JAVA 训练营毕业总结

毕业项目：

<1>对技术的关键点的思考和经验认识：

（备注：针对以下知识点 JVM,NIO,并发编程,Spring 和 ORM 等框架,MySql 数据库和 SQL,分库分表,RPC 和微服务,分布式缓存，分布式消息队列）

（1）JVM 知识点

总结如下：

加强了对字节码的了解，更加明白了源码和字节码之间的关系，以及字节码的定义。

对 java 类加载器类的生命周期过程（加载—》验证—》准备—》 解析—》初始化—》使用—》卸载）有了一定了解。明白了类的加载时机，以及三类加载器（启动类加载器，扩展加载器，应用加载器），加载器的特点双亲委托，负责依赖，缓存加载，以及如何使用 ClassLoader 加载。

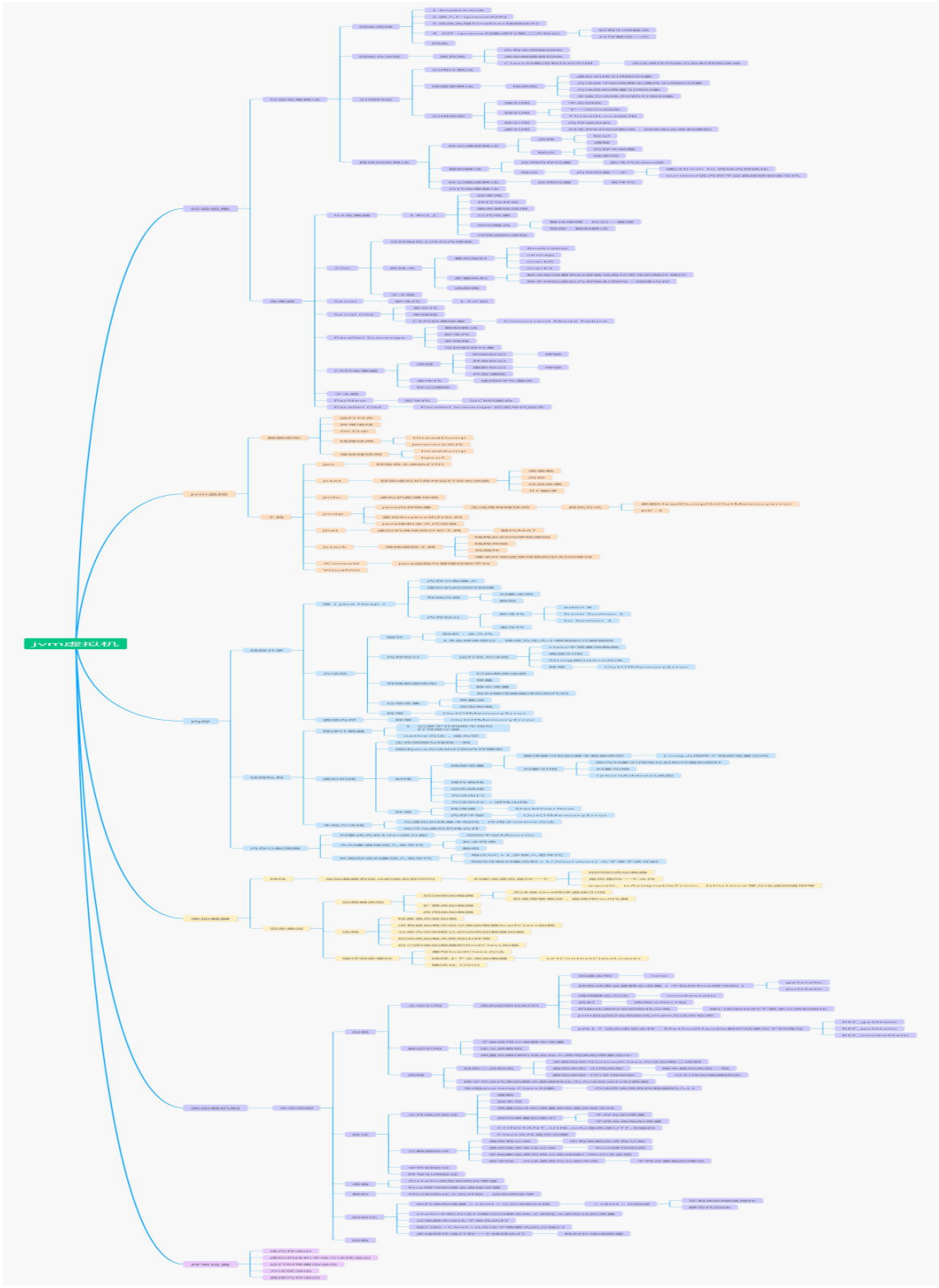
对 JVM 内存模型增进了了解，明确了线程栈和堆内存，以及 JVM 内存整体结构，并且了解 JVM 的启动参数和如何配置。

学习了 JVM 相关工具了解如何通过工具对 JVM 进行观测和调优。

深入了解了 JVM 的 GC，针对串行 GC,并行 GC,CMS GC,G1 GC 的原理&配置&使用场景，以及对于的优缺点和之后的如何选型均有深入理解，同时也更加清晰了 GC 的演进路径和 GC 的日志解读与分析&JVM 线程堆栈数据分析。

最后了解 JVM 面试汇总。

知识图谱如下：



（2）NIO

总结如下：

了解 Socket 通信模型（从建立服务端倾听的 Socket—》创建连接 Socket 向服务端发送请求—》等待并接受连接请求—》接收请求后创建连接 Socket—》开始通信（流化）—》结束通信关闭 Socket）

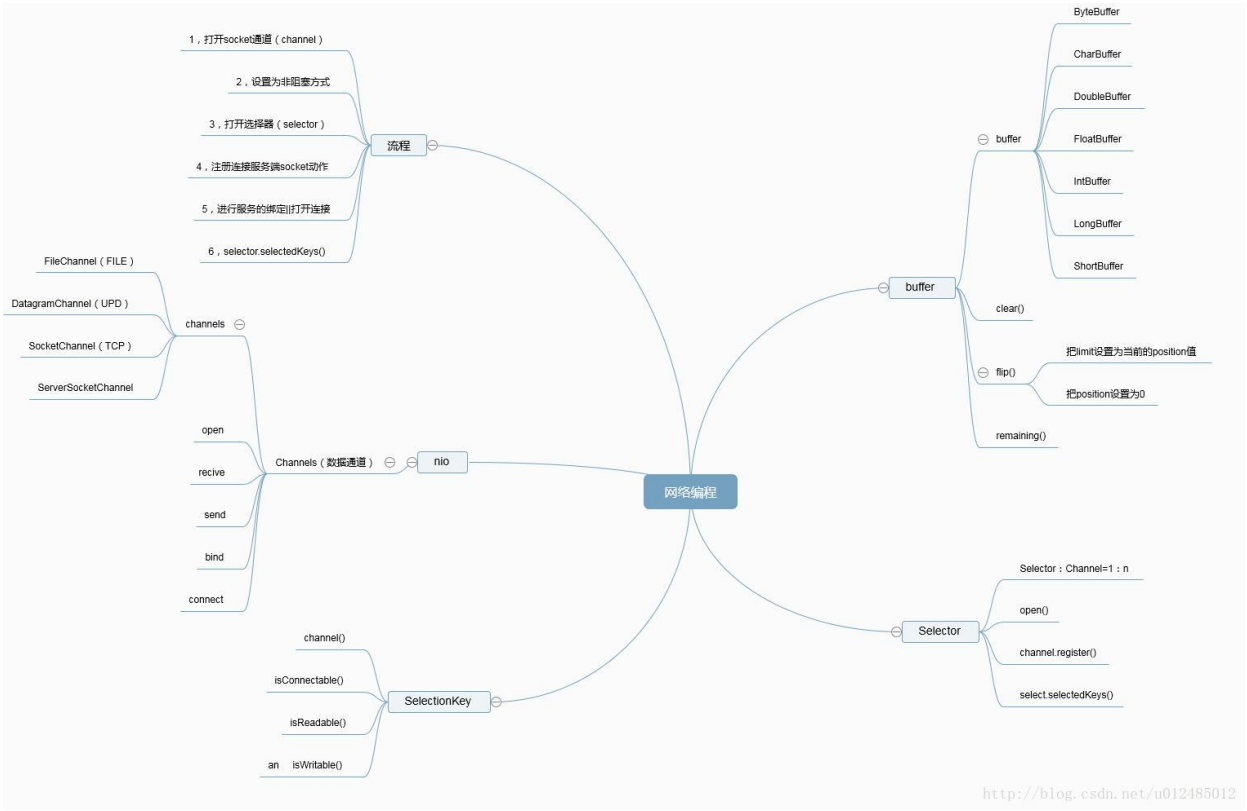
针对五种 IO 模型（阻塞 IO 模型，非阻塞 IO 模型，I/O 复用模型，信号驱动的 I/O 模型，异步 I/O 模型）有了初步的了解。

