第五章

总体设计

概述

- 总体设计的基本目的
  - 回答“概括地说，系统应该如何实现?”
- 总体设计又称为概要设计或初步设计
- 目标
  - 通过该阶段的工作将划分出组成系统的物理元素
    - 程序、文件、数据库、人工过程和文档等等
      - 但是每个物理元素仍然处于黑盒子级，黑盒子里的具体内容将在以后仔细设计
  - 设计软件的结构
    - 确定系统中每个程序是由哪些模块组成的，以及模块相互间的关系
- 过程任务
  - 寻找实现目标系统的各种不同的方案
    - 需求分析阶段得到的数据流图是设想各种可能方案的基础
  - 分析员从这些供选择的方案中选取若干个合理的方案
    - 为每个合理的方案都准备一份系统流程图，列出组成系统的所有物理元素，进行成本/效益分析，并且制定实现这个方案的进度计划
    - 分析员应该综合分析比较这些合理的方案，从中选出一个最佳方案向用户和使用部门负责人推荐
  - 如果用户和使用部门的负责人接受了推荐的方案，分析员应该进一步为这个最佳方案设计软件结构
    - 包括：设计出初步的软件结构后还要多方改进，从而得到更合理的结构，进行必要的数据库设计，确定测试要求并且制定测试计划

总体设计的过程

- 两大阶段
  - 系统设计阶段，确定系统的具体实现方案
  - 结构设计阶段，确定软件结构
- 九个步骤
  - 设想供选择的方案
  - 选取合理的方案
  - 推荐最佳方案
  - 功能分解
  - 设计软件结构
  - 设计数据库
  - 制定测试计划
  - 书写文档
  - 审查和复审

设想供选择的方案

- 在总体设计阶段分析员应该考虑各种可能的实现方案，并且力求从中选出最佳方案
  - 在总体设计阶段开始时只有系统的逻辑模型，分析员有充分的自由分析比较不同的物理实现方案
    - 一旦选出了最佳的方案，将能大大提高系统的性能/价格比
- 需求分析阶段得出的数据流图是总体设计的极好的出发点
  - 设想供选择的方案过程中一条常用的方法
  - 设想把数据流图中的处理分组的各种可能的方法，抛弃在技术上行不通的分组方法(例如，组内不同处理的执行时间不相容)，余下的分组方法代表可能的实现策略，并且可以启示供选择的物理系统

选取合理的方案

- 应该从前一步得到的一系列供选择的方案中选取若干个合理的方案，通常至少选取低成本、中等成本和高成本的三种方案
- 在判断哪些方案合理时应该考虑在问题定义和可行性研究阶段确定的工程规模和目标，有时可能还需要进一步征求用户的意见
- 对每个合理的方案分析员都应该准备下列4份资料：
  - 系统流程图
  - 组成系统的物理元素清单
  - 成本/效益分析
  - 实现这个系统的进度计划

推荐最佳方案

- 分析员应该综合分析对比各种合理方案的利弊，推荐一个最佳的方案，并且为推荐的方案制定详细的实现计划
- 用户和有关的技术专家应该认真审查分析员所推荐的最佳系统
  - 如果该系统确实符合用户的需要，并且是在现有条件下完全能够实现的，则应该提请使用部门负责人进一步审批
  - 在使用部门的负责人也接受了分析员所推荐的方案之后，将进入总体设计过程的下一个重要阶段——结构设计

功能分解

- 为了最终实现目标系统，要设计提供组成这个系统的所有程序和文件(或数据库)
  - 对程序(特别是复杂的大型程序)的设计，通常分为两个阶段完成
    - 结构设计
      - 结构设计是总体设计阶段的任务
      - 结构设计确定程序由哪些模块组成，以及这些模块之间的关系
    - 过程设计
      - 过程设计是详细设计阶段的任务
      - 过程设计确定每个模块的处理过程
- 为确定软件结构，首先依据实现软件的条件对复杂的功能进一步分解
  - 分析员结合算法描述仔细分析数据流图中的每个处理，如果一个处理的功能过分复杂，必须把它的功能适当地分解成一系列比较简单的功能
  - 一般说来，经过分解之后应该使每个功能对大多数程序员而言都是明显易懂的
- 功能分解导致数据流图的进一步细化，同时还应该用IPO图或其他适当的工具简要描述细化后每个处理的算法

设计软件结构

- 通常程序中的一个模块完成一个适当的子功能
- 软件结构的约束条件
  - 把模块组织成良好的层次系统，顶层模块调用它的下层模块以实现程序的完整功能
  - 每个下层模块再调用更下层的模块，从而完成程序的一个子功能，最下层的模块完成最具体的功能
  - 软件结构(即由模块组成的层次系统)可以用层次图或结构图来描绘
- 如果数据流图已经细化到适当的层次，则可以直接从数据流图映射出软件结构

设计数据库

- 对于需要使用数据库的那些应用系统，软件工程师应该在需求分析阶段所确定的系统数据需求的基础上，进一步设计数据库

制定测试计划

- 在软件开发的早期阶段考虑测试问题，能促使软件设计人员在设计时注意提高软件的可测试性

书写文档

- 应该用正式的文档记录总体设计的结果，相关文档通常有：
  - 系统说明，主要内容包括
    - 用系统流程图描绘的系统构成方案，组成系统的物理元素清单，成本/效益分析；
    - 对最佳方案的概括描述，精化的数据流图，用层次图或结构图描绘的软件结构，用IPO图或其他工具(例如，PDL语言)简要描述的各个模块的算法，模块间的接口关系，以及需求、功能和模块三者之间的交叉参照关系等等
  - 用户手册
    - 根据总体设计阶段的结果，修改更正在需求分析阶段产生的初步的用户手册
  - 测试计划
    - 包括测试策略，测试方案，预期的测试结果，测试进度计划等等
  - 详细的实现计划
  - 数据库设计结果

审查和复审

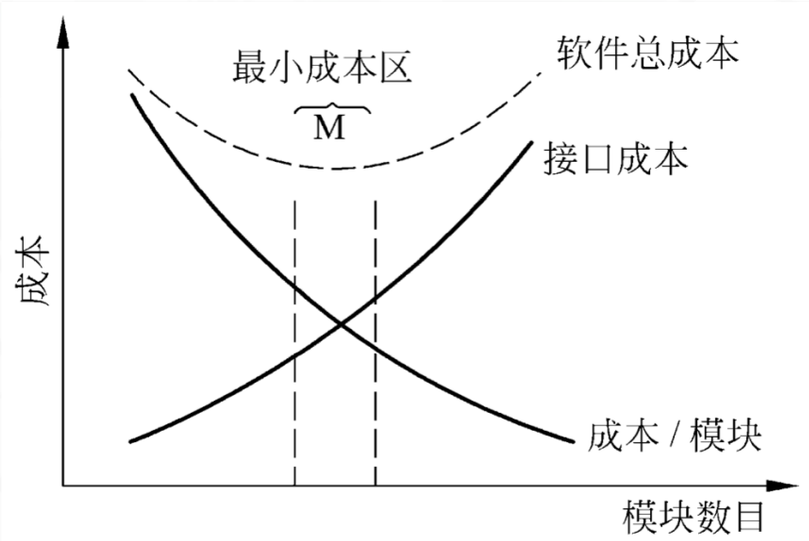
- 最后应该对总体设计的结果进行严格的技术审查
- 在技术审查通过之后再由使用部门的负责人从管理角度进行复审

设计原理

- 模块化
  - 模块
    - 是由边界元素限定的相邻程序元素（例如，数据说明，可执行的语句）的序列，而且有一个总体标识符代表它
    - 按照模块的定义，过程、函数、子程序和宏等，都可作为模块
    - 面向对象方法学中的对象是模块，对象内的方法（或称为服务）也是模块
    - 模块是构成程序的基本构件
  - 模块化
    - 是把程序划分成独立命名且可独立访问的模块，每个模块完成一个子功能，把这些模块集成起来构成一个整体，可以完成指定的功能满足用户的需求
- 模块化的理论根据
  - 设函数C(x)定义问题x的复杂程度，函数E(x)确定解决问题x需要的工作量(时间)。对于两个问题P1和P2，如果C(P1)>C(P2)，显然E(P1)>E(P2)
  - 根据人类解决一般问题的经验，另一个有趣的规律是C(P1+P2)>C(P1)+C(P2)
  - 即，如果一个问题由P1和P2两个问题组合而成，那么它的复杂程度大于分别考虑每个问题时的复杂程度之和。
  - 综上所述，得到下面的不等式E(P1+P2)>E(P1)+E(P2)

设计原理

- 模块化（cont.）
  - 模块化的理论根据
    - 如果无限地分割软件，最后为了开发软件而需要的工作量也就小得可以忽略了？？
    - 当模块数目增加时每个模块的规模将减小，开发单个模块需要的成本(工作量)确实减少了；但是，随着模块数目增加，设计模块间接口所需要的工作量也将增加
    - 每个程序都相应地有一个最适当的模块数目M，使得系统的开发成本最小
  - 虽然目前还不能精确地决定M的数值，但是在考虑模块化的时候总成本曲线确实是有用的指南



设计原理

- 模块化（cont.）
  - 模块化的优点
    - 采用模块化原理可以使软件结构清晰，不仅容易设计也容易阅读和理解
    - 程序错误通常局限在有关的模块及它们之间的接口中，所以模块化使软件容易测试和调试，因而有助于提高软件的可靠性
      - 因为变动往往只涉及少数几个模块，所以模块化能够提高软件的可修改性
    - 模块化也有助于软件开发工程的组织管理，一个复杂的大型程序可以由许多程序员分工编写不同的模块，并且可以进一步分配技术熟练的程序员编写困难的模块

