软件工程建模

Outline

- 数学建模
- 计算机建模
- 软件工程建模
- 软件开发生命周期模型
- 软件工程建模分析案例

Outline

- 数学建模
- 计算机建模
- 软件工程建模
- 软件开发生命周期模型
- 软件工程建模分析案例

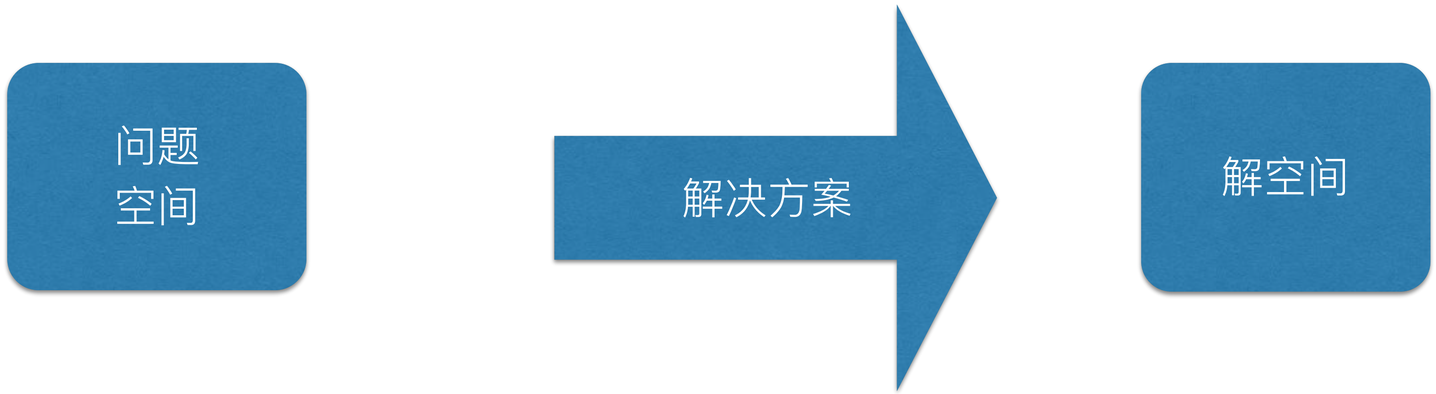
所有软件工程师都在思考的问题

- What to develop?
- Why to develop?
- How to develop?

思维的演化

数学—》计算机—》软件工程

问题求解



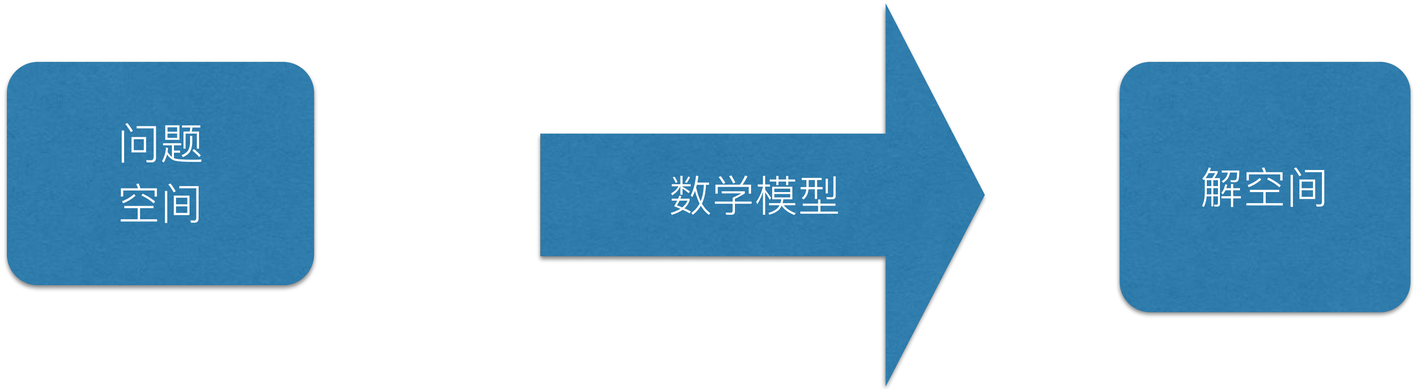
求1到100这100个数的和等于多少?

?

5050

数学的思维

数学的思维



求1+2+3+...+99+100等于多少?

?

5050

第零步 - 明确自己的武器

- 数学框架
  - 整数
    - 加法
    - 乘法

第一步 - 审题

- 提炼其中的数学问题
  - results = 1+2+3+...+99+100

第二步 - 建立数学模型

- 对50对构造成和为101的数列求和（1＋100，2＋99，3＋98……）

第三步 - 制订解决方案

- result = 50 * 101;

第四步 - 检查

- Verification - 检查解决方案的有效性
  - whether do it right?

第五步 - 实施

- 计算乘法
  - result = 5050

Outline

- 数学建模
- 计算机建模
- 软件工程建模
- 软件开发生命周期模型
- 软件工程建模分析案例

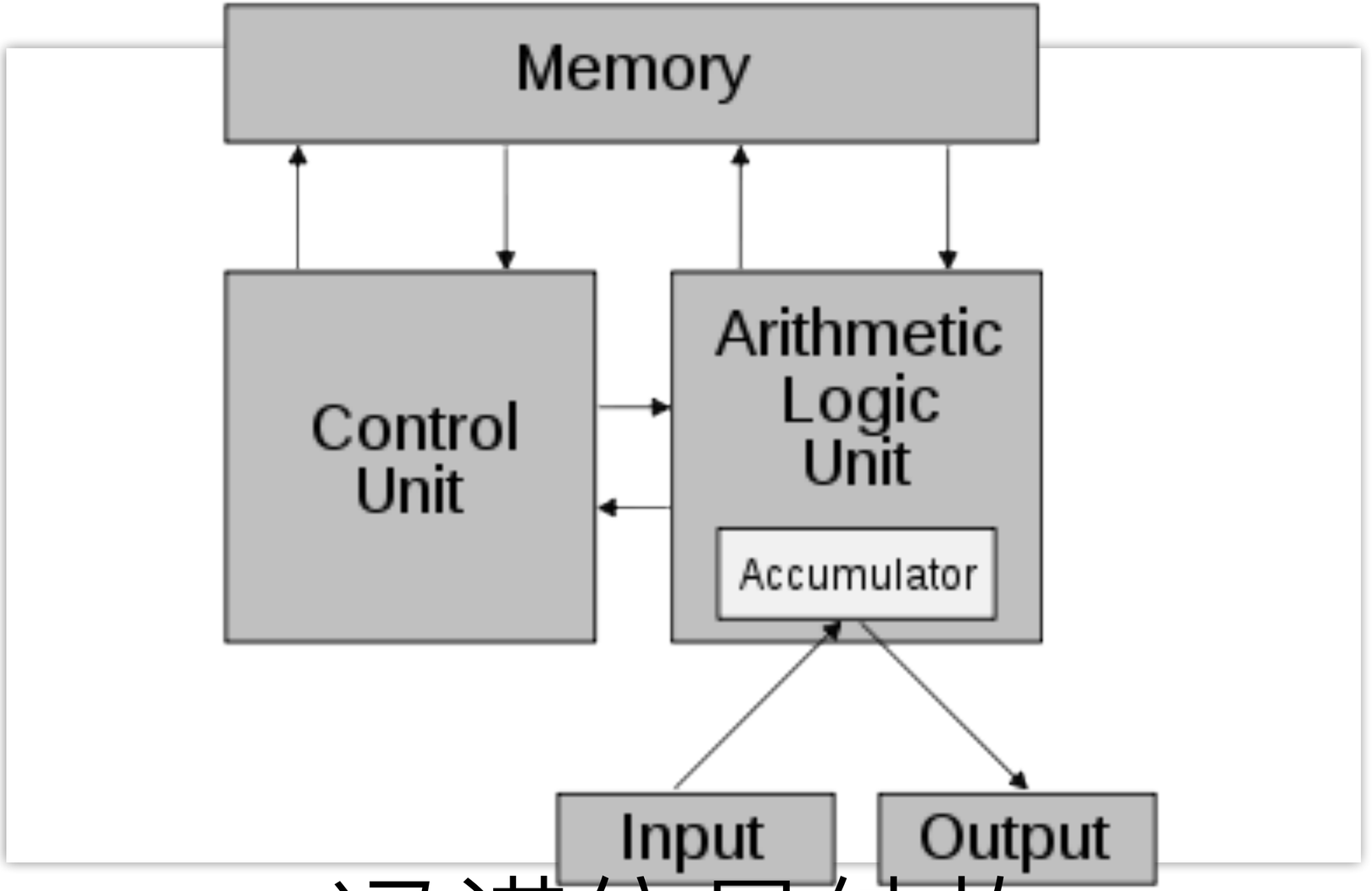
计算机的思维

计算机与数学的关系

- 数学为计算机提供了理论基础
  - 现实世界的问题先转换为一个数学问题
  - 然后再用计算机解决这个数学问题
- 计算机有特有的软件和硬件实现
  - 软件框架（编程范式：命令式、函数式）
  - 硬件框架（硬件结构：冯诺依曼结构、哈佛结构）

