# 1 java基础

## 算法

### 希尔排序

### 冒泡排序

### 快速排序

### 直接选择排序

### 堆排序

### 归并排序

### 基数排序

### 直接插入排序

### 二叉查找树

### 红黑树

### B树

### B+树

### LSM树（分别有对应的应用，数据库、HBase）

### BitSet解决数据重复和是否存在等问题

## 设计模式

### 单例模式

### 工厂模式

### 抽象工厂模式

### 适配器模式

### 代理模式

### 策略模式

### 装饰者模式

### 观察者设计模式

### ThreadLocal设计模式

## java基本

### 字符串常量池的迁移

### 字符串KMP算法

### equals和hashcode

### 泛型

### 异常

### java反射机制

### string的hash算法

### hash冲突的解决办法：拉链法

### foreach循环的原理

### static、final、transient等关键字的作用

### volatile关键字的底层实现原理

### Collections.sort方法使用的是哪种排序方法

### Future接口，常见的线程池中的FutureTask实现等

### string的intern方法的内部细节

### StringTable的实现

### 集合（set Map list heap）

HashMap、HashSet、ArrayList、LinkedList、HashTable、ConcurrentHashMap、TreeMap的实现原理

HashMap的并发问题

ConcurrentLinkedQueue的实现原理

## 正则表达式

* 4.1 捕获组和非捕获组
* 4.2 贪婪，勉强，独占模式

# 多线程及并发容器

## 内部执行原理

## 锁

## synchronized和volatile理解

## Unsafe类的原理，使用它来实现CAS。因此诞生了AtomicInteger系列等

## CAS可能产生的ABA问题的解决，如加入修改次数、版本号

## 同步器AQS的实现原理

## 独占锁、共享锁；可重入的独占锁ReentrantLock、共享锁 实现原理

## 公平锁和非公平锁

## 读写锁 ReentrantReadWriteLock的实现原理

## LockSupport工具

## Condition接口及其实现原理

## Fork/Join框架

## CountDownLatch和CyclicBarrier

# JVM

## 类的加载机制

## jvm内存结构

## GC算法与垃圾回收

## JVM调优

二.类的加载机制

1.什么是类的加载

2.类的生命周期

3.类加载器

4.双亲委派模型

三.什么是类的加载

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

四.类的生命周期

1.加载，查找并加载类的二进制数据，在Java堆中也创建一个java.lang.Class类的对象

2.连接，连接又包含三块内容：验证、准备、初始化。1）验证，文件格式、元数据、字节码、符号引用验证；2）准备，为类的静态变量分配内存，并将其初始化为默认值；3）解析，把类中的符号引用转换为直接引用

3.初始化，为类的静态变量赋予正确的初始值

4.使用，new出对象程序中使用

5.卸载，执行垃圾回收

五.类加载器

1.启动类加载器：Bootstrap ClassLoader，负责加载存放在JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库

2.扩展类加载器：Extension ClassLoader，该加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载DK\jre\lib\ext目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.\*开头的类），开发者可以直接使用扩展类加载器。

3.应用程序类加载器：Application ClassLoader，该类加载器由sun.misc.Launcher$AppClassLoader来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器

六.类加载机制

1.全盘负责，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入

2.父类委托，先让父类加载器试图加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类

3.缓存机制，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效

七.jvm内存结构

1.方法区和对是所有线程共享的内存区域；而java栈、本地方法栈和程序员计数器是运行是线程私有的内存区域。

2.Java堆（Heap）,是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

3.方法区（Method Area）,方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

4.程序计数器（Program Counter Register）,程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。

5.JVM栈（JVM Stacks）,与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

6.本地方法栈（Native Method Stacks）,本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

八.对象分配规则

1.对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。

2.大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。

3.长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。

4.动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。

5.空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Monitor GC,如果false则进行Full GC。

九.GC算法

GC最基础的算法有三种：标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

1.标记 -清除算法，“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。

2.复制算法，“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

3.标记-压缩算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

4.分代收集算法，“分代收集”（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

十.垃圾回收器

1.Serial收集器，串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。

2.ParNew收集器，ParNew收集器其实就是Serial收集器的多线程版本。

3.Parallel收集器，Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的吞吐量。

4.Parallel Old 收集器，Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法

5.CMS收集器，CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。

6.G1收集器，G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足GC停顿时间要求的同时,还具备高吞吐量性能特征