抽象

- 抽象是人类在认识复杂现象的过程中使用的最强有力的思维工具
  - 在现实世界中一定事物、状态或过程之间总存在着某些相似的方面(共性)
  - 把这些相似的方面集中和概括起来，暂时忽略它们之间的差异，即为抽象
  - 抽象就是抽出事物的本质特性而暂时不考虑它们的细节
- 由于人类思维能力的限制，如果每次面临的因素太多，是不可能做出精确思维的
  - 处理复杂系统的惟一有效的方法是用层次的方式构造和分析该系统
    - 一个复杂的动态系统首先可以用一些高级的抽象概念构造和理解
    - 这些高级概念又可以用一些较低级的概念构造和理解
    - 如此进行下去，直至最低层次的具体元素
- 考虑对任何问题的模块化解法时，可以提出许多抽象的层次
  - 在抽象的最高层次使用问题环境的语言，以概括的方式叙述问题的解法
  - 在较低抽象层次采用更过程化的方法，把面向问题的术语和面向实现的术语结合起来叙述问题的解法
  - 在最低的抽象层次用可直接实现的方式叙述问题的解法
- 软件工程过程的每一步都是对软件解法的抽象层次的一次精化
  - 在可行性研究阶段，软件作为系统的一个完整部件
  - 在需求分析期间，软件解法是使用在问题环境内熟悉的方式描述的
  - 当由总体设计向详细设计过渡时，抽象的程度也就随之减少了
  - 当源程序写出来以后，也就达到了抽象的最低层

抽象

- 逐步求精和模块化的概念，与抽象是紧密相关的
  - 随着软件开发工程的进展，在软件结构每一层中的模块，表示了对软件抽象层次的一次精化
    - 软件结构顶层的模块，控制了系统的主要功能并且影响全局
    - 在软件结构底层的模块，完成对数据的一个具体处理，用自顶向下由抽象到具体的方式分配控制，简化了软件的设计和实现，提高了软件的可理解性和可测试性，并且使软件更容易维护

逐步求精

- 逐步求精可以被看作
  - 是一项把一个时期内必须解决的种种问题按优先级排序的技术
  - 逐步求精方法确保每个问题都将被解决，而且每个问题都将在适当的时候被解决

- 逐步求精是人类解决复杂问题时采用的基本方法，也是许多软件工程技术（例如，规格说明技术，设计和实现技术）的基础
  - 可以把逐步求精定义为：“为了能集中精力解决主要问题而尽量推迟对问题细节的考虑。”
- 逐步求精之所以如此重要，是因为人类的认知过程遵守Miller法则
  - 一个人在任何时候都只能把注意力集中在（7±2）个知识块上

逐步求精

- 关于Miller法则
  - 一个人在任何时候都只能把注意力集中在（7±2）个知识块上
    - 但是，在开发软件的过程中，软件工程师在一段时间内需要考虑的知识块数远远多于7
  - 例如，一个程序通常不止使用7个数据，一个用户也往往有不止7个方面的需求
- 逐步求精方法的作用
  - 能帮助软件工程师把精力集中在与当前开发阶段最相关的方面上
  - 能帮助忽略那些对整体解决方案来说虽然是必要的，然而目前还不需要考虑的细节，这些细节将留到以后再考虑

- Miller法则是人类智力的基本局限，我们不可能战胜自己的自然本性，只能接受这个事实，承认自身的局限性，并在这个前提下尽个人的最大努力工作

局部化和信息隐藏

- 应用模块化原理时，自然会产生的一个问题是：“为了得到最好的一组模块，应该怎样分解软件呢?”
- 根据信息隐藏原理指出
  - 应该通过设计和对模块的确定，使得一个模块内包含的信息(过程和数据)对于不需要这些信息的模块来说，是不能访问的
- 局部化的概念和信息隐藏概念是密切相关的
  - 所谓局部化是指把一些关系密切的软件元素物理地放得彼此靠近
  - 在模块中使用局部数据元素是局部化的一个例子
  - 显然，局部化有助于实现信息隐藏
- “隐藏”意味着有效的模块化可以通过定义一组独立的模块而实现，这些独立的模块彼此间仅仅交换那些为了完成系统功能而必须交换的信息。
  - 如果在测试期间和以后的软件维护期间需要修改软件，那么使用信息隐藏原理作为模块化系统设计的标准就会带来极大好处
    - 因为绝大多数数据和过程对于软件的其他部分而言是隐藏的(也就是“看”不见的)，在修改期间由于疏忽而引入的错误就很少可能传播到软件的其他部分

模块独立

- 模块独立的概念是模块化、抽象、信息隐藏和局部化概念的直接结果
- 开发具有独立功能而且和其他模块之间没有过多的相互作用的模块，就可以做到模块独立
  - 即，希望这样设计软件结构，使得每个模块完成一个相对独立的特定子功能，并且和其他模块之间的关系很简单
- 重要性
  - 有效的模块化(即具有独立的模块)的软件比较容易开发出来；这是由于能够分割功能而且接口可以简化，当许多人分工合作开发同一个软件时，这个优点尤其重要
  - 独立的模块比较容易测试和维护
    - 这是因为相对说来，修改设计和程序需要的工作量比较小，错误传播范围小，需要扩充功能时能够“插入”模块。总之，模块独立是好设计的关键，而设计又是决定软件质量的关键环节
- 衡量方式
  - 模块的独立程度可以由两个定性标准度量：内聚和耦合
- 耦合衡量不同模块彼此间互相依赖(连接)的紧密程度
- 内聚衡量一个模块内部各个元素彼此结合的紧密程度

耦合

- 耦合是对一个软件结构内不同模块之间互连程度的度量
- 耦合强弱取决于模块间接口的复杂程度，进入或访问一个模块的点，以及通过接口的数据
- 在软件设计中应该追求尽可能松散耦合的系统
  - 在这样的系统中可以研究、测试或维护任何一个模块，而不需要对系统的其他模块有很多了解
  - 由于模块间联系简单，发生在一处的错误传播到整个系统的可能性就很小
  - 因此，模块间的耦合程度强烈影响系统的可理解性、可测试性、可靠性和可维护性
- 如果两个模块中的每一个都能独立地工作而不需要另一个模块的存在，那么它们彼此完全独立，这意味着模块间无任何连接，耦合程度最低
  - 但是，在一个软件系统中不可能所有模块之间都没有任何连接。

耦合

- 种类
  - 数据耦合是低耦合
    - 如果两个模块彼此间通过参数交换信息，而且交换的信息仅仅是数据，那么这种耦合称为数据耦合
    - 系统中至少必须存在这种耦合，因为只有当某些模块的输出数据作为另一些模块的输入数据时，系统才能完成有价值的功能。一般说来，一个系统内可以只包含数据耦合
  - 控制耦合是中等程度的耦合
    - 如果传递的信息中有控制信息(尽管有时这种控制信息以数据的形式出现)，则这种耦合称为控制耦合
    - 它增加了系统的复杂程度，控制耦合往往是多余的，在把模块适当分解之后通常可以用数据耦合代替它
  - 特征耦合
    - 如果被调用的模块需要使用作为参数传递进来的数据结构中的所有元素，那么，把整个数据结构作为参数传递就是完全正确的。
    - 但是，当把整个数据结构作为参数传递而被调用的模块只需要使用其中一部分数据元素时，就出现了特征耦合
    - 在这种情况下，被调用的模块可以使用的数据多于它确实需要的数据，这将导致对数据的访问失去控制，从而给计算机犯罪提供了机会
  - 公共环境耦合
    - 当两个或多个模块通过一个公共数据环境相互作用时，它们之间的耦合称为公共环境耦合
    - 公共环境可以是全程变量、共享的通信区、内存的公共覆盖区、任何存储介质上的文件、物理设备等等

耦合

- 种类
  - 公共环境耦合的复杂程度随耦合的模块个数而变化，当耦合的模块个数增加时复杂程度显著增加
    - 如果只有两个模块有公共环境，那么这种耦合有两种可能：
      - 一个模块往公共环境送数据，另一个模块从公共环境取数据。这是数据耦合的一种形式，是比较松散的耦合。
      - 两个模块都既往公共环境送数据又从里面取数据，这种耦合比较紧密，介于数据耦合和控制耦合之间。
    - 如果两个模块共享的数据很多，都通过参数传递可能很不方便，这时可以利用公共环境耦合
  - 最高程度的耦合是内容耦合
    - 如果出现下列情况之一，两个模块间就发生了内容耦合
      - 一个模块访问另一个模块的内部数据
      - 一个模块不通过正常入口而转到另一个模块的内部
      - 两个模块有一部分程序代码重叠(只可能出现在汇编程序中)
      - 一个模块有多个入口(这意味着一个模块有几种功能)
    - 应该坚决避免使用内容耦合。事实上许多高级程序设计语言已经设计成不允许在程序中出现任何形式的内容耦合
- 针对耦合的设计原则
  - 尽量使用数据耦合，少用控制耦合和特征耦合，限制公共环境耦合的范围，完全不用内容耦合

内聚

- 内聚
  - 标志一个模块内各个元素彼此结合的紧密程度，它是信息隐藏和局部化概念的自然扩展
  - 简单地说，理想内聚的模块只做一件事情
- 设计时应该力求做到高内聚
  - 通常中等程度的内聚也是可以采用的，而且效果和高内聚相差不多
    - 但是，低内聚的效果很差，尽量杜绝使用
- 内聚与耦合
  - 内聚和耦合是密切相关的，模块内的高内聚往往意味着模块间的松耦合
  - 内聚和耦合都是进行模块化设计的有力工具，但是实践表明内聚更重要，应该把更多注意力集中到提高模块的内聚程度上

