast, small, safe, gradually typed embeddable scripting language derived from Lua.

# 具有互操作性的沙盒机制

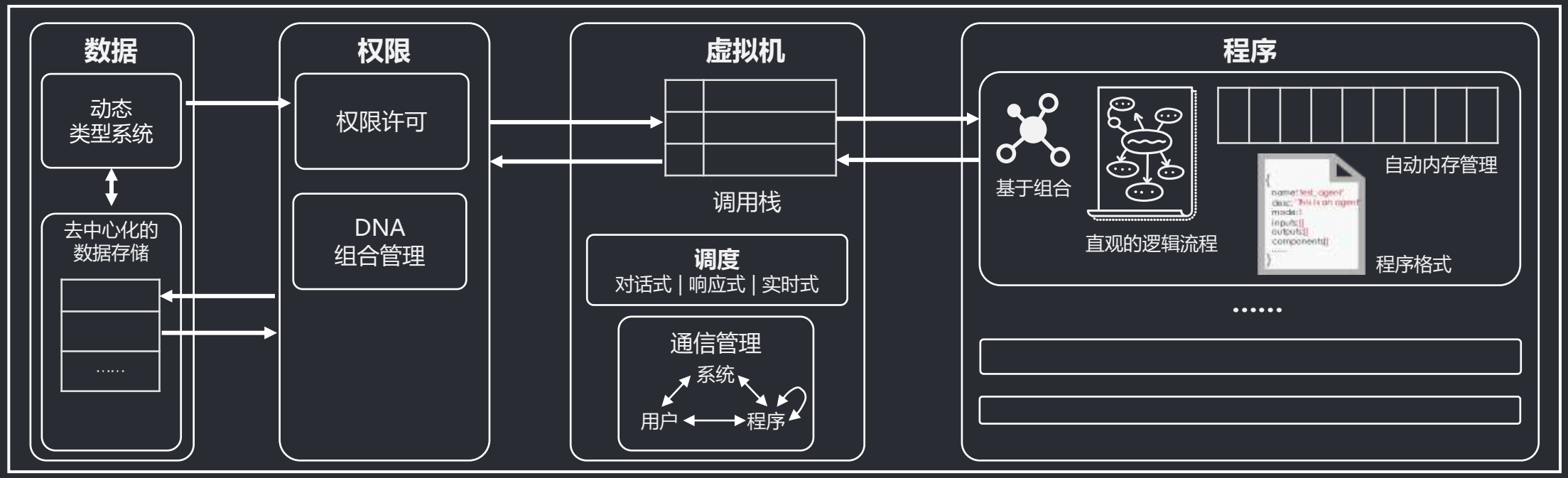
构建一个与运行时互操作性协作的沙盒机制

### 与运行时进行协作

传统的沙盒技术，为了支持通用的编译和解释过程，往往对虚拟机环境不做太多关联，导致较差的互操作性。

### 联合数据管理

通过联合Cogine的类型系统和互操作机制，程序减轻了对数据管理和程序结构组织的复杂度。

