# 多范式

目前主流的编程范式有：命令式编程(Imperative programming)、函数式编程(Functional programming)、面向对象编程(Object-oriented programming)等。我们普通码农最熟悉的应该就是面向对象编程了。

# 项目集成

## 持续集成（CI）

持续集成指的是：频繁地将代码集成到主干（一天多次）。

优点:

1. 快速发现错误。没完成一点更新，就集成到主干。快速发现错误，定位错误。
2. 防止分支大幅偏离主干。如果不经常集成，主干不断更新，导致以后集成难度变大，甚至难以集成。

目的：产品快速迭代，同时保持高质量。

核心措施：代码集成到主干之前，必须通过自动化测试。只要有一个测试用例失败，就不能集成。

流程：

1. 提交

开发者向代码仓库提交代码

1. 测试（第一轮）

代码仓库对commit操作配置了hook（钩子），只要提交代码或者合并到主干，就会跑自动化测试。

单元测试：针对函数或模块的测试

集成测试：针对整体产品的某个功能的测试（功能测试）

端对端测试：从用户界面直达数据库的全链路测试。

1. 构建

通过第一轮测试，代码就可以合并到主干，就可以交付

交付后，先进行构建（将源代码转换为可以运行的实际代码，比如安装依赖，配置各种资源等等）

一些重用的构建工具：jenkins travis codeship strider

1. 测试（第二轮）

如果第一轮就包含所有的测试，构建就应该在第一轮测试前，就没有第二轮测试。

1. 部署

通过测试，自动部署到生成服务器

## 持续交付

持续交付：频繁地将软件的新版本，交付给质量团队。

## 持续部署

持续部署：持续交付下一步。代码通过评审后，自动部署生成环境。

CLI(command-line interface)

命令行界面。

SSL(secure sockets layer)

安全套接层

NFS(network file system)

网络文件系统

文件系统的一种，允许网络中的计算机通过tcp/ip 网络共享资源。本地NFS的客户端应用可以透明的读写位于远端NFS服务器上的文件。就像访问本地文件一样。

# 注册表

注册表本质上是一个数据库，储存系统和应用程序的设置信息，直接控制windows的启动，硬件驱动程序的装载，一些windows应用程序的运行。----注册表是windows系统独有的设计。

注册表记录信息包括了软、硬件的相关配置和状态信息，比如注册表中保存有应用程序和资源管理器外壳的初始条件、首选项和卸载数据等，联网计算机的整个系统的设置和各种许可，文件扩展名与应用程序的关联，硬件部件的描述、状态和属性，性能记录和其他底层的系统状态信息，以及其他数据等

# 幂等性

## 定义

可以简单理解为：多次调用，都返回第一次调用时的处理结果。

## http幂等性

本身是一种面向资源的应用层协议

当时http协议实际上存在两种不同使用方式

一种是RESTful

忠实的遵循http协议的各种规定

一种是SOA

并没有完全把http当做应用层协议，而是把http协议当做传输层协议。在http上建立自己的应用层协议。

http幂等说的实际上是每一次相同请求，对于资源产生的作用均与第一次相同，其本身并不会产生任何副作用（注意，和返回结果无关：返回结果并不一定需要一样）。

**在http的规范中**

delete ：幂等

post ： 创建资源

put: 更新资源

注意：通常，会简单认为post 表示创建资源,put表示更新资源。但是实际上二者都可以创建资源。区别在于幂等性上，post对应的uri并非资源本身，而是资源的接受者。两次相同的post会在服务器端创建两份资源，他们具有不同的uri，因而post不具有幂等性。

而put对应的URI是要创建或者更新的资源本身，对同一个URI进行相同的put，其副作用和一次put是相同的。

# RESTful

## 定义

REST 指的是：一组架构约束条件和原则。满足这些约束条件和原则的引用程序或设计就是RESTful

REST(Representaional State Transfer ：表现层状态转换 )