针对网络开发框架 Netty 有了一定的了解，主要基于异步，事件驱动，基于 NIO，其适用于服务端，客户端，协议为 TCP/UDP。以及了解了 Netty 是如何实现高性能，并且实现了 API 网关。明白了从事件处理机制到 Reactor 模型（Reactor 模型是事件驱动的，有一个或者多个并发输入源，有一个 Service Handler 和多个 EventHandlers，并且 ServiceHandle 会同步的将输入请求多路复用的分给相应的 Event Handler）

了解了 Netty 的运行原理和关键对象

Bootstrap,EventLoopGroup,EventLoop,
SocketChannel,ChannelInitializer,ChannelPipline,
ChannelHandler。以及 Netty 的粘包和拆包。

知识图谱如下：


网络编程
流程
1，打开socket通道（channel）
2，设置为非阻塞方式
3，打开选择器（selector）
4，注册连接服务端socket动作
5，进行服务的绑定|打开连接
6，selector.selectedKeys()
nio
channels
FileChannel（FILE）
DatagramChannel（UPD）
SocketChannel（TCP）
ServerSocketChannel
Channels（数据通道）
open
recive
send
bind
connect
SelectionKey
channel()
isConnectable()
isReadable()
an isWritable()
buffer
ByteBuffer
CharBuffer
DoubleBuffer
FloatBuffer
IntBuffer
LongBuffer
ShortBuffer
clear()
flip()
把limit设置为当前的position值
把position设置为0
remaining()
Selector
Selector：Channel=1：n
open()
channel.register()
select.selectedKeys()
http://blog.csdn.net/u012485012

（3）并发编程

总结如下：

了解 Thread 的状态改变操作

Thread.sleep(long millis);

Thread.yield();

thread.join()/thread.join(long millis);

obj.wait();

obj.notify();

Thread 的中断和异常处理，以及多线程的安全问题，多线程的并发问题类似于多事务的并发问题，关键点在于 lock 的应用。

并发相关的性质：

happens-before 原则（先行发生原则）

1）程序次序规则：一个线程内，按照代码先后顺序。

2）锁定规则：一个 unlock 操作先行发生于后面对同一个锁的 lock 操作。

3)Volatile 变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作。

4）传递规则：如果操作 A 先行发生于操作 B，而操作 B 又先行发生于操作 C，则可以得出 A 先于 C。

5）线程启动规则：Thread 对象的 start()方法先行发生于此线程的每个一个动作。

6）线程中断规则：对线程 interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生。

7）线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过 Thread.join()方法结束，Thread.isAlive()的返回值手段检测到线程已经终止执行。

8)对象终结规则：一个对象的初始化完成先行发生于他的 finalize()方法的开始。

线程池：

1）Excutor ： 执行者 - 顶层接口

2）ExcutorService ： 接口 API

3）ThreadFactory ： 线程工厂

4）Excutors ： 工具类

ReadWriteLock 管理一组锁，一个读锁，一个写锁。

读锁可以在没有写锁的时候被多个线程同时持有，但是写锁是独占的。所有读写锁的实现必须确保写操作对读操作的内存影响。每次只能有个一个写线程，但是同时可以有多个线程并发的读数据，所以在 ReadWriteLock 的情况下适用于读多写少的并发情况。

针对集合的并发安全总结:

ArrayList　　并发读写不安全；

LinkedLis　　使用副本机制改进；

HashMap　　并发读写不安全；

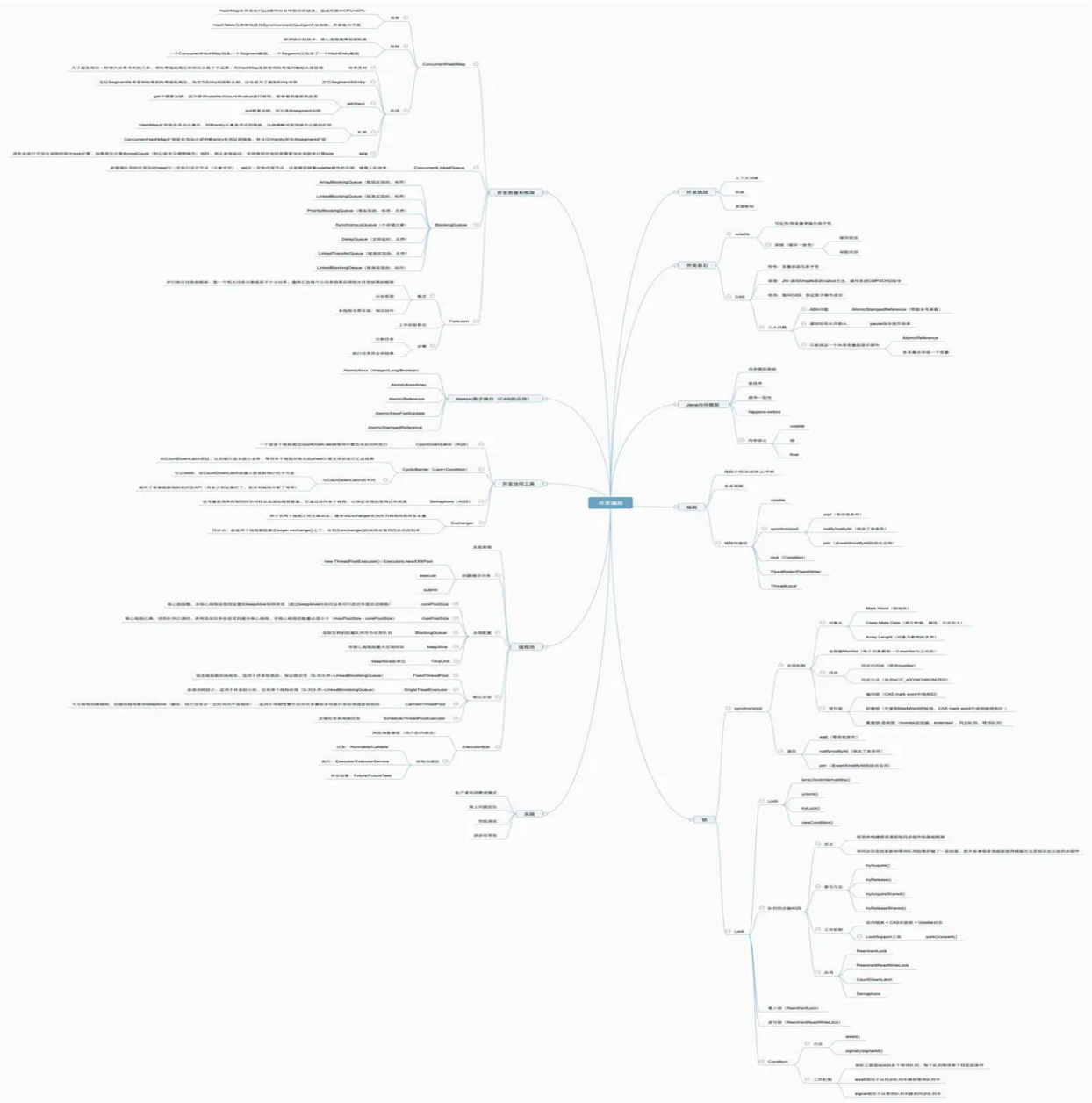
LinkedHashMap　使用分段锁或者 CAS

并发需要考虑的维度：粒度，性能，重入，公平，自旋锁（spinlock）。

线程间协作与通信：

1. 线程间共享：static/实例变量（堆内存），Lock ， synchronized
2. 线程间协作：Thread#join(), Object#wait/notify/notifyAll, Future/Callable,CountdownLatch,CyclicBarrier

知识图谱如下：



（4）Spring 和 ORM 等框架

总结如下：

对 Spring Framework 6 大模块有一定了解：

1. Core: Bean/Context/AOP
2. Testing: Mock/TestContext
3. DataAccess: Tx/JDBC/ORM
4. Spring MVC/WebFlux:web
5. Integration: remoting/JMS/WS
6. Languages: Kotlin/Groovy

了解 SpringAOP ： 其实就是 Spring 中的一个代理层，通过字节码增强来实现所有对象的托管。（备注：在我们编程设计的过程中，需要注意很多问题都可以通过增加层的方式解决）

了解 IOC-控制反转：其也称为 DI(Dependency Injection)依赖注入，这个是针对对象装配思路的改进。从对象 A 直接引用和操作对象 B，变成对象 A 里指需要依赖一个接口 IB，系统启动和装配阶段，把 IB 接口的实例对象注入到对象 A，因此 A 就不需要依赖一个 IB 接口的具体实现，也就是 B 类。

了解 SpringBean 的生命周期：构造函数—》依赖注入—》BeanNameAware—》BeanFactoryAware—》ApplicationContextAware—》BeanPostProcessor 前置方法—》InitializingBean—》自定义 init 方法—》BeanPostProcessor 后置方法—》使用—》DisposableBean—》自定义 destroy 方法

