Net知识图谱

2017-10-07 [dotNET跨平台](https://mp.weixin.qq.com/s?__biz=MzAwNTMxMzg1MA==&mid=2654069221&idx=1&sn=f540ebf722ab412ac7ed4d58ea23374e&chksm=80dbc5b0b7ac4ca62f4b8ca8b6f892d0c58a781d9d011416446fb42148f2012d4004a13a9161&mpshare=1&scene=1&srcid=1007Gs2GyTN8TLszc4qBToyR&key=65d8001a48e3b58b9e6293facce9b499c55bf78eda7efb37afe51803438dc74b980dae07e06feca346e3e05c2ddabd8659d1b4a2926db06a31c4cf01bd52b6b770c6396676ff338b5af62ea4f510b39a&ascene=1&uin=Mjg1Mzg5MzU%3D&devicetype=Windows+7&version=6205051a&pass_ticket=UiVQQ4uy%2F8i31oJChsdl6ivduxGyRuSappPhAlOpv6o%3D&winzoom=1" \l "#)

对于Web系统开发来说，Net其实也是有好多知识点需要学的，虽然目前JAVA是主流，就业市场比较大，但Net也在积极的拥抱开源，大Net Core 2 出来了，这无疑给Net开发者带来更大的希望，好了，以下是自己画的知识图，给正在找工作的自己一个时间梳理下，同时也希望给你带来些许帮助，第一次画并鉴于自己知识点有限画的不好，欢迎拍我，我及时纠正，谢谢！

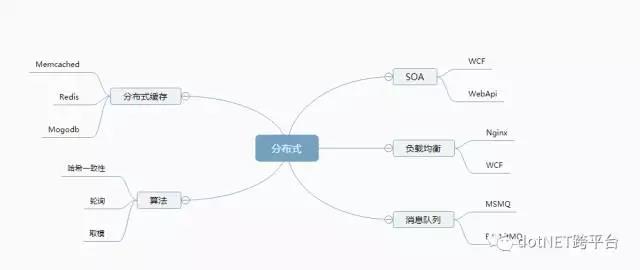
注：图是总-分结构，会针对一个知识点展开说明：

* 总结构图：



（图一）

* 分布式-结构图：



　　　　　　　　　　　　　　　　　　　　　　（图二）

* 分布式缓存 ：

1. Memcached : 当然非真正意义的分布式缓存系统，分布式需要客户端自己实现，特性：key-value 方式存储；内存缓存；不支持持久化；多台服务器负载之间不支持通信；简单，高效，可靠性弱。
2. Redis:用途较多，当缓存用，高并发用（提高吞吐量），当分布式锁用（SETNX  方式）。特性：key-value 方式存储（二进制）；集群；有事务却无事务回滚功能；持久化，可靠性强
3. Mongodb:No-SQL 数据库，当然没有数据结构，可以做海量数据存储，特性：主从复制集，集群；持久化，基本支持所有的数据类型；灵活强大的查询功能；

* 算法：

          算法是我们解决分布式问题的利器。

1. 取模：公式 a= hash(key)%N       a 是哈希值，hash 为哈希函数，key 是我们的服务器IP或名称，N是服务器数量；从公式中可以直观看出，当我们增加或删除服务器数量时会导致大部分失效，如果是缓存则命中率大部分失效，可能导致缓存雪崩，另外分库，分表 也可以采用取模算法，因为数据库可以做复制集或镜像能进行数据同步，也就是不存在失效的情况了。
2. 哈希一致：是一个环形链表，首先hash(key)---key   (可看作是key-value 的形式) 是均衡的分布在链表中，同样的问题当我们动态增加和删除服务器时，会动态重新负载到另外一台Key即服务器上，這样就保证了新的请求仍然正常工作，但会存在局部数据存在丢失的故障，那解决办法是物理冗余，做好数据同步。