低内聚的种类

- 偶然内聚
  - 如果一个模块完成一组任务，这些任务彼此间即使有关系，关系也是很松散的，就叫做偶然内聚
    - 有时在写完一个程序之后，发现一组语句在两处或多处出现，于是把这些语句作为一个模块以节省内存，这样就出现了偶然内聚的模块
- 逻辑内聚
  - 如果一个模块完成的任务在逻辑上属于相同或相似的一类，则称为逻辑内聚
- 时间内聚
  - 如果一个模块包含的任务必须在同一段时间内执行，就叫时间内聚

中内聚种类

- 过程内聚
  - 如果一个模块内的处理元素是相关的，而且必须以特定次序执行，则称为过程内聚
    - 使用程序流程图作为工具设计软件时，常常通过研究流程图确定模块的划分，这样得到的往往是过程内聚的模块
- 通信内聚
  - 如果模块中所有元素都使用同一个输入数据和(或)产生同一个输出数据，则称为通信内聚

高内聚种类

- 顺序内聚
  - 如果一个模块内的处理元素和同一个功能密切相关，而且这些处理必须顺序执行(通常一个处理元素的输出数据作为下一个处理元素的输入数据)，则称为顺序内聚
    - 根据数据流图划分模块时，通常得到顺序内聚的模块，这种模块彼此间的连接往往比较简单
- 功能内聚
  - 如果模块内所有处理元素属于一个整体，完成一个单一的功能，则称为功能内聚
  - 功能内聚是最高程度的内聚。

对内聚的评价

- 功能内聚 10分
- 顺序内聚 9分
- 通信内聚 7分
- 过程内聚 5分
- 时间内聚 3分
- 逻辑内聚 1分
- 偶然内聚 0分

- 没有必要精确确定内聚的级别。重要的是设计时力争做到高内聚，并且能够辨认出低内聚的模块，有能力通过修改设计提高模块的内聚程度降低模块间的耦合程度，从而获得较高的模块独立性

启发规则

- 启发式规则虽然不能普遍适用，但是在许多场合仍然能给软件工程师有利的启示，往往能帮助他们找到改进软件设计提高软件质量的途径
  - 改进软件结构提高模块独立性
  - 模块规模应该适中
  - 深度、宽度、扇出和扇入都应适当
  - 模块的作用域应该在控制域之内
  - 力争降低模块接口的复杂程度
  - 设计单入口单出口的模块
  - 模块功能应该可以预测

改进软件结构提高模块独立性

- 设计出软件的初步结构以后，应该审查分析这个结构，通过模块分解或合并，力求降低耦合提高内聚
  - 例如，多个模块公有的一个子功能可以独立成一个模块，由这些模块调用
  - 有时可以通过分解或合并模块以减少控制信息的传递及对全程数据的引用，并且降低接口的复杂程度

模块规模应该适中

- 经验表明，一个模块的规模不应过大，最好能写在一页纸内
  - 通常不超过60行语句
  - 从心理学角度研究得知，当一个模块包含的语句数超过30以后，模块的可理解程度迅速下降
- 过大的模块往往是由于分解不充分，但是进一步分解必须符合问题结构，一般说来，分解后不应该降低模块独立性
- 过小的模块开销大于有效操作，而且模块数目过多将使系统接口复杂
  - 因此过小的模块有时不值得单独存在，特别是只有一个模块调用它时，通常可以把它合并到上级模块中去而不必单独存在

深度、宽度、扇出和扇入都应适当

- 深度表示软件结构中控制的层数，它往往能粗略地标志一个系统的大小和复杂程度
  - 深度和程序长度之间应该有粗略的对应关系，且对应关系是在一定范围内变化
  - 如果层数过多，则应考虑是否有许多管理模块过分简单，能否适当合并
- 宽度是软件结构内同一个层次上的模块总数的最大值
  - 一般说来，宽度越大系统越复杂
  - 对宽度影响最大的因素是模块的扇出
- 扇出是一个模块直接控制(调用)的模块数目
  - 扇出过大意味着模块过分复杂，需要控制和协调过多的下级模块
    - 扇出太大一般是因为缺乏中间层次，应该适当增加中间层次的控制模块
  - 扇出过小(例如总是1)也存在问题
    - 扇出太小时可以把下级模块进一步分解成若干个子功能模块，或者合并到其上级模块中
  - 经验表明，一个设计得好的典型系统的平均扇出通常是3或4(扇出的上限通常是5～9)
  - 分解模块或合并模块必须符合问题结构，不能违背模块独立原理
- 一个模块的扇入表明有多少个上级模块直接调用它
  - 扇入越大则共享该模块的上级模块数目越多
  - 扇入大是有利的现象，但是，不能违背模块独立原理单纯追求高扇入
  - 观察大量软件系统后发现，设计得很好的软件结构通常顶层扇出比较高，中层扇出较少，底层扇入到公共的实用模块中去(底层模块有高扇入)

模块的作用域应该在控制域之内

- 模块的作用域
  - 受该模块内一个判定影响的所有模块的集合
- 模块的控制域
  - 该模块本身以及所有直接或间接从属于它的模块的集合
- 在一个设计得很好的系统中
  - 所有受判定影响的模块应该都从属于做出判定的那个模块
  - 最好局限于做出判定的那个模块本身及它的直属下级模块

力争降低模块接口的复杂程度

- 模块接口复杂是软件发生错误的一个主要原因
  - 应该仔细设计模块接口，使得信息传递简单并且和模块的功能一致
- 接口复杂或不一致(即看起来传递的数据之间没有联系)，是紧耦合或低内聚的征兆，应该重新分析这个模块的独立性

设计单入口单出口的模块

- 提醒软件工程师不要使模块间出现内容耦合
  - 当从顶部进入模块并且从底部退出来时，软件是比较容易理解的，因此也是比较容易维护的

模块功能应该可以预测

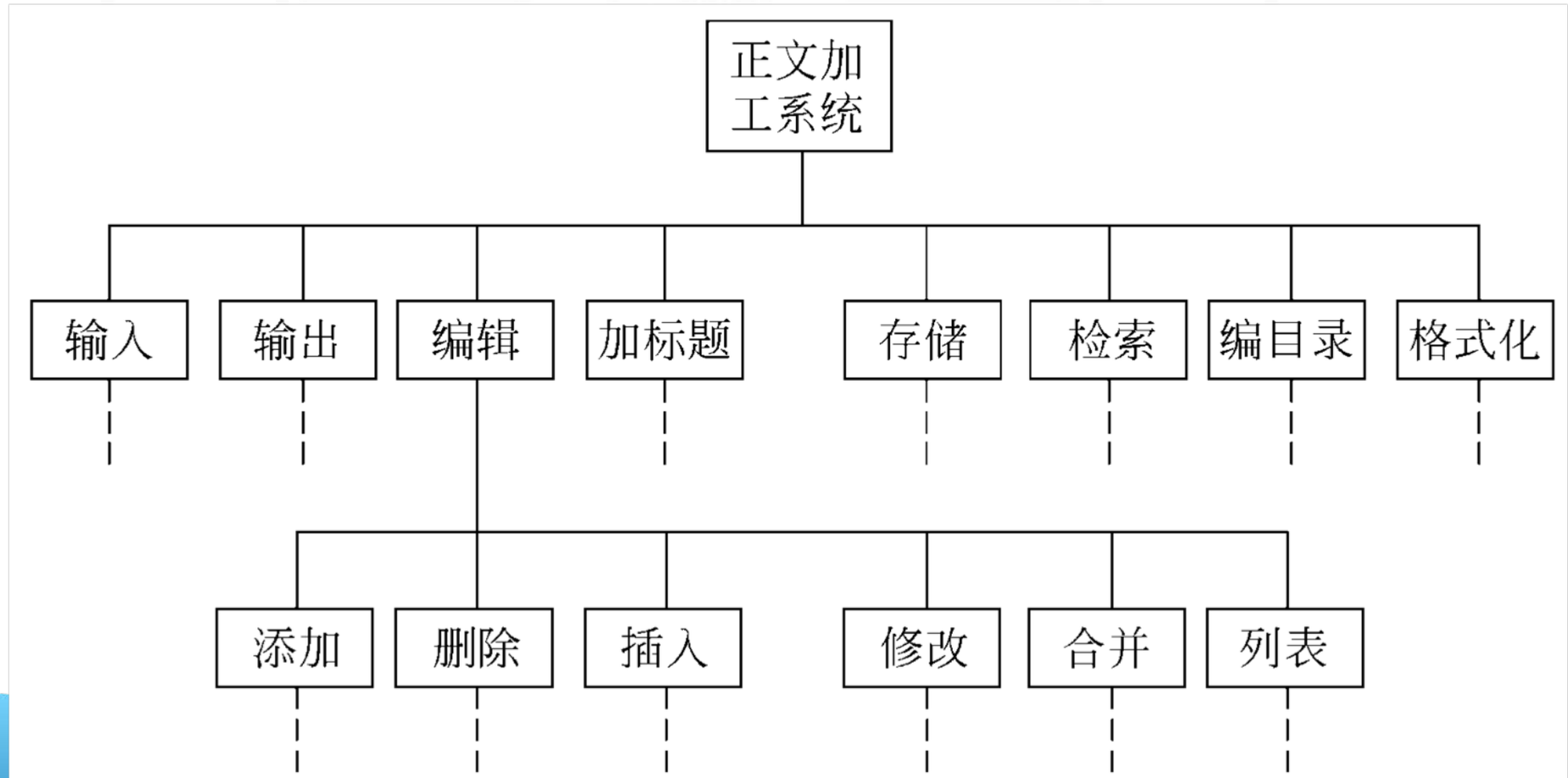
- 要防止模块功能过分局限
  - 如果一个模块可以当做一个黑盒子，即，只要输入的数据相同就产生同样的输出，该模块的功能就是可以预测的
  - 带有内部“存储器”的模块的功能可能是不可预测的，因为它的输出可能取决于内部存储器(例如某个标记)的状态
    - 由于内部存储器对于上级模块而言是不可见的，所以这样的模块既不易理解又难于测试和维护
  - 如果一个模块只完成一个单独的子功能，则呈现高内聚；但是，如果一个模块任意限制局部数据结构的大小，过分限制在控制流中可以做出的选择或者外部接口的模式，那么这种模块的功能就过分局限，使用范围也就过分狭窄