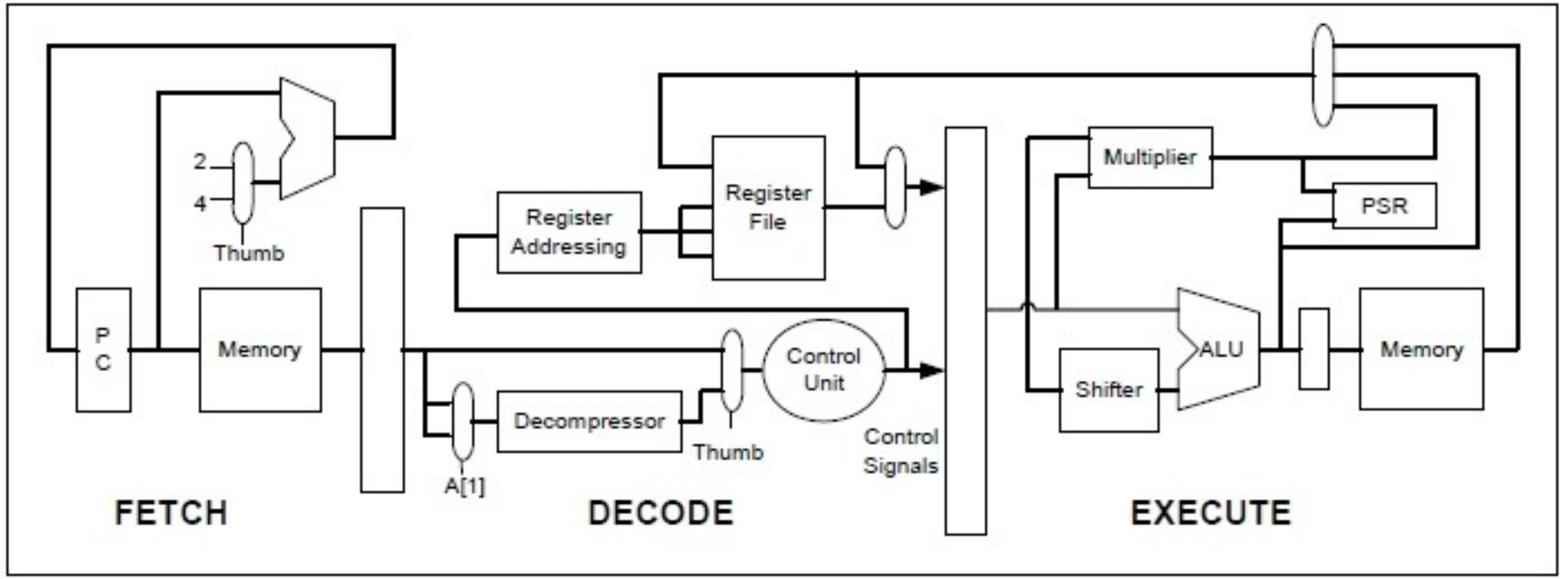
软件框架

- 编程范式
  - 命令式
  - 函数式
  - 逻辑式
- 层次性
  - 机器指令
  - 汇编指令
  - 高级语言


Memory
Control Unit
Arithmetic Logic Unit
Accumulator
Input
Output

冯诺依曼结构

ARM7TM Core and Pipeline

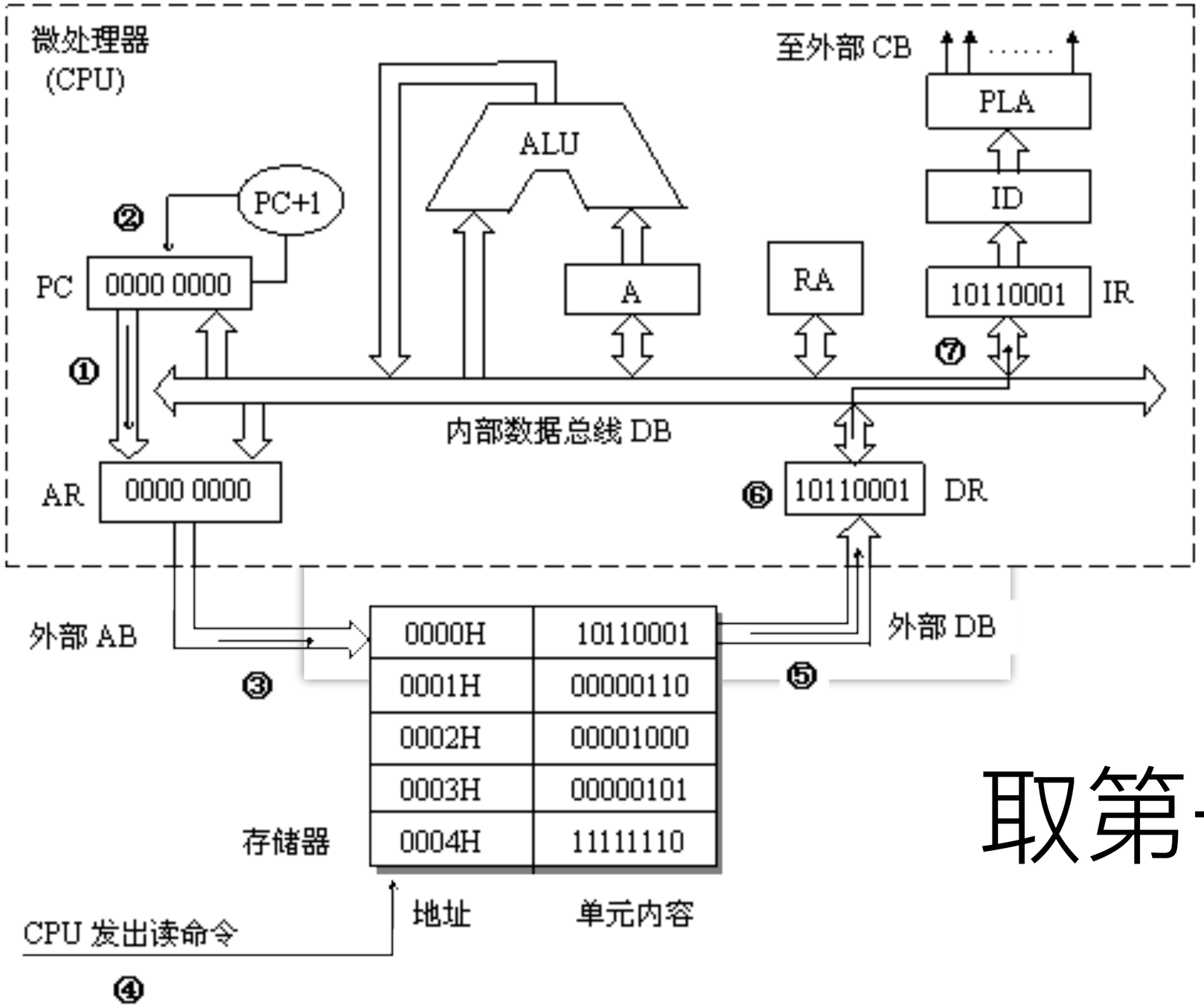


指令执行过程

取指、译指、执行

计算6+5

内存单元地址	机器语言程序	汇编语言程序	指令功能说明
0000H 0001H	10110001 00000110	MOV? A,06H	双字节指令。将数字6送累加器A
0002H 0003H	00001000 00000101	ADD? A,05H	双字节指令。将数字5与累加器A中的内容相加，结果存放在累加
0004H	11111110	HLT	停机指令


微处理器
(CPU)
至外部 CB
PLA
ID
ALU
PC+1
②
PC
0000 0000
A
RA
10110001
IR
①
⑦
内部数据总线 DB
AR
0000 0000
⑥
10110001
DR
外部 AB
外部 DB
③
⑤
0000H
10110001
0001H
00000110
0002H
00001000
0003H
00000101
0004H
11111110
存储器
地址
单元内容
CPU 发出读命令
④

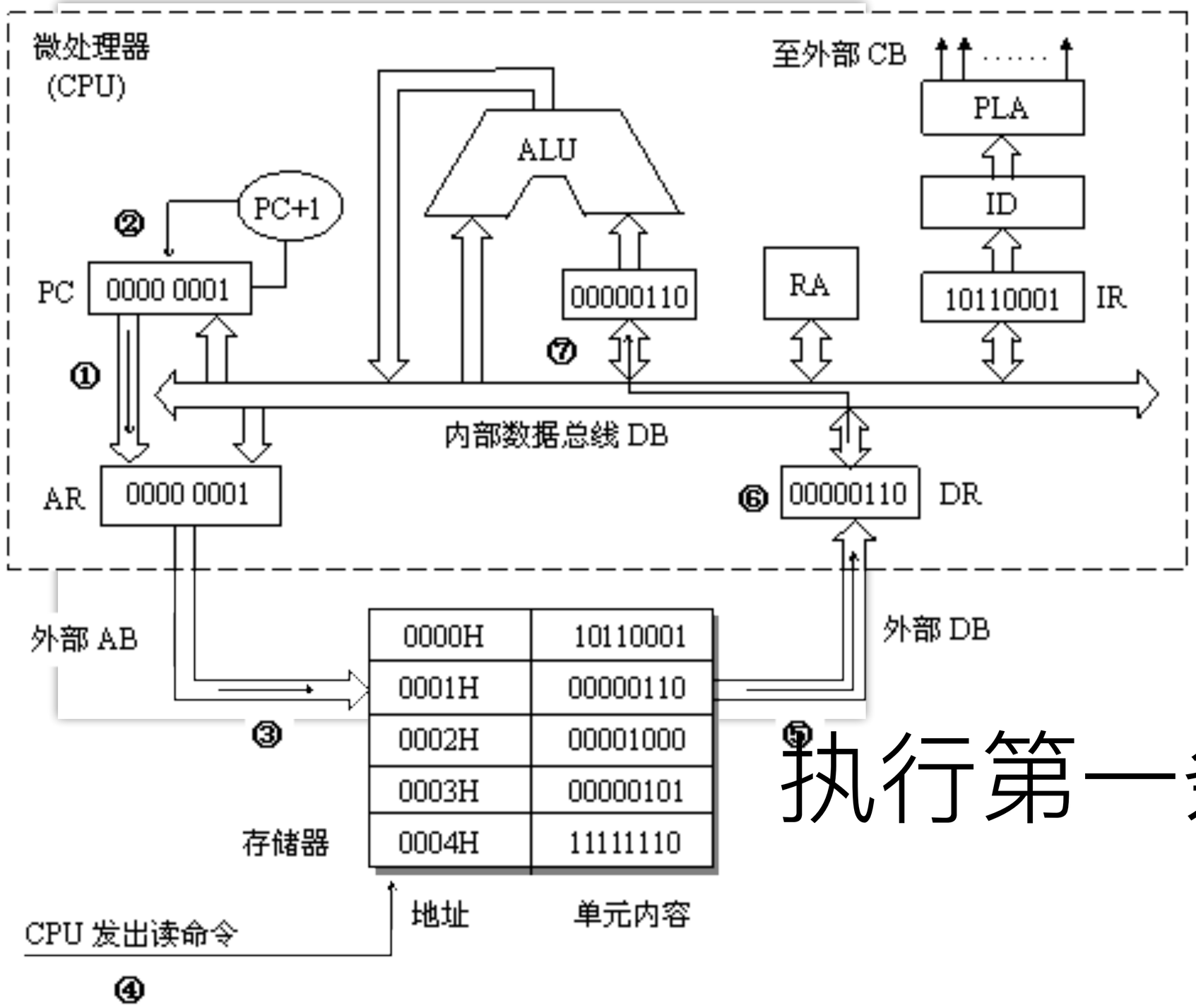
取第一条指令

取指

- ① 把PC内容送地址寄存器AR.
- ② PC内容送入AR后，PC自动加1，即由0000H变为0001H，以使PC指向下一个要读取的内存单元。注意，此时AR的内容并没有变化。
- ③ 把地址寄存器AR的内容0000H放在地址总线上，并送至存储器系统的地址译码电路（图中未画出），经地址译码选中相应的0000H单元。
- ④ CPU发出存储器读命令。
- ⑤ 在读命令的控制下，把选中的0000H单元的内容即第一条指令的操作码B1H读到数据总线DB上。
- ⑥ 把读出的内容B1H经数据总线送到数据缓冲寄存器DR。

译指

- ⑦ 指令译码。因为取出的是指令的操作码，故数据缓冲寄存器DR中的内容被送到指令寄存器IR，然后再送到指令译码器ID，经过译码，CPU“识别”出这个操作码代表的指令，于是经控制器发出执行该指令所需要的各种控制命令。

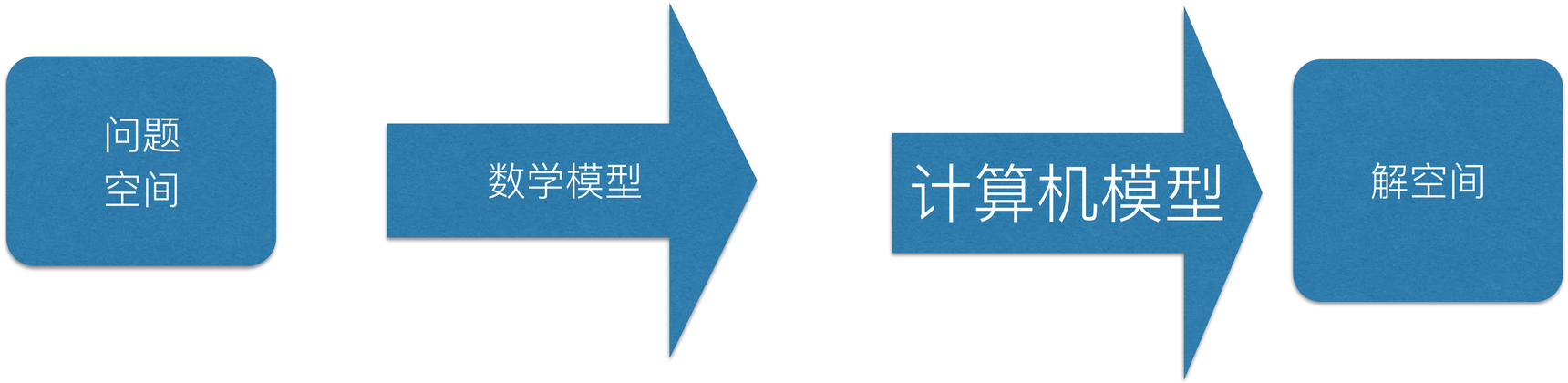

微处理器
(CPU)
至外部 CB
PLA
ID
10110001
IR
ALU
PC+1
②
PC
0000 0001
00000110
RA
⑦
①
内部数据总线 DB
AR
0000 0001
⑥
00000110
DR
外部 AB
外部 DB
0000H
10110001
0001H
00000110
0002H
00001000
0003H
00000101
0004H
11111110
③
⑤
存储器
地址
单元内容
CPU 发出读命令
④

执行第一条指令

执行

- ① 把PC内容01H送地址寄存器AR。
- ② PC内容送入AR后，PC自动加1，即由0001H变为0002H。注意，此时AR的内容0001H并没有变化。
- ③ 把地址寄存器AR的内容0001H放到地址总线上，并送至存储器系统的地址译码电路，经地址译码选中相应的0001H单元。
- ④ CPU发出存储器读命令。
- ⑤ 在读命令的控制下，把选中的0001H单元的内容06H放到数据总线DB上。
- ⑥ 把读出的内容06H经数据总线送到数据缓冲寄存器DR。
- ⑦ 数据缓冲寄存器DR的内容经内部数据总线送到累加器A。于是，第一条指令执行完毕，操作数06H被送到累加器A中。

计算机的思维



求1+2+3+...+99+100等于多少?

?

5050

第零步 - 明确自己的武器

- 计算机框架
  - 软件框架
    - 命令式编程语言
    - 汇编语言编译器
  - 硬件框架
    - 执行机器语言的计算机
    - 寄存器
    - ALU

第一步 - 审题

- 提炼其中的数学问题
  - results = 1+2+3+...+99+100

第二步 - 建立计算机模型
Imperative Programming Paradigm

- 循环100次
  - 一个寄存器保存新加的数
  - 一个寄存器保持累加和
  - 计算加法
  - 将累加和送入内存指定单元

第三步 - 制订解决方案

```
.model small
.stack
.data
sum dw ?
.code
.startup
xor ax, ax ;被加数AX清0
mov cx,100
again: add ax,cx ;从100,99,...,2,1倒序累加
loop again
mov sum,ax ;将累加和送入指定单元
.exit 0
end
```

第四步 - 检验

- Verification - 检查解决方案的有效性
  - whether do it right?