了解了 SpringBoot 的产生原因和定位，由于框架需要让开发变的简单&让配置变简单&让运行变的简单，基于上述的诉求，那么确定约定大于配置的想法。

SpringBoot 简化的手段如下：

1. Spring 本身技术的成熟与完善，各方面第三方组件的成熟集成。
2. Spring 团队在去 Web 容器化等方面的努力。
3. 基于 maven 与 pom 的 Java 生态体系，整合 pom 模板成为可能。
4. 避免大量 maven 导入和各种版本的冲突。

综上所述，Springboot 的定位是 Spring 的一套快速配置脚手架，关注于自动配置，配置驱动，其包含自动配置注解（SpringBootApplication 核心启动入口），条件化自动配置（运行时灵活组装，避免冲突），SpringBootStarter。

ORM 知识点，针对主流框架 Hibernate 与 MyBatis 的比较如下：

MyBatis 优点：原生 SQL（XML 语法），直观，对 DBA 友好

MyBatis 缺点：繁琐，可以用 Mybtis-generator,MyBatis-Plus 之类的插件。

Hibernate 优点：简单场景不用写 SQL（HQL，Cretiria, SQL）

Hibernate 缺点：对 DBA 不友好

了解了 Spring 的事务管理：

Spring 声明式事务配置参考：

事务的传播性：

@Transactional(propagation=Propagation.REQUIRED)

事务的隔离级别：

@Transactional(isolation = Isolation.READ_UNCOMMITTED)

读取未提交数据（会出现脏数据，不可重复读）基本不使用。

只读：

@Transactional(readOnly=true)

该属性用于设置当前事务是否为只读事务，设置为 true 表示只读，false 则表示可读写，默认值为 false。

事务的超时性：

@Transactional(timeout=30)

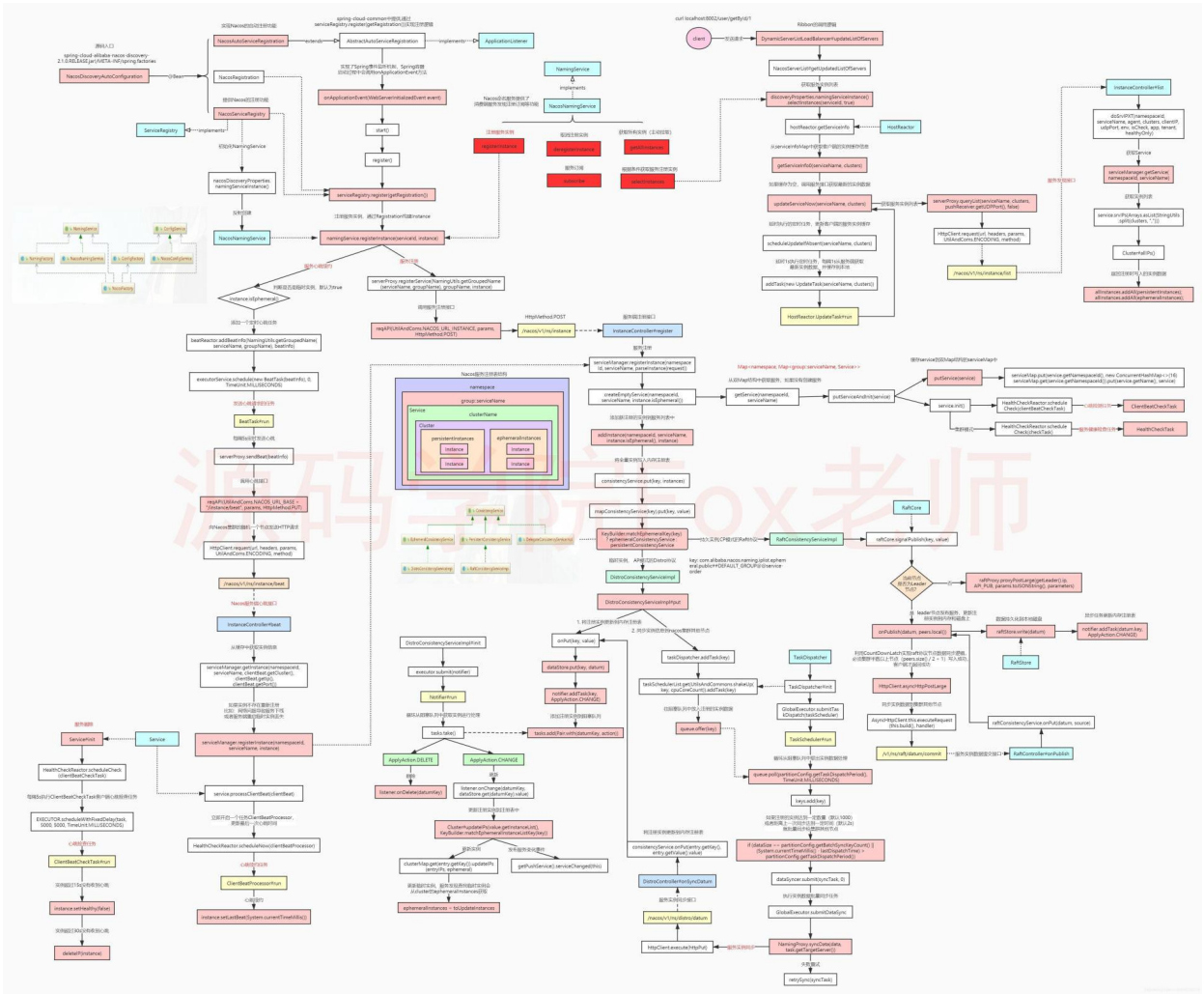
回滚：

指定单一异常类：　　@Transactional(rollbackFor=RuntimeException.Class)

指定多个异常类：

@Transactional(rollbackFor={RuntimeException.Class,Exception.Class})

知识图谱如下：



（5）MySql 数据库和 SQL

总结如下：

了解数据库设计范式：

第一范式（1NF）：关系 R 属于第一范式，当且仅当 R 中的每一个属性 A 的值域只包含原子项。

第二范式（2NF）：在满足 1NF 的基础上，消除非主属性对码的部分函数依赖。

第三范式（3NF）：在满足 2NF 的基础上，消除非主属性对码的传递函数依赖。

BC 范式（BCNF）：在满足 3NF 的基础上，消除主属性对码的部分和传递函数依赖。

第四范式（4NF）：消除非平凡的多值依赖。

第五范式（5NF）：消除一些不合适的连接依赖。

了解 MySQL 相关知识点：

MySQL 存储：

独占模式：日志组文件，表结构文件，独占表空间文件，字符集和排序规则文件，binlog 二进制日志文件（记录主数据库服务器的 DDL 和 DML 操作），二进制日志索引文件。

共享模式：innodb_file_per_table=1

MySQL 简化执行流程：

客户端—》MySQL 服务器—》查询缓存—》解析器—》解析树—》预处理器—》解析树—》查询优化器—》执行计划—》查询执行引擎—》结果

MySQL 详细执行流程：

更新记录—》连接器—》分析器—》优化器—》执行器—》写 undo log 存储回滚段指针和事务 ID—》记录所在的目标页存在于内存中—》找到数据判断数据冲突与否更新内存&找到数据更新内存—》写入 redo log—》写 binlog—》提交事务—》刷 redo log 盘处于 commit-prepare 阶段—》刷 binlog 处于 commit-commit 阶段

MySQL 对 SQL 执行顺序：

from—》on—》join—》where—》group by—》having+聚合函数—》select—》order by—》limit

MySQL 事务

事务可靠性模型 ACID：

Atomicity : 原子性，一次事务中的操作要么全部成功，要么全部失败。

Consistency : 一致性，跨表，跨行，跨事务，数据库始终保持一致状态。

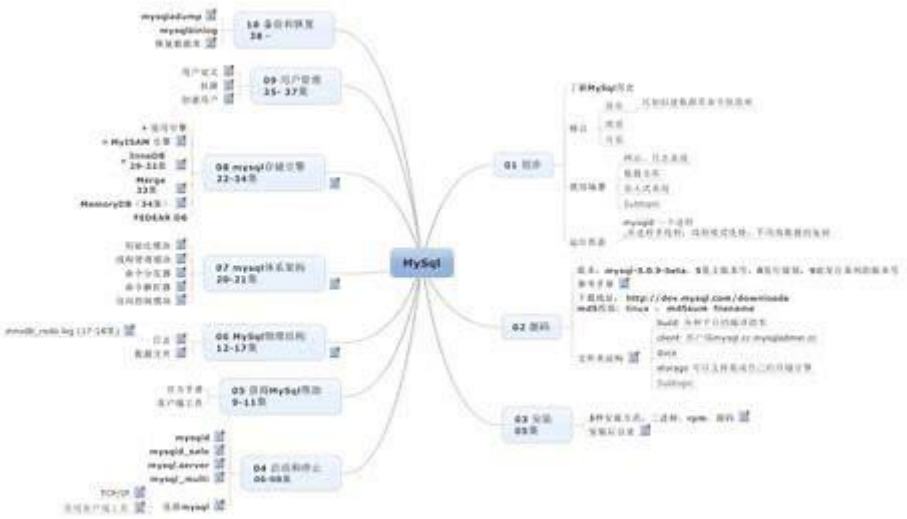
Isolation : 隔离性，可见性，保护事务不会相互干扰，包含 4 种隔离级别。