**实际上应当为：资源的表示层转换。**

**资源**

这里的**资源：**可以是文本，图片，服务....（通过统一资源定位符uri指向它）

**表现层**

‘资源’是一种信息实体，它可以有多种外在的表现形式，我们把‘资源’具体呈现出来的形式，叫做它的表现层。

如：文本可以用txt表示，也可以用xml，json，二进制表现。

uri只表示资源的实体，并不代表他的形式。

**状态转化**

访问网站，实际上就代表了客户端和服务器的互动，这个过程中，势必会涉及数据和状态变化。

而http协议是一个无状态协议，因而所有状态都保存在服务器端（？cookie ，session）

# SOA

## 定义

面向服务的架构

# 分布式计算

定义

分布式计算：

两个以上的软件互相分享信息，这些软件既可以在一台计算机上运行，也可以通过网络连接起来的多台计算机上运行。

平衡负载

稀有资源共享

将程序放在最适合运行它的计算机上

# 量子算法

相关

由于：量子质因子分解算法的出现，对RSA加密算法构成威胁

但是由于微观上操作量子态的操作十分困难，真正的量子计算器实际并没有实现。

# RSA（公钥加密算法）

## 基本原理

两个互质的数相乘很容易，但要对其进行因式分解及其困难。

## 详细原理

### 互质关系

两个正整数除了1，没有其他共因子。这两个数就是互质关系。

关于互质关系的结论

　　1. 任意两个质数构成互质关系，比如13和61。

　　2. 一个数是质数，另一个数只要不是前者的倍数，两者就构成互质关系，比如3和10。

　　3. 如果两个数之中，较大的那个数是质数，则两者构成互质关系，比如97和57。

　　4. 1和任意一个自然数是都是互质关系，比如1和99。

　　5. p是大于1的整数，则p和p-1构成互质关系，比如57和56。

　　6. p是大于1的奇数，则p和p-2构成互质关系，比如17和15。

欧拉函数

计算任意正整数n，小于的等于n的，与n构成互质关系的数的个数。

如n=8 φ(n) = 4 1,3,5,7

# 对称加密和非对称加密

## 对称加密

甲方乙方使用相同的加密规则即是对称加密

缺点：甲方必须要把加密规则告诉乙方，否则无法加密，保存和传递秘钥，就是个大问题。

## 非对称加密

加密和解密实际上可以使用不同规则，只要这两种规则之间存在某种对应关系即可。（这样就可以避免直接传递秘钥）

乙方生成两把秘钥（公钥， 私钥）。公钥公开（任何人可得），私钥则保密。

甲方获得乙方的公钥，然后对信息加密

乙方得到加密后的信息，用私钥解密

公钥加密的信息只有私钥才能解开，只要私钥不泄露，通信即是安全的。

# OpenGL ES

OpenGL ES(OpenGL for Embedded Systems)

是OpenGL ES 是 OpenGL 三维图形api的子集。针对手机、PDA 游戏主机等嵌入式设备而设计。

# 线程

## 单线程

优势：不必像多线程编程那样处处关注状态的同步问题，这里没有死锁的存在，也没有线程上下文交换带来的性能上的开销。

弱点：

无法利用多核CPU

错误会引起整个应用退出（应用的健壮性）

大量计算占用CPU导致无法继续调用异步 I/O

# 阻塞与异步

## 阻塞

线程在遇到耗时操作（磁盘读写，网络通讯I/O操作），操作系统会剥夺这个线程的CPU控制权，使其暂停执行，同时将资源让给其他工作线程。

## 异步

异步 I/O不阻塞： 将I/O请求发送个操作系统，继续执行下一个语句。当操作系统完成 I/O操作时，以事件的形式通知执行I/O操作的线程，而线程存在事件循环，不断检查有没有未处理的事件，依次处理。