第五步 - 实施

- 将汇编程序编译成机器码
- 在计算机中执行机器码
- result = 5050

为什么用数学能解决的还要用计算机做?

因为计算机

- 可以节省人力
- 计算的快速
- 存储的海量

不同的软件框架，解决问题是不一样的

第二步 - 建立计算机模型
Functional Programming Paradigm

- Sum(n) = n+Sum(n-1)
- Sum(1) = 1

第三步 - 制订解决方案

- (define (sum n)
  - (if (= n 1)
    - 1
    - (+ n (sum (- n 1)))))

第三步 - 制订解决方案

- (sum 6)
- (+ 6 (sum 5))
- (+ 6 (+ 5 (um 4)))
- (+ 6 (+ 5 (+ 4 (sum 3))))
- (+ 6 (+ 5 (+ 4 (+ 3 (sum 2)))))
- (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (sum 1))))))
- (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 1)))))
- (+ 6 (+ 5 (+ 4 (+ 3 3))))
- (+ 6 (+ 5 (+ 4 6)))
- (+ 6 (+ 5 10))
- (+ 6 15)
- 21

不同的硬件框架，解决问题是不一样的

SSE技术

- 1999年，Intel在其Pentium III微处理器中集成了SSE（Streaming SIMD Extensions）技术，有效增强了CPU浮点运算的能力。

- SSE兼容MMX指令，可以通过SIMD和单时钟周期并行处理多个浮点数据来有效提高浮点运算速度，对图像处理、浮点运算、3D运算、视频处理、音频处理等诸多多媒体应用起到全面强化作用。

- 具有SSE指令集支持的处理器有8个128位的寄存器，每一个寄存器可以存放4个单精度（32位）浮点数。SSE同时提供了一个指令集，其中的指令允许把浮点数加载到这些128位寄存器中，这些数就可以在这些寄存器中进行算术逻辑运算，然后把结果送回主存。也就是说，SSE中的所有计算都可以针对4个浮点数一次性完成，这种批处理带来了效率的提升。

解决这样一个问题

- 计算一个很长的浮点型数组中每一个元素的平方根

基于以前框架的解决方案

- for each f in array
- {
  - 把f从主存加载到浮点寄存器
  - 计算平方根
  - 再把计算结果从寄存器中取出写入主存
- }

基于SSE技术的解决方案

- for each 4 members in array //对数组中的每4个元素
- {
  - 把数组中的这4个数加载到一个128位的SSE寄存器中
  - 在一个CPU指令执行周期中完成计算这4个数的平方根的操作
  - 把所得的4个结果取出写入主存
- }

Outline

- 数学建模
- 计算机建模
- 软件工程建模
- 软件开发生命周期模型
- 软件工程建模分析案例

真实世界就是
这么简单?

我们需要一个帮我们计算1+2+3...+100的软件?

到底我们的需求是什么?

我们想要一个好用的计算器软件。

为什么需要这样一个软件?

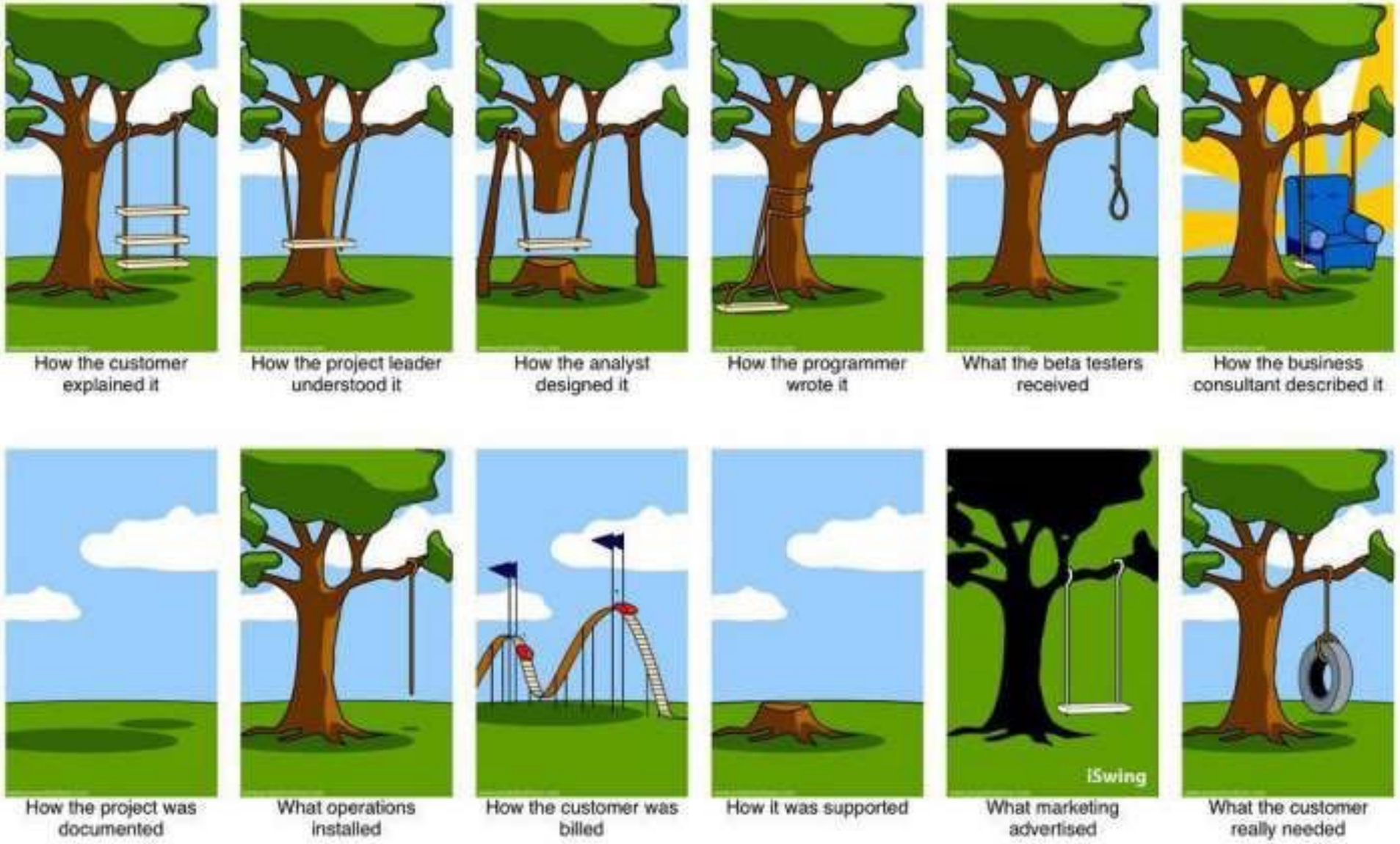
到底什么是一个好用的计算器?

怎么实现这个好的计算器?

怎么知道实现的这个计算器软件是我们需要的那个？怎么知道他是好用的？

现在的软件已经复杂到不是一个人能够开发出来的。

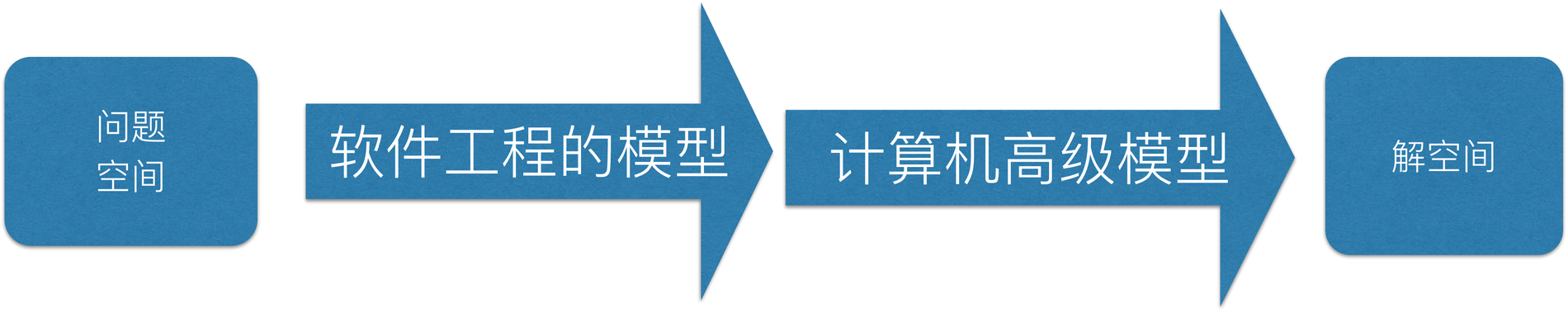
所以，如果要求在一定的时间，一定预算内，领导特定一群人来求解空间？怎么办？



What the customer wanted

这时候你需要软件工程！

软件工程的思维



?

满足真实的需求

?

利用过程、方法和工具管理
时间、金钱、人

第零步 - 明确自己的武器

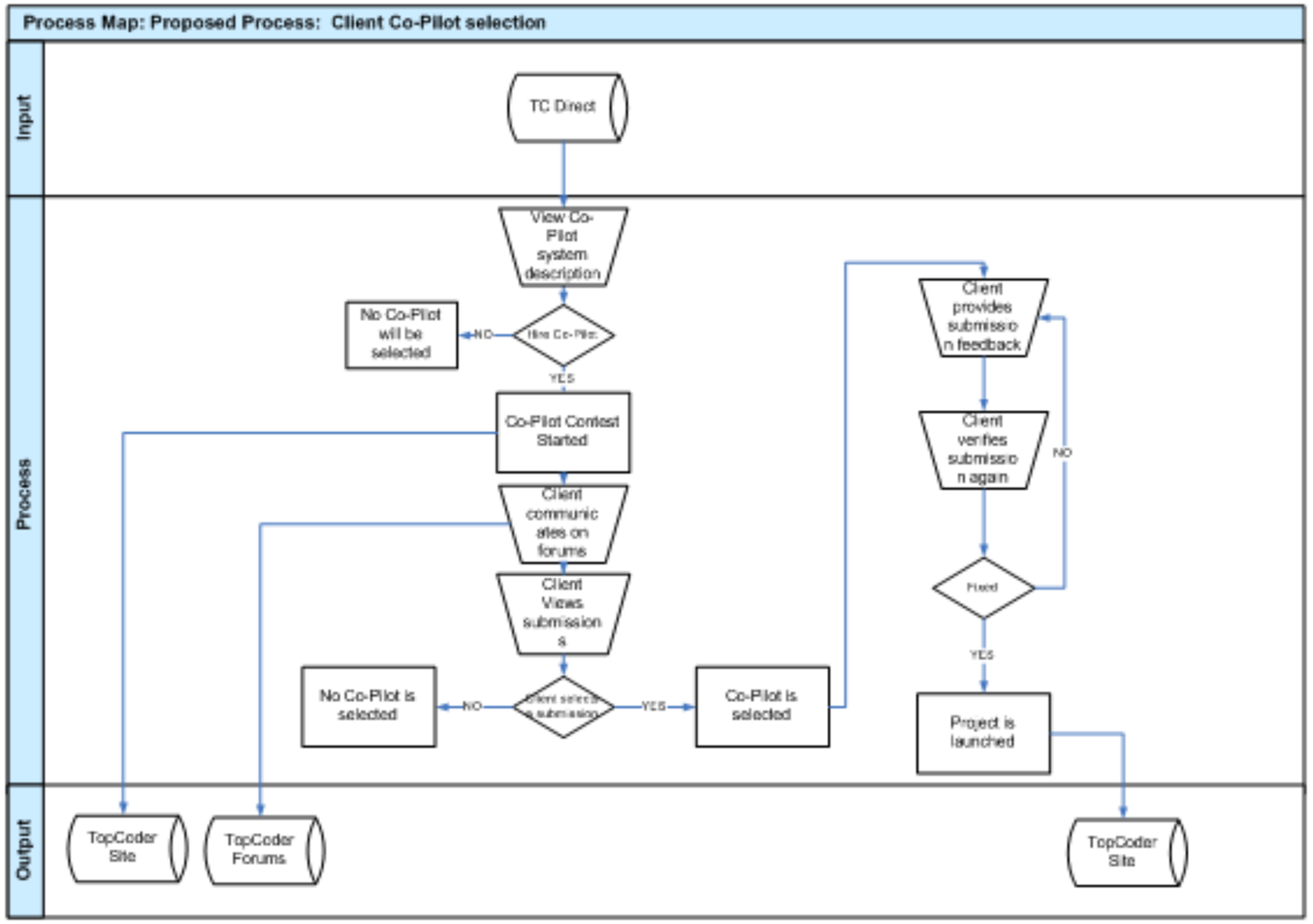
- 软件工程框架
  - 技术
    - 业务模型
    - 分析模型
    - 设计模型
  - 过程
    - 软件开发过程
- 计算机模型的进步
  - 高级语言编译器
  - 高级的模型

