

Java

集合类

List

ArrayList

概念

1. 底层使用的**动态数组**实现，能够快速根据下标定位，复杂度为O(1)
2. 如果需要频繁删除数据或者在中间插入数据的话，需要移动数据，复杂度为O(N)，代价较高
3. 该类适合数据不频繁更新的应用场景。

源码

- 默认构造函数是不创建数组的，只有当第一次插入的时候才会创建数组，默认为10
- 每次插入前会先判断数组是否满了，如果满了就需要扩容，扩容为当前的1.5倍，然后将数据拷贝到新数组
- List和数组转化：

```
List list = Arrays.asList(arr); 修改数组会改变list，直接引用return new ArrayList<>(a);
String[] strArr = list.toArrays(); 修改 list 不会改变 数组，拷贝了一份新的对象
System.arraycopy(elementData, 0, a, 0, size);
```

LinkedList

概念

1. 底层使用的是双向链表实现，插入删除效率高，复杂度为O(1)
2. 默认是头插法，如果需要指定下标插入、查找或者指定下标插入复杂度为O (n)
3. 该类型适合频繁插入删除头尾节点的情况。

源码

- 实现了List和Deque接口，除了具备List功能外，还支持双端队列功能。
- 默认构造器什么都不做，内部成员变量保存头尾指针
- remove()移除头结点，add()是头插，add(object)实现了队列接口方法为尾插

ArrayList和LinkedList的区别

1. 底层数据结构：ArrayList使用动态数组，LinkedList使用的是双向链表
2. 效率：
 - 根据下标查询：ArrayList按照下标查询复杂度为O(1)效率高，LinkedList根据下标查询是O(n)
 - 根据值查找：两者都需要遍历，时间复杂度为O(n)
 - 新增和删除：ArrayList尾部插入删除为O(1)，其他位置需要移动元素为O(n)；LinkedList头尾插入删除为O(1)，其他位置需要遍历链表为O(n)

- 内存空间占用：ArrayList底层是数组，内存连续，节省内存。LinkedList是双向链表，除了存储数据外还需要存储前后两个指针更占用空间。
- 线程是否安全：都不是线程安全的。解决办法：在方法内使用，局部变量是线程安全的；加锁或者在外部包裹锁 Collections.synchronizedList();

Set

Map

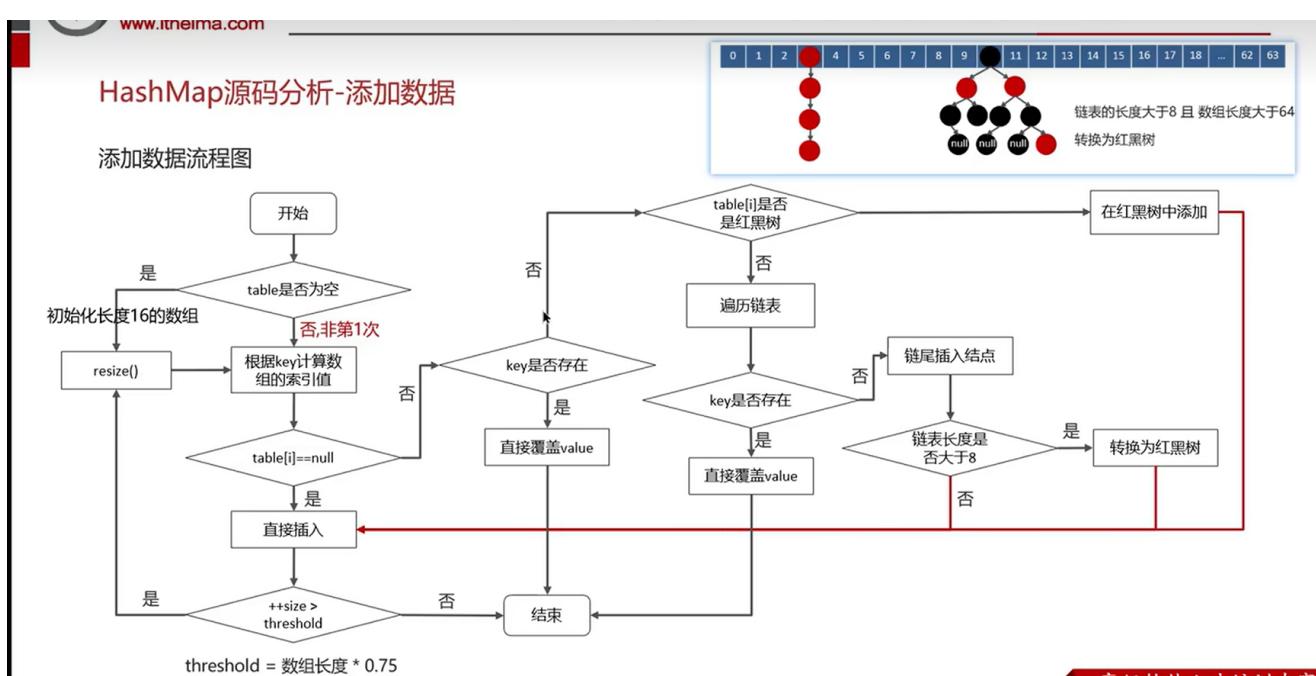
HashMap

实现原理

- 底层实现是数组+链表+红黑树（JDK8）
- 当添加数据时，首先会根据key的hashcode来定位数组下标。
如果不冲突则直接插入，如果哈希冲突，则放入链表或者红黑树中
- 链表转化为红黑树的要求是：**链表长度大于8，数组长度大于64**。如果链表节点小于等于6个，会退化为链表

put()具体流程

- 首先，计算key的hash值
- 第一次插入，需要先初始化数组，默认长度为16
- 第二次插入，如果插入相同的key，则会直接覆盖
- 如果是不同的key，则会进行hash定位，如果没有哈希冲突，直接插入；
- 如果有hash冲突，则判断是否为红黑树，如果是，则在红黑树中插入；
- 如果是链表，则遍历链表判断是否存在key，如果存在则替换；如果不存在则插入，插入后检查是否需要转化为红黑树。
- 插入成功后判断是否超过了阈值（数组长度0.75）如果超过需要扩容，扩大为当前容量的2倍

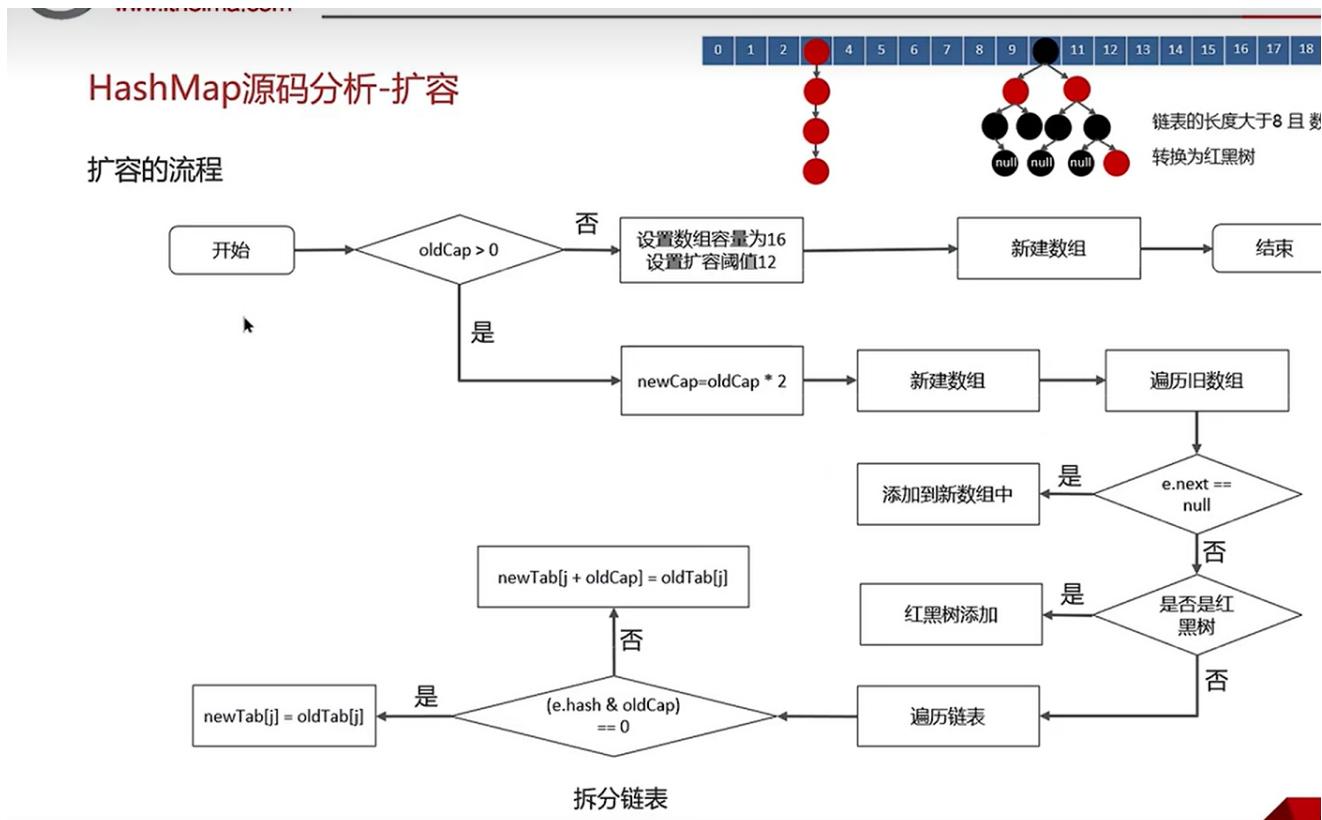


扩容机制

首先，判断旧数组容量是否为0，如果是0，进行初始化，默认容量为16，扩容阈值为12。

如果旧数组长度不为0，则扩大容量为原来的2倍，然后将旧数组里的元素移动到新数组中，共分为三种情况：

- 如果当前hash值位置只有一个元素，则重新hash后直接添加到新数组中
- 如果有多个元素，先判断下是否为红黑树，如果是的话就添加到红黑树中；
- 如果是链表，遍历链表，将元素的hash值与oldCap做&操作来判断是放在原来的位置还是新扩展对应的位置newTab[j+oldCap]（我理解的这里是为了分散同一个hashcode值下挂载的元素数）



寻址算法

使用的算法是：二次哈希法

代码有两处优化：

1. 求hash值：hash值的高16位和低16位进行异或运算(`h = key.hashCode() ^ (h >>> 16)`)，使得hash过后同时具备高16位和低16位的特征
2. 定位数组索引：寻址没有用取模运算而是用按位与操作，是因为位运算效率更高。两者作用是等价的，但是位运算效率更改。但是只能用在被除数是2的倍数上。`tab[i = (n - 1) & hash]`

为什么hashmap长度一定是2的次幂

1. 计算索引时：可以使用位运算效率更高
2. 扩容时：重新计算索引时效率更高。每一个旧元素都需要重新计算位置，位运算效率更高。

多线程

线程基础

进程和线程

1. 关系：进程是正在运行的实例，一个进程包含很多线程，每一个线程执行不同的任务
2. 内存占用：不同线程使用不同的内存空间，在同一个进程中的线程共享内存空间，线程切换上下文成本低
3. 对于JAVA来说：JVM属于进程级别，在JVM内部包含许多线程

并行和并发

1. 线程之间轮流执行称为并发，多个线程同时运行称为并行。
2. 单核CPU只能并发，多核CPU可以并发也可以并行，如果在某一时间点有多个CPU在执行线程称为并行。

创建线程

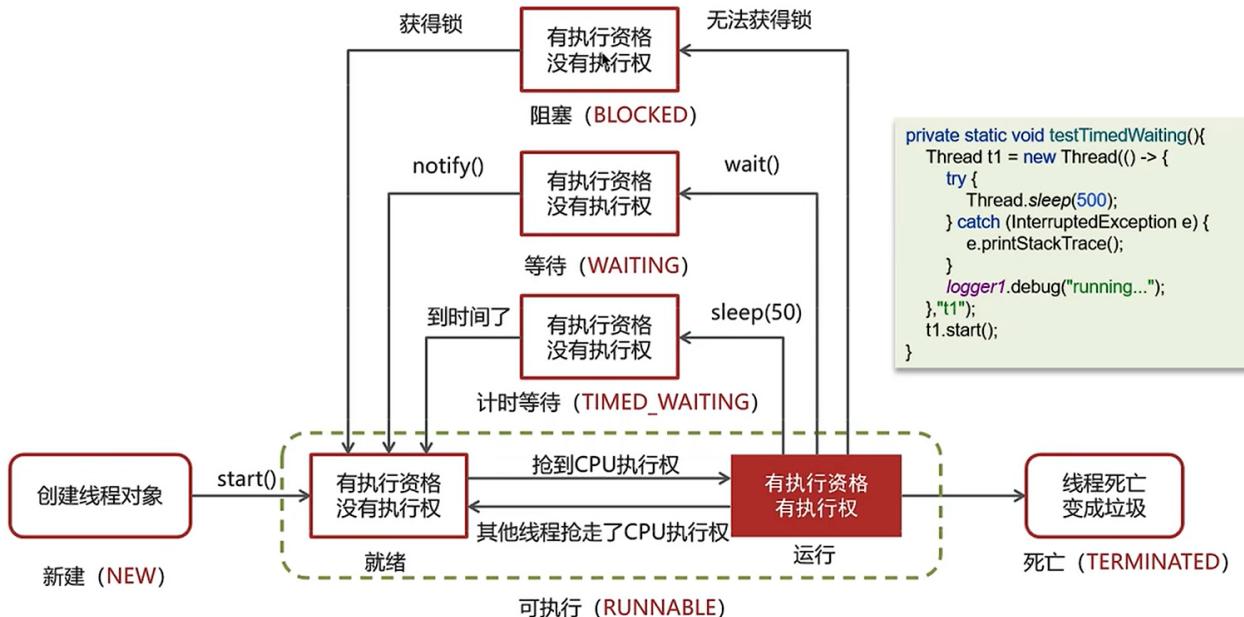
1. 继承Thread类
2. 实现Runner接口
3. 实现Callable接口，可以获取到线程执行结果
4. 使用线程池创建

线程状态

哪些状态：新建，就绪，阻塞，等待，超时等待，终止。

线程转化：

- 创建完线程后状态是新建(NEW)
- 调用start()方法后变为就绪状态(Runnable)
- 获取到CPU后变成执行状态和就绪态公用同一个状态 (Runnable)
- 如果执行完时间片还没结束则返回就绪状态，如果需要拿锁则进入阻塞态 (Blocked)，如果调用object.wait(xxx)或者 Thread.sleep(xxx)进入限期等待状态，如果调用object.wait()或者thread.join()则进入无限期等待状态。
- 运行结束将变为终止状态



如何顺序执行三个线程

- 使用join()：join方法会让出CPU给join的线程，等新加入的线程执行完毕后再继续获取CPU执行下面的代码。实现：t1线程正常运行; t2在线程体先t1.join(); t3在线程体内先t2.join
- 使用condition：创建三个condition分别在三个线程内调用singal()和await()方法来控制执行顺序。

Wait和Sleep区别

共同点：都暂时放弃CPU使用权，进入阻塞状态

不同点：

- sleep是Thread的静态方法，wait方法是object的成员方法，每个对象都有
- sleep执行固定的时间后唤醒，wait方法可以被notify唤醒，也可以指定固定时间后唤醒
- wait调用前必须先获取到锁，否则会报错，调用后释放锁；sleep调用后仍然持有锁，其他线程拿不到锁继续等待

