补充

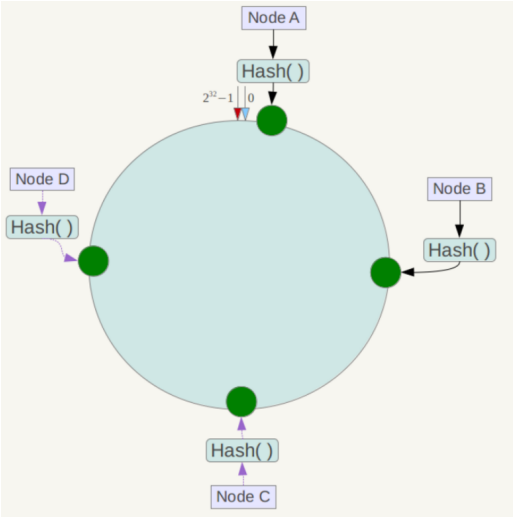
1. 一致性hash算法

传统方式是通过对对象的值取hash然后取模，映射到相应的桶。这样如果节点个数变动，绝大部分的映射关系会失效。提出一致性哈希解决节点数目变动时映射关系失效的情况。

分布式系统每个节点都有可能失效而且有节点的动态增加。

1. 具有的特性
    1. 平衡性：将结果尽可能分不到所有的缓冲中，可以使得所有缓冲空间都得到利用
    2. 单调性：hash结果保证原有已分配的内容可以映射到新的缓冲区
    3. 分散性：终端看不到所有的缓冲，只能看到一部分
    4. 平滑性：缓存服务器的数目平滑改变和缓存对象的平滑改变一致
2. 一致性hash将整个hash值空间组织成圆环

    然后将各个服务器使用hash函数进行计算（可以使用IP/主机名），确定在hash环上的位置

    将数据key使用相同的hash函数计算出hash值，确定数据在环上的位置，顺时针行走，第一台遇到的服务器就是定位到的服务器
3. 如果hash环中服务器节点受损，仅会影响此节点到逆时针上一节点之间的访问数据，其他不会影响，因此具有良好的容错性和可扩展性
4. 如果服务器节点较少，很容易出现数据访问集中的情况（数据倾斜），因此引入虚拟节点机制。对每一个服务节点计算多个hash，称之为虚拟节点。

2. Java和C++的区别

1. Java是完全面向对象的语言，所有函数和变量都是类的一部分。更加简洁
2. Java去除了所有的指针使用，避免了野指针的出现导致系统崩溃
3. Java去除了多重继承
4. Java有内存回收机制，进行自动内存管理
5. 不支持操作符重载
6. 存在goto关键字但是不支持使用
7.

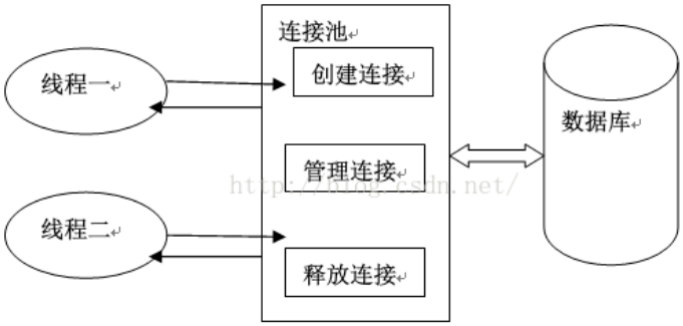
3. 负载均衡算法

1. 轮询法：请求顺序轮流分配到后端服务器上
2. 随机法：随机选取一台服务器进行访问，数量大时接近于轮询
3. 源地址hash法：根据客户端IP地址，通过hash函数计算得到值，对服务器数进行取模，然后得到访问服务器序号。（客户端IP每次访问均会访问到同一台服务器）--一致性hash算法
4. 加权轮询法：不同服务器由于配置和当前负载不相同，因此给配置高/负载低的机器配置更高权重，使其更有可能被轮询到
5. 加权随机法：根据后端机器的配置，为负载分配权重，按权重随机分配服务器
6. 最小连接数法：根据服务器当前连接数进行动态的选取当前积压连接数最小的一台服务器进行处理