# 网络传输名词

## uri

统一资源标识符

url

统一资源定位符

urn

统一资源名

## http事务

由一条请求命令和一个响应结果组成

网络分层

## web结构组件

代理

客户端和服务器之间的HTTP中间实体

缓存

HTTP的仓库，使常用页面的副本可以保存在里客户端更近的地方

网关

连接其他应用程序的特殊web服务器（如 http /ftp网关 将协议的流量相互转换）

隧道

对http通信报文进行盲转发的特殊代理

可以在一条或多条http连接上转发非http数据，转发是不会窥探数据（也就是说这些数据不会被只允许web流量通过的防火墙屏蔽）

Agent代理

发起自动http请求的半智能web客户端

如web浏览器 ， 网络机器人

# 版本控制

ps: 所有的版本控制系统，实际上只能跟踪文本文件的改动，而图片，视频这些二进制文件，能管理（只能讲二进制的每次改动串起来，也就是说只知道图片从100k到120k），但是没法跟踪文件的变动。而 word格式是二进制的，因此无法跟踪word文件的改动。

注意：

不要用Windows自带的记事本编辑文本文件，因为Microsoft会将每个文件开通添加了

oxefbbbf（16进制）的字符，用来保存utf-8文件。

## 集中式版本控制系统

如： svn

版本库存放于中央服务器。

将资源从中央服务器获取，然后修改，修改后推送到中央服务器。

弱点：

1.必须联网，非局域网下提交，下载速度感人。

## 分布式版本控制系统

如：git

分布式版本控制系统没有“中央服务器”，每个人的电脑上都是一个完整的版本库，

1. B修改了文件，他们会将各自的修改推送给对方，就可以看到对方的修改。

其实git系统通常也有一台充当“中央服务器”，用来交换各自的修改。当然没有这个，也能工作，只是推送修改不方便而已。

优点：

1. 安全，没有“中央服务器”，一个挂了，还有其他版本库
2. 强大的分支管理

# 字符编码

# Blob

Blob对象表示不可变的类似文件对象的原始数据。浏览器有对其相互转化的接口（Blob对象）

file对象是特殊类型的Blob。

二进制大对象，是一个可以存储二进制的容器，常用语存储二进制文件的字段类型。

text/plain 转换为 blob

xhr.onreadystatechange = function(e) {

if (this.readyState == 4 && this.status == 200) {

//通过 responseText 来获取图片文件对应的二进制字符串

var binStr = this.responseText;

//然后自己再想方法将逐个字节还原为二进制数据

for (var i = 0, len = binStr.length; i < len; ++i) {

var c = binStr.charCodeAt(i);

//String.fromCharCode(c & 0xff);

//这里是将其转为 32 位的补码，实际上仍然表示c这个数。

var byte = c & 0xff;

}

}

};

# Buffer

## 对比cache

cache是为了弥补高速设备与低速设备的鸿沟而引入的中间层，最终起到加速访问速度的作用

而buffer主要的目的是进行流量整形，把突发的大数量较小规模的I/O整理成平稳的小数量较大规模的I/O，以减少响应次数

1、**Buffer**（缓冲区）是系统两端处理**速度平衡**（从长时间尺度上看）时使用的。它的引入是为了减小短期内突发I/O的影响，起到**流量整形**的作用。比如生产者——消费者问题，他们产生和消耗资源的速度大体接近，加一个buffer可以抵消掉资源刚产生/消耗时的突然变化。  
2、**Cache**（缓存）则是系统两端处理**速度不匹配**时的一种**折衷策略**。因为CPU和memory之间的速度差异越来越大，所以人们充分利用数据的局部性（locality）特征，通过使用存储系统分级（memory hierarchy）的策略来减小这种差异带来的影响。  
3、假定以后存储器访问变得跟CPU做计算一样快，cache就可以消失，但是buffer依然存在。比如从网络上下载东西，瞬时速率可能会有较大变化，但从长期来看却是稳定的，这样就能通过引入一个buffer使得OS接收数据的速率更稳定，进一步减少对磁盘的伤害。  
4、TLB（Translation Lookaside Buffer，翻译后备缓冲器）名字起错了，其实它是一个cache.

