* JVM

- JVM内存整体结构

- 进程

- 栈、堆、非堆、JVM自身

- 堆：

- 年轻代

- 新生代

- S0

- S1

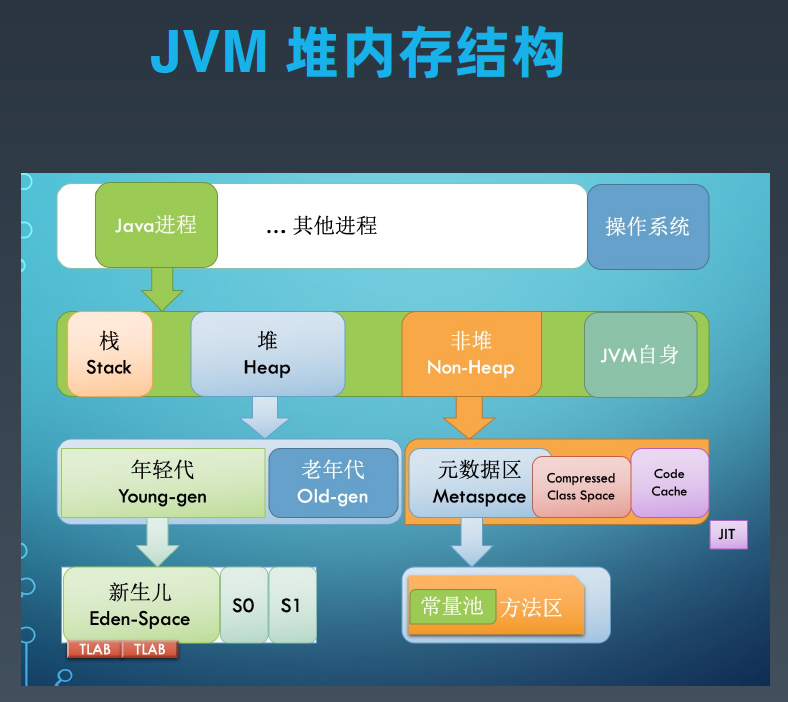
- 老年代

- 非堆

- 元数据区：常量池方法区

- 线程栈

- 帧



- CPU与内存行为

- CPU乱序执行

- volatile关键字

- 原子性操作

- 内存屏障

到目前为止，我们一共了解了Java目前支持的所有GC算法，一共有7类:

- 1. 串行GC（Serial GC）: 单线程执行，应用需要暂停；

- 2. 并行GC（ParNew、Parallel Scavenge、Parallel Old）: 多线程并行地执行垃圾回收，关注与高吞吐；

- 3. CMS（Concurrent Mark-Sweep）: 多线程并发标记和清除，关注与降低延迟；

- 4. G1（G First）: 通过划分多个内存区域做增量整理和回收，进一步降低延迟；

- 5. ZGC（Z Garbage Collector）: 通过着色指针和读屏障，实现几乎全部的并发执行，几毫秒级别的延迟，线性可扩展；

- 6. Epsilon: 实验性的GC，供性能分析使用；

- 7. Shenandoah: G1的改进版本，跟ZGC类似。

可以看出GC算法和实现的演进路线:

1. 串行 -> 并行: 重复利用多核CPU的优势，大幅降低GC暂停时间，提升吞吐量。

2. 并行 -> 并发: 不只开多个GC线程并行回收，还将GC操作拆分为多个步骤，让很多繁重的任务和应用线程 一起并发执行，减少了单次GC暂停持续的时间，这能有效降低业务系统的延迟。