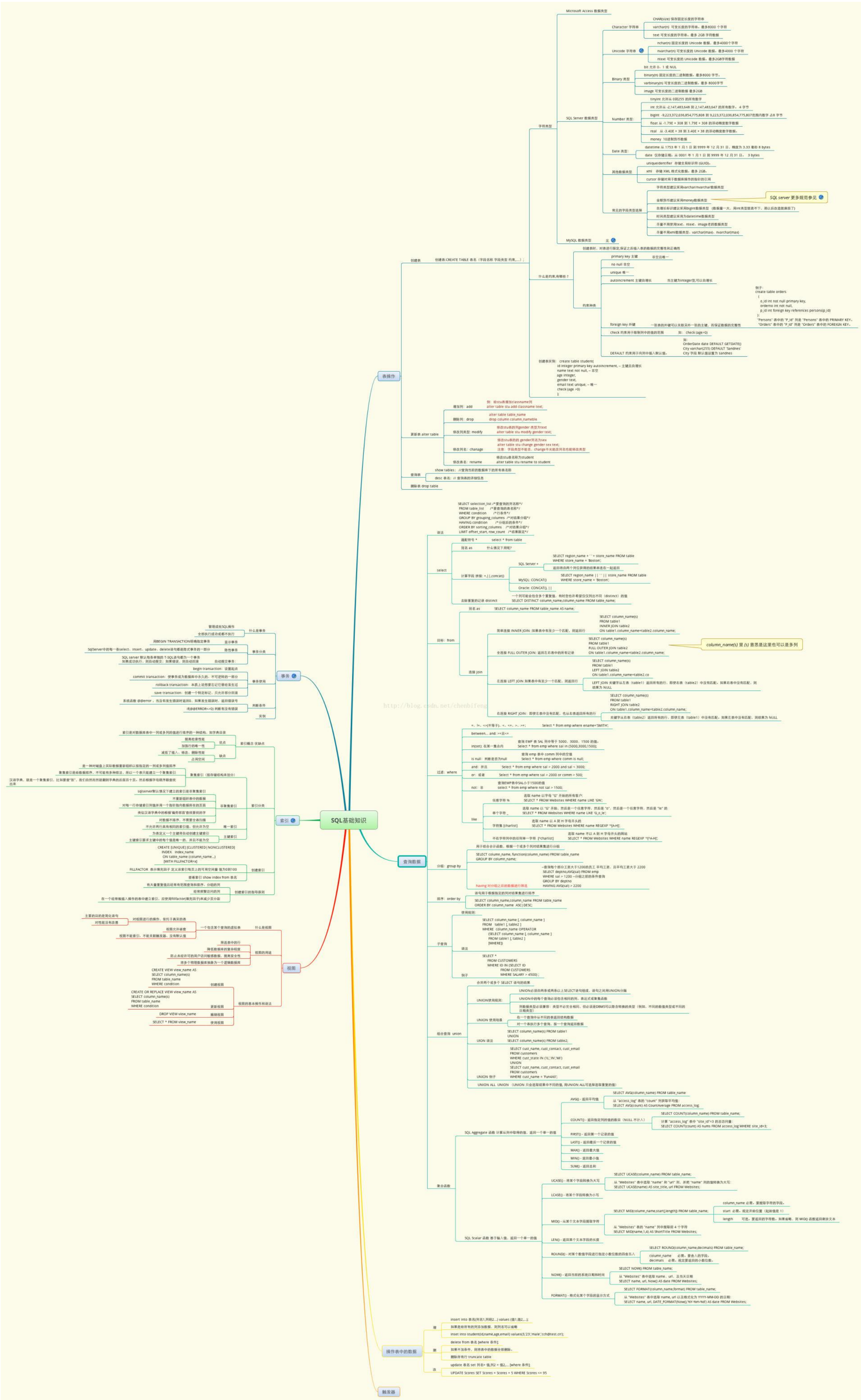
Durability : 持久性，事务提交成功后，不会丢数据。

InnoDB:

双写缓冲区，故障恢复，操作系统，fsync()，磁盘存储，缓存，UPS, 网络，备份策略.....

知识图谱如下：


MySql
01 简介
02 源码
03 安装
04 启动和停止
05 使用MySql指南
06 MySql物理结构
07 mysql体系架构
08 mysql存储引擎
09 用户管理
10 备份和恢复


SQL基础知识
表操作
查询数据
操作表中的数据
触发器
事务
索引
视图
SQL server 更多规范参见
column_name(s) 里 (s) 意思是这里也可以是多列

（6）分库分表

总结如下：

了解 MySQL 的高可用：

1）主从手动切换：如果主节点挂掉，将某个从改成主。用 LVS+Keepalived 实现多个节点的探活+请求路由。

2）MHA（Master High Availability）目前在 MySQL 高可用方面是一个相对成熟的解决方案，它由日本 DeNA 公司的 youshimaton 开发，是一套优秀的作为 MySQL 高可用性环境下故障切换和主从提升的高可用软件。

3）MGR：如果主节点挂掉，将自动选择某个从改成主；无需人工干预，基于组复制，保证数据一致性。

MGR 特点：

<1>高一致性：基于分布式 Paxos 协议实现组复制，保证数据一致性。

<2>高容错性：自动检测机制，只要不是大多数节点都宕机就可以继续工作，内置防脑裂保护机制。

<3>高扩展性：节点的增加与移除会自动更新组成员信息，新节点加入后，自动从其他节点同步增量数据，直到与其他节点数据一致；

<4>高灵活性：提供单主模式和多主模式，单主模式在主库宕机后能够自动选择主，所有写入都在主节点上进行，多主模式支持多节点写入。

4）MySQL Cluster ： MySQL InnoDB Cluster 是一个高可用的框架，它由下面这个几个组件构成：

<1> MySQL Group Replication ： 提供 DB 的扩展，自动故障转移。

<2> MySQL Router ： 轻量级中间件，提供应用程序连续目标的故障转移。

<3> MySQL Shell ： 新的 MySQL 客户端，多种接口模式。可以设置群组复制及 Router。

5）Orchestrator： 如果主节点挂掉，将某个从改成主。一款 MySQL 高可用和复制拓扑管理工具，支持复制拓扑结构的调整，自动故障转移和手动主从切换等。后端数据库用 MySQL 或者 SQLite 存储元数据，并提供 Web 界面展示 MySQL 复制的拓扑关系及状态，通过 Web 可更改 MySQL 实例的复制关系和部分配置信息，同时也提供命令行和 API 接口,方便运维管理。

特点：

（1）自动发现 MySQL 的复制拓扑，并且在 Web 上展示；

（2）重构复制关系，可以在 web 进行拖图来进行复制关系变更。

（3）检测主异常，并可以自动或手动恢复，通过 Hooks 进行自定义脚本；

（4）支持命令行和 Web 界面管理复制。

了解分库分表相关的扩展立方体：

x 轴 ： 通过 clone 整个系统复制，集群。

y 轴 ： 通过解耦不同功能复制，业务拆分。

z 轴 ： 通过拆分不同数据扩展，数据分片。

在数据库的扩展方面：

全部数据—》数据复制—》主从数据，备份与高可用。

业务分类数据—》垂直分库分表—》分布式服务化，微服务

任意数据—》水平分库分表—》分布式结构，任意扩容

针对数据库垂直拆分的优缺点：

优点：

1. 单库(单表)变小，便于管理和维护
2. 对性能和容量有提示作用
3. 改造后，系统和数据复杂度降低。
4. 可以作为微服务改造的基础。

缺点：

1. 库变多，管理变复杂。
2. 对业务系统有较强的侵入性。
3. 改造过程复杂，容易出故障。
4. 拆分到一定程度就无法继续拆分。

垂直拆分的做法：

1. 梳理清楚拆分范围和影响范围。
2. 检查评估和重新影响到的服务。
3. 准备新的数据库集群复制数据。
4. 修改系统配置并发布新版上线。

数据库水平拆分：

水平拆分就是直接对数据进行分片，有分库和分表两个具体方式，但是都只是降低单个节点数据量，但不改变数据本身的结构。这样对业务系统本身的代码逻辑来说，就不需要做特别大的改动，甚至可以基于一些中间件做到透明。

数据库水平拆分的优缺点：

优点：

1. 解决容量问题。
2. 比垂直拆分对系统影响小。
3. 部分提升性能和稳定性。

缺点：

1. 集群规模大，管理复杂。
2. 复杂 SQL 支持问题（业务侵入性，性能）
3. 数据迁移问题
4. 一致性问题

数据库中间件 ShardingSphere:

Apache ShardingSphere 是一套开源的分布式数据库中间件解决方案组成的生态圈，它由 JDBC，Proxy 和 Sidecar(规划中)这 3 款相互独立，却又能够混合部署配合使用的

产品组成。它们均提供标准化的数据分片，分布式事务和数据库治理功能，可适用于 JAVA 同构，异构语言，云原生等各种多样化的应用场景。

分布式事务：
分布式条件下，多个节点操作的整体事务一致性。
特别是在微服务场景下，业务 A 和业务 B 关联，事务 A 成功，事务 B 失败，由于跨系统，就会导致不被感知。此时从整体看，数据是不一致的。
了解如何实现分布式下的一致性：
1. 强一致性： XA
2. 弱一致性：TCC
(1) 不用事务，业务侧补偿冲正。
(2) 所谓的柔性事务，使用一套事务框架保证最终一致的事务。
XA 的事务：（两阶段提交事务）
xa_start : 负责开启或者恢复一个事务分支
xa_end : 负责取消当前线程与事务分支的关联
xa_prepare : 询问 RM 是否准备好提交事务分支
xa_commit : 通知 RM 提交事务分支
xa_rollback : 通知 RM 回滚事务分支
xa_recover : 需要恢复的 XA 事务

BASE 柔性事务：
TCC 通过手动补偿处理。
AT 通过自动补偿处理。

BASE 柔性事务 TCC
TCC 模式即将每个服务业务操作分为两个阶段：
1）第一个阶段检查并预留相关资源。
2）第二个阶段根据所有服务业务的 Try 状态来操作，如果都成功，则进行 Confirm 操作，如果任意一个 Try 发生错误，则全部 Cancel。

TCC 使用要求就是业务接口都必须实现三段逻辑：
1. 准备操作 Try：完成所有业务检查，预留必须的业务资源。
2. 确定操作 Confirm : 真正执行的业务逻辑，不做任何业务检查，只使用 Try 阶段预留的业务资源。因此，只要 Try 操作成功，Confirm 必须能成功。另外，Confirm 操作需满足幂等性，保证一笔分布式事务能且只能成功一次。
3. 取消操作 Cancel ：释放 Try 阶段预留的业务资源。同样的，Cancel 操作也需要满足幂等性。

（7）PRC 和微服务

总结如下：

了解 RPC 概念：

RPC 是远程过程调用（Remote Procedure Call）的缩写形式。

简单的说，就是像本地方法一样调用远程方法。

PRC 原理：

核心是代理机制。

1. 本地代理存根：Stub
2. 本地序列化反序列化
3. 网络通信
4. 远程序列化反序列化
5. 远程服务存根：Skeleton
6. 调用实际业务服务
7. 原路返回服务结果
8. 返回给本地调用方

（备注：注意处理异常）

设计一个 RPC 框架

共享 POJO 实体类&接口定义—》选择动态代理或者 AOP—》序列化&反序列化—》网络传输 TCP/SSL&HTTP/HTTPS—》查找实现类

从 RPC 走向服务化

客户端—》注册中心—》服务端

客户端：

服务发现—》调用模块（负载均衡&容错&透明）—》RPC 协议(序列化&协议编码&网络传输)

服务端：

服务暴露—》处理程序—》线程池—》RPC 协议（反序列化&协议解码&网络传输）

分布式框架 DUBBO

Apache Dubbo 是一款高性能，轻量级的开源 Java 服务框架

六大核心能力：面向接口代理的高性能 RPC 调用&智能容错和负载均衡&服务自动注册和发现&高度可扩展能力&运行期流量调度&可视化的服务治理与运维。

Dubbo 的主要功能：

基础功能：RPC 调用&多协议（序列化，传输，RPC）&服务注册发现&配置，元数据管理&框架分层设计，可任意组装和扩展。

扩展功能：集群（负载均衡）&治理，路由&控制台，管理与监控

Dubbo 的整体架构：

1. config 配置层：对外配置接口，以 ServiceConfig, RefercenceConfig 为中心，可以直接初始化配置类，也可以通过 spring 解析配置生成配置类。
2. proxy 服务代理层：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton,以 ServiceProxy 为中心，扩展接口为 ProxyFactory。

3. registry 注册中心层：封装服务地址的注册与发现，以服务 URL 为中心，扩展接口为 RegistryFactory,Registry,RegistryService
4. cluster 路由层：封装多个提供者的路由及负载均衡，并桥接注册中心，以 Invoker 为中心，扩展接口为 Cluster,Directory,Router,LoadBalance
5. monitor 监控层：RPC 调用次数和调用时间监控，以 Statistics 为中心，扩展接口为 MonitorFactory, Monitor, MonitorService
6. protocol 远程调用层：封装 RPC 调用，以 Invocation,Result 为中心，扩展接口为 Protocol,Invoker,Exporter。
7. exchange 信息交换层：封装请求响应模式，同步转异步，以 Request,Response 为中心，扩展接口为 Exchange,ExchangeChannel,ExchangeClient,ExchangeServer。
8. transport 网络传输层：抽象 main 和 netty 为统一接口，以 Message 为中心，扩展接口为 Channel，Transporter,Client,Server,Codec。
9. serialize 数据序列化层：可复用的一些工具，扩展接口为 Serialization,ObjectInput,ObjectOutput,ThreadPool。

RPC 与分布式服务化的区别：
RPC：技术概念
》以 RPC 来讲，我们前面的自定义 RPC 功能已经差不多了。
》可以再考虑一下性能优化，使用 spring-boot 等封装易用性。
分布式服务化：服务是业务语义，偏向于业务于系统的集成。
》以分布式服务化框架的角度来看，我们还差前面的这些非功能性需求能力。
》具体使用时，另外一个重点是如何设计分布式的业务服务。

分布式服务化配置&注册&元数据中心
配置中心（ConfigCenter）：管理系统需要的配置参数信息。
注册中心（RegistryCenter）：管理系统的服务注册，提供发现和协调能力。
元数据中心（MetadataCenter）：管理各个节点使用的元数据信息。

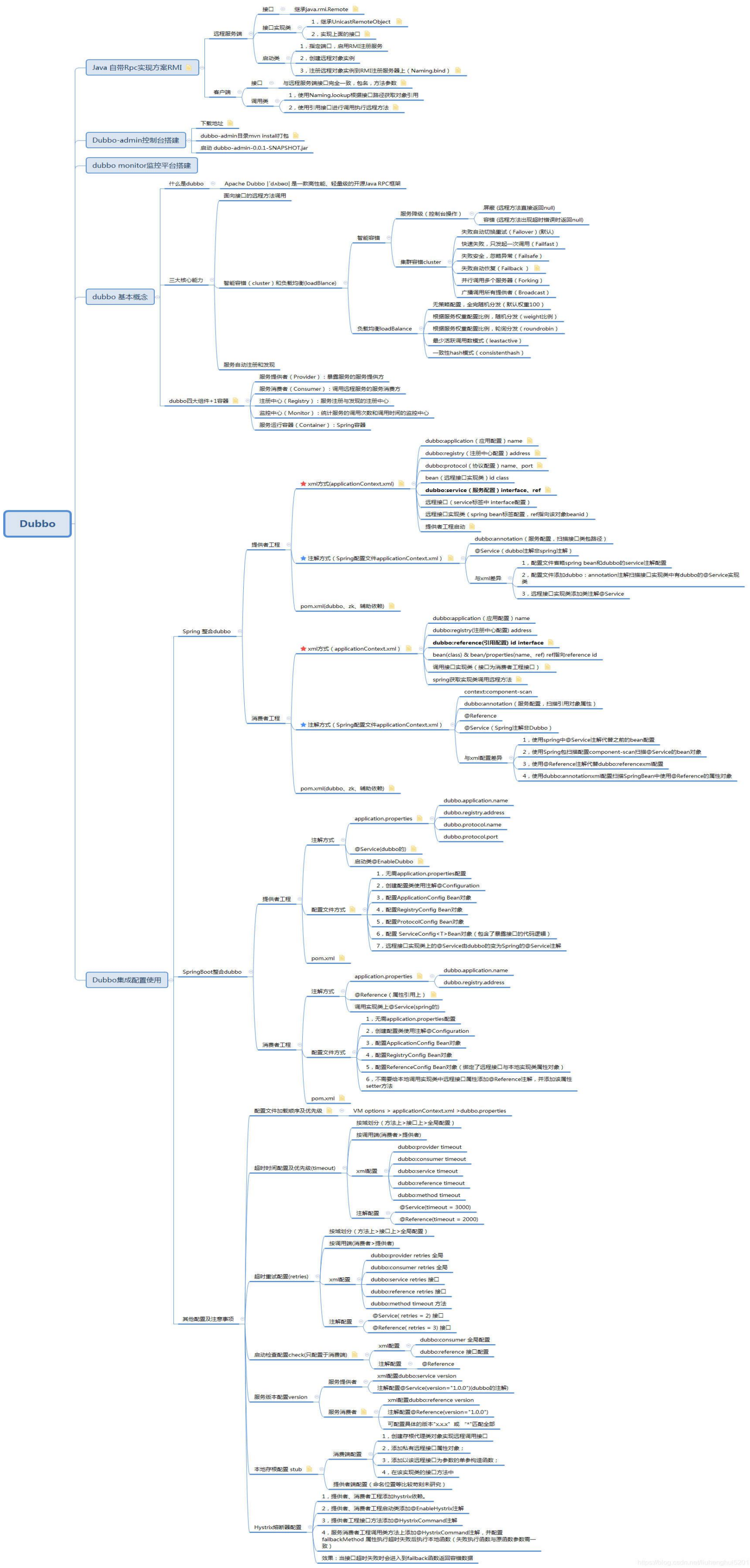
相同点：都需要保存和读取数据/状态，变更通知。
不同点：配置是全局非业务参数，注册中心是运行期临时状态，元数据是业务模型。
服务的负载均衡：（Service LoadBalance）
跟 Nginx 的负载均衡一样。
多个不同策略，原理不同，目的基本一致（尽量均匀）
1. Random (带权重)
2. RoundRobin(轮询)
3. LeastActive(快的多给)
4. ConsistentHashLoadBalance(同样参数请求到一个提供者)
服务流控：
1. 限流（内部线程数，外部调用数 OR 数据量）
2. 服务降级（去掉不必要的业务逻辑，只保留核心逻辑）
3. 过载保护（系统短时间不提供新的业务处理服务，积压处理完后在恢复输入请求）