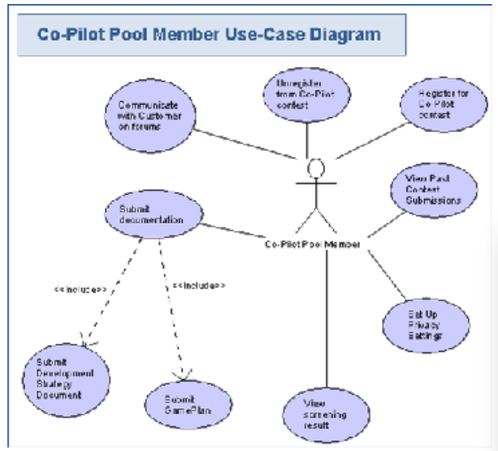
第一步 - 需求分析（审题）

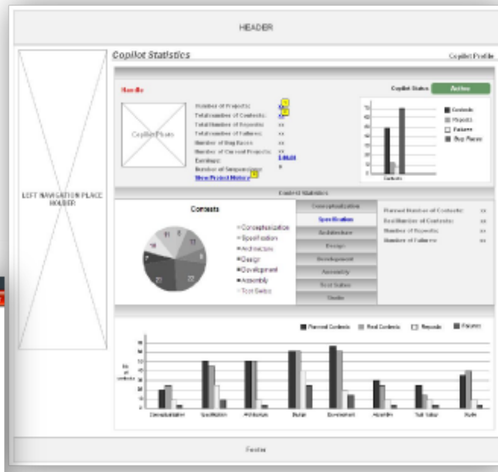
- 做什么？

分析模型 - 业务描述

- 系统管理员
  - 身份：所有系统管理员采用相同的身份和权限。
  - 操作：
    - 管理系统管理员：可以查询、添加、修改和删除系统管理员信息。
    - 管理借阅人：可以查询、添加、修改和删除借阅人信息。
    - 管理图书：可以查询、添加、修改和删除图书信息。
- 借阅人
  - 身份：分为本科生、研究生和教师三种身份，各种身份具有不同的权限。
  - 操作：
    - 查询图书：根据图书基本信息对图书进行查询。
    - 借阅图书：对选定的图书进行借阅。其中，本科生只可借阅普通图书，最多可同时借阅5本；研究生可以借阅普通图书和珍本图书，最多可同时借阅10本；教师可以借阅普通图书和珍本图书，最多可同时借阅20本。
    - 请求图书：当教师希望借阅的某种图书被借空时，可以请求图书，系统将自动通知借阅该书时间最长的本科生或研究生在7天内归还图书。
    - 查看已借图书：查看本人当前借阅图书的情况。
    - 续借图书：图书每次借阅时间为30天，本科生和研究生可以续借1次，教师可以续借2次。超期的图书和被教师请求的图书不得续借。
    - 归还图书：归还本人借阅的图书。
    - 查看消息：查看图书到期提醒、提前还书通知（本科生或研究生所借图书被请求归还时）、请求图书到馆通知（教师所请求的图书归还时）等。