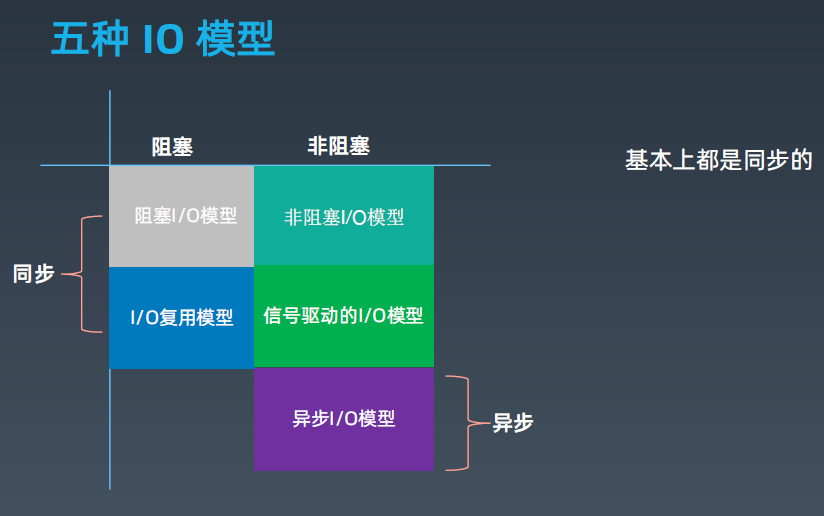
3. CMS -> G1: G1可以说是在CMS基础上进行迭代和优化开发出来的，划分为多个小堆块进行增量回收，这 样就更进一步地降低了单次GC暂停的时间

4. G1 -> ZGC: ZGC号称无停顿垃圾收集器，这又是一次极大的改进。ZGC和G1有一些相似的地方，但是底层 的算法和思想又有了全新的突破。

* NIO

三种经典的I/O模式

有三种经典的I/O模型：BIO（阻塞I/O）,上面例子中的单线程、多线程、线程池都属于BIO模型；NIO（非阻塞IO），上面例子的中Netty是NIO模式；AIO（异步I/O），由于某些原因，虽然性能理论上要比NIO好，但还没有很好的应用。



下面是吃饭场景对比I/O模式

- BIO（阻塞I/O） -- 食堂排队打饭模式：排队在窗口，打好才走

- NIO（非阻塞I/O） -- 点单、等待被叫模式：等待被叫，好了自己去端

- AIO（异步I/O） -- 包厢模式：点单后菜直接被端上桌

阻塞与非阻塞、同步与异步

阻塞和非阻塞是对于通信中的数据而言的：如果没有数据会一直等下去那就是阻塞；不等就是非阻塞

同步和异步是对于怎样取数据而言的：如果数据好了，程序自己去取，那就是同步；如果是数据就绪后回调给程序就是异步

对应吃饭的场景是：

- 阻塞与非阻塞：对于菜而言，阻塞就是一直等菜，反之

- 同步与异步：对应菜好了怎么取，自己去取就是同步，等服务员端上来就是异步

为什么删掉已经做好的AIO支持：

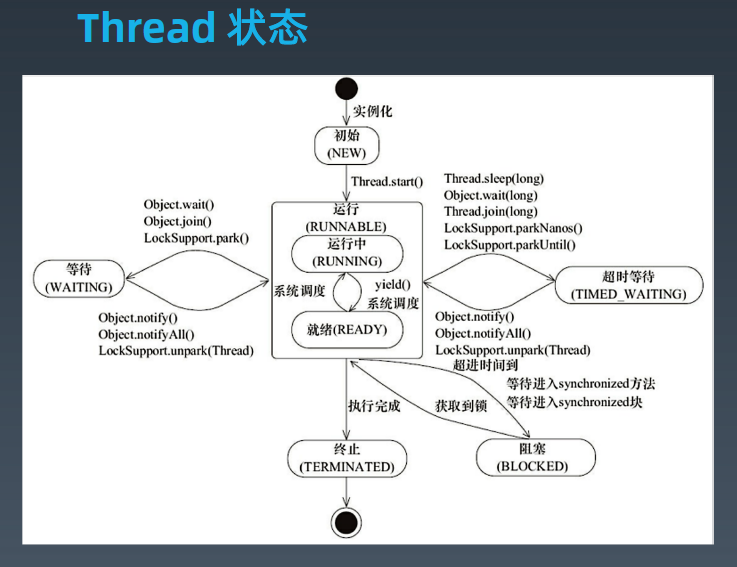
- Windows实现成熟，但很少做服务器

- Linux的AIO不够成熟

- Linux下AIO相比较NIO性能提升不明显

* 并发编程

Java 如何解决多线程的并发问题



解决可见性和有序性问题

可见性的问题是各个线程数据写入了各自的缓存中，直观的解决办法是禁止使用缓存，全部写到内存中

有序性的问题是编译优化的指令重排序，直观的解决办法是禁止编译器优化

但缓存和编译优化的目的都是为了提升程序的性能的，粗暴的全部禁用掉，那性能可能就堪忧了

那合理的方案就是合理禁用缓存和编译优化了，也可以说是局部禁用缓存和编译优化

在 Java 中提供了程序员解决这两方面的问题的方法，这些方法包括 volatile、synchronized 和 final 三个关键字，以及六项 Happens-Before 规则

