- 架构
  - 分层架构
    - 结构
      - 表现层: 用户界面
      - 业务逻辑层: 业务逻辑
      - 服务层: 通过不同业务逻辑通用接口
      - 持久层: 提供数据 sql 语句就放在这一层
      - 数据库层: 保存数据
    - 优缺点
      - 优点
        - 结构简单
        - 易于分工, 每人负责不同的层
        - 每一层可以独立测试, 其他层的接口通过模拟解决
      - 缺点
        - 一旦环境变化, 需要代码调整或增加功能时, 通常比较麻烦和费时
        - 部署比较麻烦, 即使只修改了一个小地方, 往往需要整个软件重新部署, 不容易做持续部署
        - 软件升级时, 可能需要整个服务暂停
        - 扩展性差, 用户量大量增加是, 必须扩展每一层, 由于每一层内部是耦合的, 扩展很困难
  - 事件驱动架构
    - 结构
      - 事件队列: 接收事件的入口
      - 分发器: 将不同的时间分发到把不同业务逻辑单元
      - 事件通道: 分发器与处理之间的联系渠道
      - 事件处理器: 实现业务逻辑, 处理完成后会发出事件, 触发下一步操作
    - 优缺点
      - 优点
        - 分布式的异步架构, 时间处理器之间高度解耦, 软件的扩展性好
        - 适用性广, 各类型的项目都可以用
        - 性能较好, 异步不会堵塞
        - 事件处理器可以独立的加载和卸载, 容易部署
      - 缺点
        - 异步编程(考虑远程通信, 失去响应), 开发复杂
        - 难以支持原子性操作, 事件通过会涉及多个处理器, 很难回滚
        - 分布式, 异步式导致难以测试
  - 微核架构
    - 结构
      - 内核
      - 插件
    - 优缺点
      - 优点
        - 良好的功能延伸性, 需要什么功能, 开发一个插件即可
        - 功能之间是隔离的, 插件可以独立的加载
        - 可定制型高, 适应不同的开发需要
        - 可以渐进式的开发, 逐步增加功能
      - 缺点
        - 扩展性差, 内核通常是一个独立的单元, 不容易做成分布式
        - 开发难度相对较高
  - 微服务架构
    - 实现模式
      - RESTful API 模式: 服务通过 API 提供, 云服务就属于这一类
      - RESTful 应用模式: 服务通过传统的网络协议或者应用协议提供, 背后是一个多功能的应用程序
      - 集中消息模式: 采用消息代理, 可以实现消息队列, 负载均衡, 统一日志和异常处理
    - 优缺点
      - 优点
        - 扩展性好, 各个服务之间低耦合
        - 容易部署, 软件从单一可部署, 被拆成了多个服务, 每个服务都是可部署单元
        - 容易开发, 每个组件都可以进行持续集成式的开发, 可以做到实时部署, 不间断的升级
        - 易于测试, 可以单独测试每一个服务
      - 优缺点
        - 由于强调互相独立和低耦合, 服务可能会拆分的很细, 这导致系统以来大量的微服务, 变的很凌乱和很笨重, 性也会不佳
        - 一旦服务之间需要通信, 整个架构就变得复杂, 典型的例子就是通用类, 解决方法就是拷贝到每一个服务中去, 用冗余换取架构的简单性
        - 分布式的本质使得很难实现原子性操作, 交易回滚会比较困难
  - 云架构

- python
  - 爬虫

- 技能树
  - php
    - 开发环境
      - Vagrant搭建lnmp环境, 模拟完整线上环境
    - 框架
      - 深入了解laravel服务容器环境, 设计模式, 封装laravel 扩展包, 发布composer包
    - 设计模式
      - 创建型
        - 单例模式
          1. final class
          2. 私有化构造方法
          3. 懒加载获得实例
          4. 防止实例被克隆和实例化
        - 多例模式
          1. final class
          2. 私有化构造方法
          3. 懒加载获得实例
          4. 防止实例被克隆和实例化
      - 结构型
        - 流接口模式
          - 方法链
        - 数据映射模式
          - 目的是为了关系型数据库, 内存中的数据, 数据映射本身三者相互独立
          - ORM: 一个对象对应一条数据库记录, 对象属性对应记录的字段, 对象属性改变时, 自动更新数据库
        - 依赖注入 – 控制反转
          - 一般调用, 某个类时, 需要先实例化, 依赖注入就是实例化被调用者的工作交给IOC容器处理
      - 行为型
        - 访问者模式
          - 结构
            - 数据结构
            - 结点对象
            - 访问者接口
            - 访问者实现
          - 让外部对象能够获取树型结构的每个结点的对象, 对每个对象进行操作的模式, 解耦数据结构和数据操作
        - 观察者模式
          - 定义对象间一对多的关系, 当一个对象状态发生改变时, 所有依赖于它的对象都得到通知并自动更新, 也叫发布-订阅模式
    - 异常处理