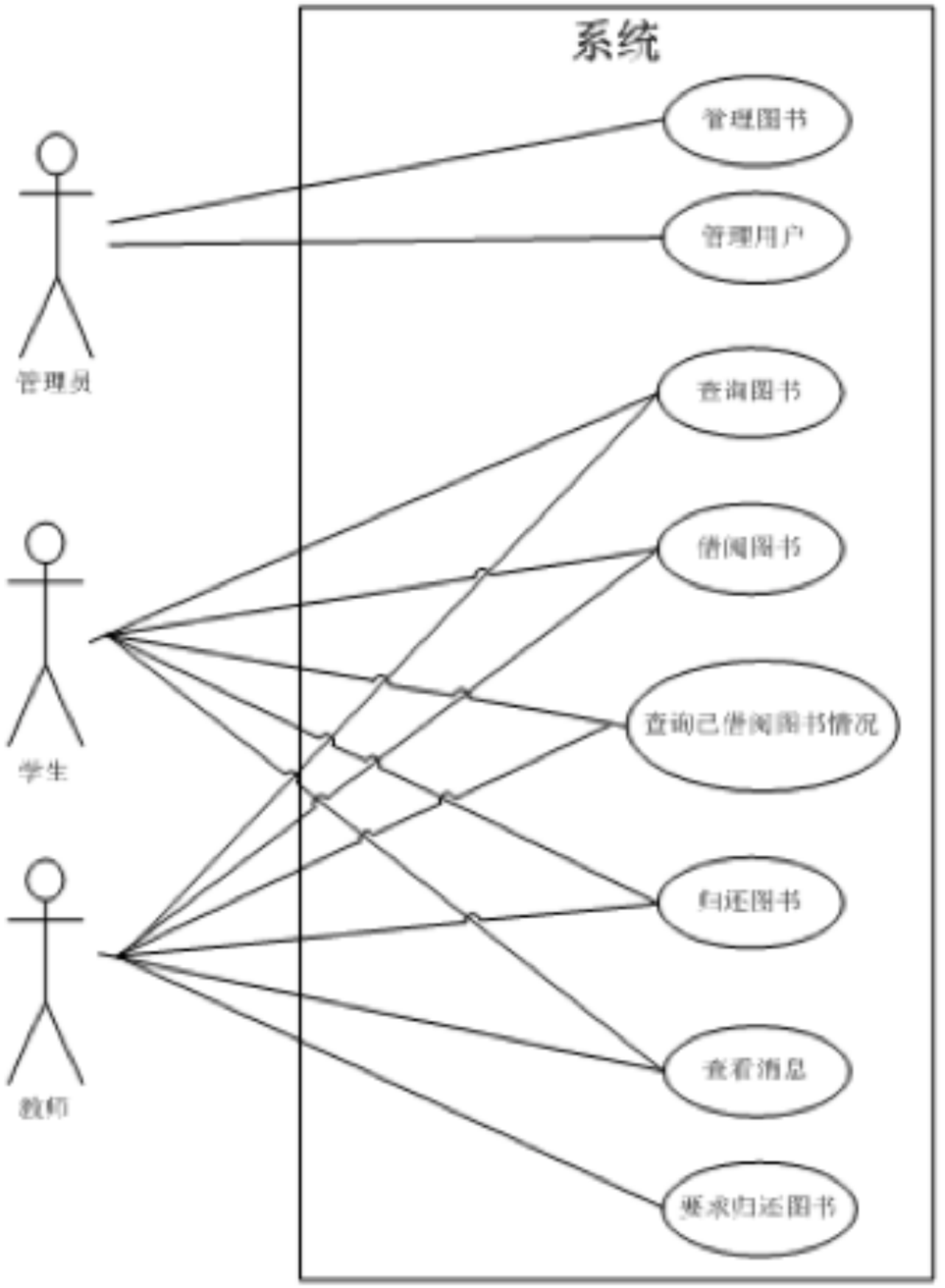


业务描述案例

流程、用例、数据、Storyboard

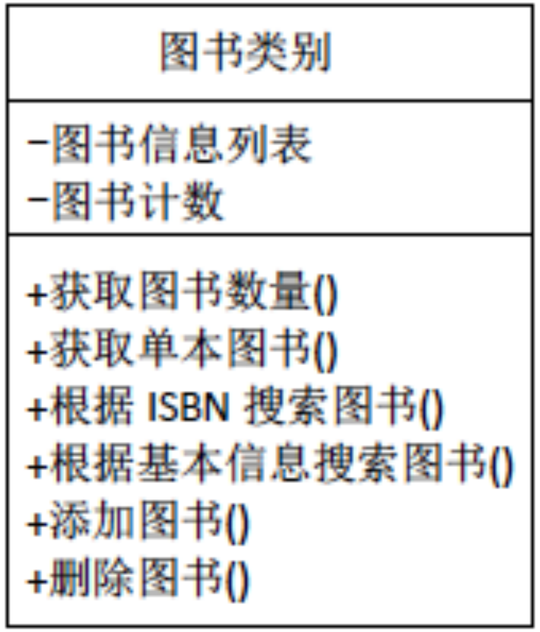
分析模型

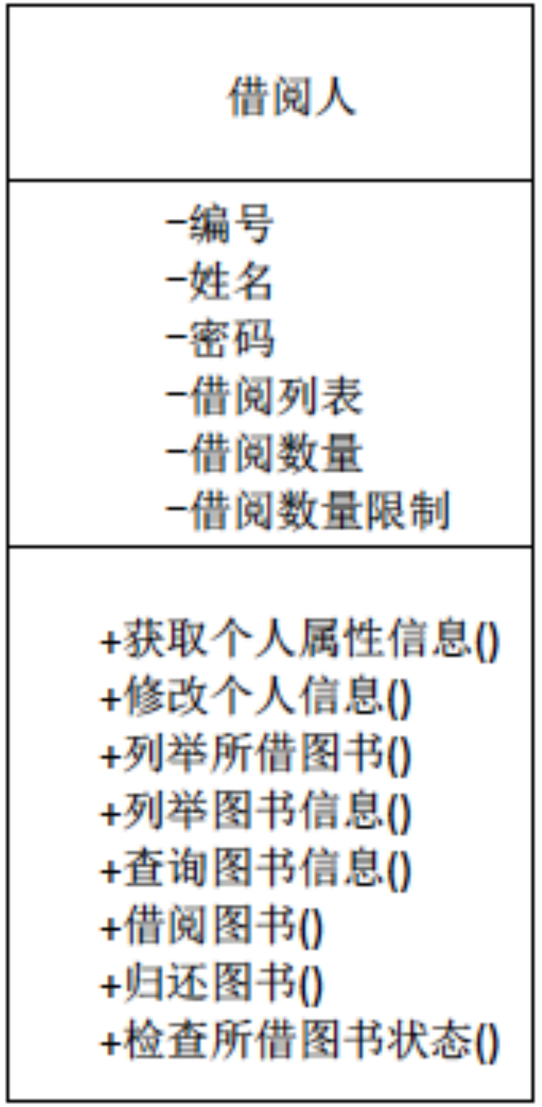
用例图



分析模型

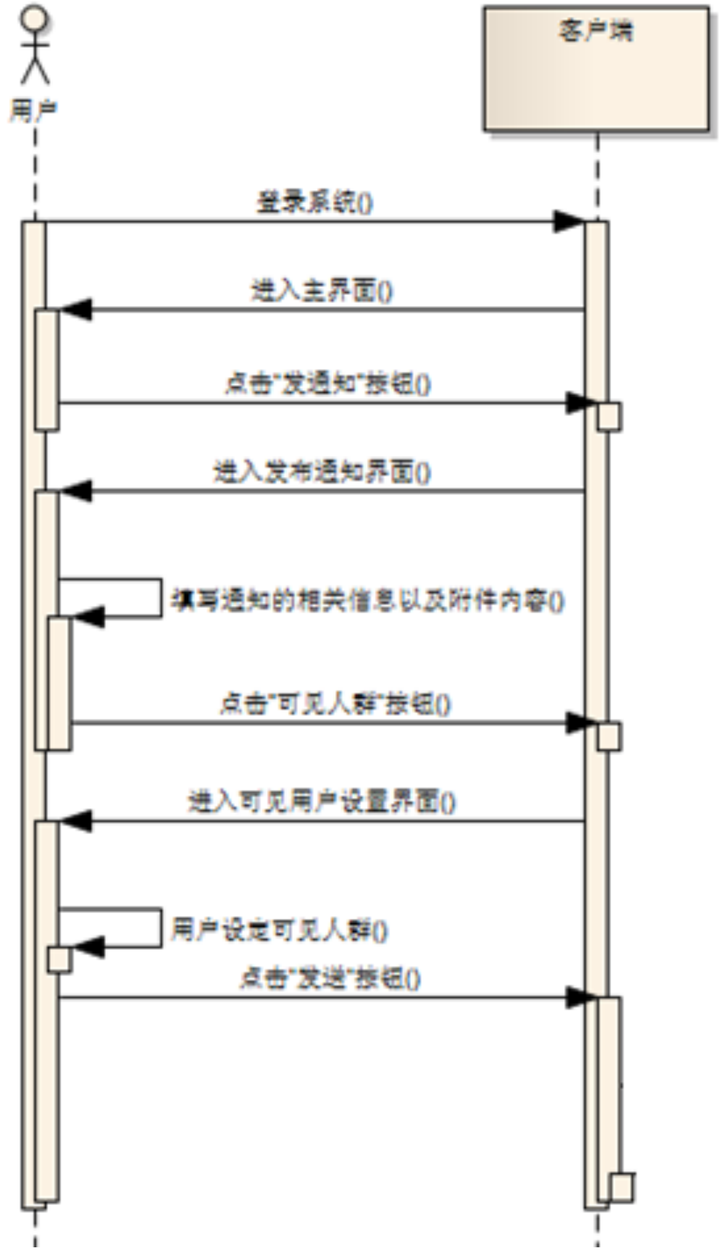
分析类图

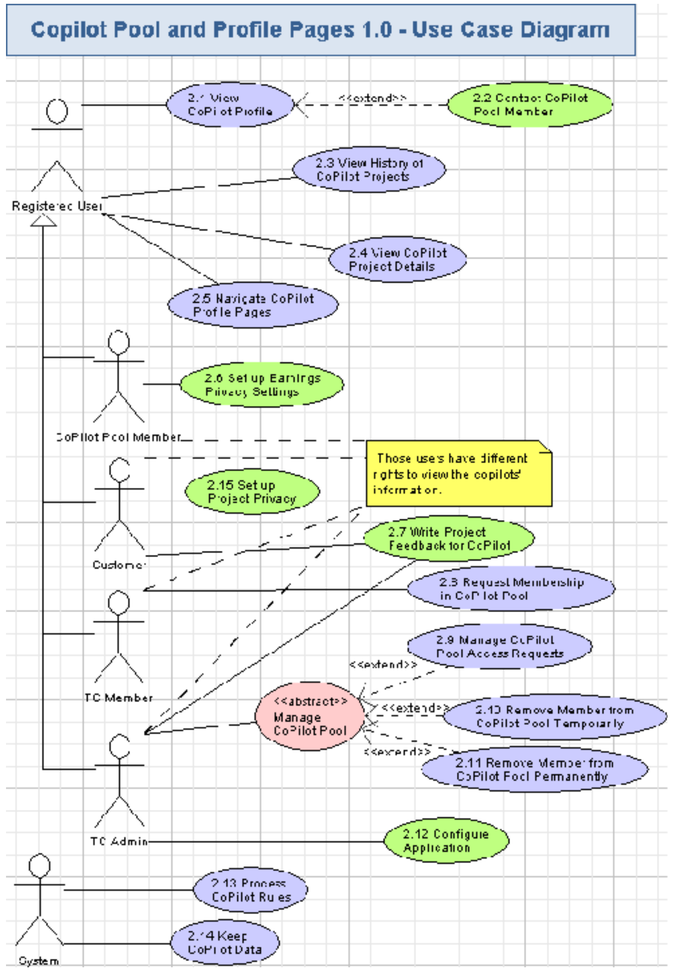

图书类别
-图书信息列表
-图书计数
+获取图书数量()
+获取单本图书()
+根据 ISBN 搜索图书()
+根据基本信息搜索图书()
+添加图书()
+删除图书()

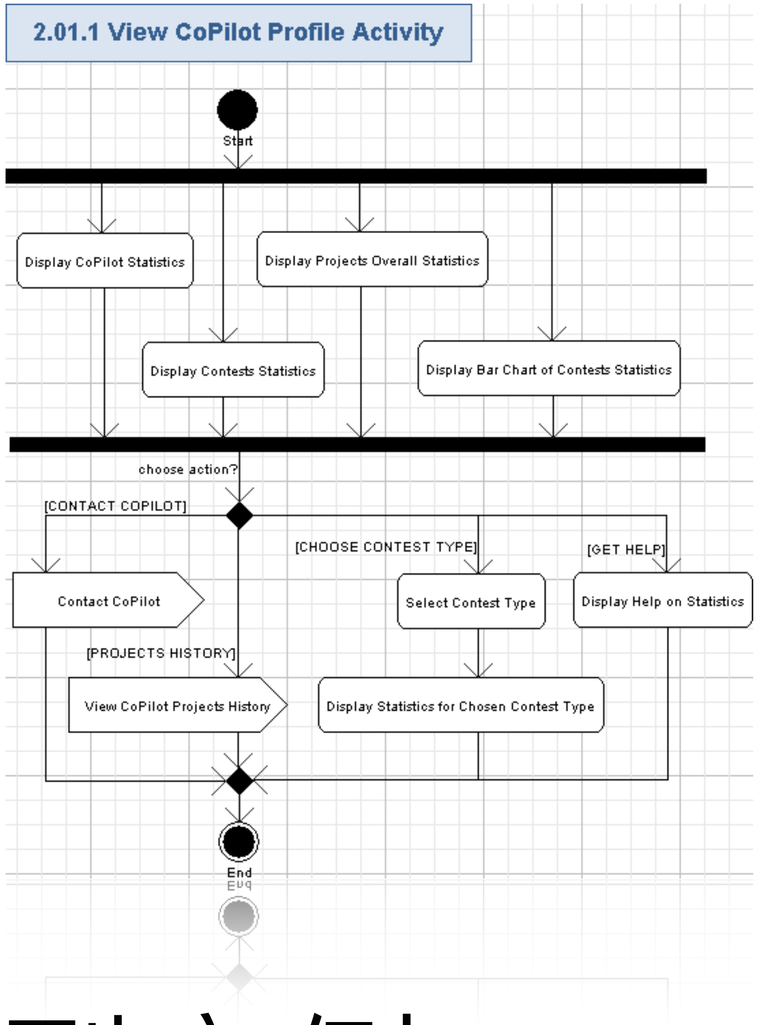

借阅人
-编号
-姓名
-密码
-借阅列表
-借阅数量
-借阅数量限制
+获取个人属性信息()
+修改个人信息()
+列举所借图书()
+列举图书信息()
+查询图书信息()
+借阅图书()
+归还图书()
+检查所借图书状态()

分析模型

系统顺序图


用户
客户端
登录系统()
进入主界面()
点击"发通知"按钮()
进入发布通知界面()
填写通知的相关信息以及附件内容()
点击"可见人群"按钮()
进入可见用户设置界面()
用户设定可见人群()
点击"发送"按钮()


Copilot Pool and Profile Pages 1.0 - Use Case Diagram
2.1 View CoPilot Profile
<<extend>>
2.2 Contact CoPilot Pool Member
2.3 View History of CoPilot Projects
Registered User
2.4 View CoPilot Project Details
2.5 Navigate CoPilot Profile Pages
2.6 Set up Earnings Privacy Settings
CoPilot Pool Member
Those users have different rights to view the copilots' information.
2.15 Set up Project Privacy
2.7 Write Project Feedback for CoPilot
Customer
2.8 Request Membership in CoPilot Pool
2.9 Manage CoPilot Pool Access Requests
TC Member
<<extend>>
<<abstract>>
Manage CoPilot Pool
<<extend>>
2.10 Remove Member from CoPilot Pool Temporarily
<<extend>>
2.11 Remove Member from CoPilot Pool Permanently
TC Admin
2.12 Configure Application
2.13 Process CoPilot Rules
2.14 Keep CoPilot Data
System

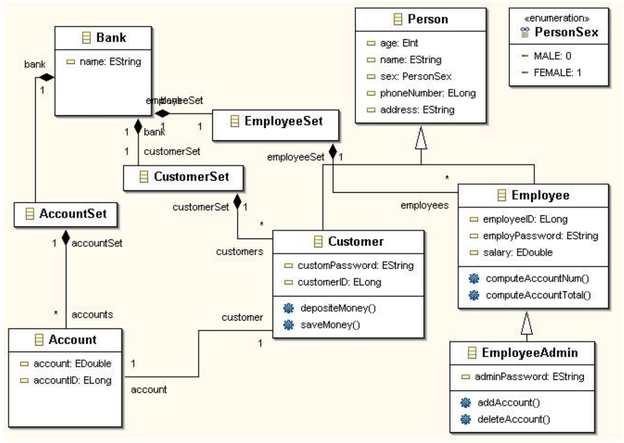

2.01.1 View CoPilot Profile Activity
Start
Display CoPilot Statistics
Display Projects Overall Statistics
Display Contests Statistics
Display Bar Chart of Contests Statistics
choose action?
[CONTACT COPILOT]
[CHOOSE CONTEST TYPE]
[GET HELP]
Contact CoPilot
Select Contest Type
Display Help on Statistics
[PROJECTS HISTORY]
View CoPilot Projects History
Display Statistics for Chosen Contest Type
End

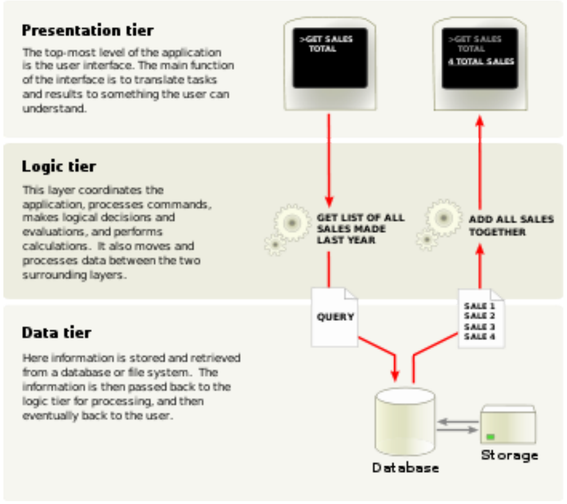
分析模型案例

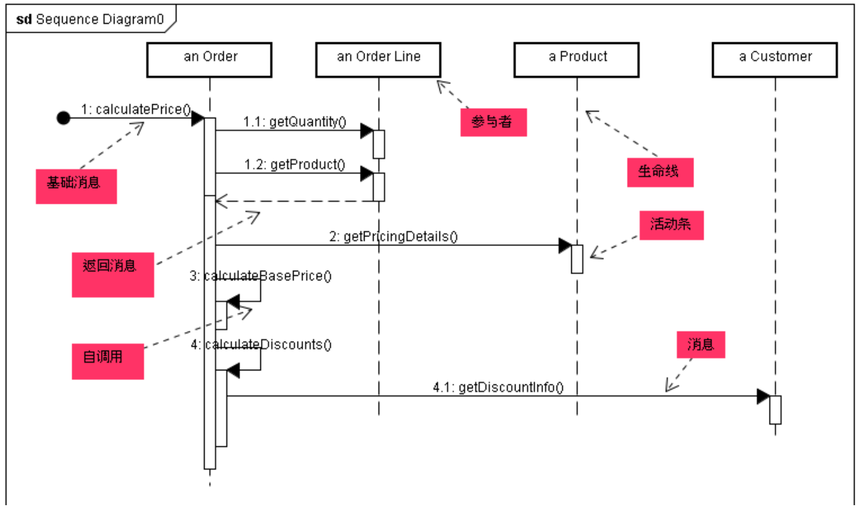
第二步 - 设计（建立计算机模型）

- 怎么做

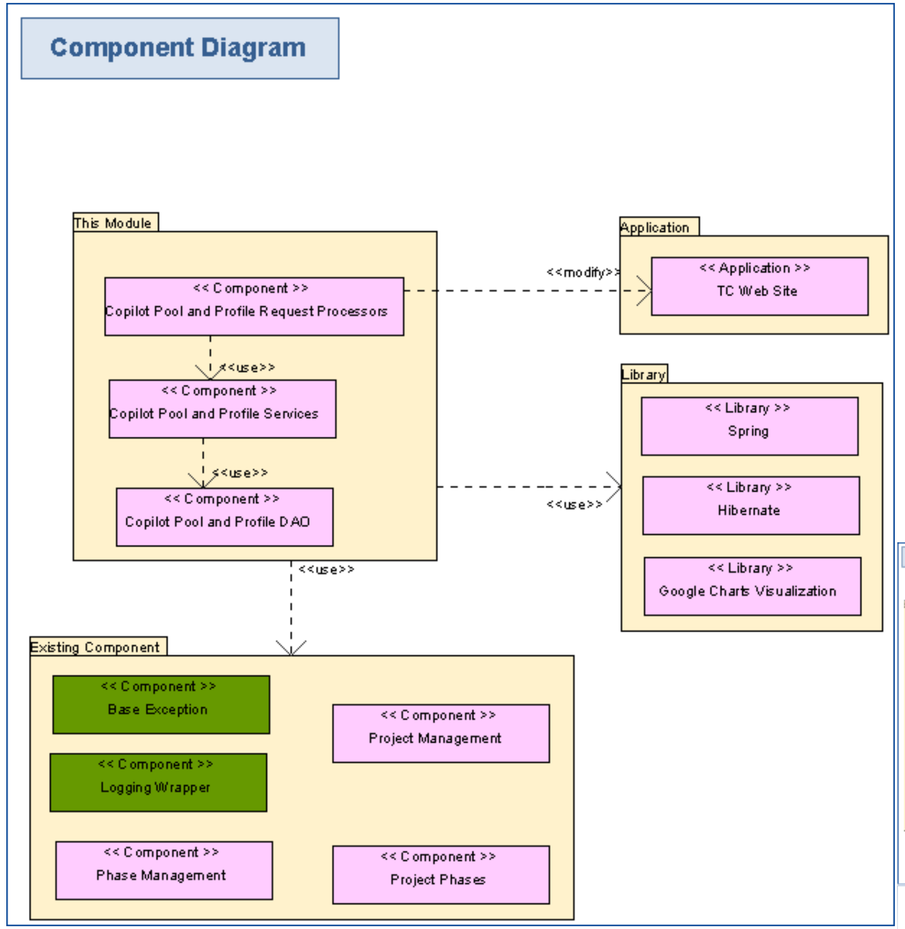

Clients
Internet
Server

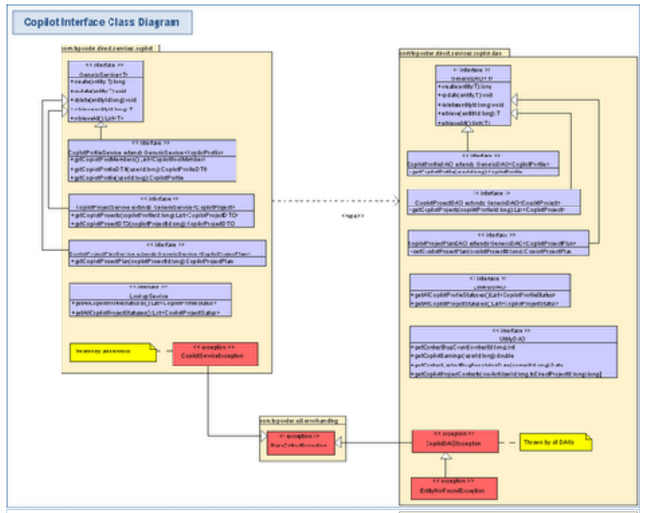

Bank
name: EString
Person
age: EInt
name: EString
sex: PersonSex
phoneNumber: ELong
address: EString
«enumeration»
PersonSex
MALE: 0
FEMALE: 1
EmployeeSet
CustomerSet
AccountSet
Customer
customPassword: EString
customerID: ELong
depositeMoney()
saveMoney()
Employee
employeeID: ELong
employPassword: EString
salary: EDouble
computeAccountNum()
computeAccountTotal()
Account
account: EDouble
accountID: ELong
EmployeeAdmin
adminPassword: EString
addAccount()
deleteAccount()
bank
customerSet
employeeSet
accountSet
customers
accounts
employees
customer
account