4. 数据库连接池

负责分配、管理、释放数据库连接

初始化时创建一定数量的数据库连接放到连接池中，连接池保证至少拥有这么多的连接数量。当应用程序的链接数超过了最大数量时，将请求加入到等待队列中

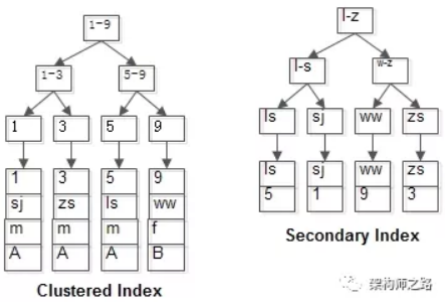


概念

1. 最小连接数：连接池一直保持的数据库连接，所以如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费。
2. 最大连接数：是连接池能申请的最大连接数，如果数据库连接请求超过此数，后面的数据库连接请求将被加入到等待队列中，这会影响之后的数据库操作。
3. 相差：如果相差大，最先的链接请求获利。超过最小时，再请求需要重新建立数据库连接，需要一定的开销

5. 数据库索引

1. 回表查询
    1. 聚集索引：仅有一个，使用候选键/生成唯一键进行 -- 稀疏索引
    2. 普通索引：叶子节点存储主键值 -- 稠密索引
    3. 从普通索引无法定位行记录，需要使用聚集索引找到行指针
    4. 回表查询：使用非聚簇索引找到主键值，然后使用聚簇索引找到主键对应的指针
    5. 非聚簇索引也不一定触发回表，因为可能只需要该键值就可返回



2. hash索引
    1. 局部性无法体现：例如查询AA，AB需要多次查询次数
    2. 尽管查询速度为O(1)，但是如果键值大量碰撞，复杂度可能到O(n)
    3. 不支持使用索引进行排序
    4. 必须进行回表查询
3. B+树
    1. B+树支持临近查询（体现了局部性）
    2. 如果满足（聚簇索引，覆盖索引时），可以不进行回表查询

       聚簇索引：表数据按照索引顺序存储，一个表只能有一个聚簇索引

       非聚簇索引：无关，叶子节点包含索引字段值和指向数据页的逻辑指针、

       InnoDB只有主键索引时聚簇索引，如果没有主见则寻找/生成唯一键
    3. 查询效率稳定

6. Mysql的锁机制

共享锁、排他锁

InnoDB使用行级锁和表级锁，默认行级锁

在不适用索引的时候，InnoDB使用表锁，而不是行锁

使用任何索引，InnoDB会添加行级锁

MylSAM不存在死锁：因为他一次性分配所有锁，要么全部满足，要么全部等待

InnoDB存在死锁：锁逐步获得，可能存在死锁（事务释放锁并回退）

1. 共享锁--S锁：对数据进行读操作，多个事务可以同时添加共享锁
2. 排他锁--X锁：对数据进行写操作，不可再添加其他锁
3. 行锁：对一行记录添加锁，只会影响一条记录（InnoDB对索引加锁）
4. 锁的粒度划分
    1. 表级锁

       开销小，加锁快，不会出现死锁，力度大，并发度低
    2. 行级锁

       开销大，加锁慢，所得粒度最小，并发度高

       （共享锁、排他锁）
    3. 页面锁

       介于表索和行锁之间，并发度一般
5. 间隙锁：在辅助索引上，只会阻塞insert操作

7. 倒排索引

根据属性的值来查找记录

8. Linux系统数据包转发

1. 常见的防火墙：3、4层的防火墙：网络层的防火墙，7层的防火墙：代理层的网关

9. 单例模式的销毁（garbage collection）

猜测：如果单例的某些函数中打开了文件、进行了动态内存分配，那么就必须在析构函数中进行解构，但是由于是动态申请的空间，

```
class singleton {
private:
    singleton() {}
    static singleton* sig;
    ~singleton() { cout << "desconstruction!"; }
public:
    static singleton* get_instance()
    {
        if (sig == NULL)
            sig = new singleton();
        return sig;
    }
    friend class garbage_collector;
};
singleton* singleton::sig = nullptr;


class garbage_collector {
public:
    ~garbage_collector()
    {
        if (singleton::sig!=nullptr)
        {
            delete singleton::sig;
            cout << "111";
        }
    }
};
static garbage_collector gc;
```