描绘软件结构的图形工具

- 层次图和HIPO图
- 结构图

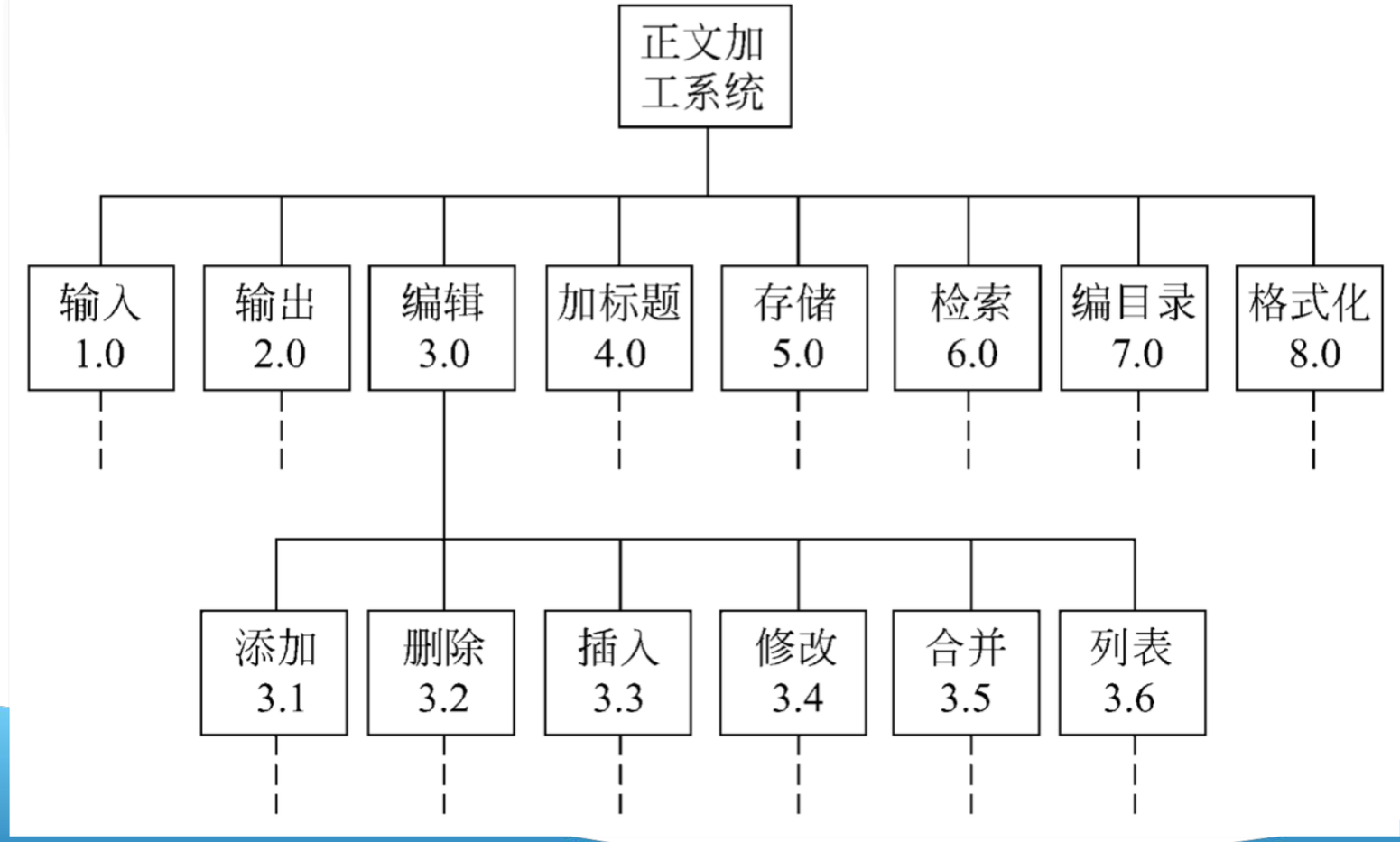
层次图和HIPO图

- 层次图用来描绘软件的层次结构
  - 层次图中的一个矩形框代表一个模块，方框间的连线表示调用关系而不像层次方框图那样表示组成关系



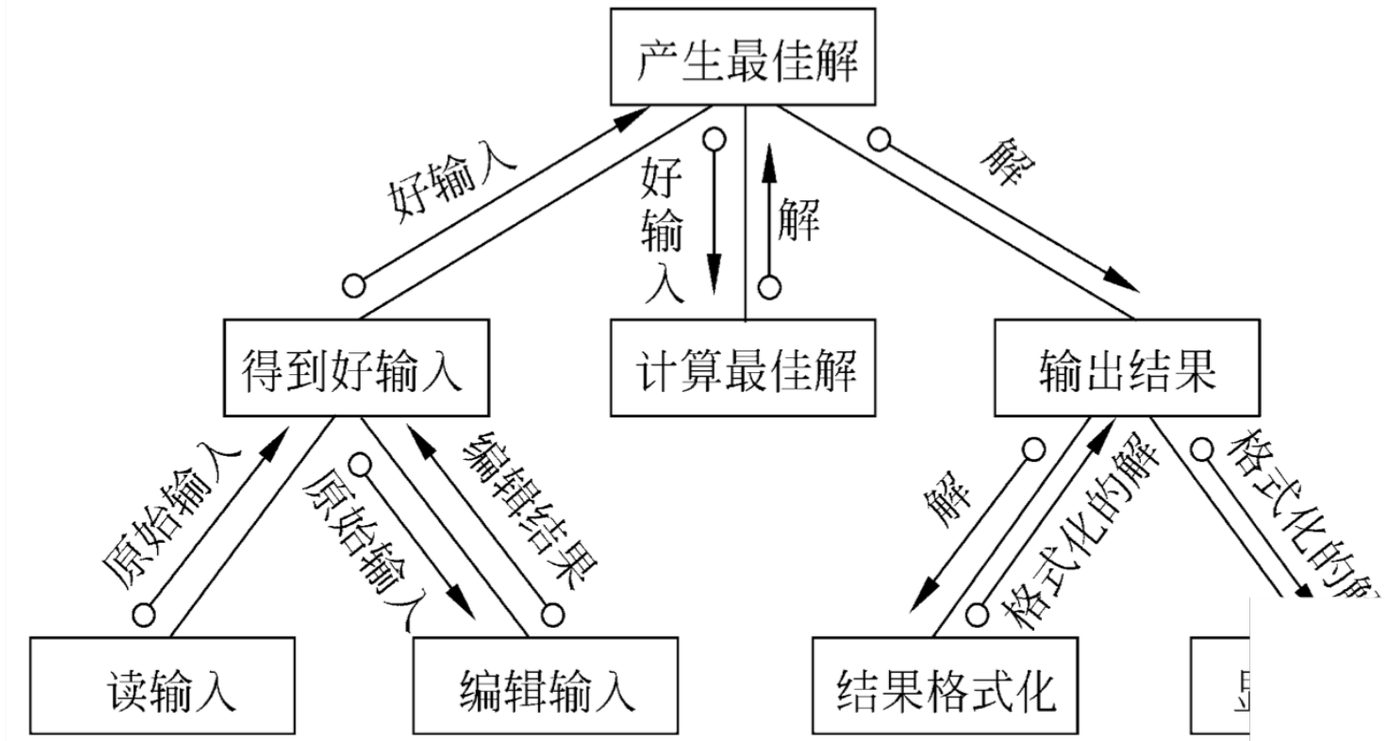
层次图和HIPO图

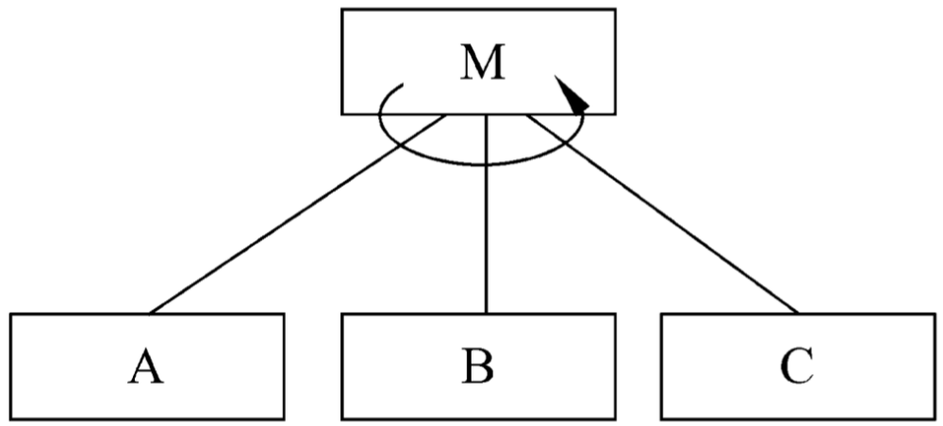
- 层次图很适于在自顶向下设计软件的过程中使用
- HIPO图是美国IBM公司发明的“层次图加输入/处理/输出图”的英文缩写
  - 为了能使HIPO图具有可追踪性，在H图(层次图)里除了最顶层的方框之外，每个方框都加了编号