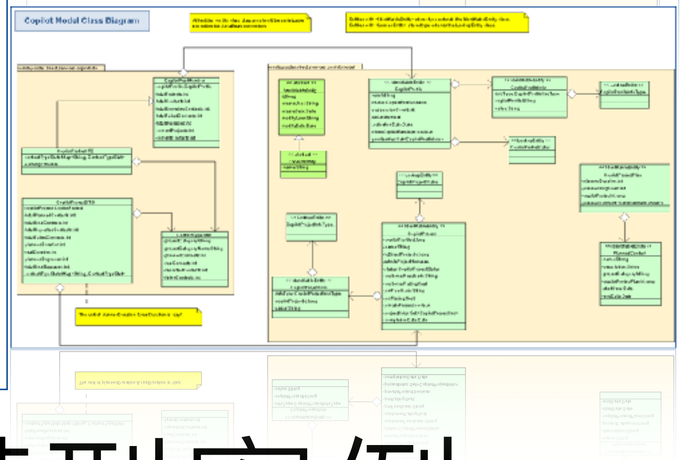

Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
QUERY
Database
Storage

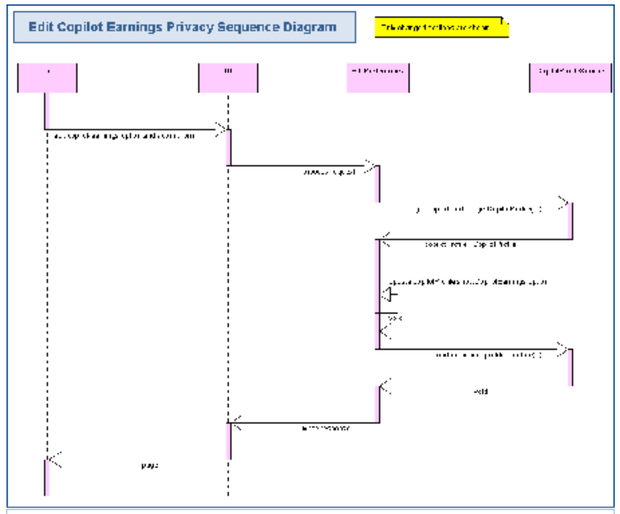

sd Sequence Diagram0
an Order
an Order Line
a Product
a Customer
1: calculatePrice()
1.1: getQuantity()
1.2: getProduct()
2: getPricingDetails()
3: calculateBasePrice()
4: calculateDiscounts()
4.1: getDiscountInfo()
参与者
生命线
基础消息
活动条
返回消息
消息
自调用

设计模型


Component Diagram
This Module
<< Component >>
Copilot Pool and Profile Request Processors
<<use>>
<< Component >>
Copilot Pool and Profile Services
<<use>>
<< Component >>
Copilot Pool and Profile DAO
<<modify>>
Application
<< Application >>
TC Web Site
<<use>>
Library
<< Library >>
Spring
<< Library >>
Hibernate
<< Library >>
Google Charts Visualization
<<use>>
Existing Component
<< Component >>
Base Exception
<< Component >>
Logging Wrapper
<< Component >>
Phase Management
<< Component >>
Project Management
<< Component >>
Project Phases


Copilot Interface Class Diagram


Copilot Modal Class Diagram


Edit Copilot Earnings Privacy Sequence Diagram

设计模型案例

第三步 - 构造（制订解决方案）

- Do it

构造模型 - 高级语言语法

- Java
- C++
- Haskell

构造结果 - 程序本身

- class HelloWorldApp {
- 	public static void main(String[] args) {
- 		System.out.println("Hello World!"); // Display the string.
- 	}
- }

第四步 - 软件测试 （检验）

- Verification - 检查解决方案的有效性
  - whether do it right?
- Validation - 检查是否解决了问题?
  - whether do the right thing?

第五步 - 移交和演化（实施）

- 在真实环境中运行
- 演化新的版本

步骤

- 需求
- 设计
- 实现
- 测试
- 部署

Outline

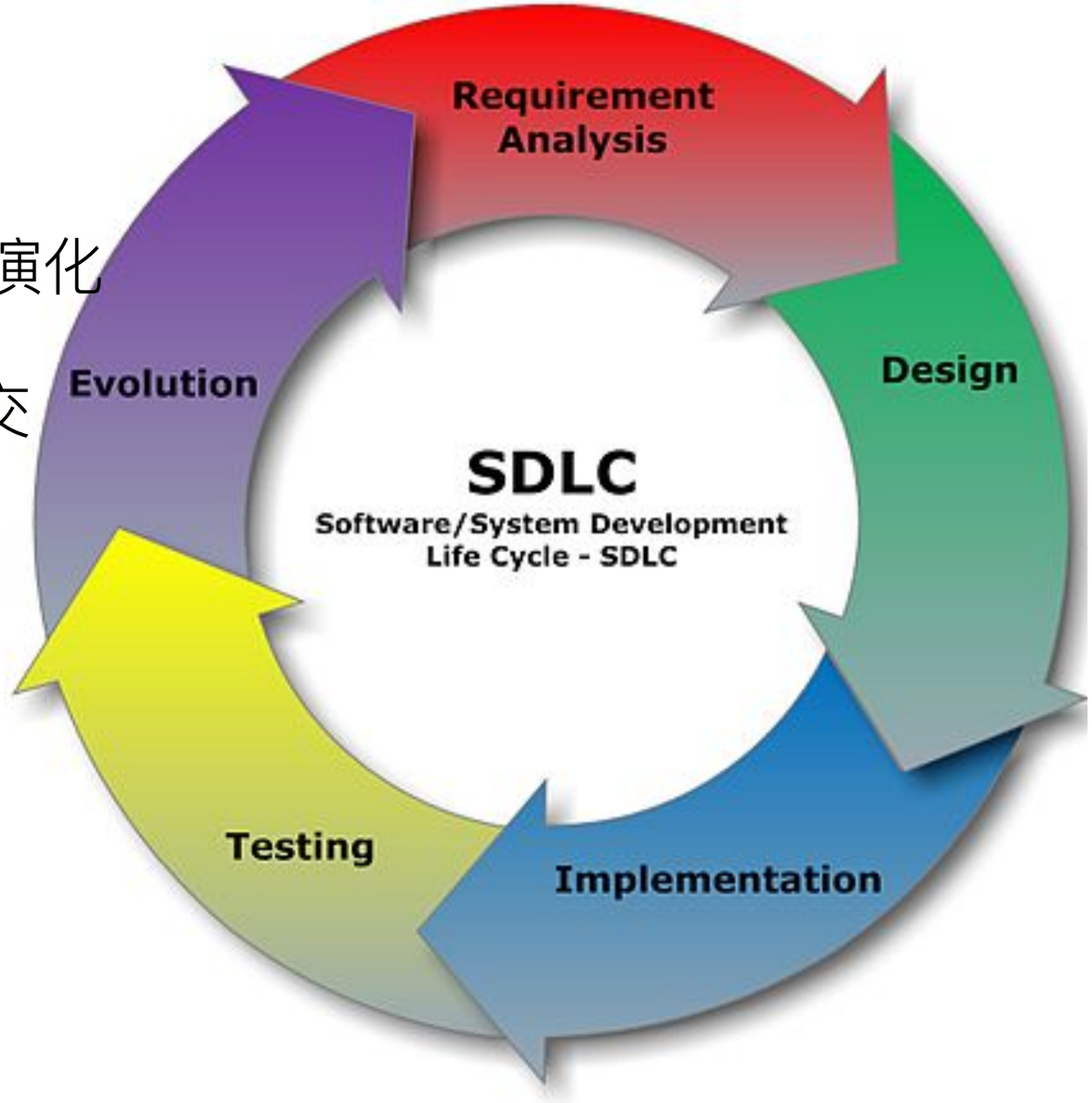
- 数学建模
- 计算机建模
- 软件工程建模
- 软件开发生命周期模型
- 软件工程建模分析案例

SDLC

Software development life-cycle

不断演化

移交



Software Development Activity

Activity	Target	Artifact
Software Requirement	What	SRS
Software Design	How	SDD
Software Construction	Build	Code and executable file
Software Testing	Are you building the “right” thing? Are you building it “right”?	Test Report
Software Deliver	Install	User Document and Svstem Document
Software Maintenance	Revolution	New version software

Outline

- 数学建模
- 计算机建模
- 软件工程建模
- 软件开发生命周期模型
- 软件工程建模分析案例
  - 如何给软件需求建模？—用例
  - 性能的分析 - String类的使用

如何给软件需求建模?

什么是需求?

需求

- IEEE对需求的定义为[IEEE610.12-1990]：
  - ⑴用户为了解决问题或达到某些目标所需要的条件或能力；
  - ⑵系统或系统部件为了满足合同、标准、规范或其它正式文档所规定的要求而需要具备的条件或能力；
  - ⑶对⑴或⑵中的一个条件或一种能力的一种文档化表述。

需求是一种期望

如何表达这种期望?

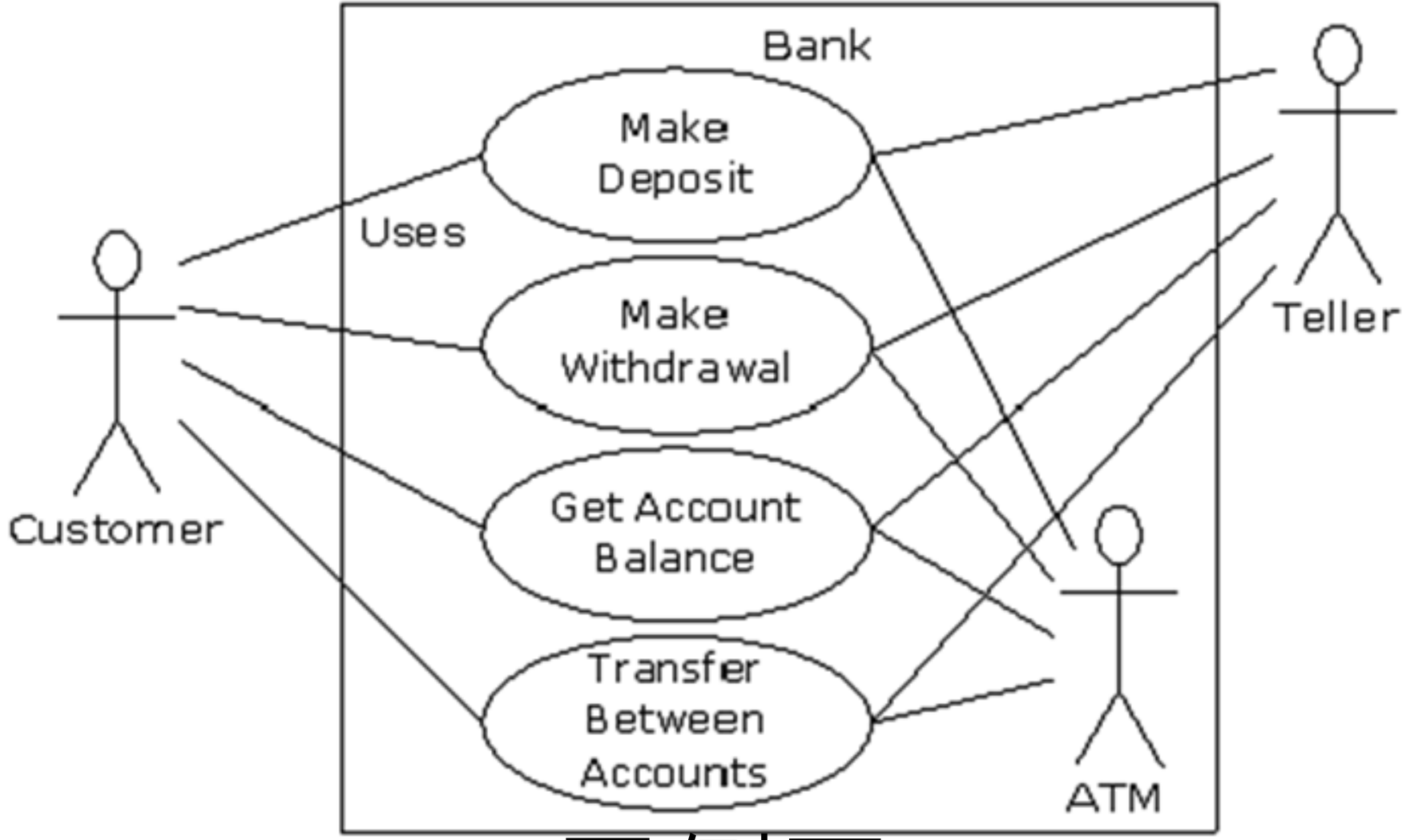
一段录音？一段文字？

如何组织这些内容?