常用的销毁方式：限于使用new创建的单例，将删除的操作挂载到操作系统的某个点上，即一个static对象的销毁（程序结束时会自动销毁静态成员变量）

如上图所示，使用garbage collection的销毁调用单例的销毁

10. magic static的弊端

1. 编译器会为类自动生成一个默认的构造函数。来支持类的拷贝
2. 需要禁止类拷贝和类赋值
3. 解决方式
    1. 引用改指针
    2. 重载拷贝构造函数和=操作符，同时声明为私有

11. map和set的区别

由于关联式容器不存在内存移动的问题，所以只会存在当前迭代器失效的问题，使用erase(iterator++)即可解决

12. map和set和自定义数据类型

1. map可以重载自定义对象的<号
2. 使用仿函数作为map的第三个参数自定义比较

二者必须必须const函数

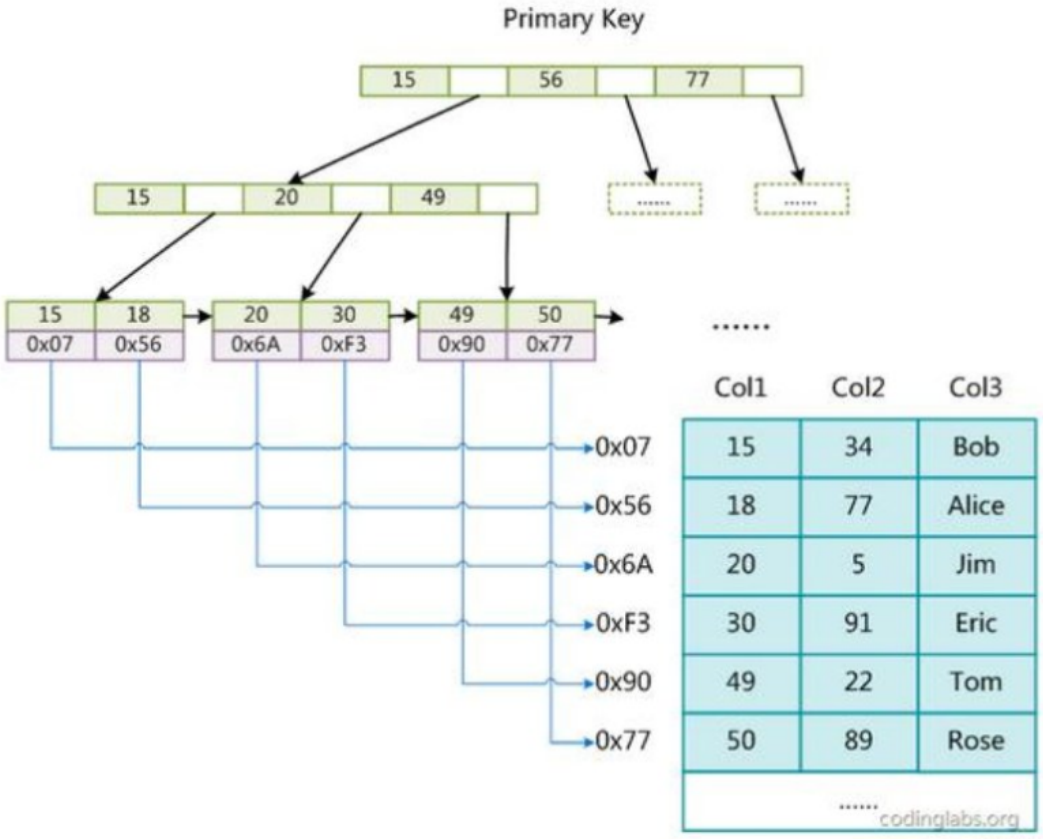
13. MYSQL索引实现原理

底层索引结构使用B+树实现，以进行全局遍历时出现全表扫描的情况

1. MYLSAM引擎使用B+树作为索引节点，叶节点的data域存放数据记录的地址，联合索引存放以节点为开始的联合索引、

   MylSAM的索引文件和数据文件分离

   非聚集索引（相邻的数据存储是分散的）



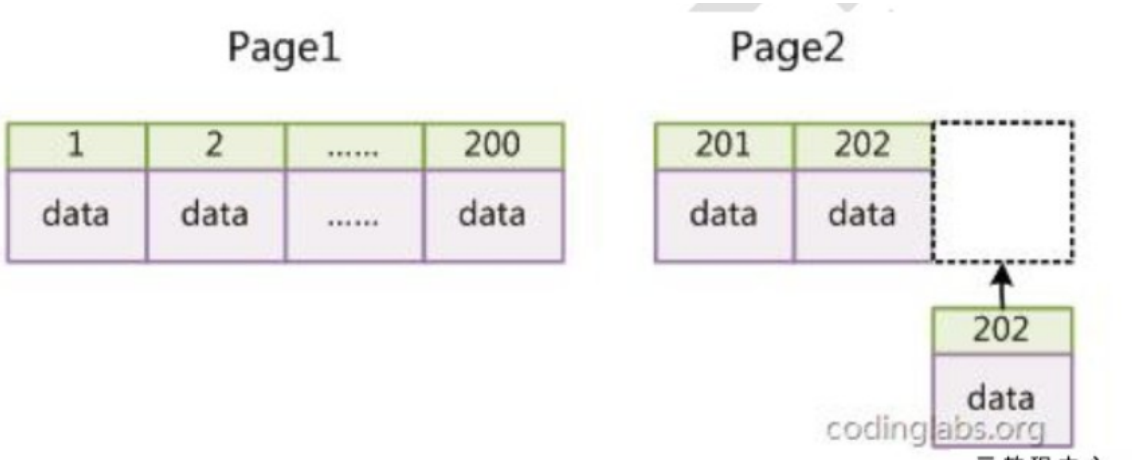
MylSAM辅助索引：主索引和辅助索引在结构上没有区别，但是辅助索引的key可以重复

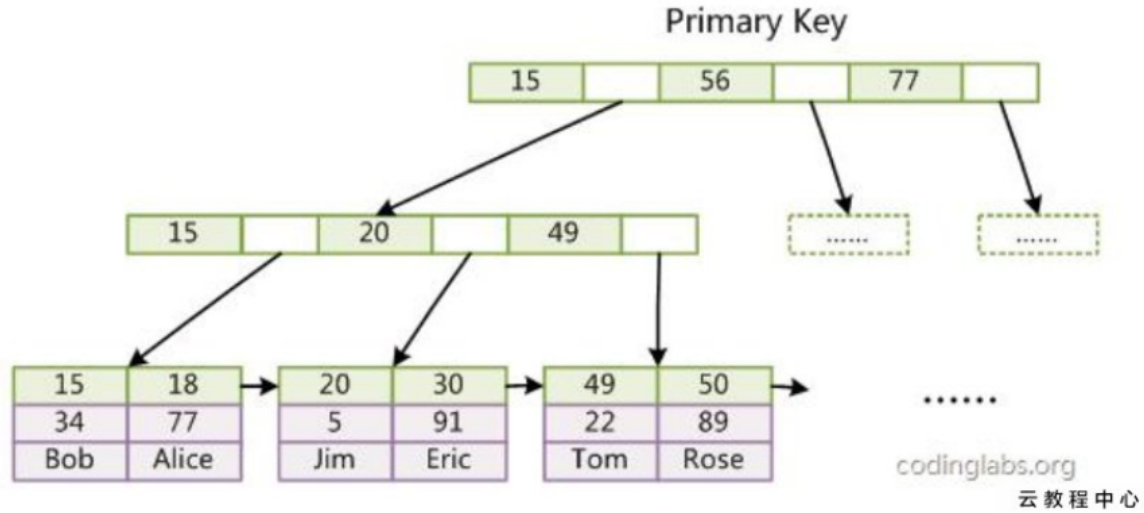
2. InnoDB索引实现(叶节点保存了完整的数据记录)