如何停止线程

- 在线程中使用循环访问一个标志位，在线程外部通过改变标志位来控制线程
- 调用interrupt方法中断线程：打断阻塞的线程，会抛出InterruptedException异常t1.interrupt()；如果是非阻塞的状态，类似第一种加flag if(current.isInterrupted())break;
- 使用stop()方法停止，但是有可能损失数据，不推荐使用

并发安全

synchronized关键字

- 是一个互斥的对象锁，在同一时刻只有一个线程持有对象锁。
- 作用在成员方法上，锁的是对象实例this；作用在静态方法上，锁的是类；作用在某个对象上，锁的是该对象。
- 底层实现是由monitor实现，有三个属性：owner（只有一个），entryList（阻塞的线程），waitSet（等待的线程）

锁升级

synchronized有三种形式：

1. 偏向锁：一段很长的时间内都只有一个线程使用锁，可以使用偏向锁。对象头MarkWord存储的是偏向线程的ID
2. 轻量级锁：线程在加锁的时间是错开的，也就是没用竞争，不同线程交替持有锁，可以使用轻量级锁来优化。对象头存储线程栈中的LockRecord指针
3. 重量级锁：多线程竞争锁，底层使用Monitor是像，涉及到用户态和内核态的切换、成本较高，性能比较低。对象头存储指向堆中的monitor对象的指针。

一旦锁发生了竞争，都会升级为重量级锁。

JMM

1. JMM(JAVA Memory Model) JAVA内存模型主要是为了解决多线程数据交互的问题。
2. JMM内存分为两块，一块是线程私有的工作内存，一块是主内存。
3. 线程的是有工作内存是相互隔离的，但可以使用主内存进行交互。

CAS

1. 全称Compare And Swap(比较再交换)，属于乐观锁，在无锁的情况下保证线程安全。
2. 有三个参数：V（当前内存值），E（预期的值），N（新值）；当V=E时，更新为N；当不等于时，会更新当前的值
3. 用处比较多：AQS框架，juc.atomic.AtomicXXX类，synchronize
4. 在操作共享变量时使用自旋锁效率更高
5. 可能出现ABA问题，即一个线程将A改成B再改成A，但是在另外一个线程看来是没用变化。可以用版本号来解决这个问题

乐观锁和悲观锁

乐观锁：最乐观的估计，认为没用别的线程来共享资源，即使使用了也可以使用其他方法来做线程安全，例如CAS。

悲观锁：最悲观的估计，防止其他线程共享变量，在任何时间都对资源上锁，例如synchronize重量级锁

Volatile

1. 可见性：volatile变量的值在各个线程中是一致的，当一个线程修改了变量，其他线程可以立刻得知。然而普通变量却不行，因为普通变量在线程之间传递需要通过主内存来完成。
2. 禁止指令重排：修饰的共享变量会在读写变量时加入屏障，阻止其他读写操作越过屏障，防止重排序。

AQS

- Abstract Queued Synchronizer，抽象队列同步器，是一种锁机制，例如像ReentrantLock，Semaphore都是基于AQS实现的。
- AQS内部维护了一个FIFO的双端队列，队列汇总存储的是排队的线程
- AQS内部有一个state属性来判断当前是否有线程获取到资源，默认是0
- 如果多个线程来争抢state资源，使用CAS来保证原子性。
- 该实现类有公平锁和非公平锁两种，非公平锁指的是新加入的线程会和队列中的线程来争抢资源。公平锁则将新加入的线程放入队列尾部。

ReentrantLock

1. 是一种可重入锁
2. 底层使用AQS+CAS实现
3. 支持公平锁和非公平锁两种形式

ReentrantLock与synchronize的区别

1. 语法层面：synchronized属于java关键字，不能显示调用，自动加锁释放锁；Lock属于API，可以直接使用，手动加锁释放锁；
2. 功能方面：两者都属于悲观锁，都具备互斥、同步和重入功能；但Lock有更多其他功能，例如：可打断，可超时，公平锁，多条件变量等。
3. 性能方面：在没用竞争时，synchronize做了很多优化例如偏向锁，轻量级锁等，性能较好；但是竞争激烈时，Lock性能更好；参考JVM第13章线程安全的事项

死锁

产生死锁的四个条件：资源有限，请求并保持，不可抢占，循环等待

如何诊断死锁：可以使用jdk自带工具：jps和jstack；也可以使用可视化工具jconsole, visualVM查看。

ConcurrentHashMap

1. 底层结构：1.7使用的是分段的数组+链表；1.8使用的是数组+链表+红黑树，和HashMap结构相同；
2. 加锁方式：1.7采用分段加锁，底层使用的是Reentrantlock，锁住每一个分段；1.8使用的是CAS+synchronized，添加新节点采用CAS，hash冲突后添加会用synchronized锁住头结点后添加到链表或红黑树。相对于segment力度更细，性能更好。

导致并发出现问题原因

1. 原子性：synchronize, lock
2. 可见性：volatile, synchronize, lock
3. 有序性：volatile

线程池

核心参数

线程池七大核心参数：核心线程、最大线程、存活时间、时间单位、阻塞队列、线程工厂（起名字）、四种拒绝策略

执行顺序

1. 提交任务后，参看核心线程数是否已满，如果没有慢，直接创建核心线程执行任务
2. 如果核心线程数满了，则放入阻塞队列中
3. 如果阻塞队列满了，则创建临时线程执行任务
4. 如果临时线程+核心线程数大于最大线程数，则执行拒绝策略

注意：

- 如果执行的核心线程和临时线程完成任务后会检查阻塞队列中是否有等待的任务；如果有，则弹出继续执行。
- 如果一个线程空闲时间超过指定的存活时间，则判断当前线程池中的线程数是否大于核心线程数，如果大于则会被停掉。

线程池中的阻塞队列

1. ArrayBlockingQueue：数组阻塞队列，底层是用数组实现的，所以在创建实例时必须指定队列容量；只有一把锁控制插入和弹出操作
2. LinkedBlockingQueue：底层使用的是链表，可以不指定容量，默认为Integer最大值，也可以指定容量。插入弹出操作由两把锁控制，一把控制插入，另一把控制弹出，效率较高，推荐使用。

如何确定核心线程数

- 如果计算量大，线程切换少的任务，线程可以少一点，避免线程切换，核心线程数可以设置为N(N是CPU核心数)
- 如果计算量小，大部分都是IO操作，线程可以多一些，因为大部分时间CPU都不起作用，例如远程调用，IO等耗时操作，可以设置为2N(N是线程数)

线程池的种类

在Executors中提供了四种快捷创建线程池的静态方法

1. FixThreadPool：核心线程数和最大线程数相同的线程池，没有临时线程，阻塞队列是Integer的Max_value
2. SingleThreadPool：只有固定的一个线程的线程池，没有临时线程，阻塞队列仍然是Integer的Max_Value.
3. CachedThreadPool：没有使用核心线程，都是临时线程，可以无限创建线程，阻塞队列SynchronousQueue不存储任何元素。
4. ScheduledThreadPool：可以设置核心线程数，最大线程数为Max_Value，可以延迟执行任务。

为什么不建议使用Executors创建线程池

根据阿里巴巴代码手册，不能使用Executors创建线程池只要有以下两个问题

4. 【强制】线程池不允许使用 `Executors` 去创建，而是通过 `ThreadPoolExecutor` 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明： `Executors` 返回的线程池对象的弊端如下：

1) `FixedThreadPool` 和 `SingleThreadPool`:

允许的请求队列长度为 `Integer.MAX_VALUE`，可能会堆积大量的请求，从而导致 OOM。

2) `CachedThreadPool` 和 `ScheduledThreadPool`:

允许的创建线程数量为 `Integer.MAX_VALUE`，可能会创建大量的线程，从而导致 OOM。

使用场景

CountDownLatch

CountDownLatch主要有两个方法：

- `cd.countDown()`, cd数量减1
- `cd.await()`, 调用的线程被阻塞，当cd减少到0时，会唤醒当前线程，继续执行。

在新建一个CountDownLatch对象时，需要指定减少的数量。当减少到0时，唤醒所有的await线程。

```
public static void main(String[] args) throws InterruptedException {
    CountDownLatch cd = new CountDownLatch(2);
    new Thread(() -> {
        try {Thread.sleep(2000);} catch (InterruptedException e) {e.printStackTrace();}
        cd.countDown();
    }, "t1").start();
    new Thread(() -> {
        try {Thread.sleep(3000);} catch (InterruptedException e) {e.printStackTrace();}
        cd.countDown();
    }, "t2").start();
    cd.await();
    System.out.println("main thread complete");
}
```

Semaohore

Semaphore['seməfɔ:r] (谐音：四毛佛) 信号量，可以用来控制当前执行线程数。

- 初始化时需要指定信号量的值
- 调用s.acquire()时请求一个信号量，如果当前信号量大于0，信号量-1，如果小于等于0时，将会被阻塞。
- 调用s.release()时释放一个信号量，信号量+1

ThreadLocal

ThreadLocal又叫做线程本地存储，目的是为了实现不同线程之间数据隔离的功能。

底层使用的是threadLocalMap，然而对于每一个线程内部都包含了一个ThreadLocalMap变量，当用户设置值时，会将threadLocal作为key，用户数据作为value放入线程中ThradLocalMap中。该类有三个主要的方法：set(), get(), remove()分别对应赋值，取值和移除。

ThreadLocal有可能会产生内存泄漏问题，因为ThreadLocalMap的key是弱引用，但是value是弱引用，key有可能会被GC线程清除但是value不会，因此在不使用的时候及时清除。

场景1：大量数据导入

例如对于需要在有限的资源内处理千万量级的数据，可以采用数据分页+线程池的方式来解决。

场景2：数据汇总

一个任务需要调用各个系统下并且没用依赖关系，可以使用线程池 + Future来实现

伪代码如下

```
public void static main(){
    Future<Object> future1 = new Thread(()->restTemplate.getObject(...));
    Future<Object> future2 = new Thread(()->restTemplate.getObject(...));
    Future<Object> future3 = new Thread(()->restTemplate.getObject(...));
    future1.get();
    future2.get();
    future3.get();
}
```

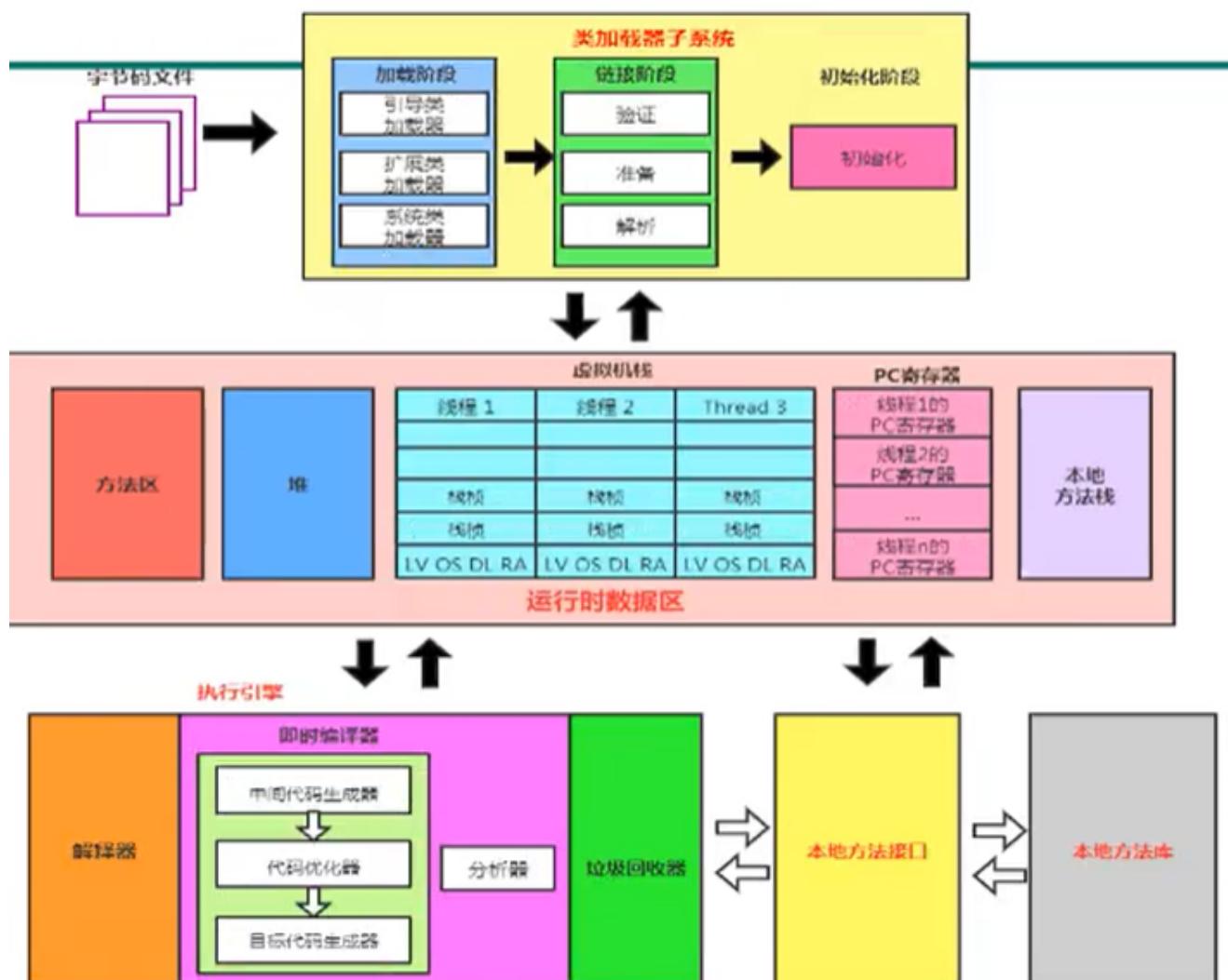
场景3：异步调用

例如数据搜索外，需要保存搜索记录，此时可以另外开辟一个线程来异步保存。

```
void searchAndInsert(){  
    search();  
    insert();  
}  
  
// 异步保存  
@Async  
void insert(){}  
}
```

JVM

类加载子系统



作用

- 类加载器子系统负责从文件系统或者网络中加载Class文件
- ClassLoader只负责class文件的加载，至于它是否可以运行，则由Execution Engine决定。
- 加载的类信息存放于一块称为方法区的内存空间。

加载过程



加载阶段

通过一个类的全限定名获取定义此类的二进制字节流

将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构

在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口

加载方式

- 从本地系统中直接加载
- 通过网络获取，典型场景：Web Applet
- 从zip压缩包中读取，成为日后jar、war格式的基础
- 运行时计算生成，使用最多的是：动态代理技术
- 由其他文件生成，典型场景：JSP应用从专有数据库中提取.class文件，比较少见
- 从加密文件中获取，典型的防Class文件被反编译的保护措施