微服务架构阶段：
准备阶段：调研—》分析—》规划—》组织
实施阶段：拆分—》部署—》治理—》改进
微服务架构关注维度：
遗留系统改造&自动化管理&恰当粒度拆分&分布式事务&扩展立方体&完善监控体系
微服务的演化过程：
1. 功能剥离，数据解耦
2. 自然演进，逐步拆分
3. 小步快跑，快速迭代
4. 灰度发布，谨慎试错
5. 提质量线，还技术债

Spring-Cloud (Feign/Ribbon)
Feign 的核心功能就是，作为 HTTP Client 访问 REST 服务接口。
优势在于：
1. 全都基于注解，简单方便。
2. 跟 XXTemplate 一样，内置了简单操作，OOP。
3. 跟其他组件，ribbon, hytrix 联合使用。

Ribbon 是用于云环境的一个客户端内部通信（IPC）库。
特性：
1. 负载均衡。
2. 容错。
3. 多协议支持（HTTP，TCP，UDP），特别是异步和反应式下
4. 缓存和批处理

知识图谱如下：



Dubbo
Java 自带Rpc实现方案RMI
远程服务端
接口
继承java.rmi.Remote
接口实现类
1，继承UnicastRemoteObject
2，实现上面的接口
启动类
1，指定端口，启用RMI注册服务
2，创建远程对象实例
3，注册远程对象实例到RMI注册服务器上（Naming.bind）
客户端
接口
与远程服务端接口完全一致，包名，方法参数
调用类
1，使用Naming.lookup根据接口路径获取对象引用
2，使用引用接口进行调用执行远程方法
Dubbo-admin控制台搭建
下载地址
dubbo-admin目录mvn install打包
启动 dubbo-admin-0.0.1-SNAPSHOT.jar
dubbo monitor监控平台搭建
dubbo 基本概念
什么是dubbo
Apache Dubbo |ˈdʌbəʊ| 是一款高性能、轻量级的开源Java RPC框架
三大核心能力
面向接口的远程方法调用
智能容错（cluster）和负载均衡(loadBlance)
智能容错
服务降级（控制台操作）
屏蔽 (远程方法直接返回null)
容错 (远程方法出现超时错误时返回null)
集群容错cluster
失败自动切换重试（Failover）(默认)
快速失败，只发起一次调用（Failfast）
失败安全，忽略异常（Failsafe）
失败自动恢复（Failback）
并行调用多个服务器（Forking）
广播调用所有提供者（Broadcast）
负载均衡loadBalance
无策略配置，全完随机分发（默认权重100）
根据服务权重配置比例，随机分发（weight比例）
根据服务权重配置比例，轮询分发（roundrobin）
最少活跃调用数模式（leastactive）
一致性hash模式（consistenthash）
服务自动注册和发现
dubbo四大组件+1容器
服务提供者（Provider）：暴露服务的服务提供方
服务消费者（Consumer）：调用远程服务的服务消费方
注册中心（Registry）：服务注册与发现的注册中心
监控中心（Monitor）：统计服务的调用次数和调用时间的监控中心
服务运行容器（Container）：Spring容器
Dubbo集成配置使用
Spring 整合dubbo
提供者工程
xml方式(applicationContext.xml)
dubbo:application（应用配置）name
dubbo:registry（注册中心配置）address
dubbo:protocol（协议配置）name、port
bean（远程接口实现类）id class
dubbo:service（服务配置）interface、ref
远程接口（service标签中 interface配置）
远程接口实现类（spring bean标签配置，ref指向该对象beanid）
提供者工程启动
注解方式（Spring配置文件applicationContext.xml）
dubbo:annotation（服务配置，扫描接口类包路径）
@Service（dubbo注解非spring注解）
与xml差异
1，配置文件省略spring bean和dubbo的service注解配置
2，配置文件添加dubbo：annotation注解扫描接口实现类中有dubbo的@Service实现类
3，远程接口实现类添加类注解@Service
pom.xml(dubbo、zk、辅助依赖)
消费者工程
xml方式（applicationContext.xml）
dubbo:application（应用配置）name
dubbo:registry(注册中心配置) address
dubbo:reference(引用配置) id interface
bean(class) & bean/properties(name、ref) ref指向reference id
调用接口实现类（接口为消费者工程接口）
spring获取实现类调用远程方法
注解方式（Spring配置文件applicationContext.xml）
context:component-scan
dubbo:annotation（服务配置，扫描引用对象属性）
@Reference
@Service（Spring注解非Dubbo）
与xml配置差异
1，使用spring中@Service注解代替之前的bean配置
2，使用Spring包扫描配置component-scan扫描@Service的bean对象
3，使用@Reference注解代替dubbo:referencexml配置
4，使用dubbo:annotationxml配置扫描SpringBean中使用@Reference的属性对象
pom.xml(dubbo、zk、辅助依赖)
SpringBoot整合dubbo
提供者工程
注解方式
application.properties
dubbo.application.name
dubbo.registry.address
dubbo.protocol.name
dubbo.protocol.port
@Service(dubbo的)
启动类@EnableDubbo
配置文件方式
1，无需application.properties配置
2，创建配置类使用注解@Configuration
3，配置ApplicationConfig Bean对象
4，配置RegistryConfig Bean对象
5，配置ProtocolConfig Bean对象
6，配置 ServiceConfig<T>Bean对象（包含了暴露接口的代码逻辑）
7，远程接口实现类上的@Service由dubbo的变为Spring的@Service注解
pom.xml
消费者工程
注解方式
application.properties
dubbo.application.name
dubbo.registry.address
@Reference（属性引用上）
调用实现类上@Service(spring的)
配置文件方式
1，无需application.properties配置
2，创建配置类使用注解@Configuration
3，配置ApplicationConfig Bean对象
4，配置RegistryConfig Bean对象
5，配置ReferenceConfig Bean对象（绑定了远程接口与本地实现类属性对象）
6，不需要给本地调用实现类中远程接口属性添加@Reference注解，并添加该属性setter方法
pom.xml
其他配置及注意事项
配置文件加载顺序及优先级
VM options > applicationContext.xml >dubbo.properties
超时时间配置及优先级(timeout)
按域划分（方法上>接口上>全局配置）
按调用端(消费者>提供者)
xml配置
dubbo:provider timeout
dubbo:consumer timeout
dubbo:service timeout
dubbo:reference timeout
dubbo:method timeout
注解配置
@Service(timeout = 3000)
@Reference(timeout = 2000)
超时重试配置(retries)
按域划分（方法上>接口上>全局配置）
按调用端(消费者>提供者)
xml配置
dubbo:provider retries 全局
dubbo:consumer retries 全局
dubbo:service retries 接口
dubbo:reference retries 接口
dubbo:method timeout 方法
注解配置
@Service(retries = 2) 接口
@Reference(retries = 3) 接口
启动检查配置check(只配置于消费端)
xml配置
dubbo:consumer 全局配置
dubbo:reference 接口配置
注解配置
@Reference
服务版本配置version
服务提供者
xml配置dubbo:service version
注解配置@Service(version="1.0.0")(dubbo的注解)
服务消费者
xml配置dubbo:reference version
注解配置@Reference(version="1.0.0")
可配置具体的版本"x.x.x" 或 "*"匹配全部
本地存根配置 stub
消费端配置
1，创建存根代理类对象实现远程调用接口
2，添加私有远程接口属性对象；
3，添加以该远程接口为参数的单参构造函数；
4，在该实现类的接口方法中
提供者端配置（命名位置等比较苛刻未研究）
Hystrix熔断器配置
1，提供者、消费者工程添加hystrix依赖。
2，提供者、消费者工程启动类添加@EnableHystrix注解
3，提供者工程接口方法添加@HystrixCommand注解
4，服务消费者工程调用类方法上添加@HystrixCommand注解，并配置failbackMethod 属性执行超时失败后执行本地函数（失败执行函数与原函数参数需一致）
效果：当接口超时失败时会进入到failback函数返回容错数据