7.GC算法和垃圾回收器算法图解以及更详细内容参考JVM（3）：Java GC算法 垃圾收集器

十一.GC日志分析

摘录GC日志一部分（前部分为年轻代gc回收；后部分为full gc回收）：

2016-07-05T10:43:18.093+0800: 25.395: [GC [PSYoungGen: 274931K->10738K(274944K)] 371093K->147186K(450048K), 0.0668480 secs] [Times: user=0.17 sys=0.08, real=0.07 secs]

2016-07-05T10:43:18.160+0800: 25.462: [Full GC [PSYoungGen: 10738K->0K(274944K)] [ParOldGen: 136447K->140379K(302592K)] 147186K->140379K(577536K) [PSPermGen: 85411K->85376K(171008K)], 0.6763541 secs] [Times: user=1.75 sys=0.02, real=0.68 secs]

通过上面日志分析得出，PSYoungGen、ParOldGen、PSPermGen属于Parallel收集器。其中PSYoungGen表示gc回收前后年轻代的内存变化；ParOldGen表示gc回收前后老年代的内存变化；PSPermGen表示gc回收前后永久区的内存变化。young gc 主要是针对年轻代进行内存回收比较频繁，耗时短；full gc 会对整个堆内存进行回城，耗时长，因此一般尽量减少full gc的次数

十二.调优命令

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

1.jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。

2.jstat，JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

3.jmap，JVM Memory Map命令用于生成heap dump文件

4.jhat，JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看

5.jstack，用于生成java虚拟机当前时刻的线程快照。

6.jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

详细的命令使用参考这里JVM（4）：Jvm调优-命令篇

十三.调优工具

常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

1.jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控

2.jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。

3.MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗

4.GChisto，一款专业分析gc日志的工具

* 5.1 类加载机制，也就是双亲委派模型
* 5.2 java内存分配模型（默认HotSpot）

线程共享的：堆区、永久区 线程独享的：虚拟机栈、本地方法栈、程序计数器

* 5.3 内存分配机制：年轻代（Eden区、两个Survivor区）、年老代、永久代以及他们的分配过程
* 5.4 强引用、软引用、弱引用、虚引用与GC
* 5.5 happens-before规则
* 5.6 指令重排序、内存栅栏
* 5.7 Java 8的内存分代改进
* 5.8 垃圾回收算法：

标记-清除（不足之处：效率不高、内存碎片）

复制算法（解决了上述问题，但是内存只能使用一半，针对大部分对象存活时间短的场景，引出了一个默认的8:1:1的改进，缺点是仍然需要借助外界来解决可能承载不下的问题）

标记整理

* 5.8 常用垃圾收集器：

新生代：Serial收集器、ParNew收集器、Parallel Scavenge 收集器

老年代：Serial Old收集器、Parallel Old收集器、CMS（Concurrent Mark Sweep）收集器、 G1 收集器（跨新生代和老年代）

* 5.9 常用gc的参数：-Xmn、-Xms、-Xmx、-XX:MaxPermSize、-XX:SurvivorRatio、-XX:-PrintGCDetails
* 5.10 常用工具： jps、jstat、jmap、jstack、图形工具jConsole、Visual VM、MAT

# web框架

## Spring

### Spring原理

### IOC

### AOP

* 2.1 AOP的实现分类：编译期、字节码加载前、字节码加载后三种时机来实现AOP
* 2.2 深刻理解其中的角色：AOP联盟、aspectj、jboss AOP、Spring自身实现的AOP、Spring嵌入aspectj。特别是能用代码区分后两者
* 2.3 接口设计：

AOP联盟定义的概念或接口：Pointcut（概念，没有定义对应的接口）、Joinpoint、Advice、MethodInterceptor、MethodInvocation

SpringAOP针对上述Advice接口定义的接口及其实现类：BeforeAdvice、AfterAdvice、MethodBeforeAdvice、AfterReturningAdvice；针对aspectj对上述接口的实现AspectJMethodBeforeAdvice、AspectJAfterReturningAdvice、AspectJAfterThrowingAdvice、AspectJAfterAdvice。

SpringAOP定义的定义的AdvisorAdapter接口：将上述Advise转化为MethodInterceptor

SpringAOP定义的Pointcut接口：含有两个属性ClassFilter（过滤类）、MethodMatcher（过滤方法）

SpringAOP定义的ExpressionPointcut接口：实现中会引入aspectj的pointcut表达式

SpringAOP定义的PointcutAdvisor接口（将上述Advice接口和Pointcut接口结合起来）

* 2.4 SpringAOP的调用流程
* 2.5 SpringAOP自己的实现方式（代表人物ProxyFactoryBean）和借助aspectj实现方式区分