   索引要求必须建立在主键之上（如果没有则创建一个unique的属性）

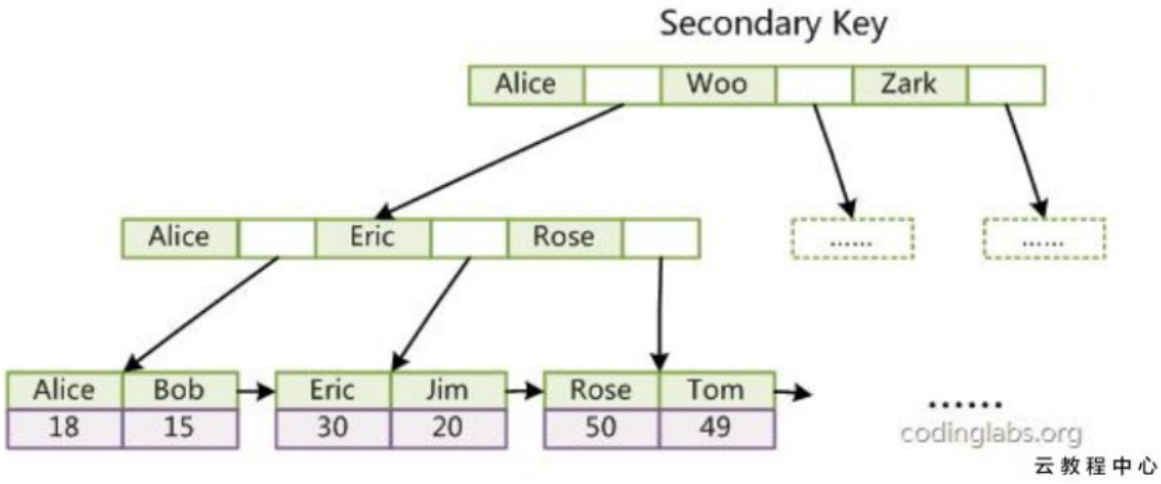
   InnoDB的数据文件本身就是索引文件，

   聚集索引：相邻的数据是连续存储的（尤其主键是自增ID的时候）

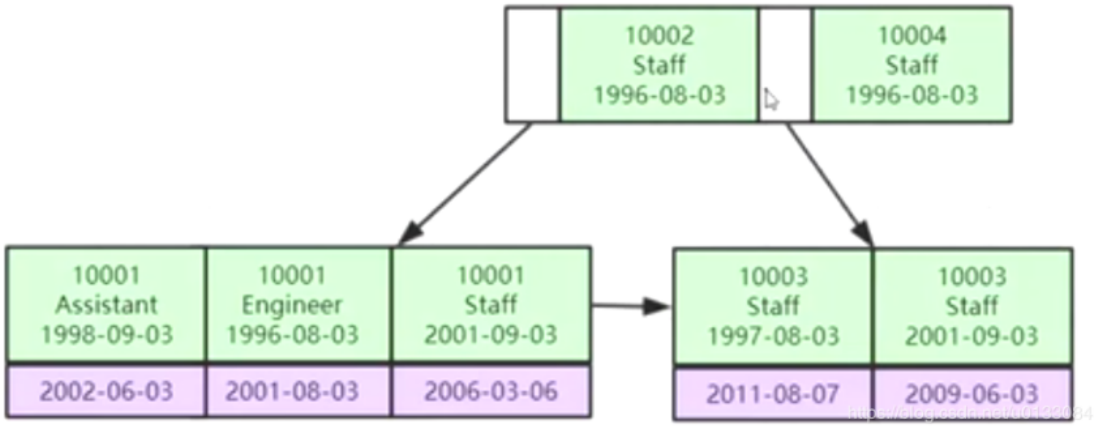




InnoDB所有的辅助索引都引用主键作为data域，即使用其他属性建立索引的时候，找到的是主键的值，如果还需要其他属性，则需要进行回表扫描去查找



3. 联合索引以及最左原则



假设建立了ABC三个属性的联合索引

（A，，）---会使用索引

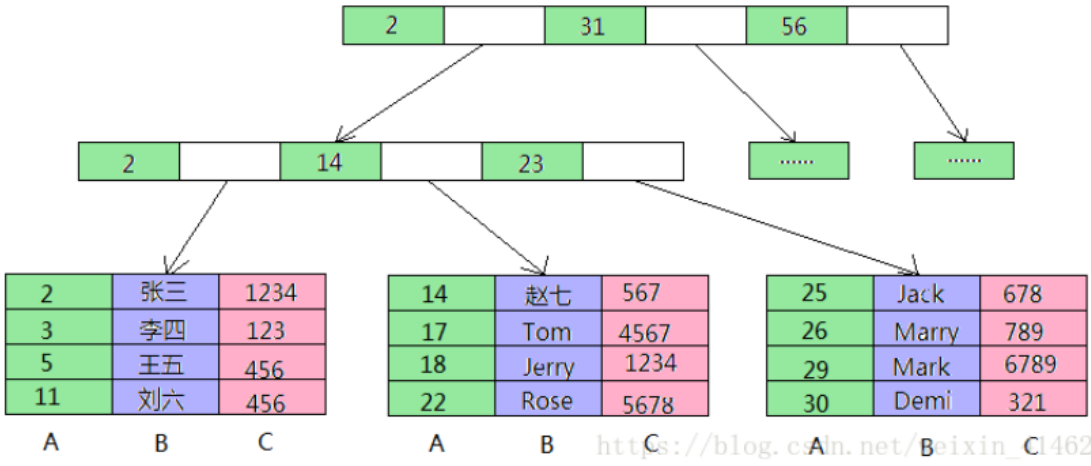
（A，B，）---会使用索引

那么（A，B，C）---会使用索引

（，B，C）---不会使用索引

（，，C）---不会使用索引

非叶子节点存储最左属性，按照最左属性的值建立B+树，在叶结点存储全部关键字的内容