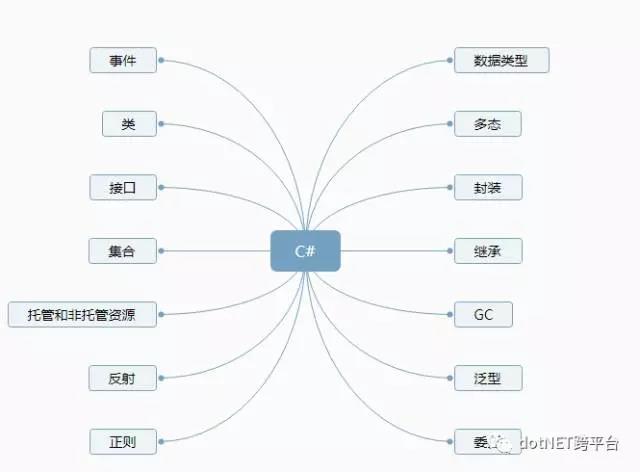
* 网络通信-结构图



（图三）

1. WebApi：基于Rest 架构风格，采用Http协议，使用Asp.net 编程模型进行实现，其常见的规范和约束：1,URLAPI 使用名称；2，使用Http行为来表述行为：Get，POST,PUT,Delete;3，请求和响应约定序列化方式，Content-Type,Accept；4，返回状态码尽可以和http返回状态一致，比如200代表成功，500 代表服务端异常；5，无状态可方便扩展；6，使用HATEOAS约束，对客户端提供一个URL，响应中返回你所有资源的URL及行为方式，后续客户端可根据响应中的链接进行资源访问，无论服务端如何变化，而对客户端而言始终透明，以此降低客户端的维护成本。
2. WCF:采用SOAP 协议，是以RPC-XML 规范为依据，使用WSDL 进行描述和定义，支持http/s ，TCP，MSMQ 等通信协议；定义契约，实施开发，客户端集成方式有多种，可以直接引用服务，可以继承ClientBase,可以通过ChannelFactory 信道工厂创建客户端实例，当然无论采用方式都是采用客户端代理模式的方式来透明实现，在代理中有很多技巧：比如实现服务端的负载均衡，比如自定义读取客户端WCF相关的配置文件和其它配置进行解耦方便维护；WCF内部很多基础实施已然有很多扩展点，对客户端的认证 可以使用 UserNamePassowrdValidator 进行扩展，授权 可以使用  IAuthorizationPolicy 进行扩展并使用CAS（Code Access Securty） 和 特性技术，全局异常点可以对 IErrorHandler  进行扩展，服务行为 可以对 IServiceBehavior 进行扩展，而且这些扩展代码实现并能基于配置完成；

* C#-结构图



（图四）

1. 数据类型，总体分为值类型和引用类型，值类型包括 int,double,enum,float等，其特点是不变性；引用类型包括：string , class, interface 等，这里面尤其说明的是string 也有不变性，strng a="1" 和 string b="1" 中 a 和 b 是指向同一个引用，而且 string a="1"+"2"  中间其实生成了临时的新对象并占用了一定内存，建议使用 StringBulider ; 值类型和引用类型的转换为装箱和拆箱，影响性能，要尽量少的强制类型转换，可以使用泛型进行强类型约定；
2. 集合，List  无序，可重复，线程非安全，存在泛型，复杂度为 O(n)  ； Dictionary  字典，key 不可重复，线程非安全，存在泛型，查找复杂度 O(1)；HashTable  字典，key 不可重复，线程安全，查找复杂度 O(1);  当然还有 Queue，Stack 等 ， 一般情况有对应的线程安全集合在前面加Concurrent，如 ConcurrentQueue,ConcurrentDictionary  ;
3. 委托和事件都可以进行内部解耦，回调，广播，只是委托更加灵活，可以做为函数参数传入，而事件只能在内部调用；当然委托和事件的定义也不同，事件是建立在委托定义之上，更强调的是消息通知。
4. 类和接口，多态，封装，继承，是面向对象的基本数据和特性，通过设计模式和设计原则能很清晰的熟悉面向对象的方式方法；如5大原则：开闭原则，单一原则，依赖倒置原则，里氏替换原则，接口隔离原则；其定义不再赘述。

