blog/编程的逻辑.md

编程的逻辑

编程可以看成是把需求用机器能懂的语言表达出来。表达的时候，会借助一些工具，先表达成工具能懂的逻辑，然后由工具转换成机器能懂的逻辑。

要把需求准确的翻译表达成工具能懂的逻辑，首先要对工具本身的逻辑和规则很清晰，否则翻译出来就是鸡同鸭讲，驴唇不对马嘴，漏洞百出。工具的逻辑和规则通过应用程序编程接口(API)和编程语言暴露出来，只有对API和编程语言都非常熟悉，才能精确的把需求翻译表达出来。

硬件的基本逻辑

计算机的三大核心部件：CPU、内存、硬盘。应用编程很少直接面向硬件，但要知道一些基本事实，比如内存和硬盘的速度差异，编程的时候如果同步操作磁盘较多，则可能影响性能，可以考虑用内存来缓存数据。 程序的内存空间分为堆区和栈区，要能对所使用的编程语言的内存模型有了解，防止编程时出现内存泄露。

操作系统的逻辑

应用编程一般是直接面向操作系统或者更上层的平台。面向操作系统编程，那么编写出来的应用运行起来就是一个独立的进程。面向一些上层平台编程，例如java虚拟机、浏览器，编写出来的应用就可能运行在平台进程内的某些线程或者平台fork出来的子进程。 进程是应用程序运行的

实例，是操作系统管理程序运行和分配资源的基本单位。但是真正调度执行的时候，是以线程为单位进行调度执行的。一个进程内一般都有个主线程，Android、iOS的UI线程就是主线程，浏览器中每个页面的js是在单线程中执行。 单个线程内的代码是线性串行执行的，执行完代码后线程就会退出并被销毁。多个线程间的执行顺序是并行的，如果线程间需要协调和交互，就涉及到线程间的同步互斥。多线程模型虽然大大提高了CPU的利用率，但同时也大大提高了编程复杂度，多线程间的资源共享和互斥非常容易造成死锁等非常难于排查的问题，非常难于掌控。编程的时候，尽量保持单线程模型。 有时候并不希望线程退出，比如App、服务器的主线程，浏览器的JS执行线程，他们需要在没事的时候挂起，有事的时候及时处理事情。他们的核心逻辑类似，都是在线程内维护了一个事件循环，等待着事件的发生，然后处理事件。耗时的事件处理必须新开线程处理，否则会阻塞整个事件循环，让其他事件得不到处理，App表现为UI卡顿。事件循环的大致逻辑是：

```
for(!quit) {
    event = getEvent()
    switch(event.type) {
        case TIME:
            processTimeout()
        case IO:
            processIO()
        case UIEvent:
            processAction()
    }
}
```

事件一般有定时事件，IO就绪事件，UI操作事件。getEvent是同步调用，如果没有事件会挂起线程，直到返回一个事件，然后就处理相应的事件。谨记耗时处理不能放在事件循环线程中，否则会导致其他事件不能得到及时处理。 事件循环让单线程看上去也具备了“多任务”的处理能力，由于没有线程同步，单线程的性能也非常高，服务端的高性能网络库也有基于这种事件循环的，如libevent、libuv、java的很多reactor模式实现。浏览器js环境中，setTimeout就是注册一个事件，到指定时间后执行某个函数，这个函数并不是在其他线程中执行的，只是在当前线程中的某次循环中执行的。APP中是包装成了runloop的概念，从其他线程切换到UI线程执行时，需要使用提供的API向UI线程提交一个回调函数，UI线程会在后续的循环中执行回调函数，完成从其他线程切换到主线程的操作。iOS可以去看libdispatch的源码，是iOS系统的基石。

通信和网络的逻辑

通信的逻辑是，两个实体间要通信，第一步就是建立通道，第二步就是约定内容格式，有了这两步，两个实体就能互相交互。通信无处不在，不同主机之间、同一台主机各硬件之间、进程之间、不同模块之间、不同页面之间、原生和H5之间，都需要建立可靠的通信机制，才能实现交互。 网络常用的API主要是TCP和更上层的HTTP。TCP可以理解为，在ab两端之间有两根管子，一条管子从a流向b，就是a往里面写b从里面读，另一条管子就是反过来b往里面写a从里面读。这两条管子只是通道，但是没有约定管子里流的内容的格式，通道里流的都是二进制数据。要

双方能互相解析对方的数据，必须约定一种内容格式。HTTP就是这样的内容格式约定，当然我们也可以定义自己的格式。有了格式，双方就知道如何读取解析对方发送的消息。这就是粘包分包问题，管子里面所有消息都是粘在一起的二进制流，但是有了格式，我们就能把二进制流拆分成独立的消息。 应用程序常采用请求应答模式，就是客户端发送请求，然后接受服务器端的应答。假如客户端顺序发送10个请求，则服务器端会按顺序收到1-10号请求，收到请求后，每个请求一般都是并行处理，先处理完的先发送响应消息给客户端，所以客户端收到的响应顺序是不确定的。

编程语言的逻辑

编程语言可以分为声明式和命令式。掌握编程语言的第一层境界是熟悉语法，这些语法就是规则，只有按照这些规则，编译解析工具才能把编写的逻辑翻译成机器码去执行。语法主要包含类型定义及作用域、条件和循环语句。第二层就是理解编程语言的类型系统，基础类型是如何实现的，面向对象语言的类是如何实现的。第三层就是理解语言的内存模型和语言被编译、优化和执行的规则和原理，例如各种类型在内存中是怎么表示的，理解内存分配回收机制，知道如何防止内存泄露，知道多个条件判断如果第一个条件就已经能确定结果那么后续条件不会执行。

运行环境

应用编程都是基于一定环境，编写运行在环境内的程序。要想程序按预期运行，那么对环境的特性要有足够的理解，明白环境会从哪些方面影响程序的运行。例如，编写

服务端程序如果不处理中断，那么中断就会让程序意外退出，App不处理网络断开连接，体验就会有问题。要对环境足够熟悉，才能在环境中得心应手。

生命周期

传统进程的生命周期比较简单，系统只会通过中断来影响应用程序。App的生命周期更复杂，操作系统会从更多方面影响App。iOS App接近于传统独立进程，但是多了前后台切换，切换到后台后，只能做少量特定的动作。Android弱化了进程的概念，组件都由系统来维护，进而有了更复杂的生命周期。web单页应用类似iOS，多页应用类似Android。

架构的逻辑

架构就是抽象出各层次所有元素的结构定义和交互规范。先梳理清楚包含哪些元素，再抽象出共性的结构和交互。例如，大部分App就是各种页面，向服务器请求或提交数据，并且需要向服务器明示身份，页面之间还需要交互。所以网络、账户是大部分App的基本服务，页面跳转也可以抽象成服务以便页面之间降低耦合度以及传递入参出参，页面本身也要抽象以便快速做出可维护高水准的页面。 多处使用的功能都应该抽象成API，各部分之间的交互也只能依赖API，API方便屏蔽实现，也方便添加hook和修改。 抽象的过程中，应该遵守事物的客观性。比如账户合并机制，账户代表的是人，人和人不能合并，所以账户合并就不太可取，所有资产都是放到账户上的，账户合并也意味着所有资产都要合并，是一个大坑。

智能化

编程活动中，重新定义程序员的角色，让程序员做更少的工作，让系统承担更多职责，并且程序员的产出受系统管控。云技术、低代码、无代码就是朝着这个方向发展。