- Happens-Before 规则：这个可以说是底层规则了，保证了程序的有效性和可见性，具体如下：

- 1.程序的顺序性：程序代码执行顺序

- 2.volatile 变量规则：变量写先与读

- 3.传递性：A 先 B，B 先 C，则 A 先 C

- 4.管程锁定规则：解锁后序与加锁

- 5.线程启动规则：start 先与子线程后续操作

- 6.线程终止规则：子线程中的操作先于 join

- 7.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，可以通过 Thread.interrupted()方法检测到是否有中断发生

- 8.对象终结规则：一个对象的初始化完成(构造函数执行结束)先行发生于它的 finalize()方法的开始

- Volatile：禁用缓存和变量优化

- synchronized：Happens-Before 保证了锁操作相关的可见性和有序性

- final：表示变量只读了，那还需要考虑啥可见性和有序性，告诉编译器随便整了

原子性

原子性的问题是在当前线程还没有完全执行完当前变量的一套操作的时候，发生了线程切换，而且其他的线程对当前变量也有操作，导致了不可预知的错误。

解决的办法就是在当前变量发送操作的时刻，只能有一个线程能进行操作,发生了线程切换那就等待到下一个时间切片，在这期间，不允许其他线程进行操作。王宝令老师专栏的描述是下面这样的，意思应该差不多：

> “同一时刻只有一个线程执行”这个条件非常重要，我们称之为互斥。如果我们能够保证对共享变量的修改是互斥的，那么，无论是单核 CPU 还是多核 CPU，就都能保证原子性了。

里面锁相关的大致是 synchronized 之类的了，这里不进行详细说明了，简单提一下。

并发锁的那些事

锁的用法

锁的基本使用步骤如下：有点像蹲坑，一个坑只能一个人用，后面的人需要等前面的人用完了才能用......

- 1.声明锁：锁可以使用现有的对象，也可以新建；在 Java 中锁有级别，如实例级、对象级等

- 2.加锁：加锁，第一个访问的可以获得锁，后面锁上门，后面的就能不能进入

- 3.访问保证资源：受保护的资源可以是一个也可以是多个，需要注意资源的封装了，集中式的保护起来更容易

- 4.释放锁：访问问以后就开门，让后面人进入

* Spring 和 ORM 等框架

Spring framework 6大模块

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



Spring Boot

Spring Boot 使创建独立运行、生产级别的Spring 应用变得容易，你可以直接运行它。我们对

Spring 平台和第三方库采用限定性视角，以此让大家能在最小的成本下上手。大部分Spring

Boot 应用仅仅需要最少量的配置。

功能特性：

1. 创建独立运行的Spring 应用

2. 直接嵌入Tomcat 或Jetty，Undertow，无需部署WAR 包

3. 提供限定性的starter 依赖简化配置（就是脚手架）

4. 在必要时自动化配置Spring 和其他三方依赖库

5. 提供生产production-ready 特性，例如指标度量，健康检查，外部配置等

6. 完全零代码生产和不需要XML 配置

ORM（Object-Relational Mapping） 表示对象关系映射。

Hibernate 是一个开源的对象关系映射框架，它对JDBC 进行了非常轻量级的对象封装，它将POJO 与数据库表建立映射关系，是一个全自动的orm 框架，hibernate 可以自动生成SQL 语句，自动执行，使得Java 程序员可以使用面向对象的思维来操纵数据库。