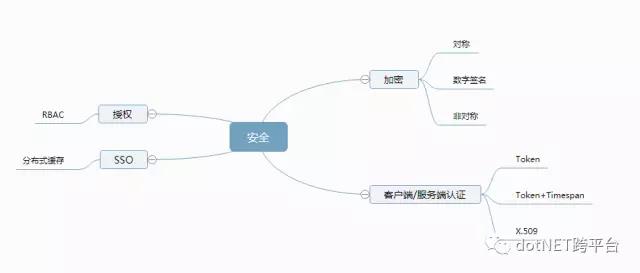
* 并发-结构图



（图五）

1. 有高并发的开发需求，首先要规划你的硬件、软件架构，那规划的依据是量化的数据，如PV,QPS;还要清楚软件的特性和用什么技术，然后还要通过性能测试进行辅助，如JMeter , 根据测试工具压测，得出整体服务的响应时间，吞吐量，另外还要对数据进行分析，看是否存在数据错乱的情况。
2. 缓存，主要是避免数据库IO的瓶颈，而在内存进行处理，当然单机缓存容量有一定局限，可以N台机器互联共享内存，则采用分布式缓存，在高并发情况下，使用缓存有一个思路是 相关业务操作及代码实现可以完全在缓存中操作，有些数据可以先预热到缓存中，服务运行的时候可以直接从缓存中读取，那缓存中数据更改后最终要同步到数据库，可以异步+队列的方式进行消息订阅和发送或者采用轮询定时从缓存中取出数据写入到数据库，在设计过程中格外注意同步失败或出现故障的情况，要有重试机制；其实在独立模块，独立服务中，有些服务并发并不高，那数据可以直接先写入到数据库中，而后再被缓存起来；最后 依据CAP定律，如果服务可用，并是容量分区这里是如果采用分布式缓存，最后只能是弱一致性，所以我们设计是解决90%以上的问题（当然如果能解决99.999%的问题更好了），另外10% 可以人工介入加以解决。
3. 限流，高并发情况下肯定不能将全部请求全部接受并一次性处理那服务有被搞崩溃的可能，那就可以将部分重复请求丢弃，可以使用Nginx 的 对客户端IP进行限制，同时高并发下肯定有些重要数据资源会存在竞争，如何保持数据一致性，使用锁机制，悲观锁会严重影响性能，但不会存在脏读，写错的情况，乐观锁，会存在脏读的情况，但能保证数据写入没有问题，一般我们采用分布式锁，使用Redis 的SETNX 特性，Redis 是单线程，存在事务，但事务没有回滚机制，Redis的事务是命令集的方式，如 SETNX 如果不存在Key 值则返回true, 如果已存在Key 值则返回false  ， 如果返回false 代表请求已存在，则请求被直接丢弃，其实Redis 的 SETNX 和 Membcached 的Add 有点类似，然后使用队列将请求串行化，到数据库基本访问不会太高，最后的数据库起码要最好主从模式，避免单一故障，最好数据冗余。
4. 算法，对请求进行限流有个很常用的手法，令牌桶，我们先申请访问的token, 并注入到一个桶容器中，设置容器的最大token 量，超过的请求直接丢弃，当请求访问时验证token 是否存在，如存在则正常处理后续业务逻辑并删除token, 否则直接丢弃请求，以上算法的实现方式可以使用 Redis 中的 SETNX +  Delete  命令实现；
5. 队列，有很对成熟的队列消息中间件，其中RabbitMQ 是较为常用，支持消息的持久化，避免中心服务DOWN 机后消息丢失；支持ACK机制，当消费者宕机或其它网络原因导致没有收到消息，则队列会进行重发消息，直到消费者确认收到通过BasicAck 命令进行发送则消息从队列中删除；有限流机制，可以通过内存大小，磁盘大小及上游流量大小三种方式对请求进行限流；可以集群，但各服务器队列进程不互相通信，所以需要客户端实现分布式，算法可以采用哈希一致性；