链接--验证阶段

目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，保证被加载类的正确性，不会危害虚拟机自身安全。

如果出现不合法的字节码文件，那么将会验证不通过

主要包括四种验证，文件**格式**验证，**元数据**验证，**字节码**验证，**符号引用**验证。

链接-准备阶段

为类变量分配内存并且设置该类变量的默认初始值，即零或空。

这里不包含用final修饰的static，因为final在编译的时候就会分配了，准备阶段会显式初始化；

链接-解析阶段

将常量池内的符号引用转换为直接引用的过程。

事实上，解析操作往往伴随着JVM在执行完初始化之后再执行。

初始化阶段

初始化阶段就是执行类构造器法（）的过程。

若该类具有父类，JVM会保证子类的（）执行前，父类的（）已经执行完毕。

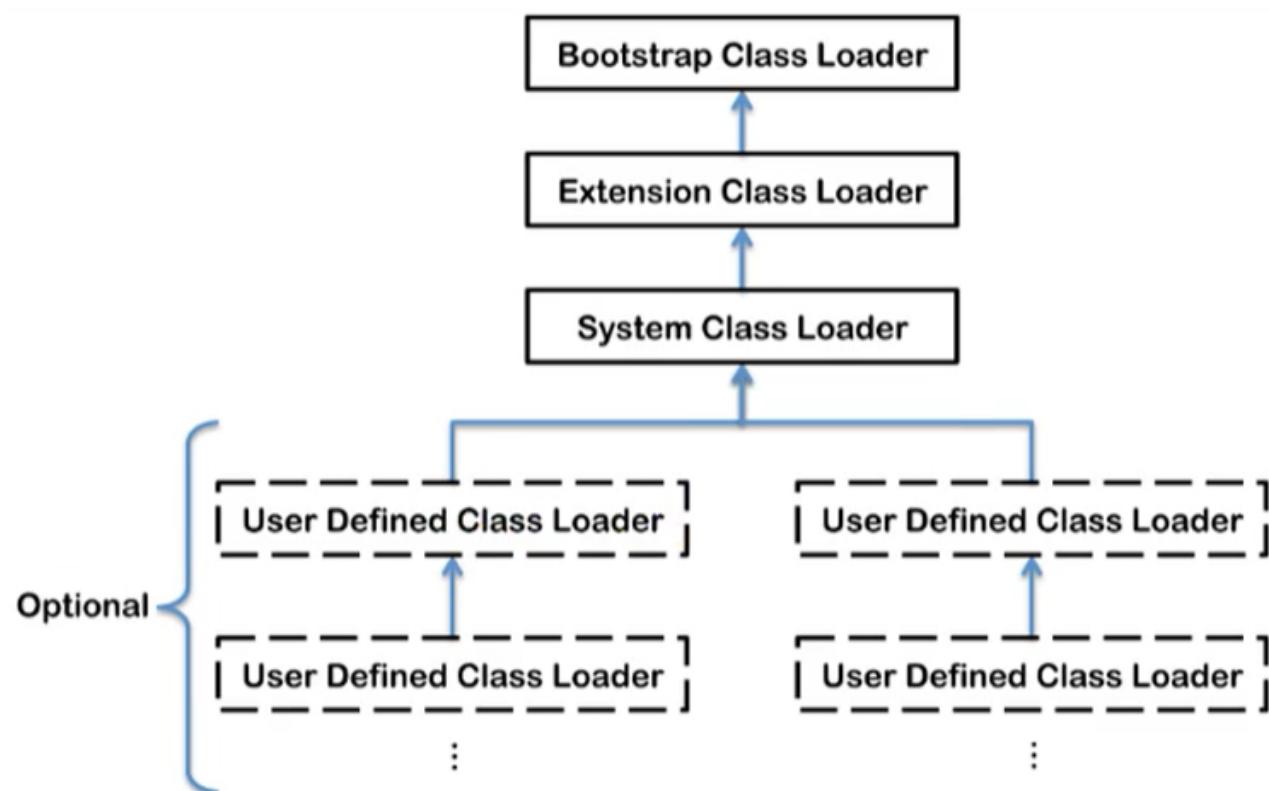
此方法不需定义，是javac编译器自动收集类中的所有类变量的赋值动作和静态代码块中的语句合并而来。

类加载器

分类

JVM支持两种类型的类加载器。分别为引导类加载器（Bootstrap Class Loader）和自定义类加载器（User-Defined Class Loader）。

从概念上来讲，自定义类加载器一般指的是程序中由开发人员自定义的一类类加载器，但是Java虚拟机规范却没有这么定义，而是将所有派生于抽象类ClassLoader的类加载器都划分为自定义类加载器。



启动类加载器

又叫引导类加载器，Bootstrap ClassLoader

- 这个类加载使用C/C++语言实现的，嵌套在JVM内部。
- 它用来加载Java的核心库（/jre/lib/rt.jar、resources.jar或sun.boot.class.path路径下的内容），用于提供JVM自身需要的类
- 并不继承自java.lang.ClassLoader，没有父加载器。
- 加载扩展类和应用程序类加载器，并指定为他们的父类加载器。
- 出于安全考虑，Bootstrap启动类加载器只加载包名为java、javax、sun等开头的类

扩展类加载器

- 这个类加载使用C/C++语言实现的，嵌套在JVM内部。
- 它用来加载Java的核心库（JAVAHOME/jre/lib/rt.jar、resources.jar或sun.boot.class.path路径下的内容），用于提供JVM自身需要的类
- 并不继承自java.lang.ClassLoader，没有父加载器。
- 加载扩展类和应用程序类加载器，并指定为他们的父类加载器。
- 出于安全考虑，Bootstrap启动类加载器只加载包名为java、javax、sun等开头的类

应用类加载器

- javl语言编写，由sun.misc.LaunchersAppClassLoader实现
- 派生于ClassLoader类
- 父类加载器为扩展类加载器
- 它负责加载环境变量classpath或系统属性java.class.path指定路径下的类库
- 该类加载是程序中默认的类加载器，一般来说，Java应用的类都是由它来完成加载
- 通过classLoader#getSystemclassLoader()方法可以获取到该类加载器

自定义类加载器

在Java的日常应用程序开发中，类的加载几乎是由上述3种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，来定制类的加载方式。

为什么要自定义类加载器？

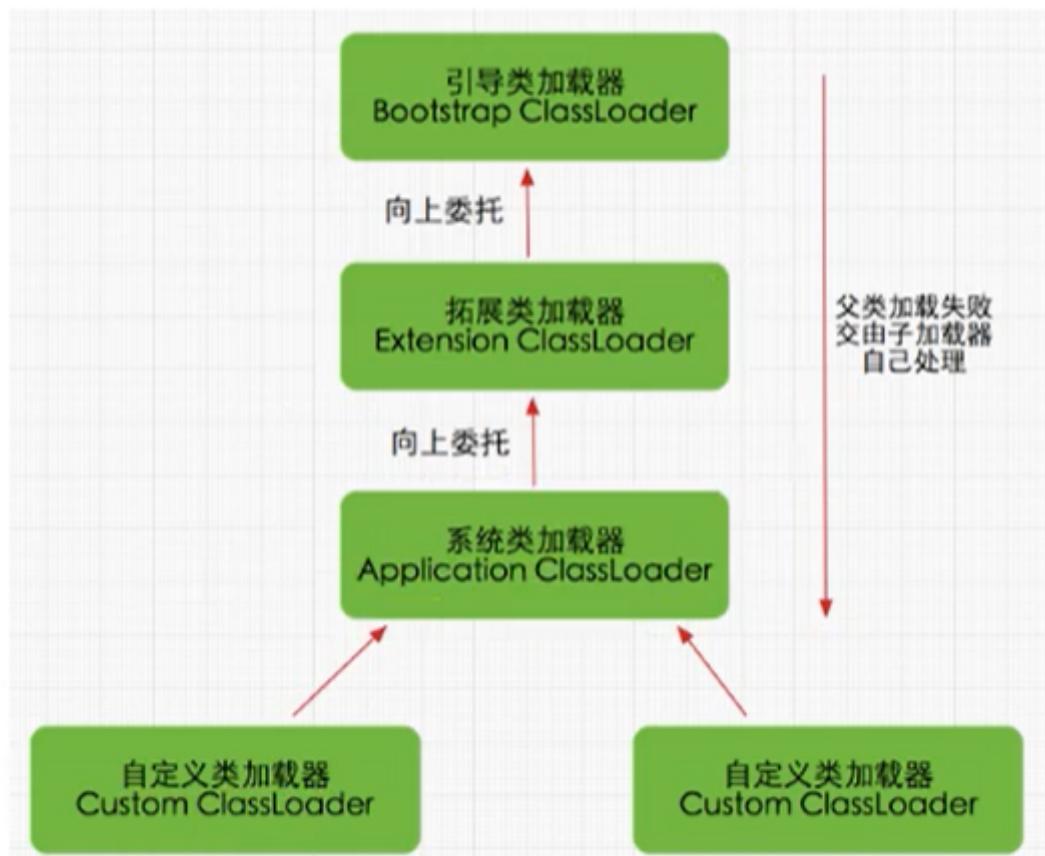
- 隔离加载类：在同一个工程里面，如果引用多个框架的话，有可能会出现某些类的路径一样、类名也相同，这样就会出现类的冲突了，像现在主流的容器类的框架一样，它们都会自定义类的加载器，实现不同的中间件隔离，**避免类的冲突**。
- 修改类加载的方式：我们可以根据实际情况中修改类的加载方式，**具体要用的时候我们再引用**
- 扩展加载源：加载的类除了可以在网络、本地物理磁盘、jar包去加载之外，我们还可以考虑通过**数据库**等等来扩展加载源
- 防止源码泄漏：我们可以对字节码文件进行**加密**，当我们需要运行这个字节码文件时候，我们需要解密来还原成内存中的类，而这个解密的操作，就需要自定义类的加载器来实现

双亲委派机制

Java虚拟机对class文件采用的是**按需加载**的方式，也就是说当需要使用该类时才会将它的class文件加载到内存生成class对象。而且加载某个类的class文件时，Java虚拟机采用的是双亲委派模式，即把请求交由父类处理，它是一种任务委派模式。

工作原理

- 如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行；
- 如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器；
- 如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式。



作用

- 避免类的重复加载
- 保护程序安全，防止核心API被随意篡改

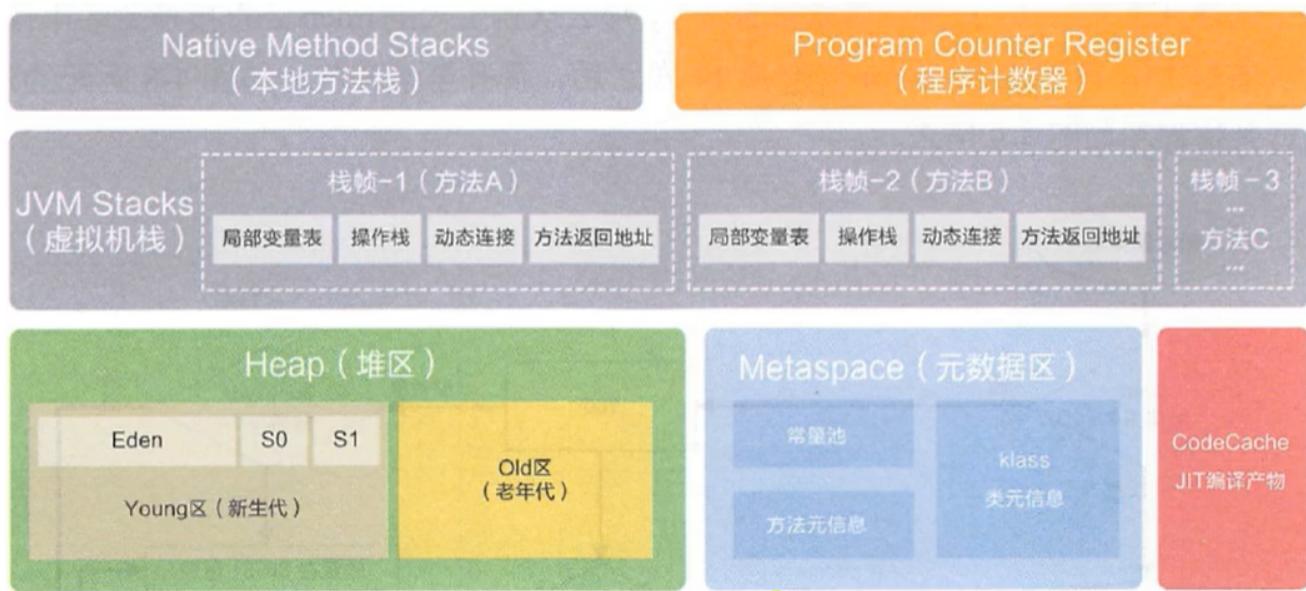
类的主动使用

- 创建类的实例
- 访问某个类或接口的**静态变量**，或者对该静态变量赋值
- 调用类的**静态方法**
- **反射**（比如：Class.forName ("com.atguigu.Test")）
- 初始化一个类的**子类**
- Java虚拟机启动时被标明为**启动类**的类
- JDK7开始提供的**动态语言支持**：

除了以上七种情况，其他使用Java类的方式都被看作是对类的被动使用，都不会导致类的初始化。

运行时数据区

运行时数据区	是否存在Error	是否存在GC
程序计数器	否	否
虚拟机栈	是	否
本地方方法栈	是	否
方法区	是 (OOM)	是
堆	是	是



程序计数器

定义

- 它是一块很小的内存空间，几乎可以忽略不记。也是运行速度最快的存储区域。
- 是线程私有的，每个线程都有它自己的程序计数器，生命周期与线程的生命周期保持一致。
- 它是唯一一个在Java虚拟机规范中没有规定任何OOM情况的区域。

作用

1. 用来存储指向下一条指令的地址，也即将要执行的指令代码。
2. 方便线程在获取到CPU后，知道当前线程执行到哪一步。

虚拟机栈

概述

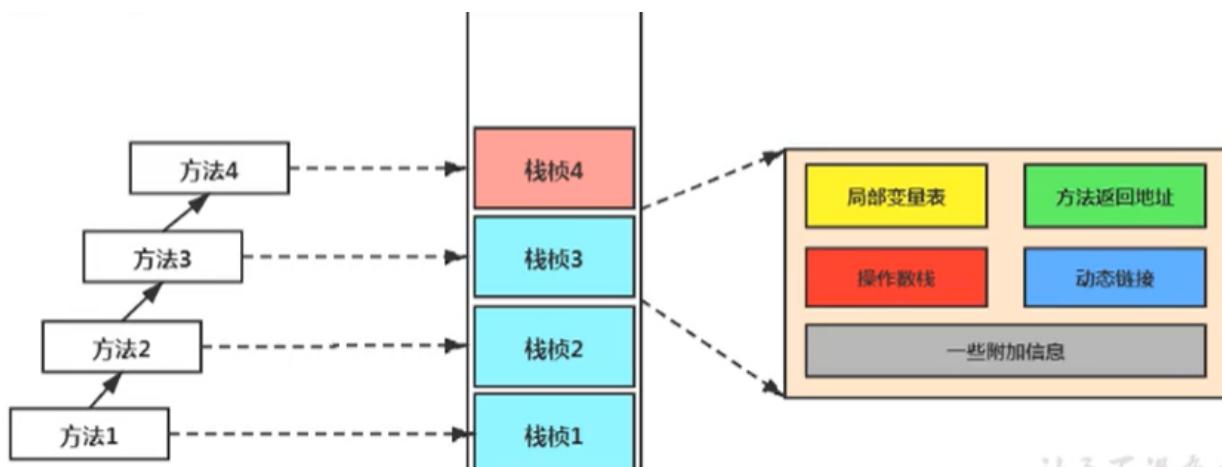
- 每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（Stack Frame），对应着一次次的Java方法调用。
- 生命周期和线程一致，也就是线程结束了，该虚拟机栈也销毁了
- 主管Java程序的运行，它保存方法的局部变量、部分结果，并参与方法的调用和返回。
- 栈是一种快速有效的分配存储方式，访问速度仅次于程序计数器。
- JVM直接对Java栈的操作只有两个：每个方法执行，伴随着进栈（入栈、压栈）；执行结束后的出栈工作。
- 不存在垃圾回收问题
- 使用参数 -Xss选项来设置线程的最大栈空间。栈的大小直接决定了函数调用的最大可达深度