14. 单继承与多继承

1. 单继承

   派生类继承基类的虚函数表项，如果重写则进行覆盖

   如果没有重写则直接继承，此外将新定义的虚函数添加到虚函数表末尾

2. 多继承（含有多个虚函数指针和多个虚函数表）

   1. 非菱形继承（菱形继承需要使用虚继承，使得每一份数据对象在派生类中只保留一份）

      1. 子类新增的虚函数放在第一个父类的虚函数表处
      2. 子类重写虚函数，所有父类的同名函数均被重写，虚函数表项全被更改

3. 父类按照声明顺序排列

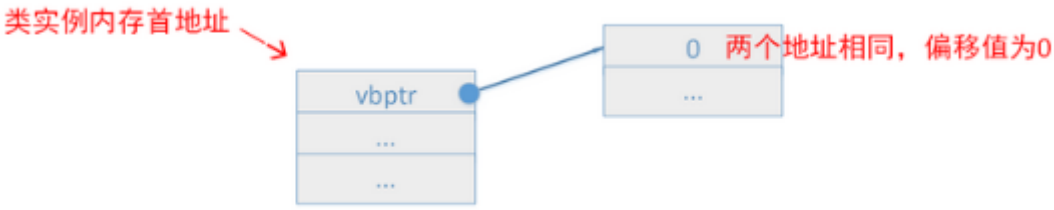
2. 虚继承

解决了菱形继承中：最末派生类含有多个间接父类的情况

1. 虚继承的子类，如果本身定义了新的虚函数，则生成一个新的虚函数指针和一张虚函数表，在对象最前面（非虚继承直接扩展父类虚函数表）
2. 虚继承子类保留父类的虚函数指针与虚函数表
3. 虚继承的子类有虚基类表指针