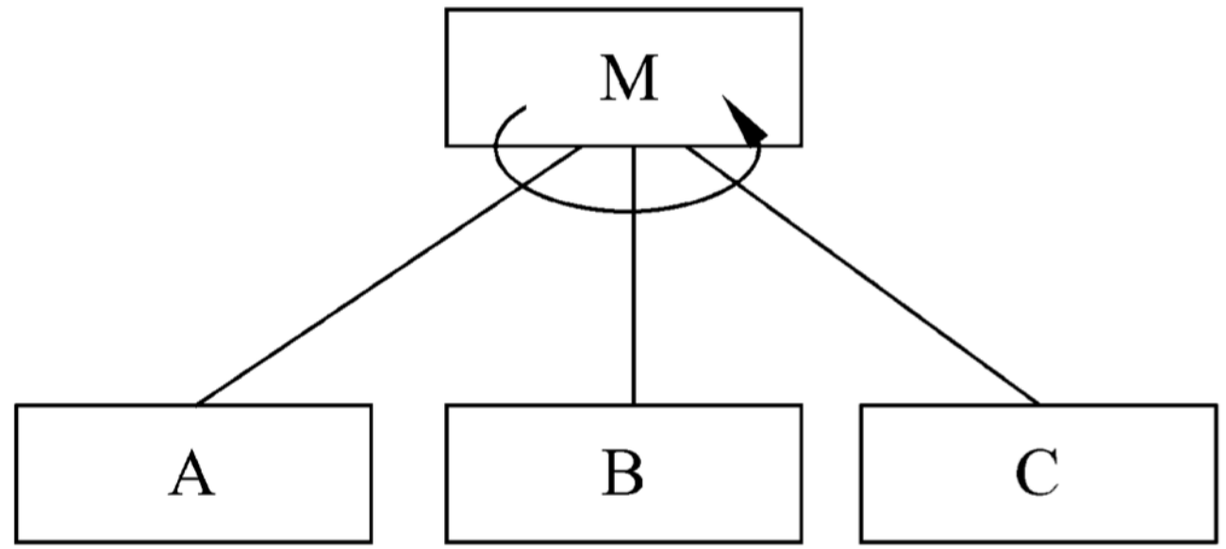


结构图

- Yourdon提出的结构图是进行软件结构设计的有力工具
- 结构图和层次图类似
  - 图中一个方框代表一个模块，框内注明模块的名字或主要功能
  - 方框之间的箭头(或直线)表示模块的调用关系
  - 用带注释的箭头表示模块调用过程中来回传递的信息
    - 如果希望进一步标明传递的信息是数据还是控制信息，则可以利用注释箭头尾部的形状来区分：尾部是空心圆表示传递的是数据，实心圆表示传递的是控制信息
- 因为按照惯例总是图中位于上方的方框代表的模块调用下方的模块，即使不用箭头也不会产生二义性


产生最佳解
好输入
好输入
解
解
得到好输入
计算最佳解
输出结果
原始输入
原始输入
编辑结果
解
格式化的解
格式化的解
读输入
编辑输入
结果格式化


M
A
B
C

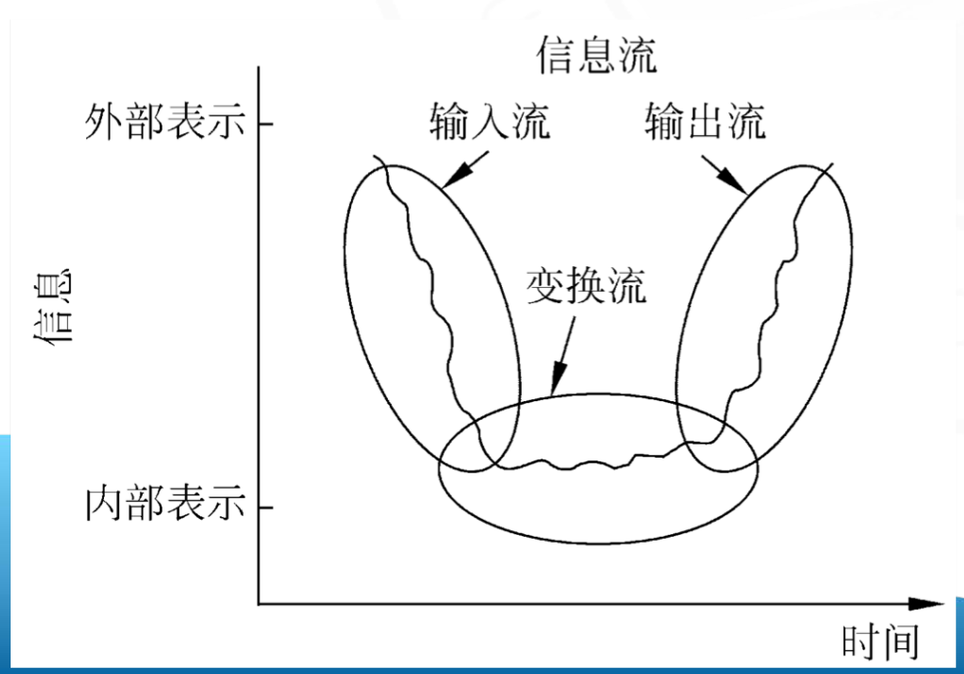

M
A
B
C

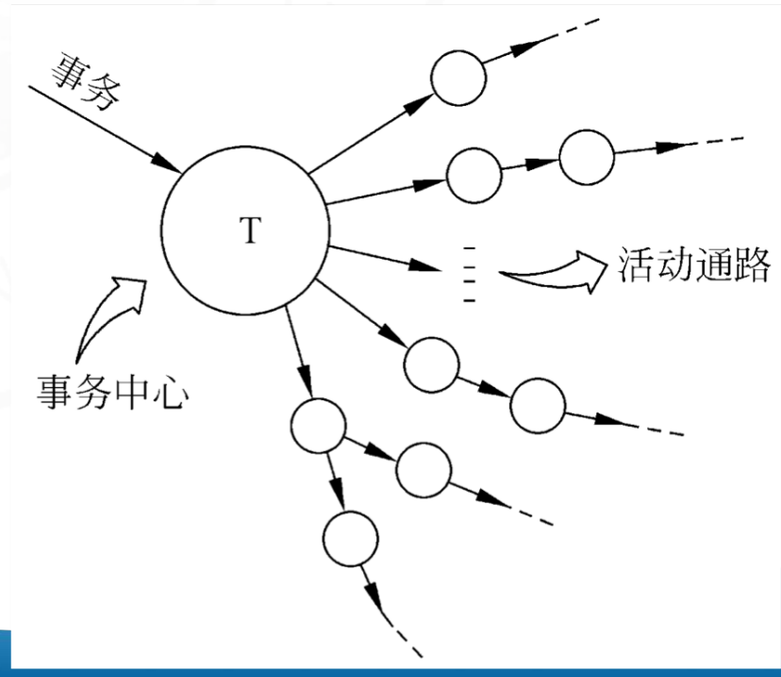
面向数据流的设计方法

- 目标
  - 给出设计软件结构的一个系统化的途径
- 在软件工程的需求分析阶段，信息流是一个关键考虑，通常用数据流图描绘信息在系统中加工和流动的情况
- 面向数据流的设计方法定义了一些不同的“映射”，利用这些映射可以把数据流图变换成软件结构
  - 因为任何软件系统都可以用数据流图表示，所以面向数据流的设计方法理论上可以设计任何软件的结构通常所说的结构化设计方法(简称SD方法)，也就是基于数据流的设计方法

面向数据流的设计方法

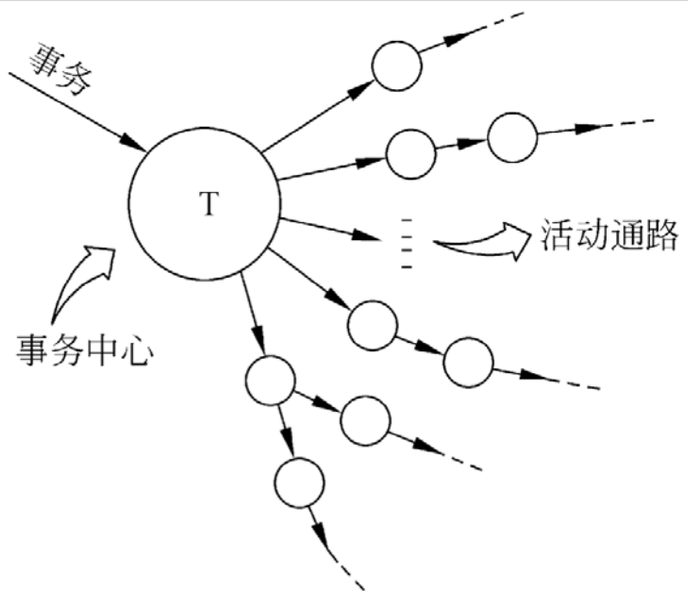
- 基本概念
  - 面向数据流的设计方法把信息流映射成软件结构
  - 信息流的类型决定了映射的方法
- 信息流的类型
  - 变换流
    - 信息沿输入通路进入系统，同时由外部形式变换成内部形式，进入系统的信息通过变换中心，经加工处理以后再沿输出通路变换成外部形式离开软件系统。当数据流图具有这些特征时，这种信息流就叫作变换流
  - 事务流
    - 基本系统模型意味着变换流，因此，原则上所有信息流都可以归结为变换流
    - 当数据流是“以事务为中心的”，即，数据沿输入通路到达一个处理T，这个处理根据输入数据的类型在若干个动作序列中选出一个来执行，则应为一类特殊的数据流，称为事务流