用例

- 用例最初由[Jacobson1992] 在 Objectory 方法中提出的,它将用例定义为“在系统(或者子系统或者类)和外部对象的交互当中所执行的行为序列的描述,包括各种不同的序列和错误的序列,它们能够联合提供一种有价值的服务”[Rumbaugh2004]。

- [Cockburn2001]认为用例描述了在不同条件下系统对 某一用户的请求的响应。根据用户的请求和请求时的系统条件,系统将执行不同的行为序列, 每一个行为序列被称为一个场景。一个用例是多个场景的集合。


Bank
Make Deposit
Uses
Make Withdrawal
Get Account Balance
Transfer Between Accounts
Customer
Teller
ATM

用例图

寻找参与者

- 谁对系统有着明确的目标和要求并且主动发出动作?
- 系统是为谁服务的?

用例的特征

- 用例是相对独立的
  - 取钱？填写取款单？
- 用例的执行结果对参与者来说是可观测的和有意义的
  - 登陆系统？后台进程监控？
- 这件事必须有一个参与者发起。
  - ATM吐钞票？
- 用例必须是以动宾短语形式出现的
  - 统计？报表？
- 一个用例就是一个需求单元、分析单元、设计单元、开发单元、测试单元，甚至部署单元

目标和步骤

- 参与者：
  - 寄信人
- 用例：
  - 买信封
  - 买邮票
  - 付钱
  - 投递

ID:	用例的标识，通常会结合用例的层次结构使用 X.Y.Z 的方式
名称:	对用例内容的精确描述，体现了用例所描述的任务，通常是“动词＋名词”
用例属性	包括创建者、创建日期、更新历史等
参与者:	描述系统的主参与者、辅助参与者和每个参与者的目标
描述:	简要描述用例产生的原因，大概过程和输出结果
优先级:	用例所描述的需求的优先级
触发条件:	标识启动用例的事件，可能是系统外部的事件，也可能是系统内部的事件，还可能是正常流程的第一个步骤
前置条件:	用例能够正常启动和工作的系统状态条件
后置条件:	用例执行完成后的系统状态条件
正常流程:	在常见和符合预期的条件下，系统与外界的行为交互序列
分支流程:	用例中可能发生的非常见的其他合理场景
异常流程:	在非预期的错误条件发生时，系统对外界进行响应的交互行为序列
相关用例:	记录和该用例存在关系的其他用例。关于用例之间的关系见 10.4.4
业务规则:	可能会影响用例执行的业务规则
特殊需求:	和用例相关的其他特殊需求，尤其是非功能性需求
假设:	在建立用例时所做的假设
待确定问题:	一些当前的用例描述还没有解决的问题

用例文本描述

ID
1
名称
处理销售
创建者

最后一次更新者

创建日期

最后更新日期

参与者
收银员，目标是快速、正确地完成商品销售，尤其不要出现支付错误。
触发条件
顾客携带商品到达销售点
前置条件
收银员必须已经被识别和授权。
后置条件
存储销售记录，包括购买记录、商品清单、赠送清单和付款信息；更新库存和会员积分；打印收据。
优先级
高
正常流程
1如果是会员，收银员输入客户编号
2系统显示会员信息，包括姓名和积分
3收银员输入商品标识
4系统记录商品，并显示商品信息，商品信息包括商品标识、描述、数量、价格、特价（如果有商品特价策略的话）和本项商品总价
5系统显示已购入的商品清单，商品清单包括商品标识、描述、数量、价格、特价、各项商品总价和所有商品总价
收银员重复3-5步，直到完成所有商品的输入
6收银员结束输入，系统计算并显示总价，计算根据总额特价策略进行
7系统根据商品赠送策略和总额赠送策略计算并显示赠品清单，赠品清单包括各项赠品的标识、描述与数量
8收银员请顾客支付账单
9顾客支付，收银员输入收取的现金数额
10系统给出应找的余额，收银员找零
11收银员结束销售，系统记录销售信息、商品清单、赠送清单和账单信息，并更新库存系统打印收据

扩展流程
1a、非法客户编号：
1、　　系统提示错误并拒绝输入
3a、非法标识：
1、　　系统提示错误并拒绝输入
3b、有多个具有相同商品类别的商品（如5把相同的雨伞）
1、　　收银员可以手工输入商品标识和数量
5-8a、顾客要求收银员从已输入的商品中去掉一个商品：
1、　　收银员输入商品标识并将其删除
1a、非法标识
1、系统显示错误并拒绝输入
2、　　返回正常流程第5步
5-8b、顾客要求收银员取消交易
1、　　收银员在系统中取消交易
9a、会员使用积分
1.　　系统显示可用的积分余额
2.　　营业员输入使用的积分数额，每50个积分等价于1元RMB
3.　　系统显示剩余的积分余额和余下的现金数额
4.　　收银员输入收取的现金数额
11a、会员
1、　　系统记录销售信息、商品清单、赠送清单和账单信息，并更新库存
2、　　计算并更新会员积分，将积分总额和积分余额都增加现金数额
特殊需求
1、系统显示的信息要在1米之外能看清
2、因为在将来的一段时间内，超市都不打算使用扫描仪设备，所以为输入方便，要使用5位0～9数字的商品标识格式。
将来如果超市采购了扫描仪，商品标识格式要修改为标准要求：13位0～9的数字

画一个图书管理系统的用例图！

Outline

- 数学建模
- 计算机建模
- 软件工程建模
- 软件开发生命周期模型
- 软件工程建模分析案例
  - 如何给软件需求建模？—用例
  - 性能的分析 - String类的使用

不可变类

- 不可改变的字符串具有一个很大的优点：编译器可以把字符串设置为共享。

- JAVA为了提高效率，所以对于String类型进行了特别的处理－－－为String类型提供了串池
  定义一个string类型的变量有两种方式：
  - String name= "tom ";
  - String name =new String("tom ")
  - 使用第一种方式的时候，就使用了串池；使用第二种方式的时候，就是一种普通的声明对象的方式 。如果你使用了第一种方式，那么当你再声明一个内容也是 "tom "的String时，它将使用串池里原来的那个内存，而不会重新分配内存，也就是说，String name= "tom ", 将会指向同一块内存
- 另外关于String类型是不可改变的问题：
  String类型是不可改变的，也就是说，当你想改变一个String对象的时候，比如
  - name= "madding "
  - 那么虚拟机不会改变原来的对象，而是生成一个新的String对象，然后让name去指向它，如果原来的那个 "tom "没有任何对象去引用它，虚拟机的垃圾回收机制将接收它。

Case 1 - String “+”

- String tempstr = "abcdefghijklmnopqrstuvwxyz";
- int times = 5000;
- long lstart1 = System.currentTimeMillis();
- String str = "";
- for (int i = 0; i < times; i++) {
- str += tempstr;
- }
- long lend1 = System.currentTimeMillis();
- long time = (lend1 - lstart1);
- System.out.println(time);

Case 2 - StringBuffer

- String tempstr = "abcdefghijklmnopqrstuvwxyz";
- int times = 5000;
- long lstart2 = System.currentTimeMillis();
- StringBuffer sb = new StringBuffer();
- for (int i = 0; i < times; i++) {
- sb.append(tempstr);
- }
- long lend2 = System.currentTimeMillis();
- long time2 = (lend2 - lstart2);
- System.out.println(time2);

Case 3 - StringBuffer more effective way

- String tempstr = "abcdefghijklmnopqrstuvwxyz";
- int times = 5000;
- long lstart2 = System.currentTimeMillis();
- StringBuffer sb = new StringBuffer(tempstr.length()*times);
- for (int i = 0; i < times; i++) {
- sb.append(tempstr);
- }
- long lend2 = System.currentTimeMillis();
- long time2 = (lend2 - lstart2);
- System.out.println(time2);

char[] 与 String

- String strPassword="Unknown";
- char[] charPassword= new char[]{'U','n','k','w','o','n'};
- System.out.println("String password: " + strPassword);
- System.out.println("Character password: " + charPassword);

- String password: Unknown
- Character password: [C@110b053

```java
package com.littletutorials.tips;

public class ConcatPerf {
    private static final int ITERATIONS = 100000;
    private static final int BUFFSIZE = 16;

    private void concatStrAdd() {
        System.out.print("concatStrAdd   -> ");
        long startTime = System.currentTimeMillis();
        String concat = "";
        for (int i = 0; i < ITERATIONS; i++) {
            concat += i % 10;
        }
        long endTime = System.currentTimeMillis();
        System.out.print("length: " + concat.length());
        System.out.println(" time: " + (endTime - startTime));
    }

    private void concatStrBuff() {
        System.out.print("concatStrBuff  -> ");
        long startTime = System.currentTimeMillis();
        StringBuffer concat = new StringBuffer(BUFFSIZE);
        for (int i = 0; i < ITERATIONS; i++) {
            concat.append(i % 10);
        }
        long endTime = System.currentTimeMillis();
        System.out.print("length: " + concat.length());
        System.out.println(" time: " + (endTime - startTime));
    }

    private void concatStrBuild() {
        System.out.print("concatStrBuild -> ");
        long startTime = System.currentTimeMillis();
        StringBuilder concat = new StringBuilder(BUFFSIZE);
        for (int i = 0; i < ITERATIONS; i++) {
            concat.append(i % 10);
        }
        long endTime = System.currentTimeMillis();
        System.out.print("length: " + concat.length());
        System.out.println(" time: " + (endTime - startTime));
    }

    public static void main(String[] args) {
        ConcatPerf st = new ConcatPerf();
        System.out.println("Iterations: " + ITERATIONS);
        System.out.println("Buffer    : " + BUFFSIZE);

        st.concatStrBuff();
        st.concatStrBuild();
        st.concatStrAdd();
    }
}
```

```
Iterations: 100000
Capacity : 16
concatStrBuff -> length: 100000 time: 16
concatStrBuild -> length: 100000 time: 15
concatStrAdd -> length: 100000 time: 10437
```

```
01. L3
02.  LINENUMBER 15 L3
03.  ICONST_0
04.  ISTORE 4
05. L4
06.  GOTO L5
07. L6
08.  LINENUMBER 17 L6
09. FRAME APPEND [J java/lang/String I]
10.  NEW java/lang/StringBuilder
11.  DUP
12.  ALOAD 3
13.  INVOKESTATIC java/lang/String.valueOf(Ljava/lang/Object;)Ljava/lang/String;
14.  INVOKESPECIAL java/lang/StringBuilder.<init>(Ljava/lang/String;)V
15.  ILOAD 4
16.  BIPUSH 10
17.  IREM
18.  INVOKEVIRTUAL java/lang/StringBuilder.append(I)Ljava/lang/StringBuilder;
19.  INVOKEVIRTUAL java/lang/StringBuilder.toString()Ljava/lang/String;
20.  ASTORE 3
21. L7
22.  LINENUMBER 15 L7
23.  IINC 4 1
24. L5
25. FRAME SAME
26.  ILOAD 4
27.  LDC 100000000
28.  IF_ICMPLT L6
29. L8
```

String +

```
01. L3
02.  LINENUMBER 43 L3
03.  ICONST_0
04.  ISTORE 4
05. L4
06.  GOTO L5
07. L6
08.  LINENUMBER 45 L6
09. FRAME APPEND [J java/lang/StringBuilder I]
10.  ALOAD 3
11.  ILOAD 4
12.  BIPUSH 10
13.  IREM
14.  INVOKEVIRTUAL java/lang/StringBuilder.append(I)Ljava/lang/StringBuilder;
15.  POP
16. L7
17.  LINENUMBER 43 L7
18.  IINC 4 1
19. L5
20. FRAME SAME
21.  ILOAD 4
22.  LDC 100000000
23.  IF_ICMPLT L6
24. L8
```

StringBuilder

StringBuilder vs String Buffer

- Increasing the initial capacity for StringBuffer and StringBuilder doesn't make much difference. Clearly with only 100,000 iterations the numbers for StringBuffer and StringBuilder are just noise.