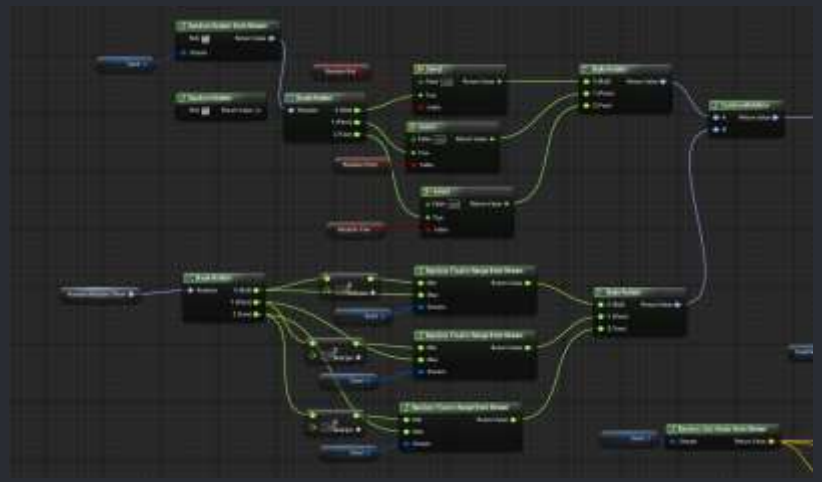


# 两种可视化编排方法

如果流程可以独立于运行时，程序的构建和管理就可以极大地简化

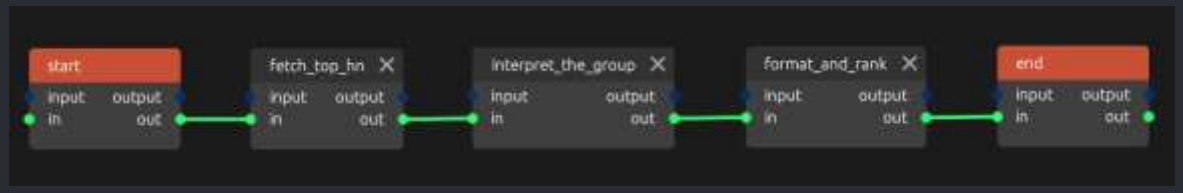
## 有数据（对象实例）线

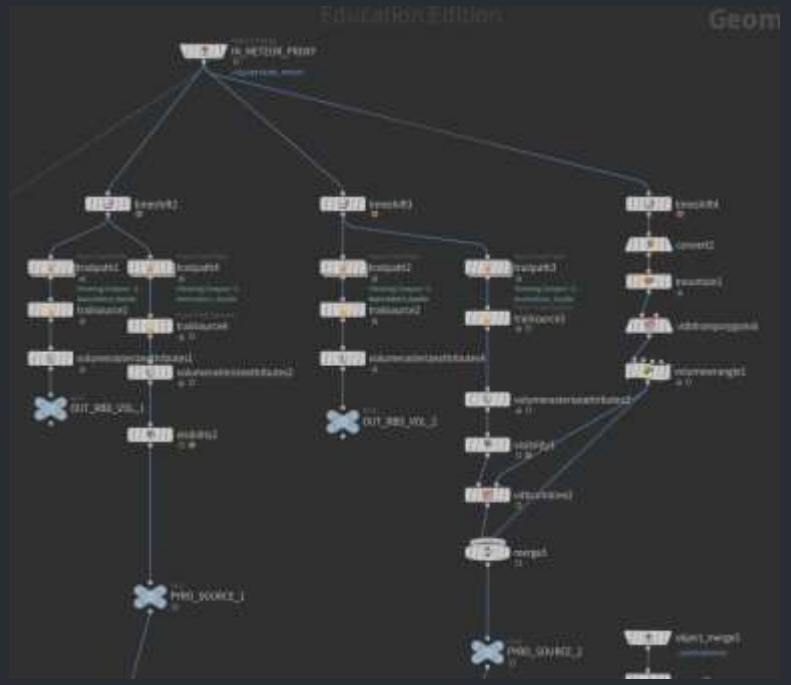
传统可视编程大部分都还是基于数据对象的，你需要关心数据的创建、存储，同时还需要关心正确地将这些数据实例的内存地址传递给正确的参数。