### Spring事务

#### jdbc事务存在的问题

#### Hibernate对事务的改进

#### 针对各种各样的事务，Spring如何定义事务体系的接口，以及如何融合jdbc事务和Hibernate事务的

#### 三种事务模型包含的角色以及各自的职责

#### 事务代码也业务代码分离的实现（AOP+ThreadLocal来实现）

#### Spring事务拦截器TransactionInterceptor全景

#### X/Open DTP模型，两阶段提交，JTA接口定义

#### Jotm、Atomikos的实现原理

#### 事务的传播属性

#### PROPAGATION\_REQUIRES\_NEW、PROPAGATION\_NESTED的实现原理以及区别

#### 事物的挂起和恢复的原理

### SpringSecurity

## Spring MVC

### SpringMVC原理详解

### SpringMVC对请求的整体处理流程

* 1.1 servlet开发存在的问题：映射问题、参数获取问题、格式化转换问题、返回值处理问题、视图渲染问题
* 1.2 SpringMVC为解决上述问题开发的几大组件及接口：HandlerMapping、HandlerAdapter、HandlerMethodArgumentResolver、HttpMessageConverter、Converter、GenericConverter、HandlerMethodReturnValueHandler、ViewResolver、MultipartResolver
* 1.3 DispatcherServlet、容器、组件三者之间的关系

## Mybatis

最原始的jdbc->Spring的JdbcTemplate->hibernate->JPA->SpringDataJPA的演进之路

## Hibernate

# 数据库

## Mysql

### 数据库隔离级别

* Read uncommitted：读未提交
* Read committed ： 读已提交
* Repeatable read：可重复读
* Serializable ：串行化

### Mysql性能优化

### Mysql锁

深入理解mysql的Record Locks、Gap Locks、Next-Key Locks

例如下面的在什么情况下会出现死锁：

start transaction; DELETE FROM t WHERE id =6; INSERT INTO t VALUES(6); commit;

* 5.3 insert into select语句的加锁情况

### Mysql事务

* 事务的ACID特性概念

### innodb的MVCC理解

### undo redo binlog

* + 1 undo redo 都可以实现持久化，他们的流程是什么？为什么选用redo来做持久化？
  + 2 undo、redo结合起来实现原子性和持久化，为什么undo log要先于redo log持久化？
  + 3 undo为什么要依赖redo？
  + 4 日志内容可以是物理日志，也可以是逻辑日志？他们各自的优点和缺点是？
  + 5 redo log最终采用的是物理日志加逻辑日志，物理到page，page内逻辑。还存在什么问题？怎么解决？Double Write
  + 6 undo log为什么不采用物理日志而采用逻辑日志？
  + 7 为什么要引入Checkpoint？
  + 8 引入Checkpoint后为了保证一致性需要阻塞用户操作一段时间，怎么解决这个问题？（这个问题还是很有普遍性的，redis、ZooKeeper都有类似的情况以及不同的应对策略）又有了同步Checkpoint和异步Checkpoint
  + 9 开启binlog的情况下，事务内部2PC的一般过程（含有2次持久化，redo log和binlog的持久化）
  + 10 解释上述过程，为什么binlog的持久化要在redo log之后，在存储引擎commit之前？
  + 11 为什么要保持事务之间写入binlog和执行存储引擎commit操作的顺序性？（即先写入binlog日志的事务一定先commit）
  + 12 为了保证上述顺序性，之前的办法是加锁prepare\_commit\_mutex，但是这极大的降低了事务的效率，怎么来实现binlog的group commit？
  + 13 怎么将redo log的持久化也实现group commit？至此事务内部2PC的过程，2次持久化的操作都可以group commit了，极大提高了效率

# 中间件及NOSQL数据库

## Kafka

### kafka的文件存储设计

kafka本身做的很轻量级来保持高效，很多高级特性没有：事务、优先级的消息、消息的过滤，更重要的是服务治理不健全，一旦出问题，不能直观反应出来，不太适合对数据要求十分严苛的企业级系统，而适合日志之类并发量大但是允许少量的丢失或重复等场景