栈帧内存结构

每个线程都有自己的栈，栈中的数据都是以栈帧（Stack Frame）的格式存在。在这个线程上正在执行的每个方法都各自对应一个栈帧（Stack Frame）。栈帧是一个内存区块，是一个数据集，维系着方法执行过程中的各种数据信息。

每个栈帧中存储着：

- 局部变量表（Local Variables）
- 操作数栈（operand Stack）
- 动态链接（DynamicLinking）（或指向运行时常量池的方法引用）
- 方法返回地址（Return Address）（或方法正常退出或者异常退出的定义）
- 一些附加信息



局部变量表

定义为一个数字数组，主要用于存储

- 方法参数
- 定义在方法体内的**局部变量**这些数据类型包括各类**基本数据类型、对象引用** (reference)，以及**returnAddress**类型。

由于局部变量表是建立在线程的栈上，是线程的私有数据，因此不存在数据安全问题。

局部变量表，最基本的存储单元是Slot (变量槽)，一个变量槽是4个字节。int/float/reference占一个，double/long占两个

栈帧中的局部变量表中的槽位是可以重用的，如果一个局部变量过了其作用域，那么在其作用域之后申明的新的局部变就很有可能会复用过期局部变量的槽位，从而达到节省资源的目的。

操作数栈

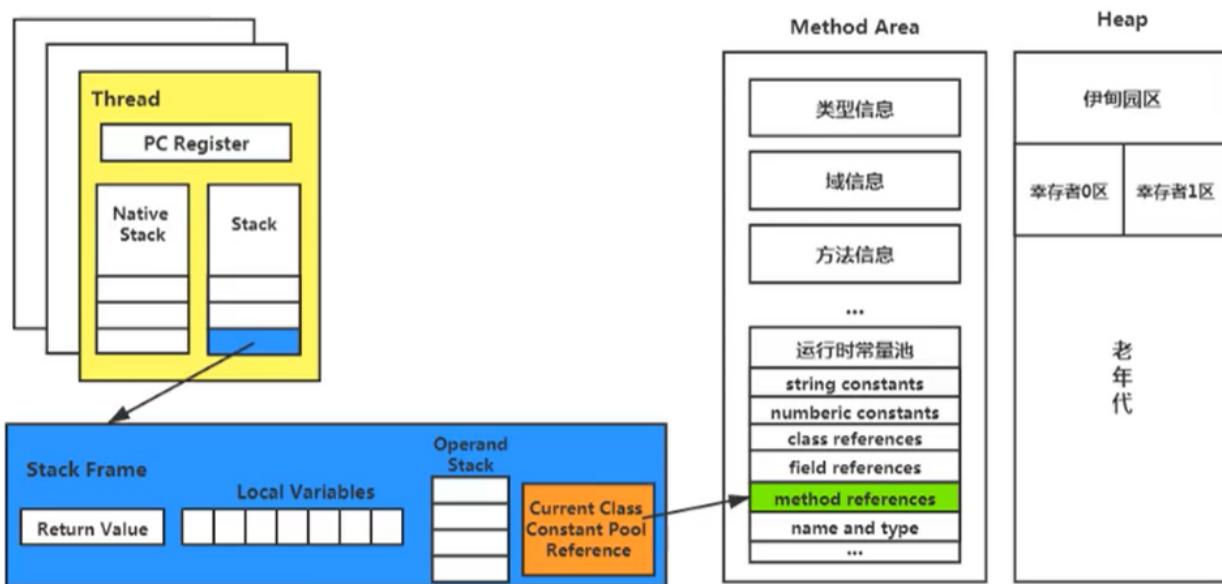
操作数栈，在方法执行过程中，根据字节码指令，往栈中写入数据或提取数据，即入栈和出栈

比如：执行复制、交换、求和等操作

动态链接

每一个栈帧内部都包含一个指向**运行时常量池**中该栈帧所属方法的引用包含这个引用的目的就是为了支持当前方法的代码能够实现动态链接 (Dynamic Linking)。

在Java源文件被编译到字节码文件中时，所有的变量和方法引用都作为符号引用保存在class文件的常量池里。



方法返回地址

存放调用该方法的pc寄存器的值。一个方法的结束，有两种方式：

- 正常执行完成
- 出现未处理的异常，非正常退出

无论通过哪种方式退出，在方法退出后都返回到该方法被调用的位置。方法正常退出时，调用者的pc计数器的值作为返回地址，即调用该方法的指令的下一条指令的地址。而通过异常退出的，返回地址是要通过异常表来确定，栈帧中一般不会保存这部分信息。

堆

概念

- 堆针对一个JVM进程来说是唯一的，也就是一个进程只有一个JVM，但是进程包含多个线程，他们是共享同一堆空间的。
- 所有的对象实例以及数组都应当在运行时分配在堆上。
- 堆内存的大小是可以调节的。
- 堆，是GC (Garbage Collection, 垃圾收集器) 执行垃圾回收的重点区域。

堆内存大小

- “-Xms”用于表示堆区的起始内存，等价于-xx:InitialHeapSize
- “-Xmx”则用于表示堆区的最大内存，等价于-XX:MaxHeapSize

一旦堆区中的内存大小超过“-xmx”所指定的最大内存时，将会抛出`OutOfMemoryError`异常。

通常会将-Xms和-Xmx两个参数配置相同的值，其目的是为了能够在Java垃圾回收机制清理完堆区后不需要重新分隔计算堆区的大小，从而提高性能。

默认情况下

- 初始内存大小：物理电脑内存大小/64
- 最大内存大小：物理电脑内存大小/4

堆内存细分

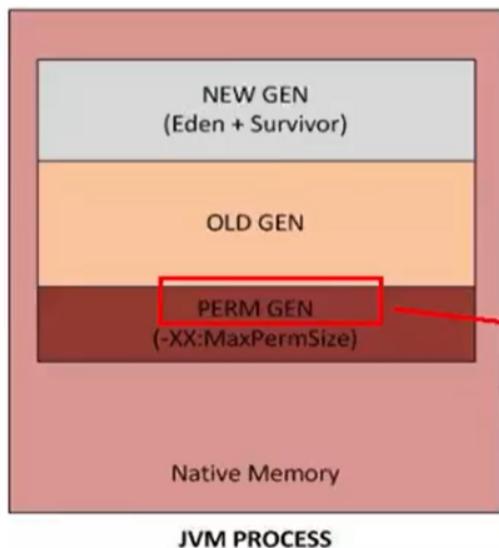
内存逻辑

分为三部分：新生区+养老区+永久区

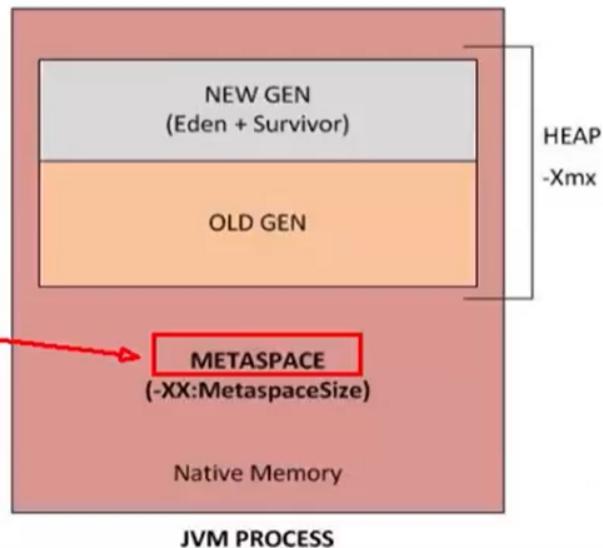
- Young Generation Space 新生区 Young/New 又被划分为Eden区和Survivor区
- Tenure generation space 养老区 Old/Tenure
- Permanent Space永久区 Perm 1.8之后改为Meta Space 元空间 Meta

堆空间内部结构，JDK1.8之前从永久代 替换成元空间。

Pre Java 8



Java 8



Java对象

- 一类是生命周期较短的瞬时对象，这类对象的创建和消亡都非常迅速
- 生命周期短的，及时回收即可
- 另外一类对象的生命周期却非常长，在某些极端的情况下还能够与JVM的生命周期保持一致

Java堆区

其中年轻代又可以划分为Eden空间、 Survivor0空间和Survivor1空间（有时也叫做from区、 to区）

JVM参数

- Eden: From: to -> 8:1:1。默认-xx:SurvivorRatio=8
- 新生代: 老年代 -> 1 : 2。默认-XX:NewRatio=2
- 默认将S区转为老年代的次数是15次。默认-Xx:MaxTenuringThreshold= 15

晋升

在Eden区满了的时候，才会触发MinorGC，而幸存者区满了后，不会触发MinorGC操作

如果Survivor区满了但还没达到转移阈值，将会直接晋升老年代。

总结

- 针对幸存者s0, s1区的总结：复制之后有交换，谁空谁是to
- 关于垃圾回收：频繁在新生区收集，很少在老年代收集，几乎不再永久代和元空间进行收集
- 新生代采用复制算法的目的：是为了减少内碎片

GC

- Minor GC：新生代的GC
- Major GC：老年代的GC
- Full GC：收集整个Java堆和方法区的垃圾收集

Minor GC

当年轻代空间不足时，就会触发MinorGC，这里的年轻代满指的是Eden代满，Survivor满不会引发GC。（每次Minor GC会清理年轻代的内存。）

Major GC

指发生在老年代的GC

出现了MajorGc，经常会伴随至少一次的Minor GC（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行MajorGC的策略选择过程）也就是在老年代空间不足时，会先尝试触发MinorGc。如果之后空间还不足，则触发Major GC

Major GC的速度一般会比MinorGc慢10倍以上，STW的时间更长，如果Major GC后，内存还不足，就报OOM了

Full GC

触发FullGC执行的情况有如下五种：

- 调用System.gc () 时，系统建议执行FullGC，但是不必然执行
- **老年代空间不足**
- **方法区空间不足**
- 通过Minor GC后进入老年代的平均大小大于老年代的可用内存
- 由Eden区、survivor spacee (From Space) 区向survivor space (To Space) 区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

说明：Full GC 是开发或调优中尽量要避免的。这样暂时时间会短一些

触发OOM的时候，一定是进行了一次Full GC，因为只有在老年代空间不足时候，才会爆出OOM异常

内存分配策略

1. 优先分配到Eden
 2. 大对象直接分配到老年代
 3. 长期存活的对象分配到老年代
 4. 动态对象年龄判断
5. **空间分配担保**：也就是经过Minor GC后，所有的对象都存活，因为Survivor比较小，所以就需要将Survivor无法容纳的对象，存放到老年代中。

TLAB

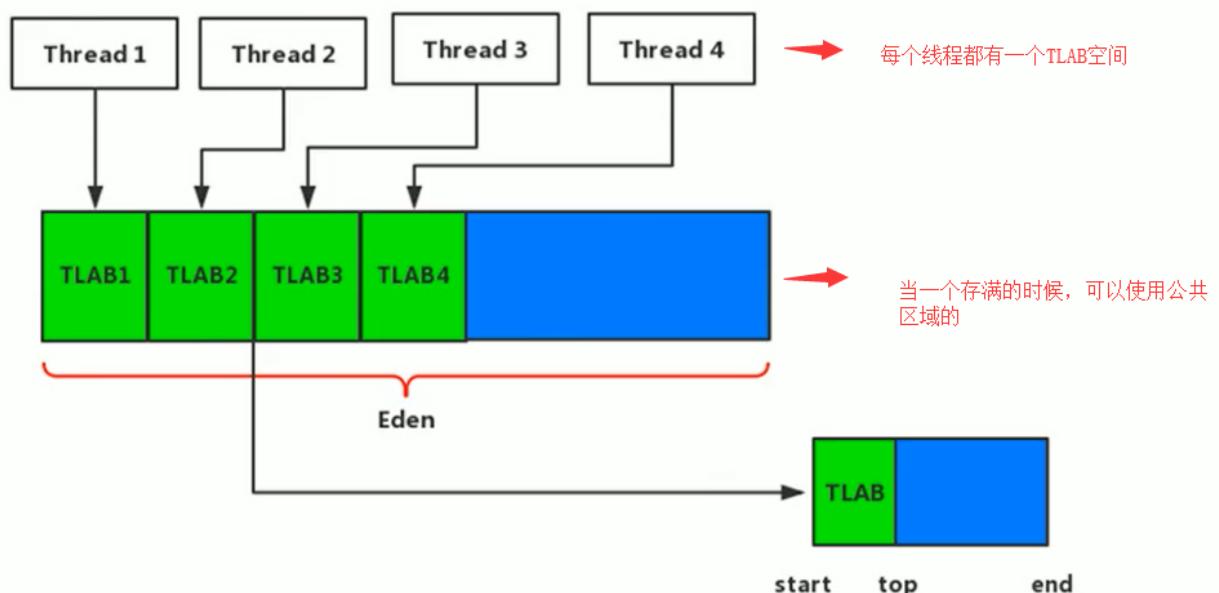
堆空间都是共享的么？不一定，因为还有TLAB这个概念，在堆中划分出一块区域，为每个线程所独占

TLAB：Thread Local Allocation Buffer，中文名是**线程私有内存分配区**。也就是为每个线程单独分配了一个缓冲区

作用

堆区是线程共享区域，任何线程都可以访问到堆区中的共享数据。由于对象实例的创建在JVM中非常频繁，因此在并发环境下从堆区中划分内存空间是线程不安全的。为避免多个线程操作同一地址，需要使用加锁等机制，进而影响分配速度。

从内存模型而不是垃圾收集的角度，对Eden区域继续进行划分，JVM为每个线程分配了一个私有缓存区域，它包含在Eden空间内。多线程同时分配内存时，使用TLAB可以避免一系列的非线程安全问题，同时还能够提升内存分配的吞吐量，因此我们可以将这种内存分配方式称之为快速分配策略。



设置

TLAB默认情况下处于启用状态。在程序中，开发人员可以通过选项“-Xx:UseTLAB”设置是否开启TLAB空间。

默认情况下，TLAB空间的内存非常小，仅占有整个Eden空间的1%，当然我们可以通过选项“-Xx:TLABWasteTargetPercent”设置TLAB空间所占用Eden空间的百分比大小。

参数设置

- -Xms：初始堆空间内存（默认为物理内存的1/64）
- -Xmx：最大堆空间内存（默认为物理内存的1/4）
- -Xmn：设置新生代的大小。（初始值及最大值）
- -XX:NewRatio：配置新生代与老年代在堆结构的占比
- -XX:SurvivorRatio：设置新生代中Eden和S0/S1空间的比例
- -XX:MaxTenuringThreshold：设置新生代垃圾的最大年龄
- -XX: +PrintGCDetails：输出详细的GC处理日志

- -XX:HandlePromotionFailure：是否设置空间分配担保

逃逸分析

随着JIT编译期的发展与逃逸分析技术逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化，所有的对象都分配到堆上也渐渐变得不那么“绝对”了。

在Java虚拟机中，对象是在Java堆中分配内存的，这是一个普遍的常识。但是，有一种特殊情况，那就是如果经过逃逸分析（Escape Analysis）后发现，一个对象并没有逃逸出方法的话，那么就可能被优化成栈上分配。这样就无需在堆上分配内存，也无须进行垃圾回收了。这也是最常见的堆外存储技术。

判断

- 当一个对象在方法中被定义后，对象只在方法内部使用，则认为没有发生逃逸。
- 当一个对象在方法中被定义后，它被外部方法所引用，则认为发生逃逸。例如作为调用参数传递到其他地方中。