## 无数据（对象实例）线

如果虚拟机对运行时数据的使用有更多信息，就可以做到自动化的数据管理，同时这可以使得流程与运行时数据解耦，不必等到正确的实例数据就可以模拟运行。



# Cogine设计哲学三：进化计算

程序的运行时结构组织带来了动态性和个性化，这是进化计算的基础

### 基于互操作性的“复制”

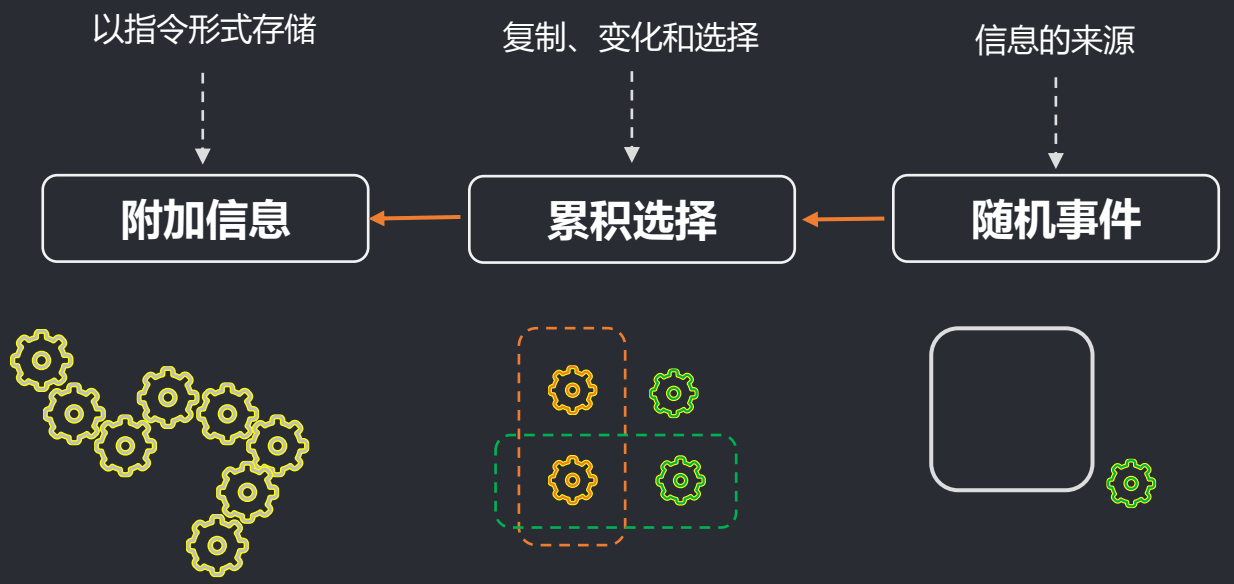
传统的程序代码都是私有的，外部无法对其进行复制；复制的目的是为了进行修改，通过互操作，就不需要直接复制原始源代码，可以对齐行为进行添加、修改和删除，这间接实现了复制的功能。

### 基于互操作的“交互”

子系统之间必须交互才能涌现更复杂的功能，互操作性使得任何个体之间的交互变得简单。

### 基于个性化组合的“变异与选择”

传统的程序功能都是固定的，用户无法进行修改，变异完全依赖于开发商自己，发生的机率很小；通过将程序变成一种个性化的组合，每个用户的程序都是不一样的，组合形成的，因此用户就可以进行广泛的自定i而形成变异与选择的功能。




### 程序定义为组合体

程序定义为一个组合体，而不是一个固定结构和功能的单一程序。

### 个性化

用户可以个性化的选择自己的组合，形成选择，那些优秀的程序将会被保留下来

### 互操作

你不是去修改已有程序，你通过互操作添加新的行为和特性

# Conversational Computing

大模型应用时代的新型软件架构

## 大模型应用痛点

基于大模型（LLM）的应用，一个函数的输入是语义化的文本而不是结构化的数据，这给程序带来了很多不确定性，这需要通过多轮对话来澄清更多信息。为了解决这个问题，我们首次提出一种称为对话式计算（Conversational Computing）的概念和技术，使得任何函数都可以任意被暂停以与用户、系统或者其它微程序进行对话以获取更多信息，并在条件满足之后自动恢复执行，整个过程就像编写传统串行程序一样简单，不需要做一些额外繁琐的工作。

## 多程序对话式协作框架

任何微程序可以以非常简单的方式与用户、系统和其它微程序进行通信，这提供一种能力通过使得程序之间通过协作的方式来完成复杂的任务。

```
function updating()
    if user == nil then
        user = input["message"]["content"]
    else
        user = string.format("%s. %s", user, input["message"]["content"])
    end

    local prompt = [[
        the input is : %s

        get the information the above input with the following <key, questions> pattern:
        name: what is the name of the person this email should send to?
        address: what is his email address?
        title: what is the title of this email?
        content: what is the content of this email?

        please return the answer in a json format only, with each key is the field name and the answer of each question is the value.
        if some key has no answer, set the value an empty string
    ]]

    local system = "get information only from the user's message, do not include any other information."

    local result = chat_completions(system, string.format(prompt, user))
    if result["ok"] == true then
        output["email"]["title"] = result["content"]["title"]
        output["email"]["content"] = result["content"]["content"]
        output["email"]["name"] = result["content"]["name"]
        output["email"]["address"] = result["content"]["address"]

        if output["email"]["address"] == nil or output["email"]["address"] == '' then
            output["message"]["content"] = "what is the address you want to send email?"
            output["message"]["receiver"] = "user"
        elseif output["email"]["content"] == nil or output["email"]["content"] == '' then
            output["message"]["content"] = "what is the content you want to sent?"
            output["message"]["receiver"] = "user"
        elseif output["email"]["title"] == nil or output["email"]["title"] == '' then
            output["message"]["content"] = "what is the title you want to sent?"
            output["message"]["receiver"] = "user"
        else
            output["message"]["content"] = ""
        end
    end
end
```