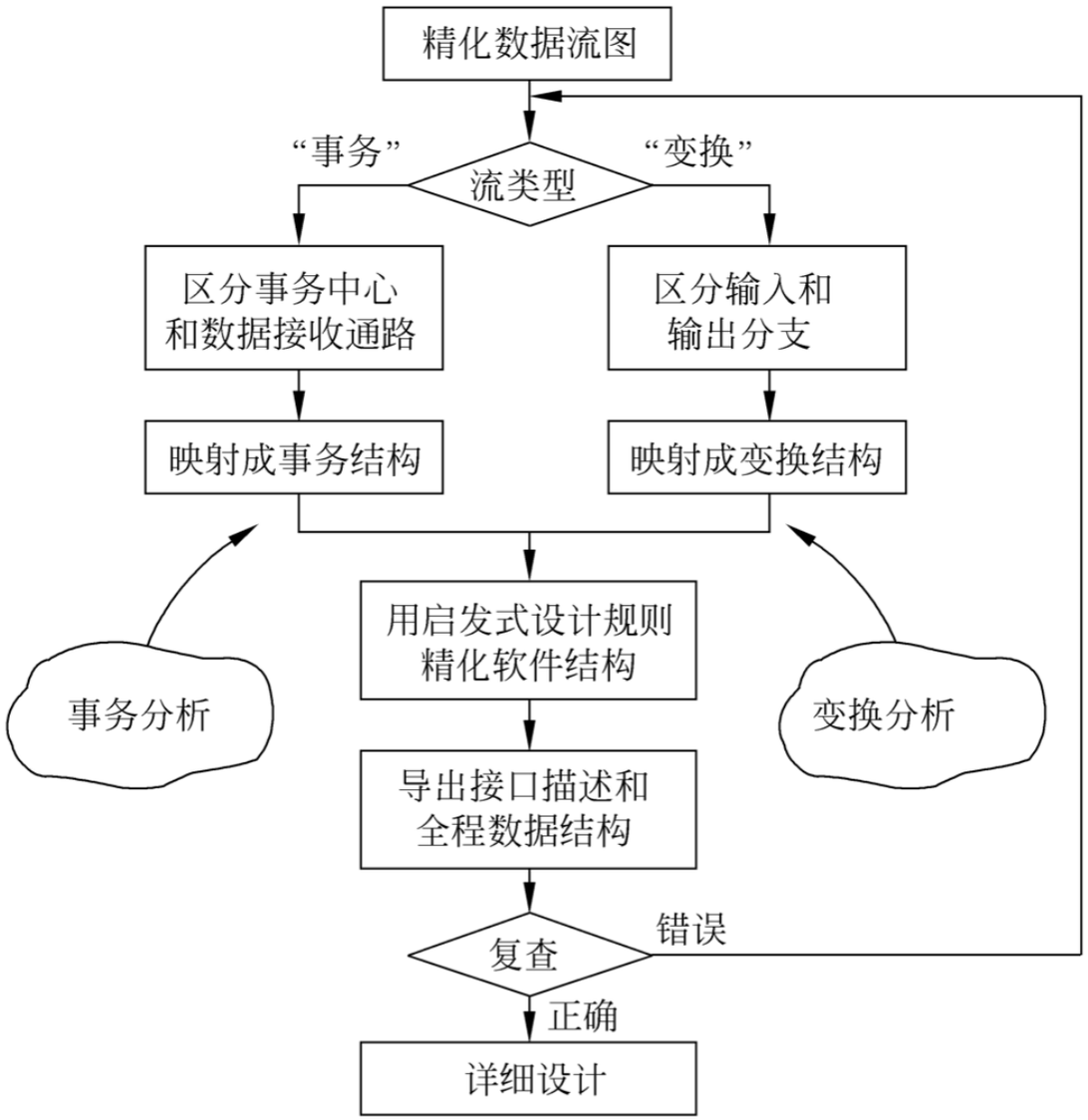




面向数据流的设计方法

- 处理T称为事务中心，它完成下述任务
  - 接收输入数据(输入数据又称为事务)
  - 分析每个事务以确定它的类型
  - 根据事务类型选取一条活动通路





面向数据流的设计方法

- 变换分析
  - 变换分析是一系列设计步骤的总称，经过这些步骤把具有变换流特点的数据流图按预先确定的模式映射成软件结构

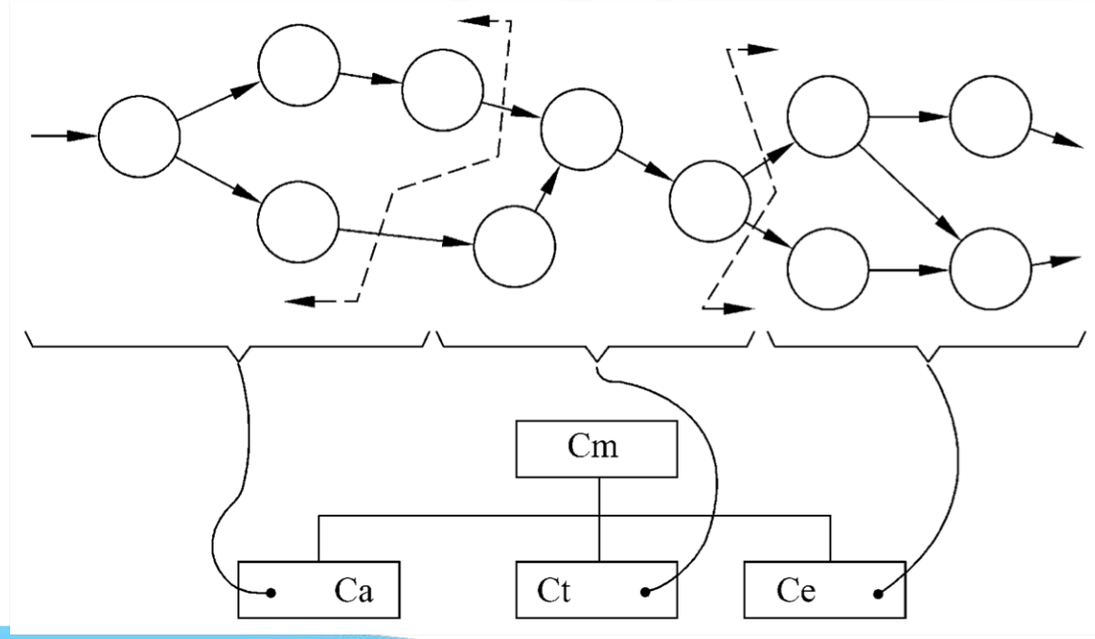
- 实例
  - 设计汽车数字仪表板，完成下述功能：
    - (1) 通过模数转换实现传感器和微处理机接口；
    - (2) 在发光二极管面板上显示数据；
    - (3) 指示每小时英里数(mph)，行驶的里程，每加仑油行驶的英里数(mpg)等等；
    - (4) 指示加速或减速；
    - (5) 超速警告：如果车速超过55英里/小时，则发出超速警告铃声。