参数

在JDK 1.7 版本之后，HotSpot中默认就已经开启了逃逸分析

如果使用的是较早的版本，开发人员则可以通过：

- 选项“-xx: +DoEscapeAnalysis”显式开启逃逸分析
- 通过选项“-xx: +PrintEscapeAnalysis”查看逃逸分析的筛选结果

结论

开发中能使用局部变量的，就不要使用在方法外定义。

使用逃逸分析，编译器可以对代码做如下优化：

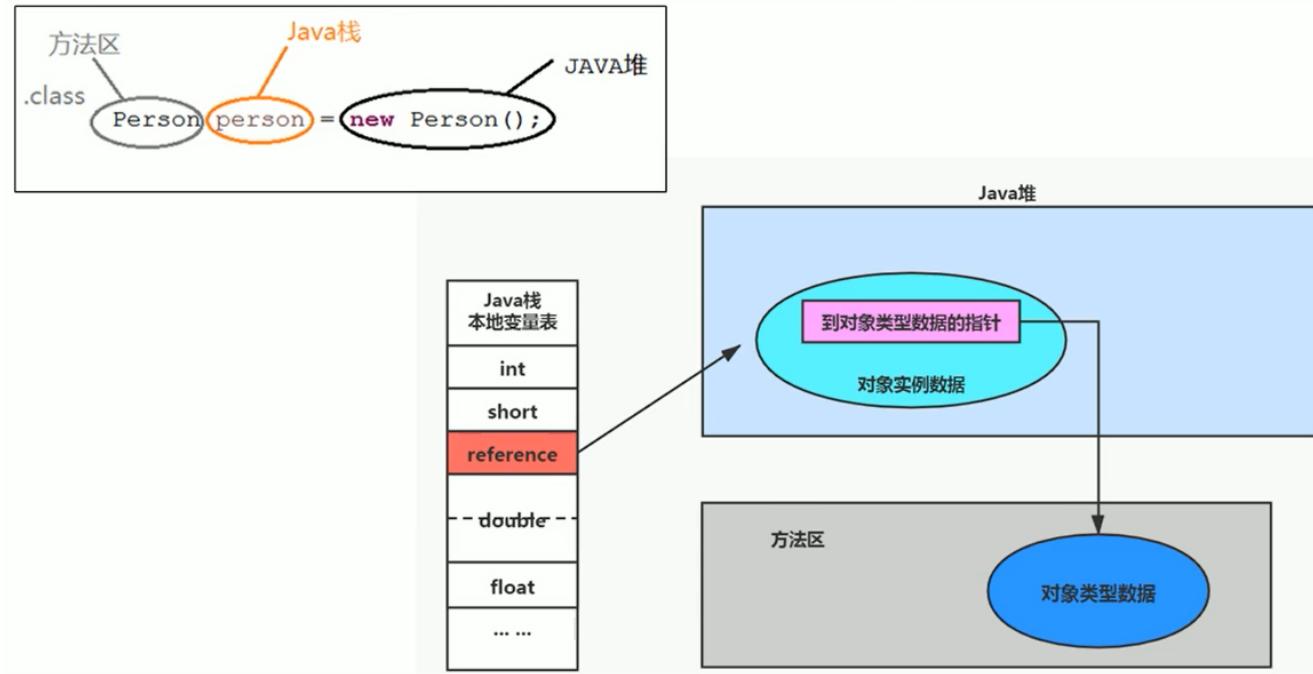
- **栈上分配**：将堆分配转化为栈分配。如果一个对象在子程序中被分配，要使指向该对象的指针永远不会发生逃逸，对象可能是栈上分配的候选，而不是堆上分配
- **同步省略**：如果一个对象被发现只有一个线程被访问到，那么对于这个对象的操作可以不考虑同步。典型的是MysqlSession在Spring中的应用
- **标量替换**：Java中的原始数据类型就是标量。Java中的对象就是聚合量，因为他可以分解成其他聚合量和标量。有的对象可能不需要作为一个连续的内存结构存在也可以被访问到，那么对象的部分（或全部）可以不存储在内存，而是存储在CPU寄存器中。

不足

其根本原因就是无法保证逃逸分析的性能消耗一定能高于他的消耗。虽然经过逃逸分析可以做标量替换、栈上分配、和锁消除。但是逃逸分析自身也是需要进行一系列复杂的分析的，这其实也是一个相对耗时的过程。一个极端的例子，就是经过逃逸分析之后，发现没有一个对象是不逃逸的。那这个逃逸分析的过程就白白浪费掉了。

方法区

栈、堆、方法区的交互关系



概念

方法区看作是一块独立于Java堆的内存空间。

在jdk7及以前，习惯上把方法区，称为**永久代**。jdk8开始，使用**元空间**取代了永久代。JDK 1.8后，元空间存放在堆外内存中，元空间不在虚拟机设置的内存中，而是使用本地内存

所以，永久代和元空间都是方法区的具体实现

方法区主要存放的是 Class，而堆中主要存放的是 实例化的对象

- 方法区（Method Area）与Java堆一样，是各个线程共享的内存区域。
- 方法区在JVM启动的时候被创建，并且它的实际的物理内存空间中和Java堆区一样都可以是不连续的。
- 方法区的大小，跟堆空间一样，可以选择固定大小或者可扩展。
- 方法区的大小决定了系统可以保存多少个类，如果系统定义了太多的类，导致方法区溢出，虚拟机同样会抛出内存溢出错误：ava.lang.OutOfMemoryError: PermGen space 或者java.lang.OutOfMemoryError:Metaspace
 - 加载大量的第三方的jar包
 - Tomcat部署的工程过多（30~50个）
 - 大量动态的生成反射类
- 关闭JVM就会释放这个区域的内存。

从JDK1.7开始，字符串常量池，静态变量从方法区中移除，保存到堆中。

字符串常量池为什么要调整位置？因为永久代的回收效率很低，在full gc的时候才会触发。

方法区的垃圾收集主要回收两部分内容：常量池中废弃的常量和不在使用的类型

空间大小

jdk7及以前

- 通过-xx:Permsize来设置永久代初始分配空间。默认值是20.75M
- -XX:MaxPermsize来设定永久代最大可分配空间。32位机器默认是64M，64位机器模式是82M

JDK8以后

元数据区大小可以使用参数

-XX:MetaspaceSize：其默认值为21MB

-XX:MaxMetaspaceSize：默认-1，即没有限制

默认值依赖于平台。与永久代不同，如果不指定大小，默认情况下，虚拟机会耗尽所有的可用系统内存。如果元数据区发生溢出，虚拟机一样会抛出异常OutOfMemoryError:Metaspace

内部结构

《深入理解Java虚拟机》书中对方法区（Method Area）存储内容描述如下：它用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等。

- 类型信息：每个加载的类型（类class、接口interface、枚举enum、注解annotation）
- 域信息：域名称、域类型、域修饰符（public, private, protected, static, final, volatile, transient的某个子集）
- 方法信息：方法名称，返回类型，参数数量和类型，修饰符，操作数栈，局部变量表，异常表
- 静态变量
- 常量

运行时常量池vs常量池

运行时常量池就是classFile中的常量池在JVM中的状态。

GC

方法区的垃圾收集主要回收两部分内容：常量池中废弃的常量和不再使用的类型。

判定一个常量是否“废弃”还是相对简单，而要判定一个类型是否属于“不再被使用的类”的条件就比较苛刻了。需要同时满足下面三个条件：

- 该类所有的实例都已经被回收，也就是Java堆中不存在该类及其任何派生子类的实例。
加载该类的类加载器已经被回收，这个条件除非是经过精心设计的可替换类加载器的场景，如osGi、JSP的重加载等，否则通常是很难达成的。

- 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。Java虚拟机被允许对满足上述三个条件的无用类进行回收，这里说的仅仅是“被允许”，而并不是和对象一样，没有引用了就必然会回收。关于是否要对类型进行回收，HotSpot虚拟机提供了-Xnoclassgc参数进行控制，还可以使用-verbose:class以及-XX:+TraceClass-Loading、-XX:+TraceClassUnLoading查看类加载和卸载信息
- 在大量使用反射、动态代理、CGLib等字节码框架，动态生成JSPI以及oSGi这类频繁自定义类加载器的场景中，通常都需要Java虚拟机具备类型卸载的能力，以保证不会对方法区造成过大的内存压力。

对象实例化

对象创建方式

- new：最常见的方式、单例类中调用getInstance的静态类方法，XXXFactory的静态方法
- Class的newInstance方法：在JDK9里面被标记为过时的方法，因为只能调用空参构造器
- Constructor的newInstance(XXX)：反射的方式，可以调用空参的，或者带参的构造器
- 使用clone()：不调用任何的构造器，要求当前的类需要实现Cloneable接口中的clone接口
- 使用序列化：序列化一般用于Socket的网络传输
- 第三方库 Objenesis

创建对象的步骤

判断对象对应的类是否加载、链接、初始化

虚拟机遇到一条new指令，首先去检查这个指令的参数能否在Metaspace的常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载，解析和初始化。如果没有，那么在双亲委派模式下加载。

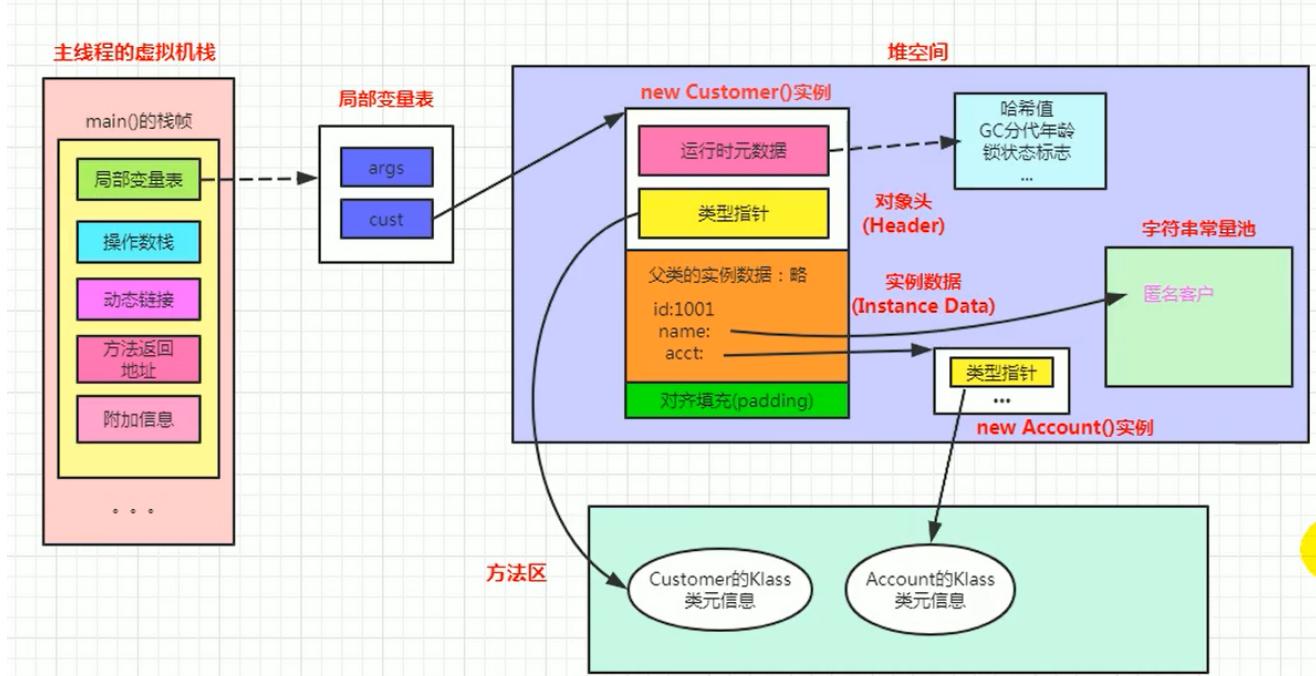
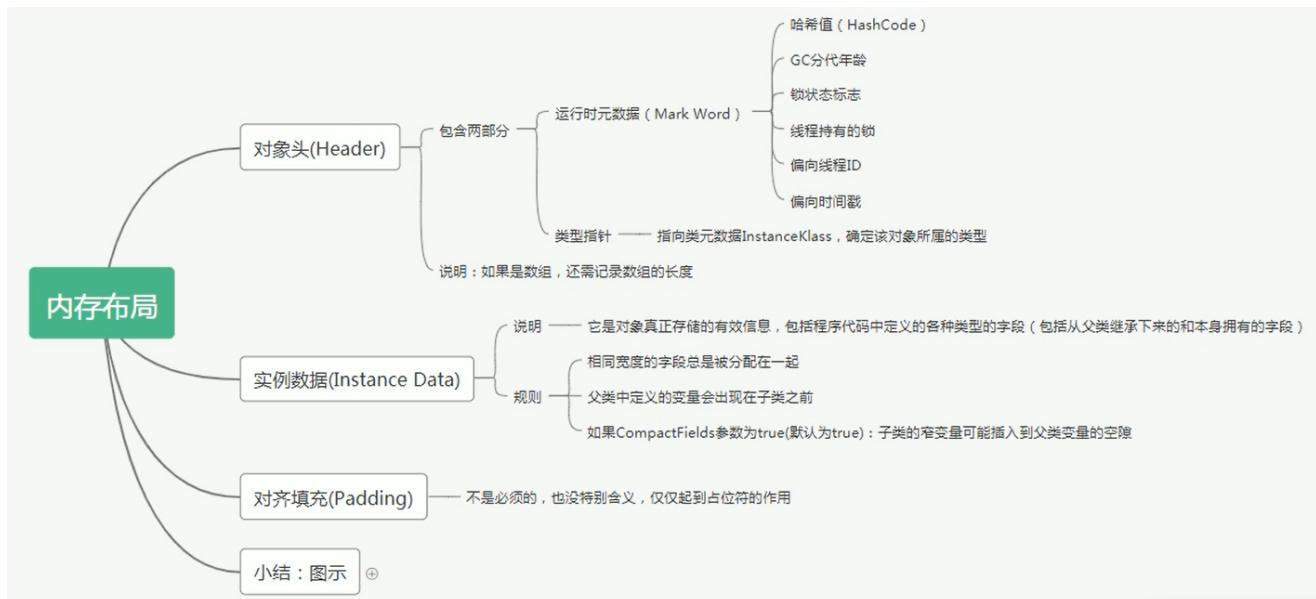
为对象分配内存

- 如果内存规整：指针碰撞。一般使用带Compact（整理）过程的收集器时，使用指针碰撞。
- 如果内存不规整
 - 虚拟机需要维护一个列表
 - 空闲列表分配

处理并发问题

- 采用CAS配上失败重试保证更新的原子性
- 每个线程预先分配TLAB - 通过设置-XX:+UseTLAB参数来设置（区域加锁机制）
 - 在Eden区给每个线程分配一块区域

对象内存布局



执行引擎

执行引擎 (Execution Engine) 的任务就是将字节码指令解释/编译为对应平台上的本地机器指令才可以。

简单来说，JVM中的执行引擎充当了将高级语言翻译为机器语言的译者。

JVM的主要任务是负责装载字节码到其内部，但字节码并不能够直接运行在操作系统之上，因为字节码指令并非等价于本地机器指令，它内部包含的仅仅只是一些能够被JVM所识别的字节码指令、符号表，以及其他辅助信息。