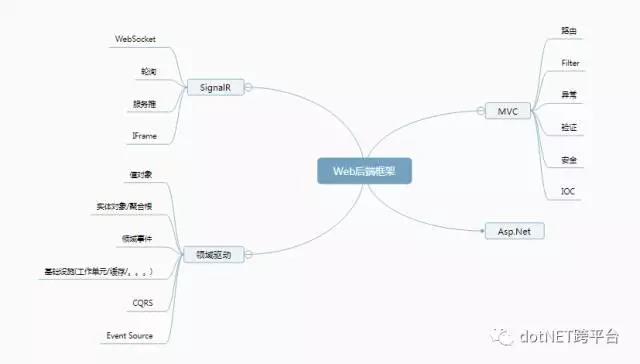
* 安全-结构图



（图六）

1. 加密，对称加密a=E(key)  , b =D(key)   a 是加密后值，E 是加密算法，key 是密钥，b 是解密后值，D 是解密算法，可以看出加密解密的key 是一致的，共享的，所以只要知道key，双方都可以进行加密解密，存在不安全因素，常用的对称加密算法 DES; 数字签名，即对内容进行加密的一个字符串（数字摘要），主要是为了保证内容的完整性及身份认证，常用的数字签名的算法MD5；非对称加密 c=E(key1),d=D(key2)  c 是加密后值，key1 为私钥，E 是加密算法，d 是解密后值，D 是解密算法，key2 为公钥；可以看出加解密所使用的密钥不同，当然也可以私用私钥解密，公钥加密，双方的家解密的私钥不公开，保证一定安全性,常见的非对称加密算法RSA.
2. 认证，常用的技术是使用Token或数字证书，Token 一般可以认为是数字签名；数字证书可以认为是X.509(是证书的标准规范和解析)；X.509 使用的是非对称加密，客户端根据证书中私钥对特定内容进行加密，然后发送到服务端，服务端通过公钥进行解密校验认证；微信公众号开发中涉及的微信端和我们服务端的双向认证是Token，但采用的对称加密.
3. 授权，Net 里面比较常用且规范的模式是RBAC,即基于角色控制授权，具体实现可以是 HttpModule + Attribute  ; HttpModule 是对所有（一般不包括静态资源）请求（线程）进行权限实例初始化并进行维持该会话，然后在每个需要授权的方法上标注Attribute及权限ID，Attribute 可以对CodeAccessSecurityAttribute进行扩展实现。
4. SSO,一般我们有這样的需求，比如有多个子系统，我们在一个系统中登陆成功后，其它系统就会共享此登陆状态而不用再进行登陆，或者 我们有一个系统，但是有多端可以使用，比如PC端，App端，当在一个客户机上登陆成功后，其它端或客户机即刻自动退出避免多端操作，由此我们想到了肯定要把一端登陆成功的状态进行保存并能够在多端共享并能全局访问，拥有此特点的想到了有分布式缓存。