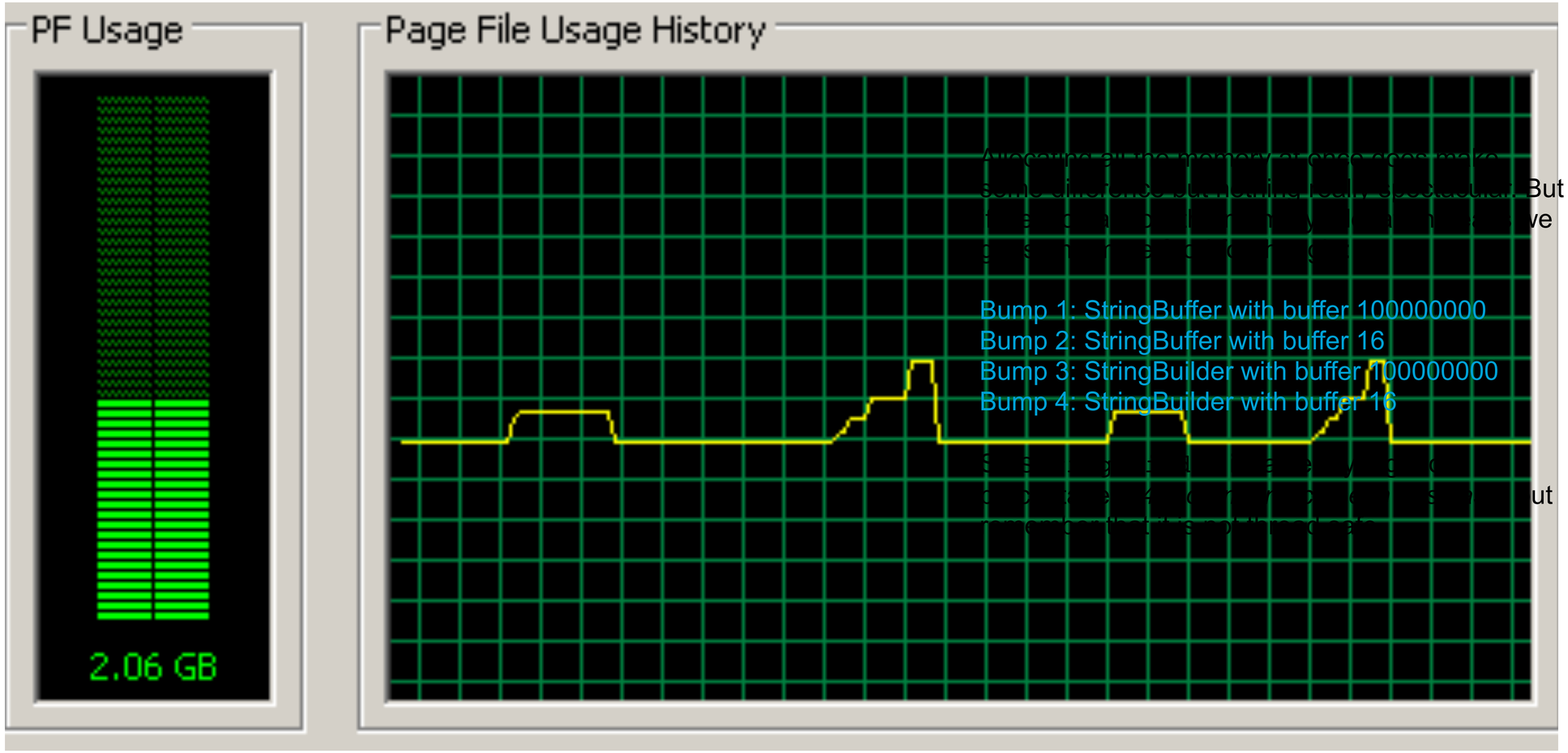
- Iterations: 100000
  Capacity : 100000
  concatStrBuff -> length: 100000 time: 15
  concatStrBuild -> length: 100000 time: 16
  concatStrAdd -> length: 100000 time: 10594

- Let's crank it up… but without the String concatenation test because it will never finish.

- Iterations: 100000000
  Capacity : 16
  concatStrBuff -> length: 100000000 time: 15142
  concatStrBuild -> length: 100000000 time: 10891

- Now it is pretty clear StringBuilder is much faster because it avoids synchronization.

- Iterations: 100000000
  Capacity : 100000000
  concatStrBuff -> length: 100000000 time: 14220
  concatStrBuild -> length: 100000000 time: 10611



Memory Allocation

String.intern()

- 存在于.class文件中的常量池，在运行期被JVM装载，并且可以扩充。String的intern()方法就是扩充常量池的一个方法；
- 当一个String实例str调用intern()方法时，Java查找常量池中是否有相同Unicode的字符串常量，
  - 如果有，则返回其的引用，
  - 如果没有，则在常量池中增加一个Unicode等于str的字符串并返回它的引用

- Java代码
- String s0= “kvill”;
- String s1=new String(”kvill”);
- String s2=new String(“kvill”);
- System.out.println(s0==s1);
- System.out.println(“**********”);
- s1.intern();
- s2=s2.intern(); //把常量池中“kvill”的引用赋给s2
- System.out.println(s0==s1);
- System.out.println(s0==s1.intern());
- System.out.println(s0==s2);

- 结果为：
  - false
  - **********
  - false //虽然执行了s1.intern(),但它的返回值没有赋给s1
  - true //说明s1.intern()返回的是常量池中”kvill”的引用
  - true

- Java代码
- String s1=new String("kvill");
- String s2=s1.intern();
- System.out.println(s1==s1.intern());
- System.out.println(s1+" "+s2);
- System.out.println(s2==s1.intern());

- 结果：
  - false
  - kvill kvill
  - true

Java对象在JVM中的存储

- 一般而言，Java 对象在虚拟机的结构如下：
  - 对象头（object header）：8 个字节
  - Java 原始类型数据：如 int, float, char 等类型的数据，各类型数据占内存如 表 1. Java 各数据类型所占内存.
  - 引用（reference）：4 个字节
  - 填充符（padding）

String在JVM中的存储

- 然而，一个 Java 对象实际还会占用些额外的空间，如：对象的 class 信息、ID、在虚拟机中的状态。在 Oracle JDK 的 Hotspot 虚拟机中，一个普通的对象需要额外 8 个字节。
- 如果对于 String（JDK 6）的成员变量声明如下：
    - private final char value[];
    - private final int offset;
    - private final int count;
    - private int hash;
- 那么因该如何计算该 String 所占的空间?
- 首先计算一个空的 char 数组所占空间，在 Java 里数组也是对象，因而数组也有对象头，故一个数组所占的空间为对象头所占的空间加上数组长度，即 8 + 4 = 12 字节 , 经过填充后为 16 字节。
- 那么一个空 String 所占空间为：
- 对象头（8 字节）+ char 数组（16 字节）+ 3 个 int（3 × 4 = 12 字节）+1 个 char 数组的引用 (4 字节) = 40 字节。
- 因此一个实际的 String 所占空间的计算公式如下：
- 8*((8+12+2*n+4+12)+7) / 8 = 8*(int) ((((n) *2)+43) /8)

- 其中，n 为字符串长度。

案例

- 在我们的大规模文本分析的案例中，程序需要统计一个 300MB 的 csv 文件所有单词的出现次数，分析发现共有 20,000 左右的唯一单词，假设每个单词平均包含 15 个字母，这样根据上述公式，一个单词平均占用 75 bytes. 那么这样 75 * 20,000 = 1500000，即约为 1.5M 左右。但实际发现有上百兆的空间被占用。 实际使用的内存之所以与预估的产生如此大的差异是因为程序大量使用String.split() 或 String.substring()来获取单词。

```
public String substring(int beginIndex, int endIndex) {
    if (beginIndex < 0) {
        throw new StringIndexOutOfBoundsException(beginIndex);
    }
    if (endIndex > count) {
        throw new StringIndexOutOfBoundsException(endIndex);
    }
    if (beginIndex > endIndex) {
        throw new StringIndexOutOfBoundsException(endIndex - beginIndex);
    }
    return ((beginIndex == 0) && (endIndex == count)) ? this :
        new String(offset + beginIndex, endIndex - beginIndex, value);
}
```

- 调用的 String 构造函数源码为：

```
String(int offset, int count, char value[]) {
    this.value = value;
    this.offset = offset;
    this.count = count;
}
```

- 仔细观察粗体这行代码我们发现 String.substring()所返回的 String 仍然会保存原始 String, 这就是 20,000 个平均长度的单词竟然占用了上百兆的内存的原因。 一个 csv 文件中每一行都是一份很长的数据，包含了上千的单词，最后被 String.split() 或 String.substring()截取出的每一个单词仍旧包含了其原先所在的上下文中，因而导致了出乎意料的大量的内存消耗。

- 当然，JDK String 的源码设计当然有着其合理之处，对于通过 String.split()或 String.substring()截取出大量 String 的操作，这种设计在很多时候可以很大程度的节省内存，因为这些 String 都复用了原始 String，只是通过 int 类型的 start, end 等值来标识每一个 String。 而对于我们的案例，从一个巨大的 String 截取少数 String 为以后所用，这样的设计则造成大量冗余数据。 因此有关通过 String.split()或 String.substring()截取 String 的操作的结论如下：

- 对于从大文本中截取少量字符串的应用，String.substring()将会导致内存的过度浪费。

- 对于从一般文本中截取一定数量的字符串，截取的字符串长度总和与原始文本长度相差不大，现有的String.substring()设计恰好可以共享原始文本从而达到节省内存的目的。

- 既然导致大量内存占用的根源是 String.substring()返回结果中包含大量原始 String，那么一个显而易见的减少内存浪费的的途径就是去除这些原始 String。办法有很多种，在此我们采取比较直观的一种，即再次调用 newString构造一个的仅包含截取出的字符串的 String，我们可调用 String.toCharArray()方法：

- String newString = new String(smallString.toCharArray());

String 构造的方法选择

- 常见的创建一个 String 可以用赋值操作符"=" 或用 new 和相应的构造函数。初学者一定会想这两种有何区别，举例如下：
- String a1 = “Hello”;
- String a2 = new String(“Hello”);

- 第一种方法创建字符串时 JVM 会查看内部的缓存池是否已有相同的字符串存在：如果有，则不再使用构造函数构造一个新的字符串，直接返回已有的字符串实例；若不存在，则分配新的内存给新创建的字符串。
- 第二种方法直接调用构造函数来创建字符串，如果所创建的字符串在字符串缓存池中不存在则调用构造函数创建全新的字符串，如果所创建的字符串在字符串缓存池中已有则再拷贝一份到 Java 堆中。

使用构造函数 string() 带来的内存性能隐患和缓解

- 仍然以之前的从 csv 文件中截取 String 为例，先前我们通过用 new String() 去除返回的 String 中附带的原始 String 的方法优化了subString导致的内存消耗问题。然而，当我们下意识地使用 newString去构造一个全新的字符串而不是用赋值符来创建（重用）一个字符串时，就导致了另一个潜在的性能问题，即：重复创建大量相同的字符串。说到这里，您也许会想到使用缓存池的技术来解决这一问题，大概有如下两种方法：
  - 方法一，使用 String 的 intern()方法返回 JVM 对字符串缓存池里相应已存在的字符串引用，从而解决内存性能问题，但这个方法并不推荐！原因在于：首先，intern() 所使用的池会是 JVM 中一个全局的池，很多情况下我们的程序并不需要如此大作用域的缓存；其次，intern() 所使用的是 JVM heap 中 PermGen 相应的区域，在 JVM 中 PermGen 是用来存放装载类和创建类实例时用到的元数据。程序运行时所使用的内存绝大部分存放在 JVM heap 的其他区域，过多得使用 intern()将导致 PermGen 过度增长而最后返回 OutOfMemoryError，因为垃圾收集器不会对被缓存的 String 做垃圾回收。所以我们建议使用第二种方式。
  - 方法二，用户自己构建缓存，这种方式的优点是更加灵活。创建 HashMap，将需缓存的 String 作为 key 和 value 存放入 HashMap。假设我们准备创建的字符串为 key，将 Map cacheMap 作为缓冲池，那么返回 key 的代码如下：
    - private String getCacheWord(String key) {
    - String tmp = cacheMap.get(key);
    - if(tmp != null) {
    - return tmp;
    - } else {
    - cacheMap.put(key, key);
    - return key;
    - }
    - }