GC算法

识别垃圾算法

引用计数法

引用计数算法（Reference Counting）比较简单，对每个对象保存一个整型的引用计数器属性。用于记录对象被引用的情况。

对于一个对象A，只要有任何一个对象引用了A，则A的引用计数器就加1；当引用失效时，引用计数器就减1。只要对象A的引用计数器的值为0，即表示对象A不可能再被使用，可进行回收。

优点：实现简单，判定效率高，回收没有延迟性。

缺点：它需要单独的字段存储计数器，这样的做法增加了存储空间的开销。一个严重的问题，即无法处理循环引用的情况

可达性分析法

基本思路：

- 可达性分析算法是以根对象集合（GC Roots）为起始点，按照从上至下的方式搜索被根对象集合所连接的目标对象是否可达。
- 使用可达性分析算法后，内存中的存活对象都会被根对象集合直接或间接连接着，搜索所走过的路径称为引用链（Reference Chain）
- 如果目标对象没有任何引用链相连，则是不可达的，就意味着该对象已经死亡，可以标记为垃圾对象。
- 在可达性分析算法中，只有能够被根对象集合直接或者间接连接的对象才是存活对象。

GC Roots

- 虚拟机栈中引用的对象
- 本地方法栈内引用的对象
- 方法区中常量引用的对象
- 所有被同步锁synchronized持有的对象
- Java虚拟机内部的引用。一些常驻的异常对象（如：NullPointerException、outofMemoryError），系统类加载器。

总结

- 除了堆空间外的一些结构，比如虚拟机栈、本地方法栈、方法区、字符串常量池等地方对堆空间进行引用的，都可以作为GC Roots进行可达性分析
- 如果一个指针保存了堆内存里面的对象，但是自己又不存放在堆内存里面，那它就是一个Root。

回收垃圾算法

- 标记—清除算法（Mark-Sweep）
- 复制算法（copying）
- 标记-压缩算法（Mark-Compact）

	标记清除	标记整理	复制算法
速率	中等	最慢	最快
空间开销	少 (但会堆积碎片)	少 (不堆积碎片)	通常需要活对象的2倍空间 (不堆积碎片)
移动对象	否	是	是

标记-清除

- **标记**: Collector从引用根节点开始遍历，**标记所有被引用的对象**。一般是在对象的Header中记录为可达对象。
- **标记的是引用的对象，不是垃圾！！**
- **清除**: Collector对堆内存从头到尾进行线性的遍历，如果发现某个对象在其Header中没有标记为可达对象，则将其回收

优点

- 不用移动指针

缺点

- **产生内碎片**，需要维护一个空闲列表

复制算法

将活着的内存空间分为两块，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后清除正在使用的内存块中的所有对象，交换两个内存的角色，最后完成垃圾回收

优点

- 实现简单，运行高效
- 不会出现“碎片”问题

缺点

- 需要两倍空间
- 需要移动指针

标记-整理

第一阶段和标记清除算法一样，从根节点开始标记所有被引用对象

第二阶段将所有的存活对象压缩到内存的一端，按顺序排放。之后，清理边界外所有的空间。

优点

- 不会产生碎片
- 不需要额外空间

缺点

- 效率低
- 需要移动指针

分代收集算法

年轻代：复制算法

老年代：一般是由标记-清除或者是标记-清除与标记-整理的混合实现。

分区算法

一般来说，在相同条件下，堆空间越大，一次Gc时所需要的时间就越长，有关GC产生的停顿也越长。为了更好地控制GC产生的停顿时间，将一块大的内存区域分割成多个小块，根据目标的停顿时间，每次合理地回收若干个小区间，而不是整个堆空间，从而减少一次GC所产生的停顿。

分代算法将按照对象的生命周期长短划分成两个部分，分区算法将整个堆空间划分成连续的不同小区间。每一个小区间都独立使用，独立回收。这种算法的好处是可以控制一次回收多少个小区间。

内存管理

内存溢出

javadoc中对outofMemoryError的解释是，没有空闲内存，并且垃圾收集器也无法提供更多内存。原因有二：

1. Java虚拟机的堆内存设置不够。
2. 代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在被引用）

内存泄漏

严格来说，只有对象不会再被程序用到了，但是GC又不能回收他们的情况，才叫内存泄漏。

举例

- 单例模式

单例的生命周期和应用程序是一样长的，所以单例程序中，如果持有对外部对象的引用的话，那么这个外部对象是不能被回收的，则会导致内存泄漏的产生。

- 一些提供close的资源未关闭导致内存泄漏

数据库连接（dataSource.getConnection()），网络连接（socket）和io连接必须手动close，否则是不能被回收的。

Stop The World

stop-the-world，简称STw，指的是GC事件发生过程中，会产生应用程序的停顿。停顿产生时整个应用程序线程都会被暂停，没有任何响应，有点像卡死的感觉，这个停顿称为STW。

可达性分析算法中枚举根节点（GC Roots）会导致所有Java执行线程停顿。因为分析工作必须在一个能确保一致性的快照中进行。如果出现分析过程中对象引用关系还在不断变化，则分析结果的准确性无法保证。

安全点与安全区域

安全点

程序执行时并非在所有地方都能停顿下来开始GC，只有在特定的位置才能停顿下来开始GC，这些位置称为“安全点（Safepoint）”。

Safe Point的选择很重要，如果太少可能导致GC等待的时间太长，如果太频繁可能导致运行时的性能问题

如何在cc发生时，检查所有线程都跑到最近的安全点停顿下来呢？

- **抢先式中断：**（目前没有虚拟机采用了）首先中断所有线程。如果还有线程不在安全点，就恢复线程，让线程跑到安全点。
- **主动式中断：**设置一个中断标志，各个线程运行到Safe Point的时候主动轮询这个标志，如果中断标志为真，则将自己进行中断挂起。（有轮询的机制）

安全区域

安全区域是指在一段代码片段中，对象的引用关系不会发生变化，在这个区域中的任何位置开始Gc都是安全的。我们也可以把Safe Region看做是被扩展了的Safepoint。

再谈引用

- Reference子类中只有终结器引用是包内可见的，其他3种引用类型均为public，可以在应用程序中直接使用
 - **强引用（StrongReference）：**最传统的“引用”的定义，是指在程序代码之中普遍存在的引用赋值，即类似“object obj=new Object ()”这种引用关系。无论任何情况下，只要强引用关系还存在，垃圾收集器就永远不会回收掉被引用的对象。
 - **软引用（SoftReference）：**在系统将要发生内存溢出之前，将会把这些对象列入回收范围之中进行第二次回收。如果这次回收后还没有足够的内存，才会抛出内存流出异常。
 - **弱引用（WeakReference）：**被弱引用关联的对象只能生存到下一次垃圾收集之前。当垃圾收集器工作时，无论内存空间是否足够，都会回收掉被弱引用关联的对象。
 - **虚引用（PhantomReference）：**一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用获得一个对象的实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

垃圾回收器

分类

线程数

按线程数分（垃圾回收线程数），可以分为串行垃圾回收器和并行垃圾回收器。

- 在诸如单CPU处理器或者较小的应用内存等硬件平台不是特别优越的场合，串行回收器的性能表现可以超过并行回收器和并发回收器。所以，串行回收默认被应用在客户端的Client模式下的JVM中
- 在并发能力比较强的CPU上，并行回收器产生的停顿时间要短于串行回收器。

工作模式

按照工作模式分，可以分为并发式垃圾回收器和独占式垃圾回收器。

- 并发式垃圾回收器与应用程序线程交替工作，以尽可能减少应用程序的停顿时间。
- 独占式垃圾回收器（Stop the world）一旦运行，就停止应用程序中的所有用户线程，直到垃圾回收过程完全结束。

碎片处理方式

按碎片处理方式分，可分为压缩式垃圾回收器和非压缩式垃圾回收器。

- 压缩式垃圾回收器会在回收完成后，对存活对象进行压缩整理，消除回收后的碎片。
- 非压缩式的垃圾回收器不进行这步操作。

性能指标

吞吐量、暂停时间、内存占用 这三者共同构成一个“不可能三角”。一款优秀的收集器通常最多同时满足其中的两项。

- **吞吐量**：运行用户代码的时间占总运行时间的比例（总运行时间 = 程序的运行时间 + 内存回收的时间）
- **垃圾收集开销**：吞吐量的补数，垃圾收集所用时间与总运行时间的比例。
- **暂停时间**：执行垃圾收集时，程序的工作线程被暂停的时间。
- **收集频率**：相对于应用程序的执行，收集操作发生的频率。
- **内存占用**：Java堆区所占的内存大小。
- **快速**：一个对象从诞生到被回收所经历的时间。

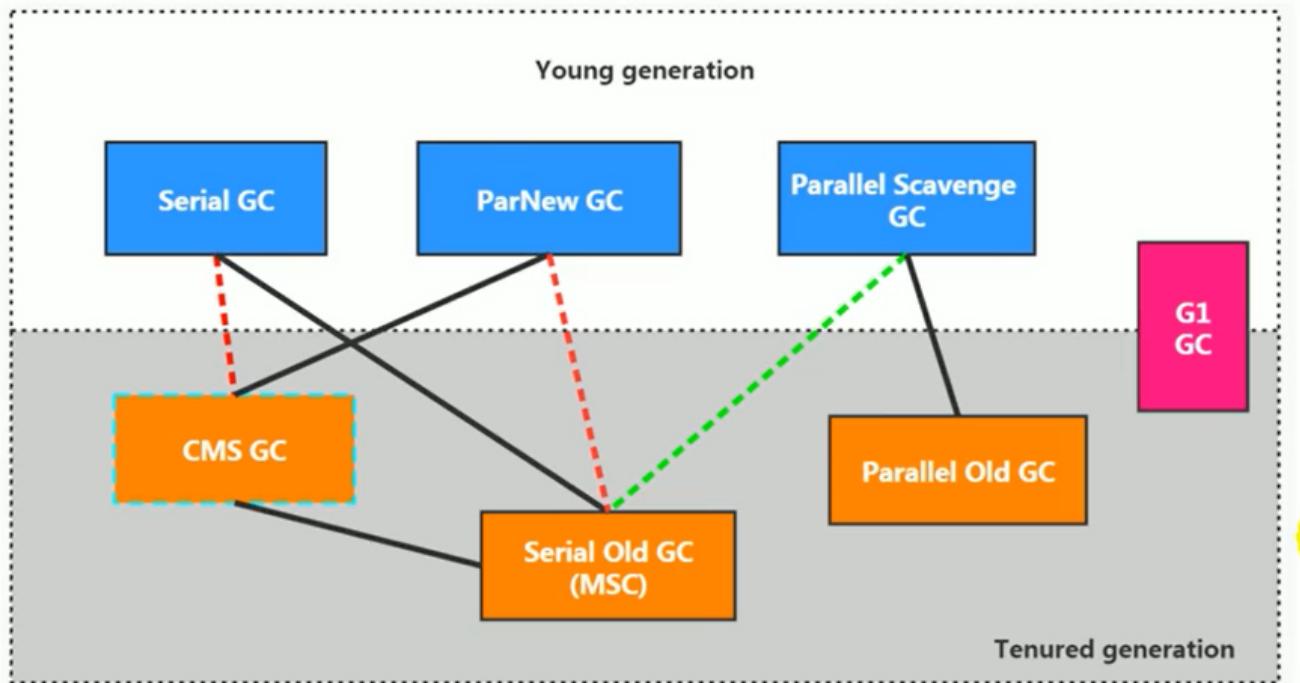
高吞吐量较好因为这会让应用程序的最终用户感觉只有应用程序线程在做“生产性”工作。

低暂停时间（低延迟）较好因为从最终用户的角度来看不管是GC还是其他原因导致一个应用被挂起始终是不好的。

不幸的是“高吞吐量”和“低暂停时间”是一对相互竞争的目标（矛盾）。

现在标准：**在最大吞吐量优先的情况下，降低停顿时间**

7款经典收集器



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

老年代收集器：Serial old、Parallel old、CMS；

整堆收集器：G1；

Serial回收器：串行回收

Serial收集器是最基本、历史最悠久的垃圾收集器了。JDK1.3之前回收新生代唯一的选择。

Serial收集器作为HotSpot中client模式下的默认新生代垃圾收集器。

Serial收集器采用复制算法、串行回收和"stop-the-World"机制的方式执行内存回收。

Serial old收集器同样也采用了串行回收和"stop the World"机制，只不过内存回收算法使用的是标记-压缩算法。

ParNew回收器：并行回收

如果说serialGC是年轻代中的单线程垃圾收集器，那么ParNew收集器则是serial收集器的多线程版本。

Par是Parallel的缩写，New：只能处理的是新生代

新生代除Serial外，目前只有ParNew GC能与CMS收集器配合工作

Parallel回收器：吞吐量优先

HotSpot的年轻代中除了拥有ParNew收集器是基于并行回收的以外，Parallel Scavenge收集器同样也采用了复制算法、并行回收和"Stop the World"机制。

那么Parallel收集器的出现是否多此一举？

- 和ParNew收集器不同，Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput），它也被称为吞吐量优先的垃圾收集器。
- 自适应调节策略也是Parallel Scavenge与ParNew一个重要区别。

Parallel old收集器采用了标记-压缩算法，但同样也是基于并行回收和"stop-the-World"机制。

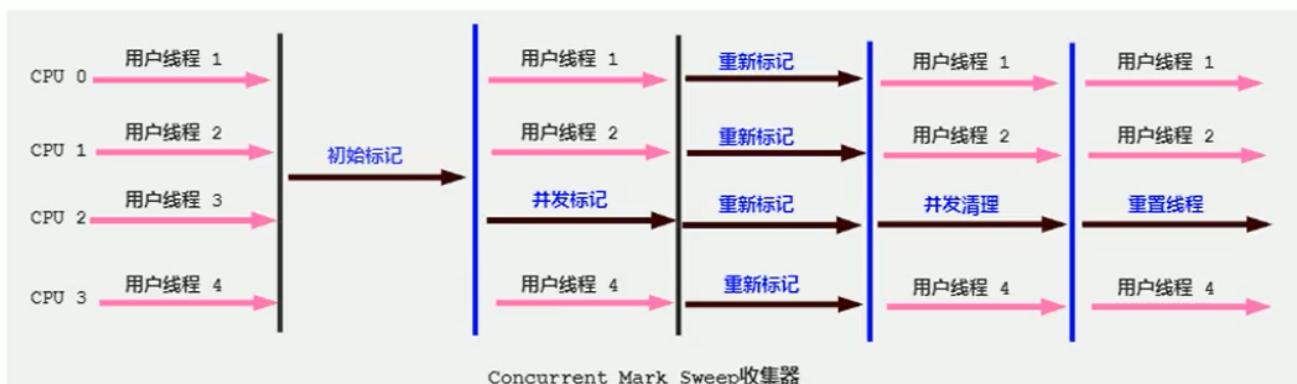
CMS回收器：低延迟

在JDK1.5时期，Hotspot推出了一款在强交互应用中几乎可认为有划时代意义的垃圾收集器：CMS（Concurrent-Mark-Sweep）收集器，这款收集器是HotSpot虚拟机中第一款真正意义上的并发收集器，**它第一次实现了让垃圾收集线程与用户线程同时工作**。

CMS收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间。停顿时间越短（低延迟）就越适合与用户交互的程序，良好的响应速度能提升用户体验。

CMS的垃圾收集算法采用标记-清除算法，并且也会"stop-the-world"

CMS整个过程比之前的收集器要复杂，整个过程分为4个主要阶段，即初始标记阶段、并发标记阶段、重新标记阶段和并发清除阶段。（涉及STW的阶段主要是：初始标记和重新标记）