## ArrayBuffer

### 意义

由于js的Array，有许多功能，而且是不限制类型，也可能是稀疏的。（是托管的，内部有比较复杂的实现），如果从XHR, File API , Canvas等等地方读取了一大串字节流，如果用js里的Array去存，性能消耗高，低效。

### 含义

ArrayBuffer是一块内存

var buf = new ArrayBuffer(1024) ：开辟1kb的内存

但是不能直接访问Arraybuffer里面的字节

通过TypedArray，提供一个‘view’（multiple views on the same data），对它们进行下标读写，最终都会反映到它所建立的ArrayBuffer上

如：

var int8 = new Int8Array(buf)

int8[0] =12;

# Source Map (js)

## 作用

由于前端会对代码进行压缩，合并，转码，这会导致代码的报错信息很难被定位。这时，source map可以有效还原报错位置。map文件只会在开发模式被加载，普通浏览不会加载map，不会浪费宽带和流量。

## 包含信息

map文件是一个JSON文件，包含信息

version 版本

sources 转换前文件列表

names 转变前所有变量名和属性名

mappings: 记录位置信息的字符串

souceRoot: 转换前的文件所在的目录。如果与转化前是同一目录，则为空。

sourcesContent 转换前的文件内容列表（与source一一对应）

## 如何映射map

在转换后的代码的尾部指定map路径即可：

//@ sourceMappingURL=/path/to/file.js.map

# RFC

request for comment 征求意见稿

最终演变为记录互联网规范、协议、过程的标准文件

# 协议簇

## UDP

User Datagram Protocal 用户数据报文协议 面向数据报的传输层协议

在 TCP/IP模型中，UDP为网络层以下和应用层以下提供了一个简单的接口。

是不安全传输

## ICMP

Internet Control Message Protocol 互联网协议簇的核心协议之一

用于TCP/IP网络中发送控制信息，提供可能发生在通信环境中的各种问题反馈。

每个ICMP信息都是直接封装在一个IP数据报中，但对ICMP信息会特殊处理，于一般的IP数据包的处理不同(不会作为IP的子协议处理)

也是不安全传输

## GRE

RFC 2784

Generic Routing Encapsulation

一种隧道协议，可以在虚拟点对点链路中封装多种网络层协议。

## Telnet

RFC 854

一种应用层协议（适用于互联网和局域网），使用虚拟终端机的形式，提供双向，以文字字符串为主的交互功能。属于TCP/IP协议簇的其中之一。是Internet远程登录服务的标准协议和主要方式

而传统的talnet会话所传输的数据并未加密，账号和密码等敏感数据容易被窃听，因而一般服务器会封锁Telnet服务，改用更安全的ssh

## RDP

Remote Desktop Protocol 远程桌面协议

一个多通道协议，让本地电脑连接终端机服务，服务端监听3389端口。

# 柯里化

在一个函数中首先填充几个参数（然后在返回一个新函数）的技术就是柯里化

偏应用方程返回一个含有预处理参数的新函数

# 重绘与回流

关于浏览器渲染

页面时 dom树 + 样式 来确立盒模型

解析html文档时，会先查看link的css，等到css加载完后才会渲染dom树。（这也是为什么需要将css写在页面顶部）

repaint --重绘

浏览器收到元素产生了不影响排版的变化，对这个元素进行重新绘制的过程（如加一个下划线，改变颜色）

reflow -- 回流

浏览器收到元素产生了对文档树排版有影响的样式变化，会对所有受到影响的dom节点进行重新排版工作。一般一个reflow后都会有repaint

# 网络安全

0day漏洞

指的是：

代码发布到漏洞补丁发布之间的时间

1day漏洞

指的是：

漏洞补丁发布的第一天。（许多用户还没来得及打上补丁，攻击者可以对补丁进行差量分析，从而猜测漏洞原理）

# 线程和进程

# 规范

## AMD

Asynchronous Modules Definition

异步模块定义

define(id?, dependencies?, factory)

define(function(){

var exports = {}

exports.sayHell = function(){}

return exports

})

## CMD

在定义模块时，指定所有依赖。

define([‘dep1’, ‘dep2’], function(dep1, dep2){

return funciton(){}

})