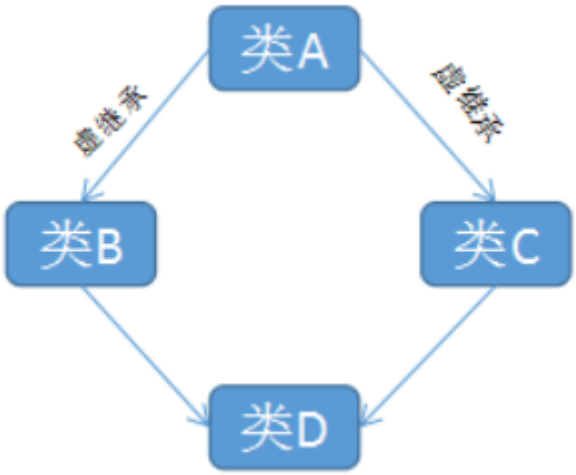
虚继承的派生类会生成一个虚基类指针vbptr，常在虚函数表之后，

类实例含有vptr的情况



类实例不含vptr的情况

虚基类表存放偏移值

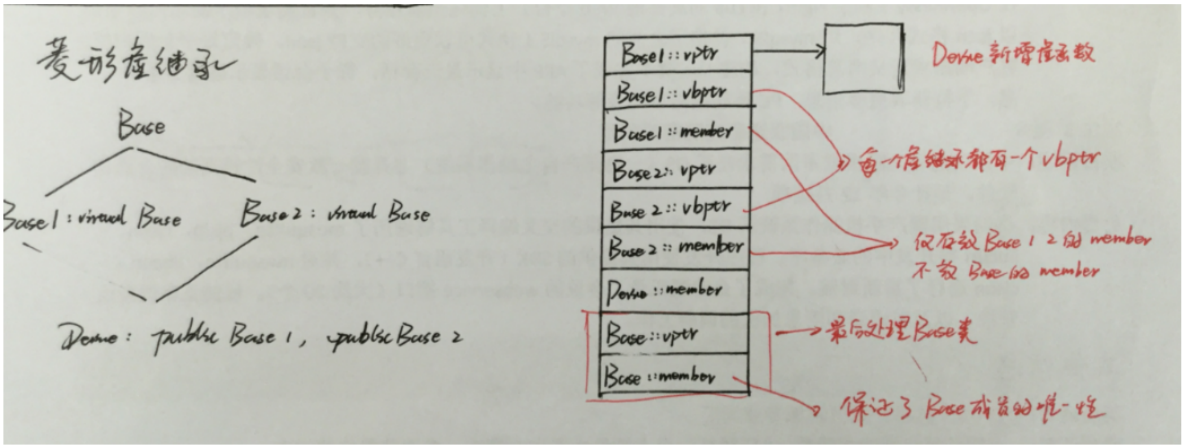


每一个虚继承，在父类的虚函数指针后面都有一个虚基类指针

在最后有最先的基类，没有虚基类指针

第一个条目存放虚基类表指针到该类内存首地址的偏移量

第二、三条目为该类的最左虚继承父类，次左虚继承父类的公用基类数据对象内存地址相对于虚基类表的偏移



虚基类指针占用4B

就是说继承的父类内容没有包含基类对象

15. 类与结构体

1. 什么时候用类，什么时候用结构体