# 对话式计算带来交互变革

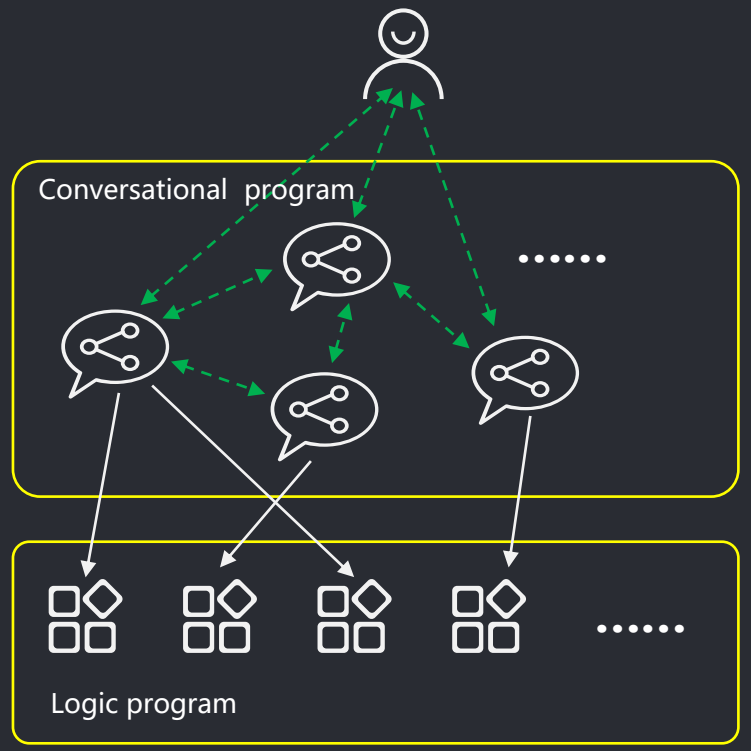
未来的应用程序不是固定功能，而是会思考和沟通的智能体程序

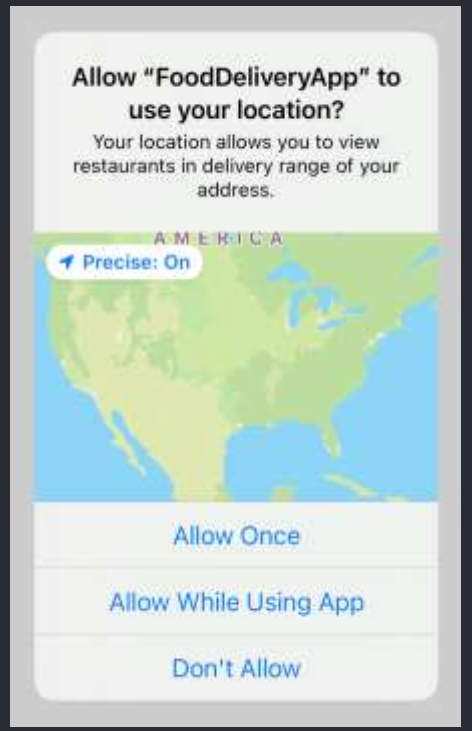
## 会沟通的智能程序

人类做事遵循两种过程：
1）人与人之间的交流澄清
2）执行明确的任务

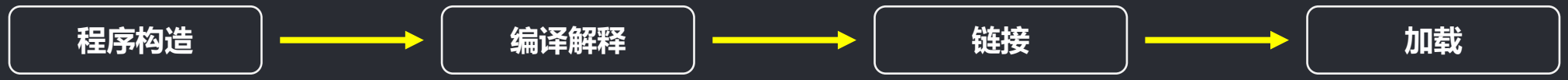
未来的程序不再是一个固定的功能，而是像一个人一样，它会根据自己的需求与用户或其它程序进行交流以澄清或者获取更多信息，然后执行确定的程序计算。