MyBatis 是一款优秀的持久层框架，它支持定制化SQL、存储过程以及高级映射。

MyBatis 与Hibernate 比较

考虑为什么大公司都用MyBatis？

MyBatis 与Hibernate 的区别与联系？

Mybatis 优点：原生SQL（XML 语法），直观，对DBA 友好

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

Mybatis 缺点：繁琐，可以用MyBatis-generator、MyBatis-Plus 之类的插件

Hibernate 缺点：对DBA 不友好

Spring/Spring Boot 使用ORM 的经验

1、本地事务（事务的设计与坑）

2、多数据源（配置、静态制定、动态切换）

3、数据库连接池池配置（大小、重连、超时）

4、ORM 内的复杂SQL，级联查询

5、ORM 辅助工具和插件

* MySQL 数据库和 SQL

行级锁（InnoDB)

• 记录锁（Record）: 始终锁定索引记录，注意隐藏的聚簇索引

• 间隙锁（Gap）: 锁住一个范围

• 临键锁（Next-Key）: 记录锁+间隙锁的组合; 可“锁定”表中不存在记录

• 谓词锁（Predicat）: 空间索引

表级锁

意向锁: 表明事务稍后要进行哪种类型的锁定

•共享意向锁（IS）: 打算在某些行上设置共享锁

•排他意向锁（IX）: 打算对某些行设置排他锁

•Insert 意向锁: Insert 操作设置的间隙锁

其他：

•自增锁（AUTO-IN）

•LOCK TABLES/DDL

死锁:

-阻塞与互相等待

-增删改、锁定读

-死锁检测与自动回滚

-锁粒度与程序设计

存储引擎的选择

InnoDB

1. 聚焦索引

2. 锁粒度是行锁

3. InnoDB 支持事务

没有其他特别因素就用InnoDB

ToKuDB

1. 高压缩比，尤其适用于压缩和归档（1:12）

2. 在线添加索引，不影响读写操作

3. 支持完整的ACID 特性和事务机制归档库

* 分库分表



从读写分离到数据库拆分

还有没有其他办法？

主从结构解决了高可用，读扩展，但是单机容量不变，单机写性能无法解决。

提升容量-->分库分表，分布式，多个数据库，作为数据分片的集群提供服务。

降低单个节点的写压力。

提升整个系统的数据容量上限。

数据库垂直拆分

垂直分库分表=> 分布式服务化=> 微服务架构

垂直拆分（拆库）：将一个数据库，拆分成多个提供不同业务数据处理能力的数据库。

垂直拆分（拆表）：如果单表数据量过大，还可能需要对单表进行拆分。

垂直拆分的优缺点

1、单库（单表）变小，便于管理和维护

2、对性能和容量有提升作用

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

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

垂直拆分的一般做法

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

注意：

1、先拆分系统，还是先拆分数据库？

2、先拆分多大范围？

数据库水平拆分

水平分库分表分为，分库、分表、分库分表三类

水平拆分（按主键分库分表）：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。

水平拆分（按时间分库分表）：很多时候，我们的数据是有时间属性的，所以自然可以按照时间维度来拆分。比如当前数据表和历史数据表，甚至按季度，按月，按天来划分不同的表。这样我们按照时间维度来

查询数据时，就可以直接定位到当前的这个子表。更详细的分析参考下一个小节。

强制按条件指定分库分表：比如配置好某些用户的数据进入单独的库表，其他数据默认处理。

自定义方式分库分表：指定某些条件的数据进入到某些库或表。

数据库水平拆分

分库分表有什么优缺点：

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

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

* RPC 和微服务

RPC 是远程过程调用（Remote Procedure Call）的缩写形式。简单来说，就是“像调用本地方法一样调用远程方法”。

RPC 的简化版原理如下图。

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

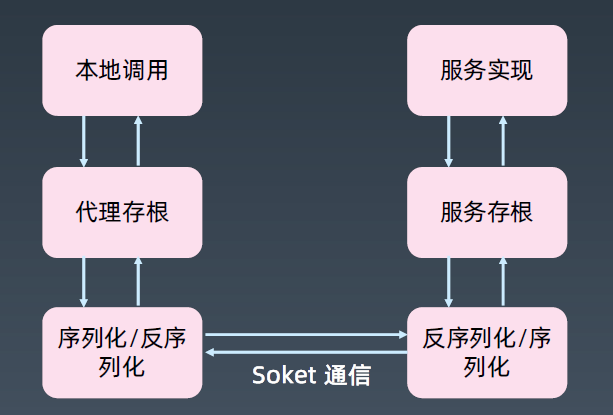
4.远程序列化反序列化

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



常见的RPC 技术

- Corba/RMI/.NET Remoting

- JSON RPC, XML RPC，WebService(Axis2, CXF)