   结构体轻量，可以直接使用栈进行分配，栈的效率较高。

   类的灵活性较高，复用性很强，例如继承和封装等等，当然类也可以使用栈内存，但是会使用更多的空间

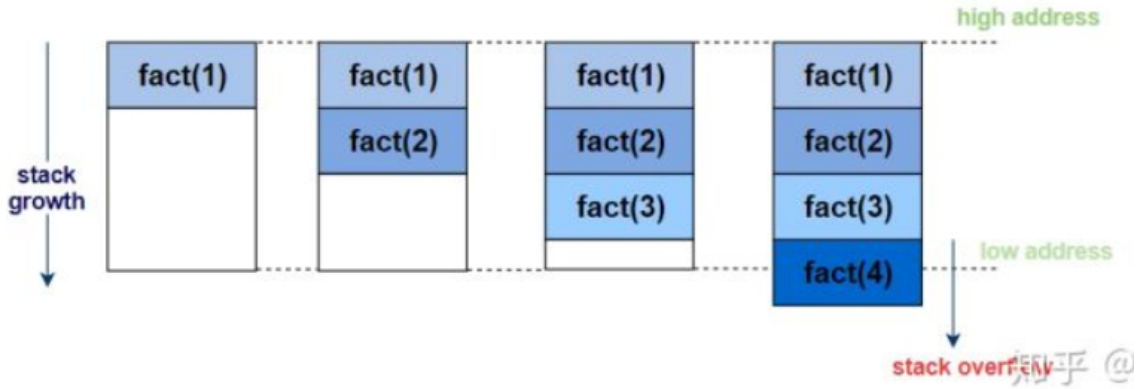
   轻量级的数据类型使用结构体，比如数据之间存在一定的关联性

   需要使用面向对象内容的通常使用类，例如继承关系

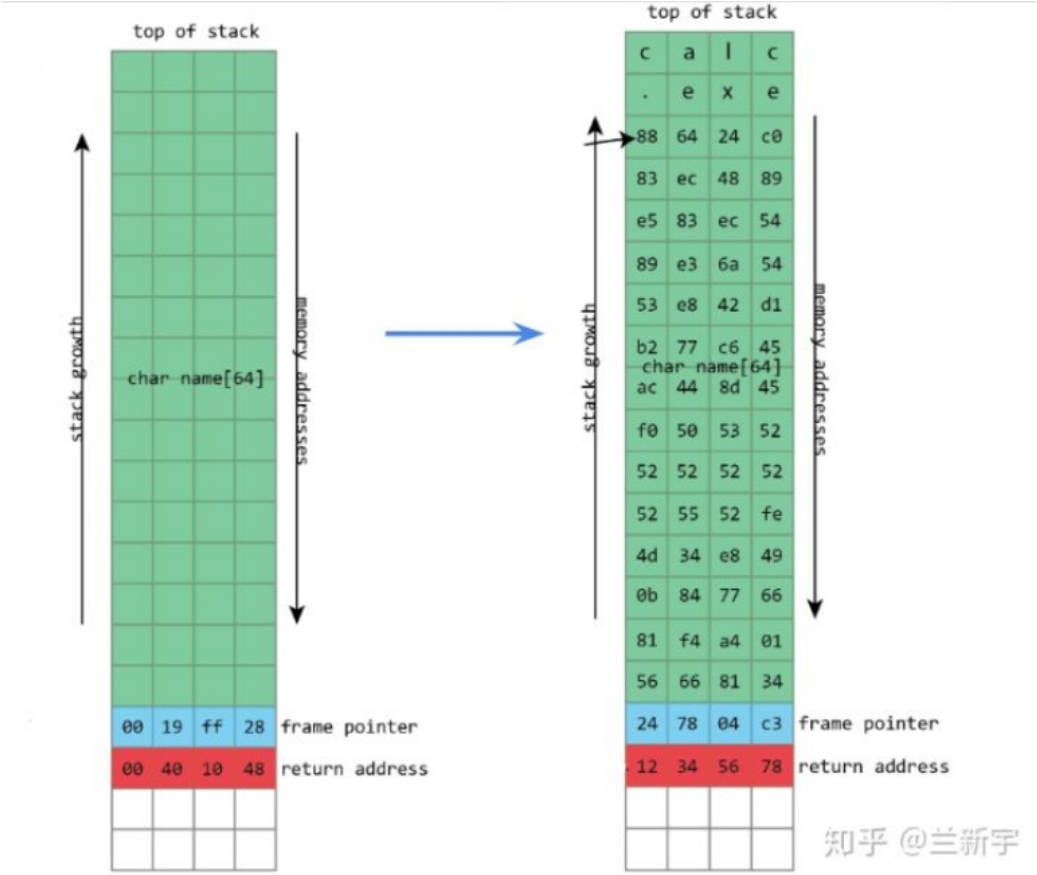
16. 判断、解决栈溢出

栈的溢出

1. 上溢：函数调用过多导致



2. 下溢：数组越界



3. 检查栈溢出：使用guard page：保护页放置在栈的两侧，如果访问到该地址触发异常

   设置变量存储栈的大小，申请空间时自动减少，如果减少至0抛出异常