（8）分布式缓存

总结如下：

缓存的本质：系统各个处理速度不匹配，导致利用空间换时间。

缓存是提升系统性能的一个简单有效的办法。

了解了缓存的加载时机：

1. 启动全量加载 ==》全局有效，使用简单
2. 懒加载

同步使用加载==》先看缓存是否有数据，没有的话从数据库读取；读取的数据，先放到内存，然后返回给调用方。

延迟异步加载==》从缓存获取数据，不管是否为空直接返回==》策略异步）如果为空，则发起一个异步加载的线程，负责加载数据==》策略解耦）异步线程负责维护缓存的数据，定期或者根据条件触发更新

需要明确，对于数据一致性，性能，成本的综合衡量，是引入缓存的必须指标。

缓存不当容易引发的问题：

1. 系统预热导致启动慢
2. 系统内存资源耗尽

本地缓存 Spring Cache

1. 基于注解和 AOP，使用非常方便。
2. 可以配置 Condition 和 SPEL,非常灵活。
3. 需要注意：绕过 Spring 的话，注解无效。

远程缓存 Redis/Memcached 缓存中间件

Remote Dictionary Server(Redis)是一个由 Salvatore Sanfilippo 写的 key-value 存储系统。

Redies 的使用场景：

1）业务数据缓存

2）业务数据处理

3）全局一致计数

4）高效统计计数

5）发布订阅与 Stream

6）分布式锁

Spring 与 Redis 的结合

核心是 RedisTemplate(可以配置基于 Jedis,Lettuce,Redisson)

使用方法类似 MongoDBTemplate,JDBCTemplate 或者 JPA 封装了基本 redis 命令操作。

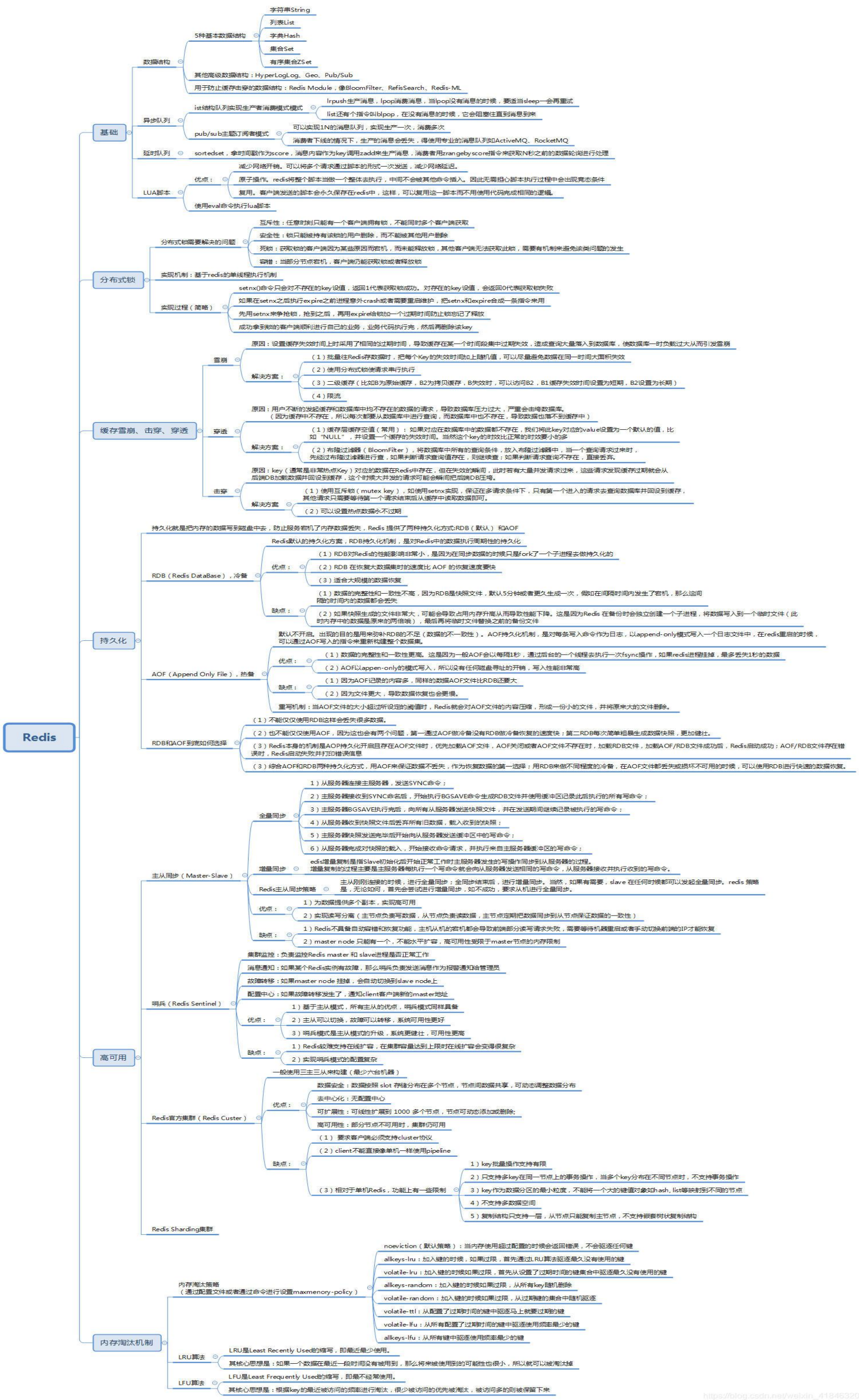
Redis 高级功能：Redis 事务，Redis 管道技术，Redis 数据备份与恢复 --RDB-frm,Redis 数据备份与恢复--AOP-binlog

了解 Redis 使用的经验：

1. 性能：
1） 线程数鱼连接数。
2） 监控系统读写比和缓存命中率
2. 容量：
1）做好容量评估，合理使用缓存资源。
3. 资源管理和分配：
1）尽量每个业务集群单独使用自己的 Redis，不混用。
2）控制 Redis 资源的申请与使用，规范环境和 Key 的管理
3）监控 CPU 100%,优化高延迟的操作。

了解 Redis 的集群和高可用
1）Redis 主从复制：从单机到多节点 类似 MYSQL 主从
2）Redis Sentinel 主从切换： 走向高可用-MHA
3）Redis Cluster: 走向分片 全自己分库分表
4）Java 中配置使用 Redis Sentinel
5) Java 中配置使用 Redis Cluster

知识图谱如下：



（9）分布式消息队列

总结如下：

了解了系统间通信方式：

基于文件，基于共享内存，基于 IPC，基于 SOCKET，基于数据库，基于 PRC

各个模式的缺点：

文件：明显不方便，不及时。

SOCKET：使用麻烦，多数情况下不如 RPC。

数据库：不实时，但是经常有人拿数据库来模拟消息队列。

RPC：调用关系复杂，同步处理，压力大的时候无法缓冲。

MQ 的作用

异步通信，系统解耦，削峰填谷，可靠通信

消息处理的保障：

At most once　至多一次

At least once　至少一次

Exactly once　精确一次

消息处理的事务性：

-通过确定机制实现事务性。

-可以被事务管理器管理，甚至可以支持 XA。

Kafka 是一个分布式的，基于发布/订阅的消息系统。

Broker:kafka 集群包含一个或者多个服务器，这种服务器被称为 broker。

Topic:每条发布到 Kafka 集群的消息都有一个类别，这个类别被称为 Topic。

Partition：Partition 是物理上的概念，每个 Topic 包含一个或多个 Partition。

Producer:Partition 是物理上的概念，每个 Topic 包含一个或多个 Partition。

Consumer：消息消费者，向 Kafka broker 读取消息的客户端。

Consumer Group：每个 Consumer 属于一个特定的 ConsumerGroup(可为每个 Consumer 指定 group name)若不指定 group name 则属于默认的 group。