### kafka的副本复制过程

### kafka的leader选举机制

### kafka的消息丢失问题

### kafka的消息顺序性问题

### kafka的isr设计和过半对比

## Redis

### redis对客户端的维护和管理，读写缓冲区

### redis事务的实现

### Jedis客户端的实现

### JedisPool以及ShardedJedisPool的实现

### redis epoll实现，循环中的文件事件和时间事件

### redis的RDB持久化，save和bgsave

### redis AOF命令追加、文件写入、文件同步到磁盘

### redis AOF重写，为了减少阻塞时间采取的措施

### redis的LRU内存回收算法

### redis的master slave复制

### redis的sentinel高可用方案

### redis的cluster分片方案

## RabbitMQ

## mongodb

## ActiveMQ

# web服务器

## Tomcat

### tomcat的整体架构设计

### tomcat对通信的并发控制

### http请求到达tomcat的整个处理流程

## Nginx

### Nginx工作原理

TODO

# 分布式

## ZooKeeper

* 1.1 客户端架构
* 1.2 服务器端单机版和集群版，对应的请求处理器
* 1.3 集群版session的建立和激活过程
* 1.4 Leader选举过程
* 1.5 事务日志和快照文件的详细解析
* 1.6 实现分布式锁、分布式ID分发器
* 1.7 实现Leader选举
* 1.8 ZAB协议实现一致性原理

**3.3 dubbo**

* 3.1 dubbo扩展机制的实现，对比SPI机制
* 3.2 服务的发布过程
* 3.3 服务的订阅过程
* 3.4 RPC通信的设计

# 序列化和反序列化框架

## Avro

## Thrift

## Protobuf

# 通信

## TCP/IP

原理（三次握手）

TCP握手和断开及有限状态机

## HTTP

## IO

### 阻塞/非阻塞的区别

### 同步/异步的区别

### 阻塞IO

### 非阻塞IO

### 多路复用IO

### 异步IO

## BIO

## NIO

### backlog

### Reactor线程模型

### jdk的poll、epoll与底层poll、epoll的对接实现

### Netty自己的epoll实现

### 内核层poll、epoll的大致实现

### epoll的边缘触发和水平触发

### Netty的EventLoopGroup设计

### Netty的ByteBuf设计

### Netty的ChannelHandler

### Netty的零拷贝

### Netty的线程模型，特别是与业务线程以及资源释放方面的理解

# 微服务

* 11.1 SOA与微服务
* 11.2 服务的合并部署、多版本自动快速切换和回滚

详见[基于Java容器的多应用部署技术实践](https://oss-cn-hangzhou.aliyuncs.com/yqfiles/03e7087f9d56b7b674df903ad9345827.%E5%9F%BA%E4%BA%8EJava%E5%AE%B9%E5%99%A8%E7%9A%84%E5%A4%9A%E5%BA%94%E7%94%A8%E9%83%A8%E7%BD%B2%E6%8A%80%E6%9C%AF%E5%AE%9E%E8%B7%B5.pdf)

* 11.3 服务的治理：限流、降级

具体见 [张开涛大神的架构系列](http://jinnianshilongnian.iteye.com/category/344044)

服务限流：令牌桶、漏桶

服务降级、服务的熔断、服务的隔离：netflix的hystrix组件

* 11.4 服务的线性扩展

无状态的服务如何做线性扩展：

如一般的web应用，直接使用硬件或者软件做负载均衡，简单的轮训机制

有状态服务如何做线性扩展：

如Redis的扩展：一致性hash，迁移工具

* 11.5 服务链路监控和报警：CAT、Dapper、Pinpoint

**3.12 Spring Cloud**

* 12.1 Spring Cloud Zookeeper:用于服务注册和发现
* 12.2 Spring Cloud Config:分布式配置
* 12.2 Spring Cloud Netflix Eureka：用于rest服务的注册和发现
* 12.3 Spring Cloud Netflix Hystrix：服务的隔离、熔断和降级
* 12.4 Spring Cloud Netflix Zuul:动态路由，API Gateway