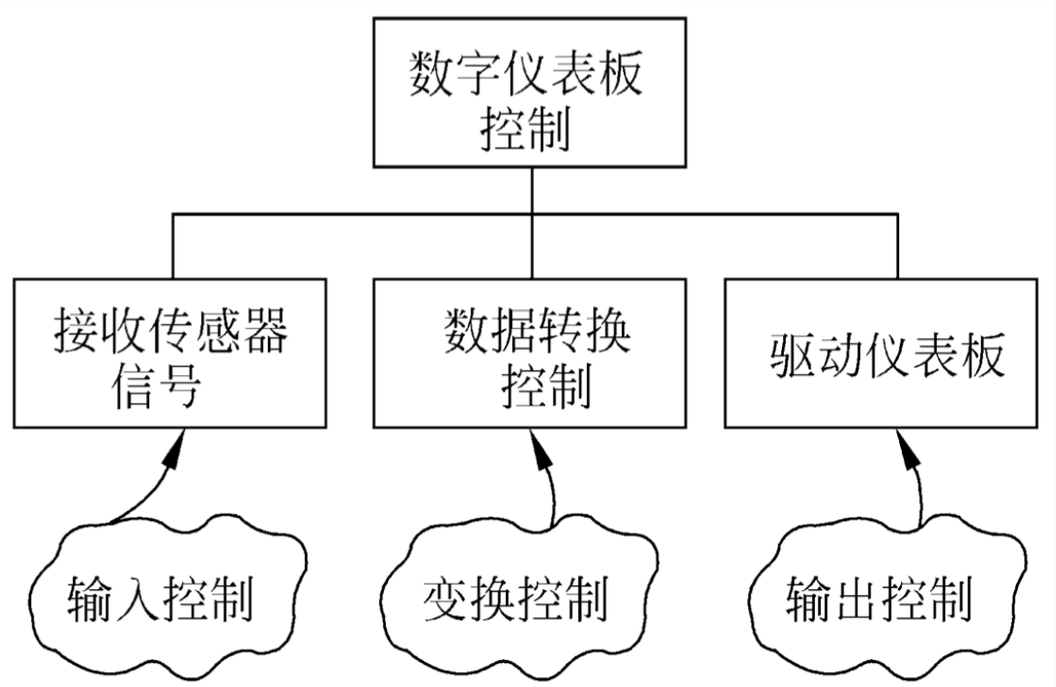
面向数据流的设计方法

- 设计步骤
  - 第1步：复查基本系统模型
    - 复查的目的是确保系统的输入数据和输出数据符合实际。
  - 第2步：复查并精化数据流图。
    - 应该对需求分析阶段得出的数据流图认真复查，并且在必要时进行精化。不仅要确保数据流图给出了目标系统的正确的逻辑模型，而且应该使数据流图中每个处理都代表一个规模适中相对独立的子功能。
  - 第3步：确定数据流图具有变换特性还是事务特性。
    - 一个系统中的所有信息流都可以认为是变换流，但是，当遇到有明显事务特性的信息流时，建议采用事务分析方法进行设计。
    - 在这一步，设计人员应该根据数据流图中占优势的属性，确定数据流的全局特性。此外还应该把具有和全局特性不同的特点的局部区域孤立出来，以后可以按照这些子数据流的特点精化根据全局特性得出的软件结构
  - 第4步：确定输入流和输出流的边界，从而孤立出变换中心。
    - 输入流和输出流的边界和对它们的解释有关，也就是说，不同设计人员可能会在流内选取稍微不同的点作为边界的位置。当然在确定边界时应该仔细认真，但是把边界沿着数据流通路移动一个处理框的距离，通常对最后的软件结构只有很小的影响

面向数据流的设计方法

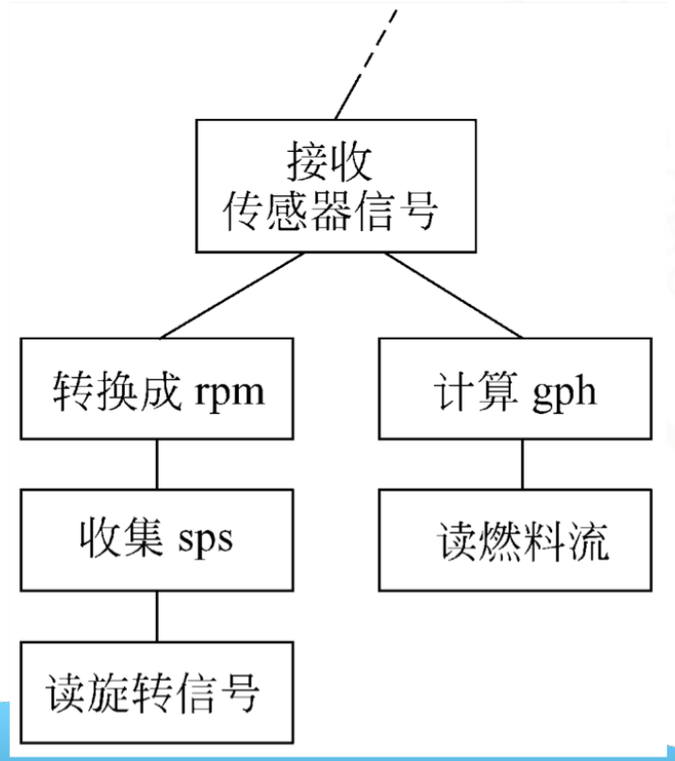
- 设计步骤
  - 第5步：基完成“第一级分解”
    - 软件结构代表对控制的自顶向下的分配，所谓分解就是分配控制的过程。
    - 对于变换流的情况，数据流图被映射成一个特殊的软件结构，这个结构控制输入、变换和输出等信息处理过程。

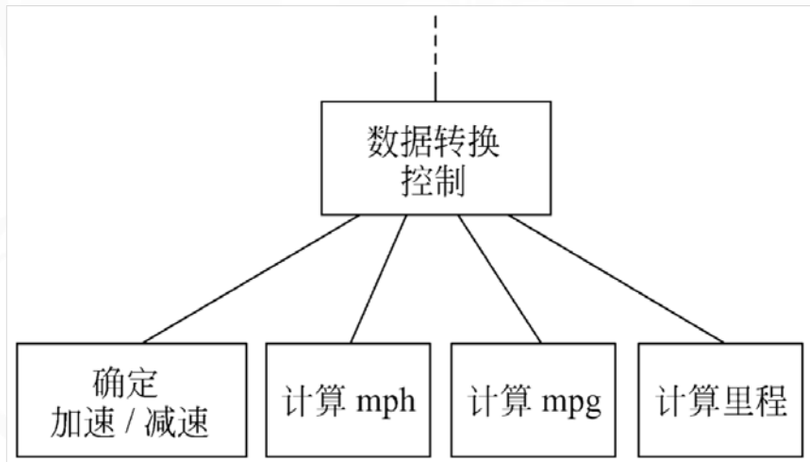


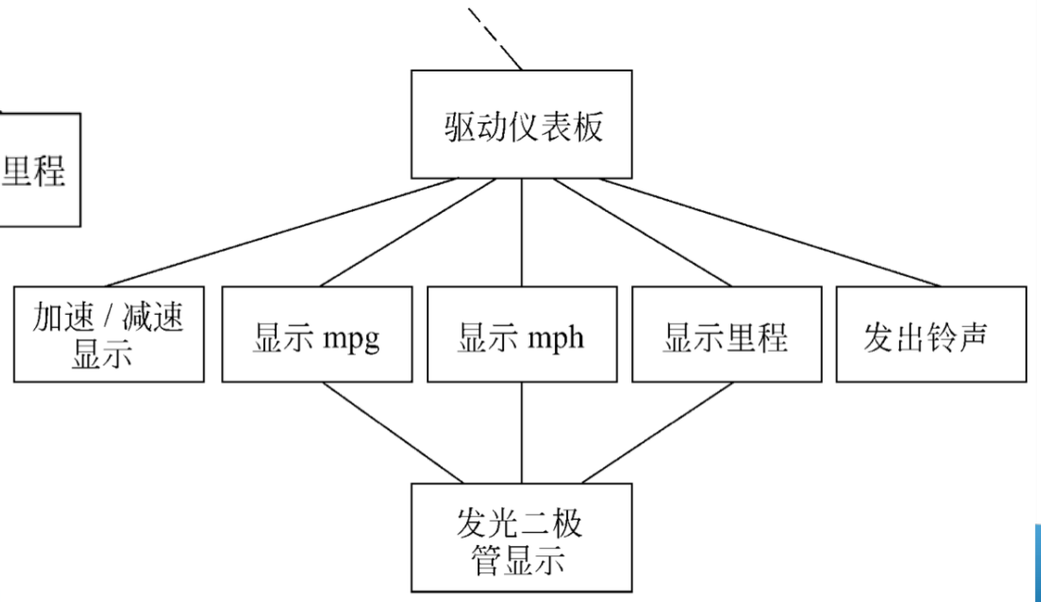


面向数据流的设计方法

- 设计步骤
  - 第6步：基完成“第二级分解”
    - 所谓第二级分解就是把数据流图中的每个处理映射成软件结构中一个适当的模块
    - 完成第二级分解的方法是，
      - 从变换中心的边界开始沿着输入通路向外移动，把输入通路中每个处理映射成软件结构中Ca控制下的一个低层模块；
      - 然后沿输出通路向外移动，把输出通路中每个处理映射成直接或间接受模块Ce控制的一个低层模块；
      - 最后把变换中心内的每个处理映射成受Ct控制的一个模块。