生产者的执行步骤：客户端实现序列化，分区，压缩操作。

生产者的确认模式：

ack=0: 只发送不管有没有写入到 broker。

ack=1: 写入到 leader 就认为成功。

ack=-1/all:写入到最小的复本数则认为成功。

生产者特性-同步发送&异步发送&顺序保证&消息可靠性传递。

消费者-Consumer Group

消费者特性-Offset 同步提交&异步提交&自动提交&SEEKS

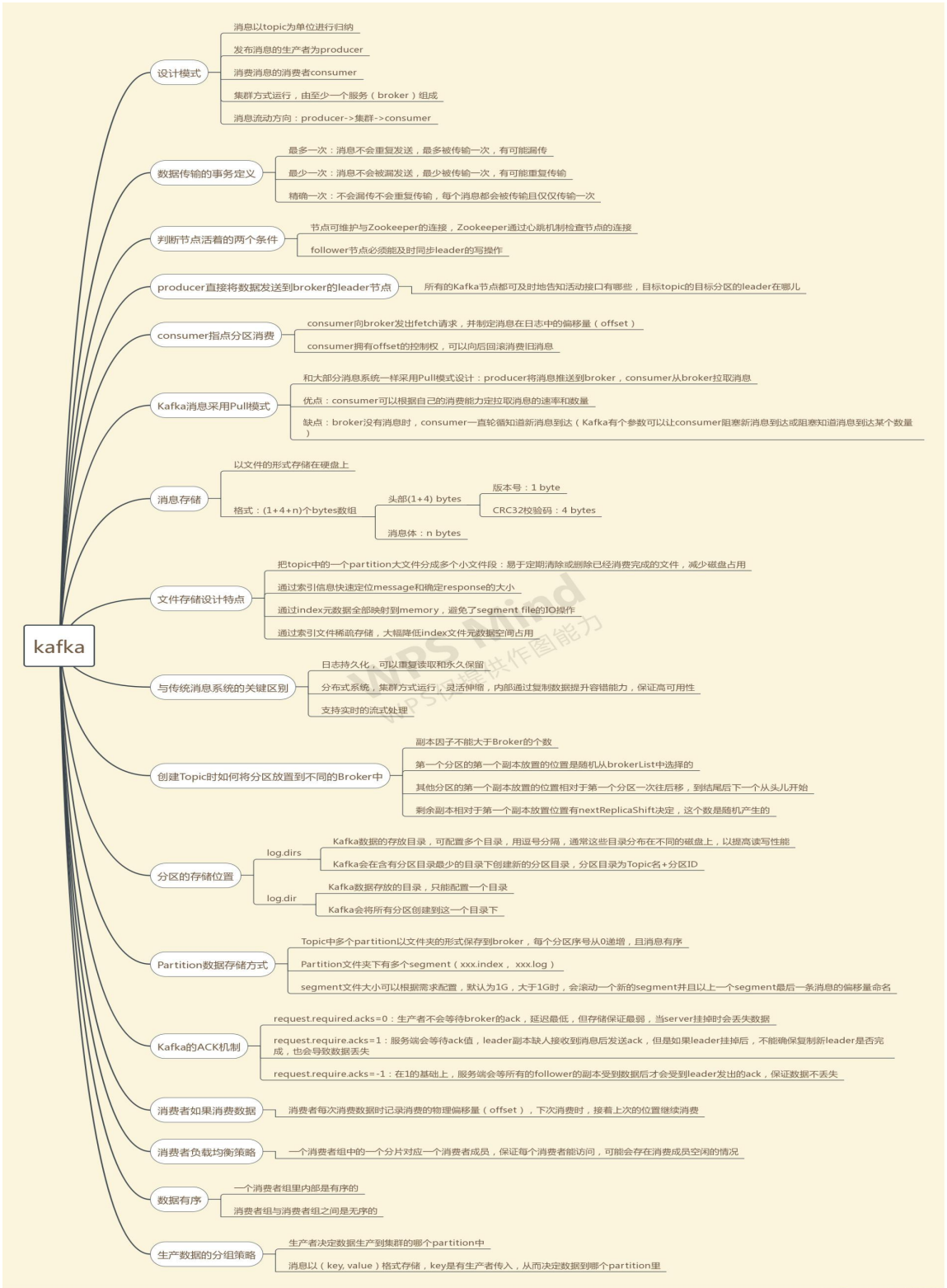
RocketMQ 与 KAFKA 的关系：

从本质上说，两者没有本质区别。

1）纯 JAVA 开发，用不用 ZK

2）支持延迟投递，消息追溯
3）多个队列使用一个日志文件，所以不存着 KAFKA 过多 TOPIC 问题

知识图谱如下：


kafka
设计模式
消息以topic为单位进行归纳
发布消息的生产者为producer
消费消息的消费者consumer
集群方式运行，由至少一个服务（broker）组成
消息流动方向：producer->集群->consumer
数据传输的事务定义
最多一次：消息不会重复发送，最多被传输一次，有可能漏传
最少一次：消息不会被漏发送，最少被传输一次，有可能重复传输
精确一次：不会漏传不会重复传输，每个消息都会被传输且仅仅传输一次
判断节点活着的两个条件
节点可维护与Zookeeper的连接，Zookeeper通过心跳机制检查节点的连接
follower节点必须能及时同步leader的写操作
producer直接将数据发送到broker的leader节点
所有的Kafka节点都可及时地告知活动接口有哪些，目标topic的目标分区的leader在哪儿
consumer指点分区消费
consumer向broker发出fetch请求，并制定消息在日志中的偏移量（offset）
consumer拥有offset的控制权，可以向后回滚消费旧消息
Kafka消息采用Pull模式
和大部分消息系统一样采用Pull模式设计：producer将消息推送到broker，consumer从broker拉取消息
优点：consumer可以根据自己的消费能力定拉取消息的速率和数量
缺点：broker没有消息时，consumer一直轮循知道新消息到达（Kafka有个参数可以让consumer阻塞新消息到达或阻塞知道消息到达某个数量）
消息存储
以文件的形式存储在硬盘上
格式：(1+4+n)个bytes数组
头部(1+4) bytes
版本号：1 byte
CRC32校验码：4 bytes
消息体：n bytes
文件存储设计特点
把topic中的一个partition大文件分成多个小文件段：易于定期清除或删除已经消费完成的文件，减少磁盘占用
通过索引信息快速定位message和确定response的大小
通过index元数据全部映射到memory，避免了segment file的IO操作
通过索引文件稀疏存储，大幅降低index文件元数据空间占用
与传统消息系统的关键区别
日志持久化，可以重复读取和永久保留
分布式系统，集群方式运行，灵活伸缩，内部通过复制数据提升容错能力，保证高可用性
支持实时的流式处理
创建Topic时如何将分区放置到不同的Broker中
副本因子不能大于Broker的个数
第一个分区的第一个副本放置的位置是随机从brokerList中选择的
其他分区的第一个副本放置的位置相对于第一个分区一次往后移，到结尾后下一个从头儿开始
剩余副本相对于第一个副本放置位置有nextReplicaShift决定，这个数是随机产生的
分区的存储位置
log.dirs
Kafka数据的存放目录，可配置多个目录，用逗号分隔，通常这些目录分布在不同的磁盘上，以提高读写性能
Kafka会在含有分区目录最少的目录下创建新的分区目录，分区目录为Topic名+分区ID
log.dir
Kafka数据存放的目录，只能配置一个目录
Kafka会将所有分区创建到这一个目录下
Partition数据存储方式
Topic中多个partition以文件夹的形式保存到broker，每个分区序号从0递增，且消息有序
Partition文件夹下有多个segment（xxx.index， xxx.log）
segment文件大小可以根据需求配置，默认为1G，大于1G时，会滚动一个新的segment并且以上一个segment最后一条消息的偏移量命名
Kafka的ACK机制
request.required.acks=0：生产者不会等待broker的ack，延迟最低，但存储保证最弱，当server挂掉时会丢失数据
request.require.acks=1：服务端会等待ack值，leader副本缺人接收到消息后发送ack，但是如果leader挂掉后，不能确保复制新leader是否完成，也会导致数据丢失
request.require.acks=-1：在1的基础上，服务端会等所有的follower的副本受到数据后才会受到leader发出的ack，保证数据不丢失
消费者如果消费数据
消费者每次消费数据时记录消费的物理偏移量（offset），下次消费时，接着上次的位置继续消费
消费者负载均衡策略
一个消费者组中的一个分片对应一个消费者成员，保证每个消费者能访问，可能会存在消费成员空闲的情况
数据有序
一个消费者组里内部是有序的
消费者组与消费者组之间是无序的
生产数据的分组策略
生产者决定数据生产到集群的哪个partition中
消息以（key, value）格式存储，key是有生产者传入，从而决定数据到哪个partition里

毕业总结：

综上所述，经过三个月的学习，针对 JVM,NIO,并发编程,Spring 和 ORM 等框架,MySql 数据库和 SQL,分库分表,RPC 和微服务,分布式缓存，分布式消息队列的知识点有了系统性的了解，并且很好的整合之前的知识点，并且通过实践的练习和反馈，确实起到了进阶的作用，非常感谢老师和各位助教的付出。