**3.13 分布式事务**

* 13.1 JTA分布式事务接口定义，对此与Spring事务体系的整合
* 13.2 TCC分布式事务概念
* 13.3 TCC分布式事务实现框架案例1：[tcc-transaction](https://github.com/changmingxie/tcc-transaction)
* 13.3.1 TccCompensableAspect切面拦截创建ROOT事务
* 13.3.2 TccTransactionContextAspect切面使远程RPC调用资源加入到上述事务中，作为一个参与者
* 13.3.3 TccCompensableAspect切面根据远程RPC传递的TransactionContext的标记创建出分支事务
* 13.3.4 全部RPC调用完毕，ROOT事务开始提交或者回滚，执行所有参与者的提交或回滚
* 13.3.5 所有参与者的提交或者回滚，还是通过远程RPC调用，provider端开始执行对应分支事务的confirm或者cancel方法
* 13.3.6 事务的存储，集群共享问题13.3.7 事务的恢复，避免集群重复恢复
* 13.4 TCC分布式事务实现框架案例2：[ByteTCC](https://github.com/liuyangming/ByteTCC)
* 13.4.1 JTA事务管理实现，类比Jotm、Atomikos等JTA实现
* 13.4.2 事务的存储和恢复，集群是否共享问题**调用方创建CompensableTransaction事务，并加入资源**
* 13.4.3 CompensableMethodInterceptor拦截器向spring事务注入CompensableInvocation资源
* 13.4.4 Spring的分布式事务管理器创建作为协调者CompensableTransaction类型事务，和当前线程进行绑定，同时创建一个jta事务
* 13.4.5 在执行sql等操作的时候，所使用的jdbc等XAResource资源加入上述jta事务
* 13.4.6 dubbo RPC远程调用前，CompensableDubboServiceFilter创建出一个代理XAResource，加入上述 CompensableTransaction类型事务，并在RPC调用过程传递TransactionContext**参与方创建分支的CompensableTransaction事务，并加入资源，然后提交jta事务**
* 13.4.7 RPC远程调用来到provider端，CompensableDubboServiceFilter根据传递过来的TransactionContext创建出对应的CompensableTransaction类型事务
* 13.4.8 provider端，执行时遇见@Transactional和@Compensable，作为一个参与者开启try阶段的事务，即创建了一个jta事务
* 13.4.9 provider端try执行完毕开始准备try的提交，仅仅是提交上述jta事务，返回结果到RPC调用端**调用方决定回滚还是提交**
* 13.4.10 全部执行完毕后开始事务的提交或者回滚，如果是提交则先对jta事务进行提交（包含jdbc等XAResource资源的提交），提交成功后再对CompensableTransaction类型事务进行提交，如果jta事务提交失败，则需要回滚CompensableTransaction类型事务。
* 13.4.11 CompensableTransaction类型事务的提交就是对CompensableInvocation资源和RPC资源的提交，分别调用每一个CompensableInvocation资源的confirm，以及每一个RPC资源的提交**CompensableInvocation资源的提交**
* 13.4.12 此时每一个CompensableInvocation资源的confirm又会准备开启一个新的事务，当前线程的CompensableTransaction类型事务已存在，所以这里开启事务仅仅是创建了一个新的jta事务而已
* 13.4.13 针对此，每一个CompensableInvocation资源的confirm开启的事务，又开始重复上述过程，对于jdbc等资源都加入新创建的jta事务中，而RPC资源和CompensableInvocation资源仍然加入到当前线程绑定的CompensableTransaction类型事务
* 13.4.14 当前CompensableInvocation资源的confirm开启的事务执行完毕后，开始执行commit,此时仍然是执行jta事务的提交，提交完毕，一个CompensableInvocation资源的confirm完成，继续执行下一个CompensableInvocation资源的confirm，即又要重新开启一个新的jta事务**RPC资源的提交（参与方CompensableTransaction事务的提交）**
* 13.4.15 当所有CompensableInvocation资源的confirm执行完毕，开始执行RPC资源的commit，会进行远程调用，执行远程provider分支事务的提交，远程调用过程会传递事务id
* 13.4.16 provider端，根据传递过来的事务id找到对应的CompensableTransaction事务，开始执行提交操作，提交操作完成后返回响应**结束**
* 13.4.17 协调者收到响应后继续执行下一个RPC资源的提交，当所有RPC资源也完成相应的提交，则协调者算是彻底完成该事务

**3.14 一致性算法**

需要补充----刘良林

# 大数据(高阶扩展)

## Hadoop

## HIVE

## HBASE

## SPARK

## STORM

## Flume

## Sqoop

## 机器学习算法

## 自然语言处理

# 其他

## 数据库的分库分表mycat

## ELK日志实时处理查询系统

Elasticsearch、Logstash、Kibana

## SpringSecurity、shiro、SSO（单点登录）

* 7.1 Session和Cookie的区别和联系以及Session的实现原理
* 7.2 SpringSecurity的认证过程以及与Session的关系
* 7.3 CAS实现SSO（详见[Cas（01）——简介](http://haohaoxuexi.iteye.com/blog/2128728)）