交互的界面和方式都变得灵活，就像真人面对问题，我们的思维是任意的，而不是某种固定界面。



# 总结




1）基于组合构造程序结构

1）编程语言语法不变
2）使用自定义的调用规则

1）动态类型
2）动态函数调用顺序

1）虚拟机动态程序组合
2）个性化的程序

# 完整Demo分析

在Reality Create中

1. 新建一个Agent程序，定义其运行模式
2. 创建Agent内部需要互操作的标准数据结构
3. 创建每一个流程，称为一个组件，并定义其数据的输入和输出
4. 创建整体流程
5. 为每个组件编写lua代码
6. 完成，

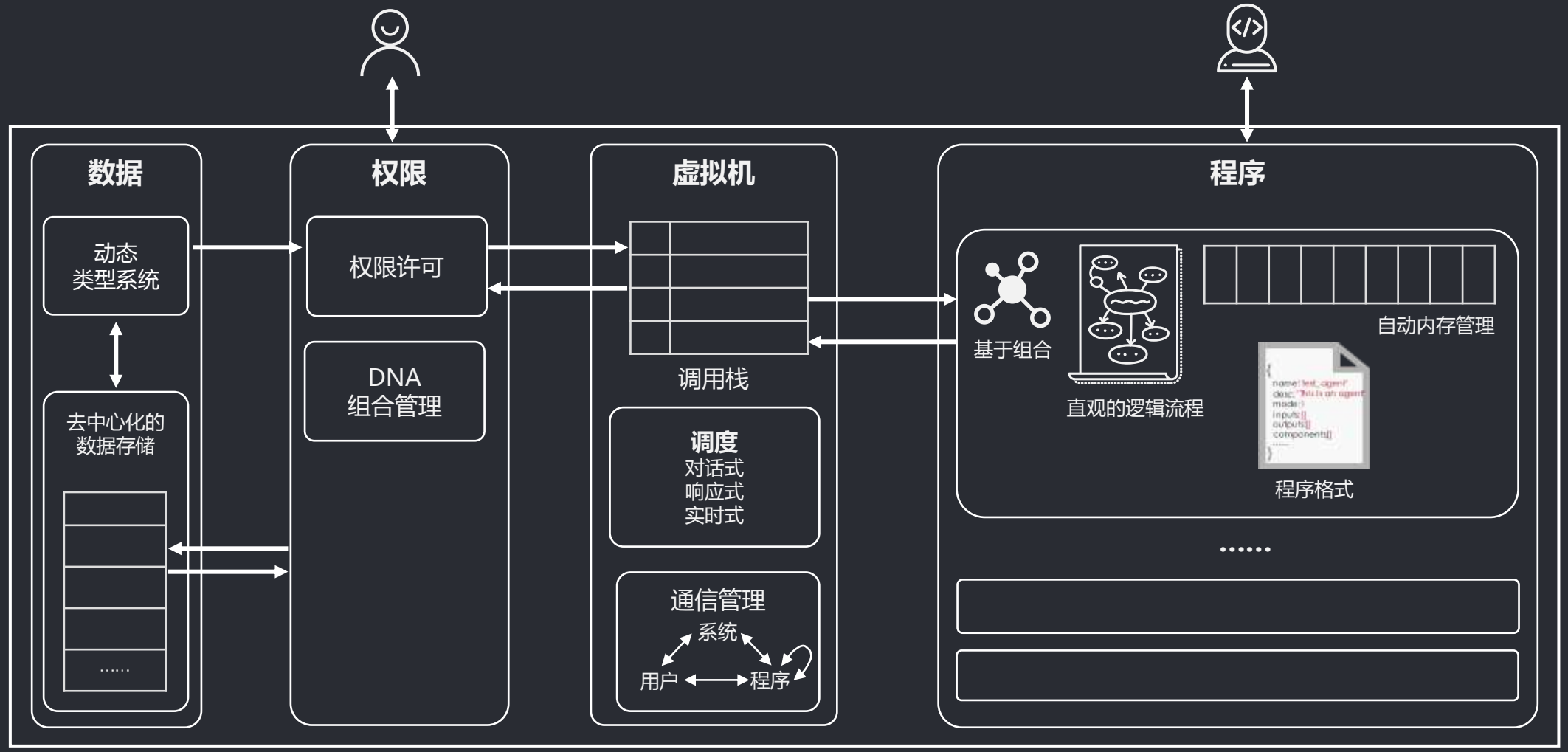
在Reality World中

7. 进行模拟测试

其它：

1. 调度模式
2. Agent通信
3. Conversational Computing
4. Standard
5. Component
6. Agent
7. Graph