- 初始标记** (Initial-Mark) 阶段：在这个阶段中，程序中所有的工作线程都将会因为"stop-the-world"机制而出现短暂的暂停，这个阶段的主要任务仅仅只是**标记出GC Roots能直接关联到的对象**。一旦标记完成之后就会恢复之前被暂停的所有应用线程。由于直接关联对象比较小，所以这里的速度非常快。
- 并发标记** (Concurrent-Mark) 阶段：从GC Roots的直接关联对象开始遍历整个对象图的过程，这个过程耗时较长但是不需要停顿用户线程，可以与垃圾收集线程一起并发运行。
- 重新标记** (Remark) 阶段：由于在并发标记阶段中，程序的工作线程会和垃圾收集线程同时运行或者交叉运行，因此为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间通常会比初始标记阶段稍长一些，但也远比并发标记阶段的时间短。
- 并发清除** (Concurrent-Sweep) 阶段：此阶段清理删除掉标记阶段判断的已经死亡的对象，释放内存空间。由于不需要移动存活对象，所以这个阶段也是可以与用户线程同时并发的。

小结

Serial GC、Parallel GC、Concurrent Mark Sweep GC这三个Gc有什么不同呢？请记住以下口令：

- 如果你想要最小化地使用内存和并行开销，请选Serial GC；
- 如果你想要最大化应用程序的吞吐量，请选Parallel GC；
- 如果你想要最小化GC的中断或停顿时间，请选CMS GC。

JDK 1.8默认的垃圾回收器组合是Parallel收集器+Parallel Old收集器

-XX: +UseConcMarkSweepGC手动指定使用CMS收集器执行内存回收任务。

开启该参数后会自动将-xx: +UseParNewGC打开。即：ParNew（Young区用）+CMS（Old区用）+Serial old的组合。

G1回收器：区域化分代式

官方给G1设定的目标是在延迟可控的情况下获得尽可能高的吞吐量，所以才担当起“全功能收集器”的重任与期望。

G1是一个并行回收器，它把堆内存分割为很多不相关的区域（Region）（物理上不连续的）。使用不同的Region来表示Eden、幸存者0区、幸存者1区、老年代等。

G1 GC有计划地避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region。

在jdk8中还不是默认的垃圾回收器，需要使用-xx: +UseG1GC来启用。

优点

- 并行与并发
- 分代收集
- 空间整合：内存的回收是以region作为基本单位的。Region之间是复制算法，整体上实际可看作是标记-压缩算法。尤其是当Java堆非常大的时候，G1的优势更加明显。
- 可预测的停顿时间模型：G1除了追求低停顿外，还能建立可预测的停顿时间模型
 - 由于分区的原因，G1可以只选取部分区域进行内存回收，这样缩小了回收的范围
 - G1跟踪各个Region里面的垃圾堆积的价值大小

缺点

G1无论是为了垃圾收集产生的内存占用（Footprint）还是程序运行时的额外执行负载（overload）都要比CMS要高。

从经验上来说，在小内存应用上CMS的表现大概率会优于G1，而G1在大内存应用上则发挥其优势。平衡点在6-8GB之间。

参数

- -XX:+UseG1GC：手动指定使用G1垃圾收集器执行内存回收任务
- -XX:G1HeapRegionSize设置每个Region的大小。值是2的幂，范围是1MB到32MB之间，目标是根据最小的Java堆大小划分出约2048个区域。默认是堆内存的1/2000。
- -XX:MaxGCPauseMillis 设置期望达到的最大Gc停顿时间指标（JVM会尽力实现，但不保证达到）。默认值是200ms
- -XX:+ParallelGcThread 设置STW工作线程数的值。最多设置为8
- -XX:ConcGCThreads 设置并发标记的线程数。将n设置为并行垃圾回收线程数（ParallelGcThreads）的1/4左右。
- -XX:InitiatingHeapoccupancyPercent 设置触发并发Gc周期的Java堆占用率阈值。超过此值，就触发GC。默认值是45。

G1垃圾回收器的回收过程

G1GC的垃圾回收过程主要包括如下三个环节：

- 年轻代GC (Young GC)
- 老年代并发标记过程 (Concurrent Marking)
- 混合回收 (Mixed GC)

1. 当年轻代的Eden区用尽时开始年轻代回收过程。G1GC暂停所有应用程序线程，启动多线程执行年轻代回收。然后从年轻代区间移动存活对象到Survivor区间或者老年区间，也有可能是两个区间都会涉及。
2. 当堆内存使用达到一定值（默认45%）时，开始老年代并发标记过程。老年代的G1回收器和其他GC不同，G1的老年代回收器不需要整个老年代被回收，一次只需要扫描/回收一小部分老年代的Region就可以了。

总结

垃圾收集器	分类	作用位置	使用算法	特点	适用场景
Serial	串行运行	作用于新生代	复制算法	响应速度优先	适用于单CPU环境下的client模式
ParNew	并行运行	作用于新生代	复制算法	响应速度优先	多CPU环境Server模式下与CMS配合使用
Parallel	并行运行	作用于新生代	复制算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
Serial Old	串行运行	作用于老年代	标记-压缩算法	响应速度优先	适用于单CPU环境下的Client模式
Parallel Old	并行运行	作用于老年代	标记-压缩算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
CMS	并发运行	作用于老年代	标记-清除算法	响应速度优先	适用于互联网或B/S业务
G1	并发、并行运行	作用于新生代、老年代	标记-压缩算法、复制算法	响应速度优先	面向服务端应用

Spring Farmowork

Spring

IOC

Bean线程安全吗

Spring容器中的Bean对象可以是单例或者多例。一般来说容器中如果注入的是无状态的对象，是线程安全的，但是如果是有状态的或者定义可修改的成员变量时，则不上线程安全的，需要考虑加锁。

例如在Spring容器中注入一个Bean，该Bean中定义一个ArrayList成员变量，并且在Bean中的成员方法中涉及到list变量的修改查询操作，这种情况不是线程安全的，需要加锁。

例如下面的例子，在1000个并发请求下，会出现线程安全的问题：可能出现插入相同size的值或者抛出并发修改异常（ConcurrentModificationException）。

```
@Component
public class XXXService{
    private final List<String> list = new ArrayList<>();

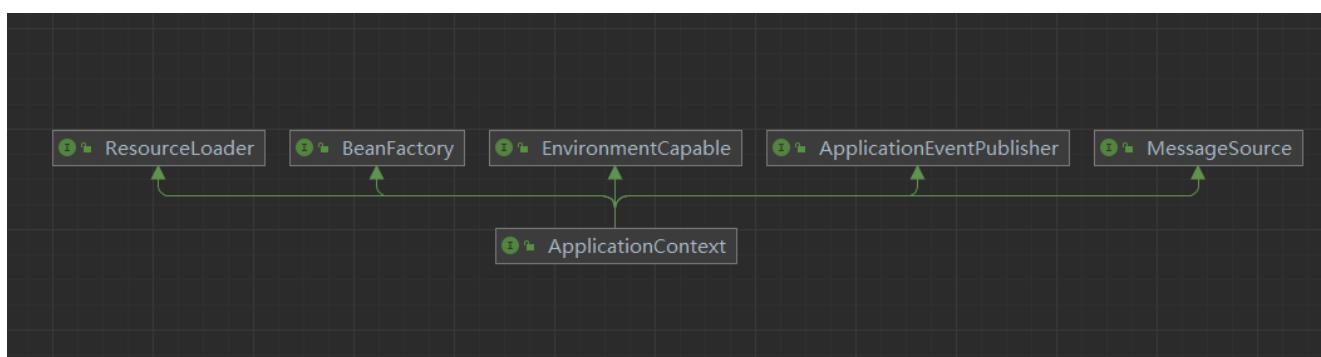
    @GetMapping("/test")
    public String add() {
        list.add(String.valueOf(list.size()));
        return list.toString();
    }
}
```

底层接口

BeanFactory

Spring的顶层接口，也是Spring的核心容器，表面上只是getBean，但是控制控制反转、依赖注入，包括Bean的生命周期都是由它的实现类提供的

ApplicationContext



由BeanFactory派生出来的，扩展了spring容器的功能，底层是实现下面的接口

- ApplicationEventPublisher: 事件发布功能
- BeanFactory: spring容器

- EnvironmentCapable: 环境信息, 系统变量, properties文件, yml文件等配置的值
- MessageSource: 国际化
- ResourcecLoader: 解析资源文件

BeanFactoryPostProcessor

增强beanDefination

- ConfigurationClassPostProcessor: 能够解析@ComponentScan、@Bean、@Import、@ImportResource注解
- MapperScannerConfigurer: 解析@MapperScan注解

BeanPostProcessor

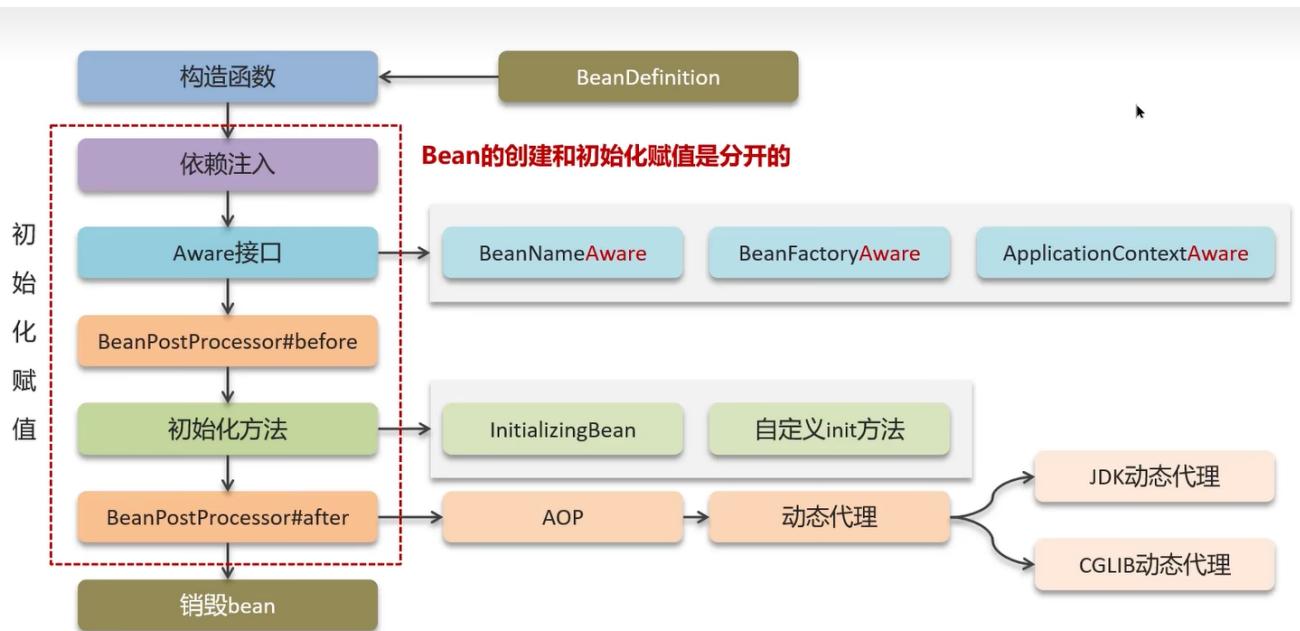
增强bean

- AutowiredAnnotationBeanPostProcessor: 能够解析@Autowired
- CommonAnnotationBeanPostProcessor: 解析@Resource、@PostConstruct、@PreDestroy
- ConfigurationPropertiesBindingPostProcessor: 解析@ConfigurationProperties注解

Bean生命周期

```
AbstractAutowireCapableBeanFactory{
    createBean(){
        //1. 解析BeanDefinition
        beanDefinition = new RootBeanDefinition(mbd);
        doCreateBean(){
            // 2. 根据BeanDefinition实例化对象
            instance = createBeanInstance(beanName, beanDefinition, args);
            // 3. 填充bean对象的属性
            populateBean(beanName, BeanDefinition, instance);
            initializeBean(){
                // 4. 调用各种Aware, 注入相关容器信息
                invokeAwareMethods();
                // 5. 调用后置处理器-前置处理
                applyBeanPostProcessorsBeforeInitialization();
                // 6. 调用初始化方法: InitializingBean#afterPropertiesSet()和自定义的初始化方法<bean
                id="xxx" initMethodName="xxx"/>
                invokeInitMethods();
                // 7. 调用后置处理器-后置处理
                applyBeanPostProcessorsAfterInitialization();
            }
        }
    }
}
```

解析BeanDefinition -> 构造函数(new) -> 依赖注入 -> Aware接口 -> BeanPostProcessor#before -> 初始化方法 -> BeanPostProcessor#after -> 使用 -> 销毁



BeanDefinition

将XML文件中定义的bean解析为BeanDefinition对象，根据这个类来创建对象。

构造函数

实例化对象

依赖注入

三种方式： XML注入；注解@Value注入；@Configuration手动注入

Aware接口

底层实现：`invokeAwareMethods`

可以用来注入一些与容器相关的信息，例如BeanNameAware获取bean的名字，BeanFactoryAware获取bean工厂，ApplicationContextAware获取应用上下文

Spring循环引用

什么是循环引用

当对象A引用对象B，并且对象B引用对象A，或多个对象之间的引用形成了闭环时，成为循环引用。

循环引用有哪几种形式

属性注入，Setter注入，构造器注入。Spring内部只能自动解决前两种，构造器注入形式无法解决。为什么不能解决呢？是因为spring内部解决依赖冲突时在bean生命周期的构造函数和依赖注入中间阶段来解决的，如果在实例化时就发生了循环依赖，就无法解决。但是可以手动在构造器中添加@Lazy注解来解决，等真正使用的时候再初始化。

```
public class A{
    public A(@Lazy B){}
}
```

如何解决的

参考博客：<https://blog.csdn.net/lzb348110175/article/details/125086262>

引入三级缓存，一级缓存存储已经创建好的对象，二级缓存存储已经实例化但是没用初始化的对象，三级缓存存储实例化后的对象的ObjectFactory对象

如果A被AOP代理，那么通过ObjectFactory获取到的就是A代理后的对象，如果A没有被AOP代理，那么这个工厂获取到的就是A实例化的对象。

1. 首先A对象实例化，然后依赖注入B对象前，会先把A对象对应的ObjectFactory对象放入到三级缓存中
2. 开始实例化B对象，然后创建B对象对应的ObjectFactory放入到三级缓存中
3. 当B对象需要注入A对象时，B对象会从三级缓存中获取到A对象的ObjectFactory，创建A对象后放入二级缓存。
4. 当A对象注入到B对象后，B对象创建完，然后将B对象放入一级缓存并注入到A对象中。
5. 最后完成创建A对象，清除二级缓存中的A对象，并放入一级缓存。