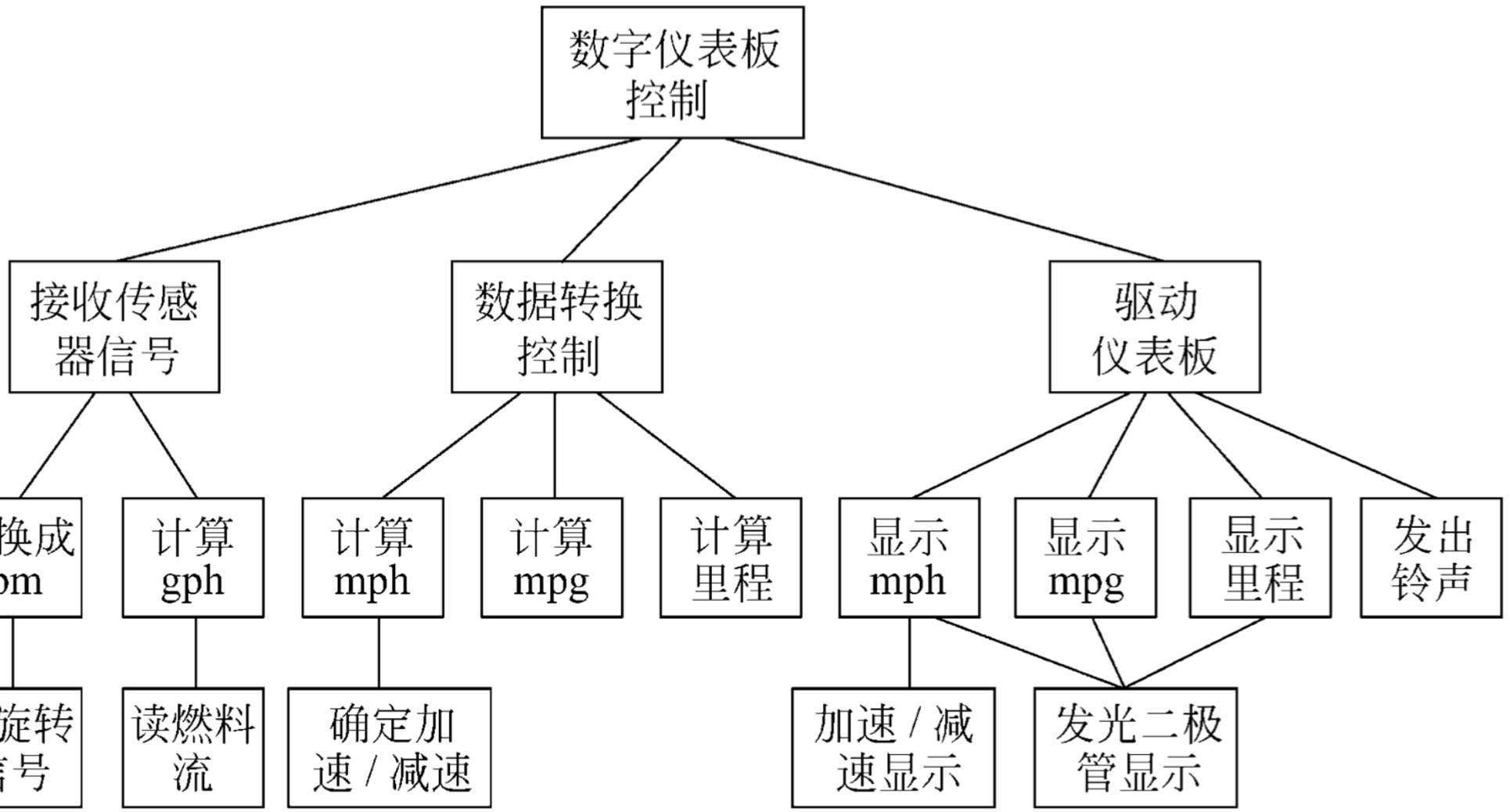






面向数据流的设计方法

- 设计步骤
  - 第7步：使用设计度量和启发式规则对第一次分割得到的软件结构进一步精化
    - 对第一次分割得到的软件结构，总可以根据模块独立原理进行精化，对于从前面的设计步骤得到的软件结构进行修改
      - 输入结构中的模块“转换成rpm”和“收集sps”可以合并
      - 模块“确定加速/减速”可以放在模块“计算mph”下面，以减少耦合
      - 模块“加速/减速显示”可以相应地放在模块“显示mph”的下面


数字仪表板控制
接收传感器信号
数据转换控制
驱动仪表板
转换成rpm
计算gph
计算mph
计算mpg
计算里程
显示mph
显示mpg
显示里程
发出铃声
读旋转信号
读燃料流
确定加速/减速
加速/减速显示
发光二极管显示

面向数据流的设计方法

- 上述7个设计步骤的目的是，开发出软件的整体表示
  - 确定了软件结构后就可以将其作为一个整体来复查，从而能够评价和精化软件结构
  - 在此时段进行修改只需要很少的附加工作，但是却能够对软件的质量特别是软件的可维护性产生深远的影响

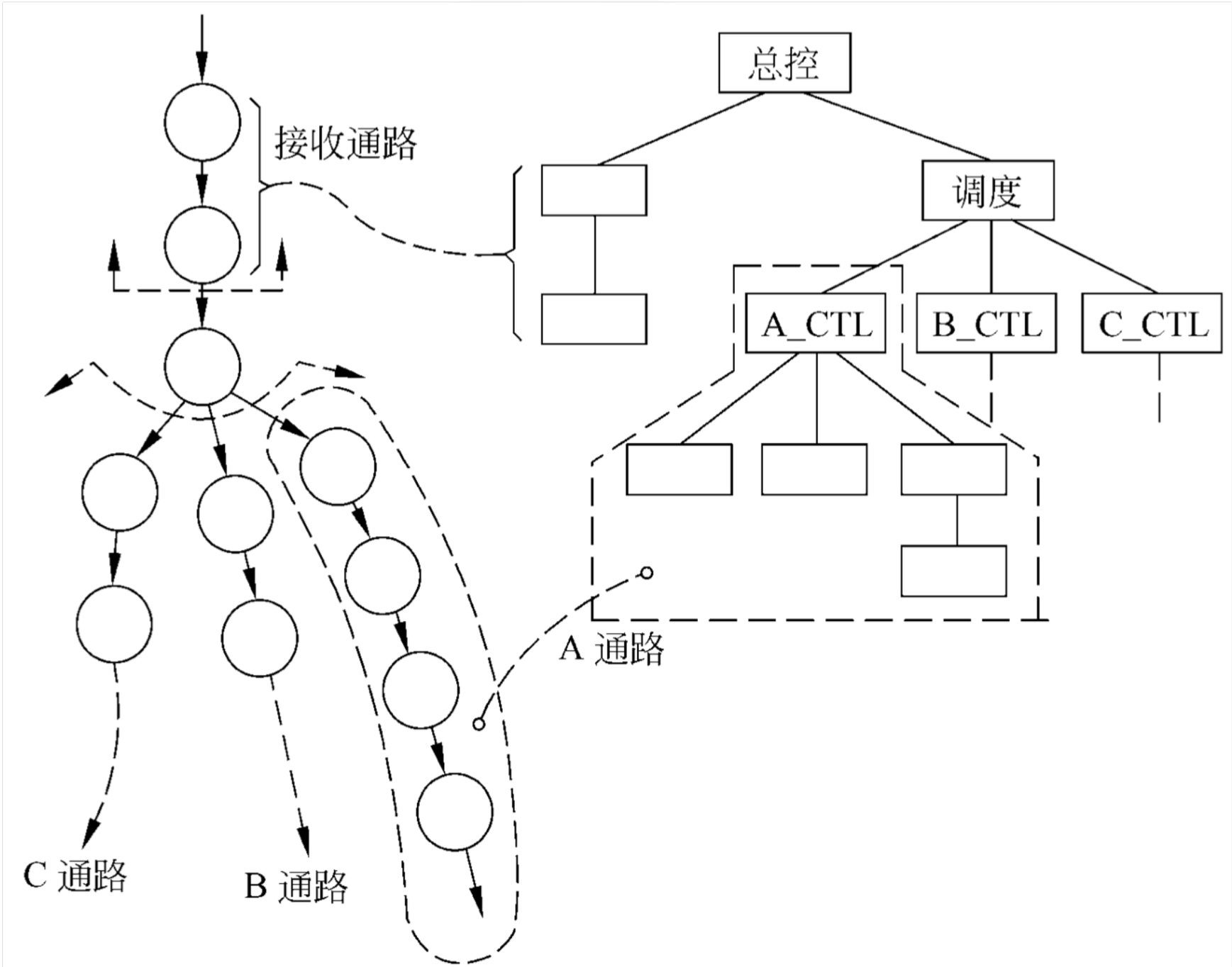
事务分析

- 当数据流具有明显的事务特点时，即有一个明显的“发射中心”(事务中心)时，可以采用事务分析方法设计软件结构

- 事务分析的设计步骤和变换分析的设计步骤大部分相同或类似
  - 主要差别仅在于由数据流图到软件结构的映射方法不同

事务分析

- 由事务流映射成的软件结构包括一个接收分支和一个发送分支
  - 接收分支结构的方法和变换分析映射出输入结构的方法很相像
    - 即从事务中心的边界开始，把沿着接收流通路的处理映射成模块
  - 发送分支的结构包含一个调度模块，它控制下层的所有活动模块；然后把数据流图中的每个活动流通路映射成与它的流特征相对应的结构
- 对于一个大系统，常常把变换分析和事务分析应用到同一个数据流图的不同部分，由此得到的子结构形成“构件”，可以利用它们构造完整的软件结构


总控
调度
A_CTL
B_CTL
C_CTL
接收通路
A 通路
B 通路
C 通路

事务分析

- 一般说来，如果数据流不具有显著的事务特点，最好使用变换分析
- 反之，如果具有明显的事务中心，则应该采用事务分析技术
- 使用过程中，机械地遵循变换分析或事务分析的映射规则，很可能会得到一些不必要的控制模块
  - 如果它们确实用处不大，那么可以而且应该把它们合并
  - 如果一个控制模块功能过分复杂，则应该分解为两个或多个控制模块，或者增加中间层次的控制模块

设计优化

- “一个不能工作的’最佳设计’的价值是值得怀疑的
  - 应该致力于开发能够满足所有功能和性能要求，而且按照设计原理和启发式设计规则衡量是值得接收的软件
- 应该在设计的早期阶段尽量对软件结构进行精化
  - 可以导出不同的软件结构，然后对它们进行评价和比较，力求得到“最好”的结果
  - 这种优化的可能，是把软件结构设计和过程设计分开的真正优点之一

设计优化

- 用下述方法对时间起决定性作用的软件进行优化是合理的
  - 在不考虑时间因素的前提下开发并精化软件结构；
  - 在详细设计阶段选出最耗费时间的那些模块，仔细地设计它们的处理过程(算法)，以求提高效率；
  - 使用高级程序设计语言编写程序；
  - 在软件中孤立出那些大量占用处理机资源的模块；
  - 必要时重新设计或用依赖于机器的语言重写上述大量占用资源的模块的代码，以求提高效率。
  - 上述优化方法遵守了一句格言：“先使它能工作，然后再使它快起来。”