- Hessian, Thrift, Protocol Buffer, gRPC

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。

Dubbo 的主要功能

扩展功能：集群、高可用、管控

- 集群，负载均衡

- 治理，路由，

- 控制台，管理与监控

灵活扩展+简单易用，是Dubbo 成功的秘诀。

* 分布式缓存

缓存策略

资源有限

- 缓存数据容量是必须要考虑的问题

- 思考系统的设计容量、使用容量、峰值，应该是我们做架构设计的一个常识

过期策略

- 按FIFO 或LRU

- 按固定时间过期

- 按业务时间加权：例如3+5x

缓存常见问题

缓存穿透

问题：大量并发查询不存在的KEY，导致都直接将压力透传到数据库。

分析：为什么会多次透传呢？不存在一直为空。

需要注意让缓存能够区分KEY 不存在和查询到一个空值。

解决办法：

1、缓存空值的KEY，这样第一次不存在也会被加载会记录，下次拿到有这个KEY。

2、Bloom 过滤或RoaringBitmap 判断KEY 是否存在。

3、完全以缓存为准，使用延迟异步加载的策略2，这样就不会触发更新。

缓存击穿

问题：某个KEY 失效的时候，正好有大量并发请求访问这个KEY。

分析：跟前面一个其实很像，属于比较偶然的。

解决办法：

1、KEY 的更新操作添加全局互斥锁。

2、完全以缓存为准，使用延迟异步加载的策略2，这样就不会触发更新。

缓存雪崩

问题：当某一时刻发生大规模的缓存失效的情况，会有大量的请求进来直接打到数据库，导致数据

库压力过大甚至宕机。

分析：一般来说，由于更新策略、或者数据热点、缓存服务宕机等原因，可能会导致缓存数据同一

个时间点大规模不可用，或者都更新。所以，需要我们的更新策略要在时间上合适，数据要均匀分

散，缓存服务器要多台高可用。

解决办法：

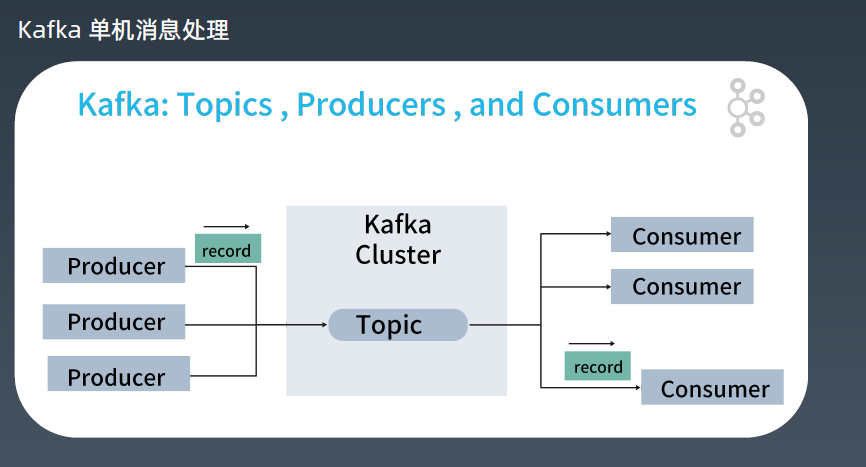
1、更新策略在时间上做到比较均匀。

2、使用的热数据尽量分散到不同的机器上。

3、多台机器做主从复制或者多副本，实现高可用。

4、实现熔断限流机制，对系统进行负载能力控制。

* 分布式消息队列



Kafka 的基本概念

1. Broker：Kafka 集群包含一个或多个服务器，这种服务器被称为broker。

2. 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。（物理上不同Topic 的消息分开存储，逻辑上一个Topic 的消息虽然保存于一个或多个broker 上，但用户只需指定消息的Topic 即可生产或消费数据而不必关心数据存于何处）。

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

4. Producer：负责发布消息到Kafka broker。

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

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

Topic 特性

1. 通过partition 增加可扩展性

2. 通过顺序写入达到高吞吐

3. 多副本增加容错性