      - Php默认情况下, 在代码出现了错误, 如notice warning等消息, 错误消息会直接打印到浏览器上, 这个时候通过try catch是捕获不到错误信息的
      - set_exception_handler: 自定义异常处理函数
      - register_shutdown_function: 当脚本执行完成时,意外死掉导致php进程结束,这个函数会调用, 该函数是从内存中调用的, 要写绝对路径
      - display_error: 是否将错误显示在屏幕上
      - error_reporting: 设置php的报错级别
    - 运行流程
      - cgi
        - 协议, fork execute, 每次请求启动一个进程来处理
        - PHP–CGI: cgi协议的实现, 只会处理解析请求,返回结果, 不会管理进程
      - fast-cgi
        - 一种协议, 通过守护进程来管理cgi进程, 将cgi程序常驻与内存中, 不必每次都初始配置和扩展, 提高cgi性能
        - php–fpm: fast–cgi协议的实现, 负责管理进程池, 本身不直接处理请求, 会交给php–cgi去处理
      - zend内核
    - web安全
      - sql注入
        - 定义: 通过操作输入来修改后台sql语句执行攻击
        - 方式:
          - and/or 大法
          - 字符串注入, #``'—`
        - 子查询
      - xss攻击
        - 向有漏洞的网站中插入恶意的html代码, 用户浏览该网站时, html代码自动执行, 如获取cookie信息, 破坏页面结构, 重定向其他网站

- mysql
  - 查看mysql版本:
    1. show version()
    2. status
  - 驱动程序/JDBC(java database connection)
    1. 注册加载一个驱动
    2. 创建数据库连接
    3. 构造sql语句
    4. 创建statement, 发送sql语句
    5. 执行sql语句
    6. 处理sql结果
    7. 关闭statement和connection
  - 存储引擎
    - 定义: 帮助mysql高效获取数据的数据结构
    - InnoDB：聚集索引
      - 支持行锁, 表锁, 外键约束
      - 事务隔离级别
        - 级别
          - 读未提交: 事务1能读取到事务2未提交的修改
          - 读已提交: 事务1能读取到事务2提交后的修改
          - 可重复读: 当前会话可以重复读, 每次读取的结果集都相同, 不管其他事务有没有提交
          - 串行化: 当设置为串行时, 其他事务对该表的写操作都会被挂起
        - 副作用
          - 脏读: 事务1能读到事务2未提交修改, 当事务2回滚时会造成脏读
          - 不可重复读: 事务1连续两次读同一条记录可能因为其他事务修改导致读取到的结果不一致
          - 幻读: 事务1对数据进行修改, 涉及到全部的数据行, 事务2新增了一条记录
      - InnoDB的数据文件就是主索引
      - InnoDB的辅助索引 引用主键作为key, 辅助索引搜索会检索两遍
      - 过长的主索引会使辅助索引变得过大
    - MyISAM：非聚集索引
      - MyISAM的索引文件和数据文件是分离的
      - 索引 值 域 保存数据的地址
      - MyISAM的主索引和辅助索引没有区别, 主索引只是唯一的key, 辅助索引可以多个key
    - Heap
      - 内存表使用哈希散列索引把数据保存在内存中, 表结构存放在磁盘, 扩展名为.frm, 重启不会丢失
      - 内存表存放在内存中, 重启会丢失
      - 对所有用户是可见的, 适合做缓存
      - 内存表不支持事务, 内存是表锁, 不易修改频繁
      - 不支持blob, text, 无法使用vachar与text
  - 键
    - 超键: 唯一标识属性的集合/联合主键
    - 候选键: 超键集中的属性
    - 主键: 候选键里面的一个
    - 外键: 表里面的属性是另一张关系表的主键
  - 分析/检查/优化
    - 分析表
      - analyze table 表1
      - 分析过程中, 会对表加只读锁
      - 更新索引, 改善性能
      - mysqlcheck –a 数据库名 表1 表2 –u root –p123456
        - 只能对 MyISAM 表器作用
    - 检查表
      - check table 表1 表2
      - 检查InnoDB 和 MyISAM 是否存在错误, 检查视图是否存在错误
    - 优化表
      - optimize table 表1
      - 可以消除删除和更新造成的磁盘碎片, 会加只读锁
  - 性能分析/优化
    - 查询与索引优化
      - 状态检查
        - 显示状态信息: show status;
        - 显示系统变量: show variables;
        - 显示存储引擎状态: show engine innodb status\G;
        - 查看当前sql执行, 包括执行状态, 是否锁表: show processlist
      - 慢查询日志
        - 检查是否开启慢sql日志: show variable like 'slow_query_log'