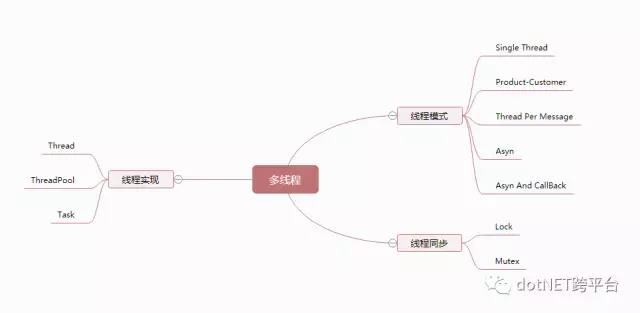
* Web后端框架-结构图



 （图七）

1. 从最开始的Asp.net 到 现在的MVC3 以上版本，对开发越发灵活和成熟，MVC 是一种模式而非技术实现，此处具体技术不再阐述，可以重点关注下一些比较实用的高级特性，比如 Area , Filter 等。
2. SignalR  是实时性框架，即主动通知，客户端和服务端基于Http 建立长连接，当服务端有新的消息时可以主动推送到客户端，常见客户端订阅可以基于JavaScript , C# 实现，Android 端也有叫SignalA但没用过，SignalR 内部通信机制有4种，最好的是WebSocket ,  客户端可以指定采用何种通信机制，一般是自动切换模式，采用或切换到那种通信模式依据是客户端的环境，理论上是越高配的性能就越好。
3. 领域驱动设计，是架构模式，由此引申出来一套理论和方法，值对象像C#里面的值类型也是不变性的，只是值对象是一个Class实例的对象，此Class 里面有1个或多个属性，为此我们必须覆写它的Equal 和 GetHashCode 保证它的不可变，GetHashCode 是在用到 字典时进行验证的，所以保险起见也要覆写，实体对象 是唯一的，每次我们New 完都是使用的不同的对象实体，避免内存对象不一致的情况，所以需要给定一个唯一标识，即也要对Equal 和 GetHashCode 进行覆写；聚合根是包含了值对象和实体对象，一个聚合根可以理解为一个领域内的模型，一般聚合根可以通过接口进行标注说明，而多个聚合根之间的会话通过领域事件来完成，领域事件的实现一般采用发布-订阅模式，事件通过接口进行标注，发布者可以将订阅者注入到容器中，容器的实现可以使用IOC框架如Unity,Autofac等，发布时根据不同的事件通知给不同的订阅者进行触发，由于事件通知的是另外一个聚合根，而很可能出现跨网络边界进行通知，依据CAP定律，会出现数据不一致的情况，所以使用Event Source 进行事件回放，可以理解为重试机制，Event Source 最好和当前聚合根在同一个事务中，进行事件回放可以另一个进程来完成，而另一个聚合根和Event Source （比如处理完成后修改其状态）最好也是在一个事务中，这样保证数据一致性，当然这里面会存在事件重复发的可能，解决方法增加一个字段标识，当事件发出去后，即可将此标识标记为一个特定状态比如发送中，下次调度就不再取发送中的事件即可；在高并发情况不建议使用事务所以另外一个Event Source 实现方法是 使用队列；我们对操作进行抽象无非就是CRUD，而CUD 我们可以抽象成命令也就是写操作，R 我们知道是查询也就是读操作，而CQRS就是命令查询职责分离，一般用在读写分离的架构当中，写操作一般使用的工作单元模式，而读操作可以很灵活，可以使用原生的Ado.Net，而读写操作可能会存在一定的时间延迟，比如写后异步处理，读的时候可能还是脏数据，所以这个延迟或者要求不能延迟具体项目要具体设计和实现。

* 多线程-结构图



（图八）

1. 为什么会有线程，为了进程的稳定，一个进程死掉了，不能影响到其他进程，而为什么会有进程，为了操作系统的稳定，不能在操作系统上操作一个记事本就把整个系统给搞崩溃了；线程是进程的逻辑单元，多个线程存在共用一个CPU的情况，会出现资源竞争，为了保证数据不错乱，要对线程进行调度，每隔30毫秒进行一次切换，当然切换的过程存在性能磨损，也是为了用户体验和系统的可靠性牺牲一点性能，在我们多核时代这一点性能其实也是可以忽略的；
2. 线程实现，Net 中关键字 Thread , 分前台线程和后台线程，通过IsBackgournd来标记，一般我们都使用后台线程即异步，线程分优先级去处理，在调度过程中优先级高的优先处理，Net 中提供了比较丰富的线程API，比如Sleep,Join，Abort 就不再一一赘述了，线程池是为了解决线程来回切换损失的性能，它的主要作用就是 线程复用，当一个线程处理完成任务后重新回到线程池中，等待下次再用，有时我们异步处理数据时为了提高性能，对数据进行分组，然后分别放在不同的线程中去处理，比如 一个线程池中有5个线程，一批数据有 1000 条，则每一个线程处理200 条，当然可能会出现不整除的情况，（取模），当线程处理完成后重新回到池中等待下一个任务，线程池也有不足比如无法实时查看线程进度，不能反馈执行结果，這样Task 就出现了，Task 是在线程池的基础上进行了封装，优化，Net 同样也提供了丰富的API ,比如 Run, StartFactory, Wait,WaitAll,ContinueWith 等；
3. 线程模式，有很多模式可以做为模板进行实现，Single Thread 即我们常见的 线程内加锁的方式，多线程时通过加锁来保证数据同步；Product-Customer 即我们发布订阅的模式，生产者、消费者可以是多线程，而中间的数据容器可以使用内存列表，数据库，或者队列；Thread-Pre-Message , 即线程池模式，多个线程在池中进行生命周期的维护；Asyn和Asyn And CallBack 即是我们常用的Task 来异步处理和带委托进行回调的方式，当然你也可以架构方面更高层面进行理解；

打的手有点疼了，就先写到这吧，以上是自己的一点心得体会，如有不当还请指教，谢谢。