# 结构总结


数据
动态
类型系统
去中心化的
数据存储
……
权限
权限许可
DNA
组合管理
虚拟机
调用栈
调度
对话式
响应式
实时式
通信管理
系统
用户
程序
程序
基于组合
直观的逻辑流程
自动内存管理
程序格式
……

# 特性总结

| 特性名称 | 描述 |
| --- | --- |
| 去中心化 | 用户的数据存储在应用程序之外，不受应用程序控制，并且用户可以控制所有数据的访问 |
| 互操作性 | 每个程序可以跟其它程序进行互操作，即使这些程序来自不同的开发者 |
| 多智能体协作框架 | 有标准的机制使得Agent在用户，系统和其它Agent之间进行通信 |
| 函数级别沙盒机制 | 每个Agent或者函数都可以被隔离，这样使用多个匿名的Agent程序可以在一个内存环境安全运行 |
| 对话式计算 | 每个函数可以任意跟用户、系统或其它程序进行任意通信以获取更多信息，该程序会被系统暂停直到获得响应后自动恢复执行，这种计算模型对LLM应用非常重要 |
| 语义抽象 | 运行时将对整个程序进行重新组织，这使得开发者仅需要关注逻辑，而不需要关心硬件、内存数据管理以及其它编程语言相关的细节，从而建立一种逻辑抽象 |
| 可视编辑 | 我们构建出一种像Houdini一样简单的可视逻辑编辑交互，同时又和其它任何图灵完备的编程语言一样具有灵活的表达能力 |
| Agent标准格式 | 我们定义了一个微程序（如AI智能体）的标准格式，它是图灵完备的程序，这个程序可以被动态加载和运行 |
| 虚拟机和动态加载运行 | 可以动态加载和运行 |
| 个性化动态组合 | 用户程序不是固定的结构，每个人都可以动态组合自己特定的组合，形成个性化的应用程序 |
| 统一应用程序 | 你不再需要分别管理和使用多个独立App，所有App是一个整体，通过自然自然语言进行交互 |