        - 动态开启慢sql日志: set global slow_query_log=ON;
        - 设置 慢查询 耗时时长: set long_query_time=100（秒）
        - 慢日志查看: more /var/lib/mysql/localhost-slow.log
    - explain
    - 参数配置
      - 连接请求的变量
        - max_connections
          - 响应的连接数: max_used_connections
            - show status like 'max_used_connections'
          - 最大连接数: max_connections
            - Show variables like 'max_connections'
          - 理想值: max_used_connections / max_connections * 100% = 85%
        - back_log
          - Mysql能暂存的链接数
          - Show full processlist; 发现大量带连接进程时, 杀掉阻塞的进程, 加大back_log的值
          - 默认值是 50, 可以调优为128
        - interactive_timeout
          - 交互连接在被服务器关闭前等待行动的秒数
          - 默认数值是28800, 可调优为7200
      - 缓冲区变量
        - key_buffer_size：
          - 索引缓冲区的大小, 决定索引处理的速度, 尤其是索引读的速度
          - key_reads / key_read_requests 尽可能的低, 至少 1:100
        - query_cache_size:
          - 查询缓冲，将查询结果放在缓冲区, 对于同样的select语句(区分大小写), 直接从缓冲区读
  - sql优化
    - 索引字段尽量小
    - 最左前缀匹配原则
      - 表c2c_db.t_credit_detail建有索引(`Flistid`,`Fbank_listid`)
      - select * from t_credit_detail where Fbank_listid='200199'
      - 该sql直接用了第二个索引字段Fbank_listid，跳过了第一个索引字段Flistid，不符合最左前缀匹配原则。
    - 索引字段尽量选择区分度高的
    - 扩展索引可以在原有索引的基础上修改索引
      - A字段 有索引, 现在要加(A,B)索引, 可以直接修改 A字段原有的索引
    - 避免索引列参与运算或使用函数, 会放弃使用索索引
    - 所查皆所需, 避免select *
    - join 联合的字段类型尽量一致, 并添加索引
    - Like %不出现在开头可以使用索引
    - 使用exists代替in
      - select num from a where num in (select num from b)
      - Select num from a where num exists(select num from b where num = a.num)
    - distinct 可以替代 group by
      - Select order_id from order where price > 10 group by order_id
      - select distinct(order_id) from order where price > 10
    - 避免使用 != 或 <>，> 或 < 更高效, <> 会放弃使用使用索引而使用全局扫面
    - 可以使用 union 替代 or, 否则会放弃使用索引
      - select id,name from user where name = "abc" or name = "lmn"
      - Select id, name from user where name = "abc" union select id, name from user where name = "lmn"
  - 技术特点
    - 关系型数据库管理系统
    - 开发语言c/c++
    - 良好的内存管理机制
    - 核心线程是完全多线程, 支持多处理器
    - 多种数据类型: signed/unsigned, int, double, float, blob
    - 支持多平台
    - B+Tree数据结构, 索引
    - 所有列都有默认值
  - 函数
    - length/char_length
      - Lenght: 一个汉字算三个字符, 一个数字一个字符算一个字符
      - char_length: 汉字, 字母, 数字 都算一个字符
  - 正则(REGEXP)
    - `Select name from user where name regexp "^st"`
  - 列类型
    - 字符串类型
      - varchar: 变长字段类型, 字符串长度 = 1(表示字符串长度)
      - char : 定长字段类型, char(M) 每个值都是M个字节, 不足前面补空
      - text
      - blob
      - enum
      - set
    - update current_timestamp: 记录被修改时自动更新timestamp时间为当前时间

- HBase
  - 定义