# 挑战 & 未来研究方向

## 技术挑战

**1）动态类型带来的性能问题**
改进虚拟机，优化函数栈的控制机制，优化组件的调度和数据管理，组件编译为像WASM的字节码。

**2）程序的持续运行模式**
传统的应用程序主要有基于GUI的响应式和类游戏的实时更新机制，这两种机制都比较简单，在多程序协同的情况下，程序的调度会更加复杂，需要结合应用慢慢形成更好的调度机制。

**3）对涌现机制的预测和管理**
传统软件的功能都是可预测的，管理起来比较容易，怎样设计更好的流程和机制便于理解和管理涌现机制，需要做很多研究和探索。

## 用户体验挑战

**1）从功能管理到数据管理**
现代移动操作系统中已经比较普遍，只是管控的是API，但是API代表的就是一种数据的获取。

**2）应用由主动执行到被动持续执行**
跟用户的主动输入不是同步的，可能是自动执行的，用户怎么感知系统变化状态？我们需要被通知，而不是一个我要做什么的数字世界。

被动式交互才能使数字世界的功能有巨大提升

**3）动态的数字世界**
人天生就是实时观察动态世界的，但是现实世界通常是一致的，信息世界的信息远远多于现实世界，交互方式就显得非常重要。

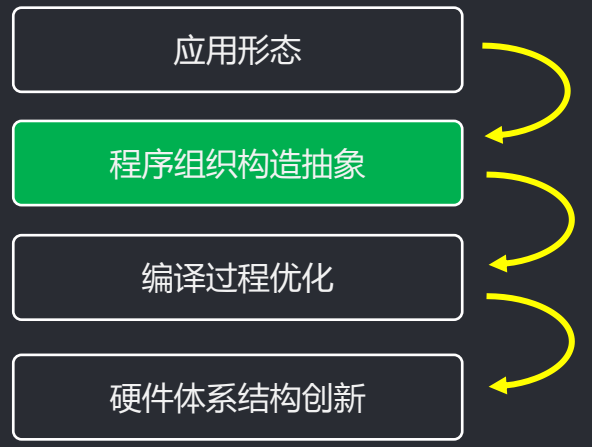
# 层级抽象

## 为什么不开发新的编程语言?

1）在语法层面，当代编程语言理论已经非常成熟
2）新语言带来学习成本，如果没有语言层面的理论进步，收益较少
3）相比语法而言，我们更需要程序结构组织方面的抽象


## 从上至下的体系结构变革

历史上，硬件结构体系的变革都是从上至下的，由应用层的软件形态，到程序组织的方式变化，最后引起编译器和硬件层的体系结构变化。
1）GPU
2）NPU



# 性能问题 & 体系架构启示

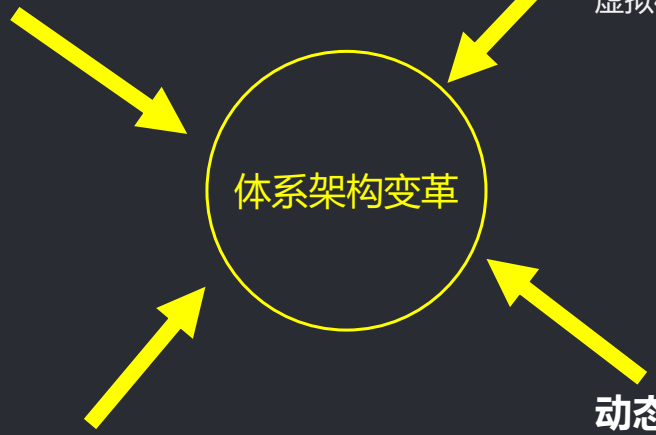
可以从多个层面全面促进编译器和体系结构的创新

**解释器的性能问题**

解释器主要发生在Lua层面，这部分后续可以将组件lua代码全部编译为Wasm，lua仅仅是充当语法构造代码。

**函数调用的性能问题**

自定义虚拟栈带来一定的复制拷贝性能开销，这部分可以考虑在虚拟机层面将rust内存地址直接映射到lua对象，实现零拷贝。




**数据内存占用**

相对传统应用，有更多的用于互操作的中间数据驻留内存，从三个当面可以优化：
1）这部分数据仅仅是用于互操作的数据，通过小心区分外部交互数据和内部对象状态数据，大部分内部状态数据是不受影响的。
2）数据缓存带来了更少的垃圾回收，或者手动创建和销毁的性能开销。
3）更好的数据缓存管理，大大减少缓存命中失败的次数，大幅提升性能

**动态寻址计算**

相比较于静态编译计算的代码地址，动态的寻址计算带来一定的开销，这部分可以设计一些性能更高效的数据结构，比如对编译器增加相应的寄存器来存储一些地址用于相对寻址等等。

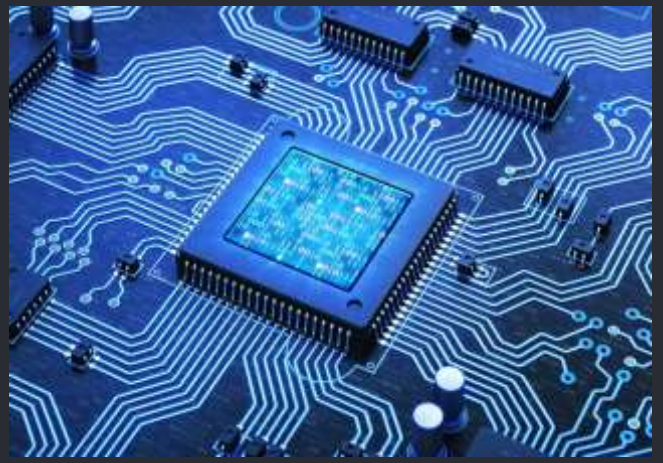
# 展望

## 驱动软件架构创新

为了迈向更加智能的数字世界，未来的软件构造会越来越复杂，需要新的方式去构造和管理复杂软件系统。

## 驱动硬件体系结构创新

通过对程序的组织结构、编译、解释以及链接过程的全方位调整，可以催生出一些新的体系结构。



## 驱动数字形态和体验创新

新的互操作性和多应用协同的新能力，将为未来元宇宙、AI、去中心化及XR应用带来全新体验。

## 驱动操作系统创新

操作系统是应用的引擎，这种应用形态的变化可以促进操作系统层面的演进，使在新的系统上开发复杂应用变得更加简单，从而延伸出更繁荣的数字化和数字世界。

谢 谢

# 敢于梦想就是力量

《为什么伟大不能被计划》

随机是创新和创造性的唯一来源