二级缓存能解决吗？

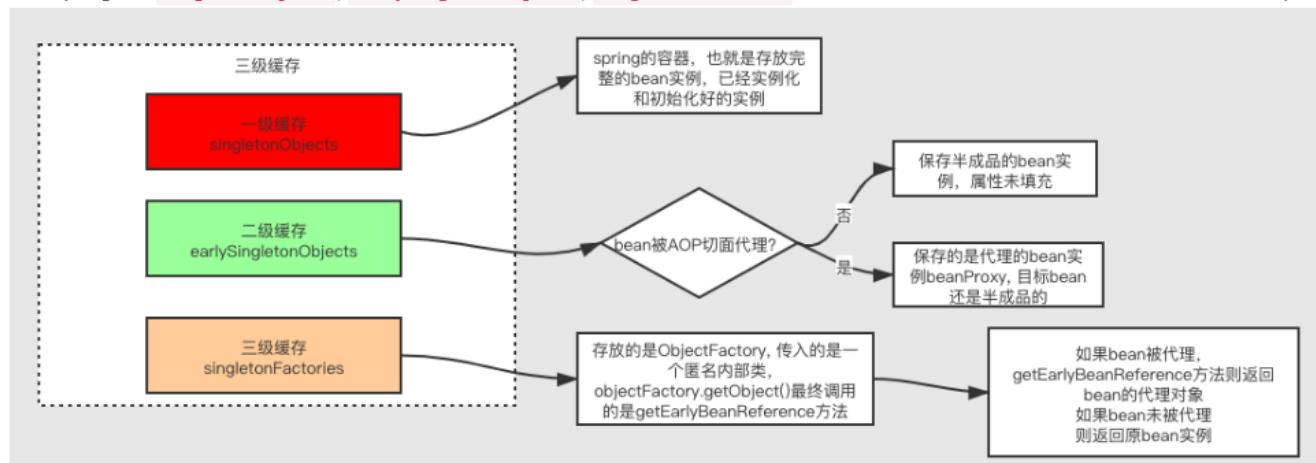
如果只是创建普通对象，不是代理对象，只需要二级缓存就可以，如果是代理对象，当B需要注入A对象时，需要从三级缓存获取到ObjectFactory对象然后调用getObject()方法完成注入。

如果没用三级缓存，有两种解决方式：

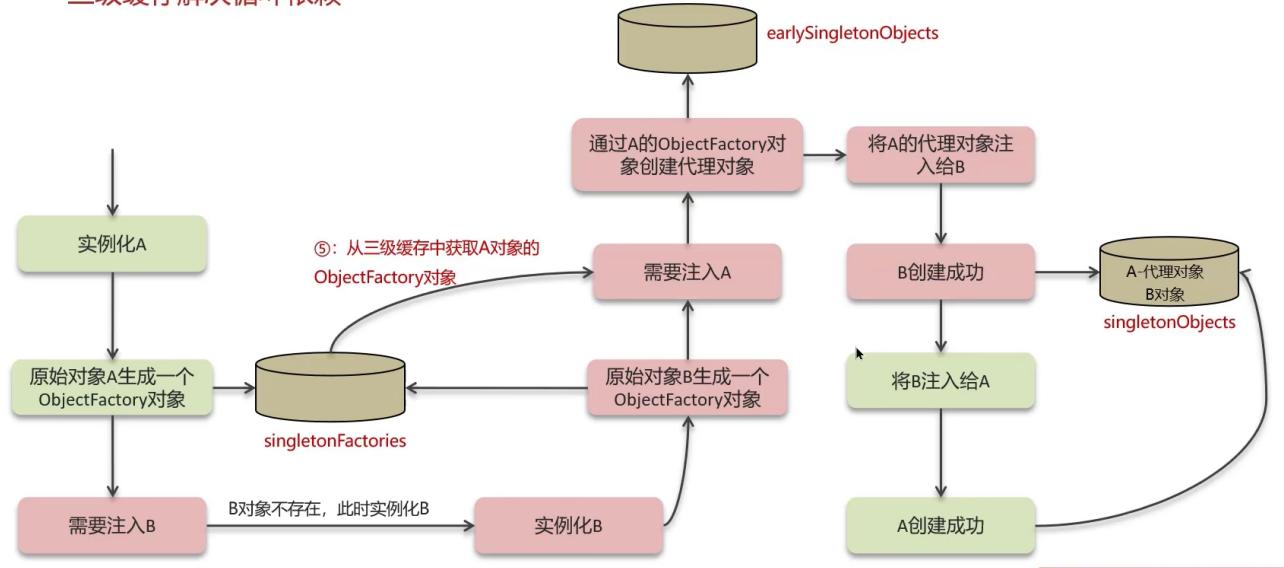
第一种：每次A对象在实例化之后都完成AOP代理，放入到二级缓存。

第二种：每次都在需要注入A对象时判断是否需要对A进行代理，如果需要代理，从二级缓存中取出A对象完成代理。如果同时有很多引用A的代理对象，需要每次都调用ObjectFactory.getObject()方法，**但是每次获取到的A的代理对象是不同的**，导致不同的对象引用了不同的A代理对象，很明显和A对象是单例的像冲突。

spring利用 `singletonObjects`, `earlySingletonObjects`, `singletonFactories` 三级缓存去解决的，所说的缓存其实也就是三个Map



三级缓存解决循环依赖



三级缓存解决循环依赖

Spring解决循环依赖是通过三级缓存，对应的三级缓存如下所示：

```
//单实例对象注册器
public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements SingletonBeanRegistry {
    private static final int SUPPRESSED_EXCEPTIONS_LIMIT = 100;
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(256);
    private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<String, ObjectFactory<?>>(16);
    private final Map<String, Object> earlySingletonObjects = new ConcurrentHashMap<String, Object>(16);
}
```



缓存名称	源码名称	作用
一级缓存	singletonObjects	单例池，缓存已经经历了完整的生命周期，已经初始化完成的bean对象
二级缓存	earlySingletonObjects	缓存早期的bean对象（生命周期还没走完）
三级缓存	singletonFactories	缓存的是ObjectFactory，表示对象工厂，用来创建某个对象的

AOP

什么是AOP

面向切面编程，指的是在现有的代码逻辑前后加入前置增强、后置增强或者环绕增强代码，实现增强代码逻辑和功能代码逻辑的解耦。例如，日志，事务等场景。

项目中如何使用的

在项目中，有一项任务需要通过restful方式调用另外一个部门的API接口，在调用数据接口之前需要登录。因此在所有的调用数据的请求前加入后置处理，分为三步：

1. 调用取数据代码，如果正常返回数据，则结束。
2. 如果返回的http状态码是401，则会调用登录的api来取cookie，然后加入到httpClient中，接着会重新调用取数据的api代码后然后返回。

事务是如何实现的

底层使用AOP实现，对标注有声明式事务注解@Transaction的方法添加环绕增强，执行方法前开启事务，执行完毕后根据执行情况来判断是提交还是回滚事务。

事务失效场景

异常捕获

在添加@Transaction的方法上手动捕获异常，并没有把异常抛出。解决办法是在catch代码块中添加throw new Exception(e)抛出异常。

```
@Transaction  
public void test1(){  
    try{  
        queryDB();  
        int i = 1/0;  
        insertDB();  
    }catch(Exception e){  
        throw new Runtime(e);  
    }  
}
```

抛出检查时异常

Spring事务默认只检查运行时异常，不检查其他异常。解决方法是更改transaction回滚的异常类型。

```
@Tranaction(rollbackFor=Exception.class)  
public void test2() throw FileNotFoundException{  
    query();  
    new FileInputStream("");  
    insert();  
}
```

非public方法

```
@Tranaction(rollbackFor=Exception.class)  
void test2() throw FileNotFoundException{  
    query();  
    new FileInputStream("");  
    insert();  
}
```

SpringMVC

执行流程

简约流程

1. 请求发送给DispatchServlet后，调用handleMapping获取到handleExecutionChain对象
2. DispatchServlet通过处理器获取到处理适配器，并调用处理适配器处理请求产生mv对象。如果返回的mv对象是空，直接结束。
3. DispatchServlet调用视图解析器获取到视图
4. DispatchServlet进行视图渲染

视图流程

1. 用户发送请求到前端控制器DispatchServlet
2. DispatchServlet收到请求后调用处理映射器HandleMapping，
3. HandleMapping找到具体的处理器，生成HandleExecutionChain返回给DispatchServlet，包含拦截器和处理器
4. DispatchServlet调用HandleAdapter
5. HandleAdapter调用具体的处理器
6. 处理完成后返回modelAndView对象给HandleAdapter
7. HandleAdapter将结果 ModelAndView 返回给 DispatchServlet，如果是空直接结束，不进行试图渲染。
如果加了 @ResponseBody 注解，将会把处理后的诗句利用 messageConvert 转化后存入到 response 中
8. DispatchServlet 将 mv 传给视图解析器
9. 视图解析器处理完成后返回 View 对象给 DispatchServlet
10. DispatchServlet 渲染视图

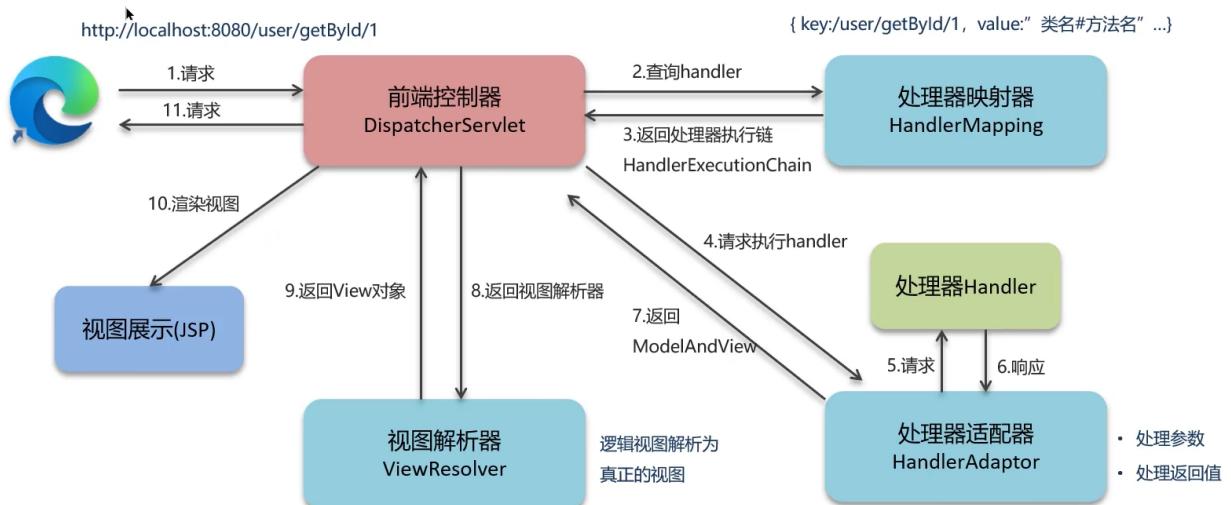
核心代码

```

public class DispatcherServlet extends HttpServlet{
    doDispatch(){
        HandlerExecutionChain mappedHandler = getHandler(processedRequest);
        HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
        ModelAndView mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
        if(mv != null){
            render(mv, request, response){
                View view = resolveViewName(viewName, mv.getModelInternal(), locale, request);
                view.render(mv.getModelInternal(), request, response);
            }
        }
    }
}

```

视图阶段 (JSP)



组件

DispatchServlet

作用：接受请求，响应结果。相当于中央处理器。

HandlerMapping

根据请求的URL找到对应的handle，支持多种方式，例如配置文件，实现接口，注解方式等。

HandlerAdapter

查看博客：https://blog.csdn.net/liuhaibo_ljf/article/details/106000158

处理适配器，在流程中根据处理器来获取到对应的处理适配器，该接口有下面几个实现类

- SimpleControllerHandlerAdapter：当处理器实现了 Controller 接口，使用该处理适配器
- SimpleServletHandlerAdapter：当处理器 Servlet 接口，使用该处理适配器
- HttpRequestHandlerAdapter：当处理器实现 HttpRequestHandler 接口，使用该处理适配器
- RequestMappingHandlerAdapter：当处理器使用 @RequestMapping 注解时进行 URL 和方法映射时，使用该处理适配器

ViewResolver

视图解析器，根据逻辑视图名解析为真正的视图。

SpringBoot

自动配置原理

在Springboot项目中的引导类上有一个注解@`SpringBootApplication`，该注解包含了三个注解

1. `@SpringBootConfiguration`: 和`@Configuration`注解作用相同，表示当前引导为配置类
2. `@ComponentScan`: 扫描引导类所在包并注入到Spring容器中
3. `@EnableAutoConfiguration`: 自动配置注解，通过`@Import`注解导入配置选择器。内部实现是通过解析META-INF/Spring.factories文件中配置类，并结合`@conditional`注解判断是否导入到Spring容器中

常用注解

Spring

Spring 的常见注解有哪些？

注解	说明
<code>@Component</code> 、 <code>@Controller</code> 、 <code>@Service</code> 、 <code>@Repository</code>	使用在类上用于实例化Bean
<code>@Autowired</code>	使用在字段上用于根据类型依赖注入
<code>@Qualifier</code>	结合 <code>@Autowired</code> 一起使用用于根据名称进行依赖注入
<code>@Scope</code>	标注Bean的作用范围
<code>@Configuration</code>	指定当前类是一个 Spring 配置类，当创建容器时会从该类上加载注解
<code>@ComponentScan</code>	用于指定 Spring 在初始化容器时要扫描的包
<code>@Bean</code>	使用在方法上，标注将该方法的返回值存储到Spring容器中
<code>@Import</code>	使用 <code>@Import</code> 导入的类会被Spring加载到IOC容器中
<code>@Aspect</code> 、 <code>@Before</code> 、 <code>@After</code> 、 <code>@Around</code> 、 <code>@Pointcut</code>	用于切面编程（AOP）

Spring MVC

SpringMVC常见的注解有哪些？

注解	说明
<code>@RequestMapping</code>	用于映射请求路径，可以定义在类上和方法上。用于类上，则表示类中的所有的方法都是以该地址作为父路径
<code>@RequestBody</code>	注解实现接收http请求的json数据，将json转换为java对象
<code>@RequestParam</code>	指定请求参数的名称
<code>@PathVariable</code>	从请求路径下中获取请求参数 <code>(/user/{id})</code> ，传递给方法的形式参数
<code>@ResponseBody</code>	注解实现将controller方法返回对象转化为json对象响应给客户端
<code>@RequestHeader</code>	获取指定的请求头数据
<code>@RestController</code>	<code>@Controller + @ResponseBody</code>

SpringBoot

Springboot常见注解有哪些？

注解	说明
@SpringBootConfiguration	组合了- @Configuration注解，实现配置文件的功能
@EnableAutoConfiguration	打开自动配置的功能，也可以关闭某个自动配置的选
@ComponentScan	Spring组件扫描

Spring Cloud

Consul

概念

是一款用于实现分布式系统服务注册与发现的开源工具。主要有五个特点：**服务注册与发现、健康检查、KV存储、安全通信和多数据中心。**

底层架构使用Client-Server模式，server一般3台或者五台，client是server的代理，可以扩展很多台。

Server一般是3台或者5台，因为当存活节点大于全部节点的一半时，集群才能正常工作，太少不满足可靠性，太多机器的话需要花费大量时间做数据同步。

每一个数据中心都是相对独立的一部分，在每一个数据中心内部选取单个leader，来负责处理该数据中心的所有查询和事务。

优点

数据一致性：当leader节点宕机时会停止所有数据节点，然后选举新的Leader后重启集群，在重启过程中，会拒绝服务。保证了数据的一致性。

多数据中心：Consul可以搭建多数据中心，每一个数据中心都是一个独立的集群，默认情况下，每一个集群里的数据是不同步的，也就是不相互影响。但是也可以支持多数据中心的数据同步，官方也给出响应的解决方案。

整合

第一步，添加spring-cloud-consul 依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

第二步，配置服务注册与发现的地址，同时也可以配置健康检查路径

```
spring.cloud.consul.host=wpc.vm
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.hostname=wpc.vm
spring.cloud.consul.discovery.heartbeat.enabled=true
spring.cloud.consul.discovery.health-check-path=
```

第三步，在主启动类上添加@EnableDiscoveryClient注解，开启自动配置

APIGateway

概念

参考博客：<https://blog.csdn.net/crazymakercircle/article/details/125057567>

该项目是基于SpringBoot和Reactor开发的网关，

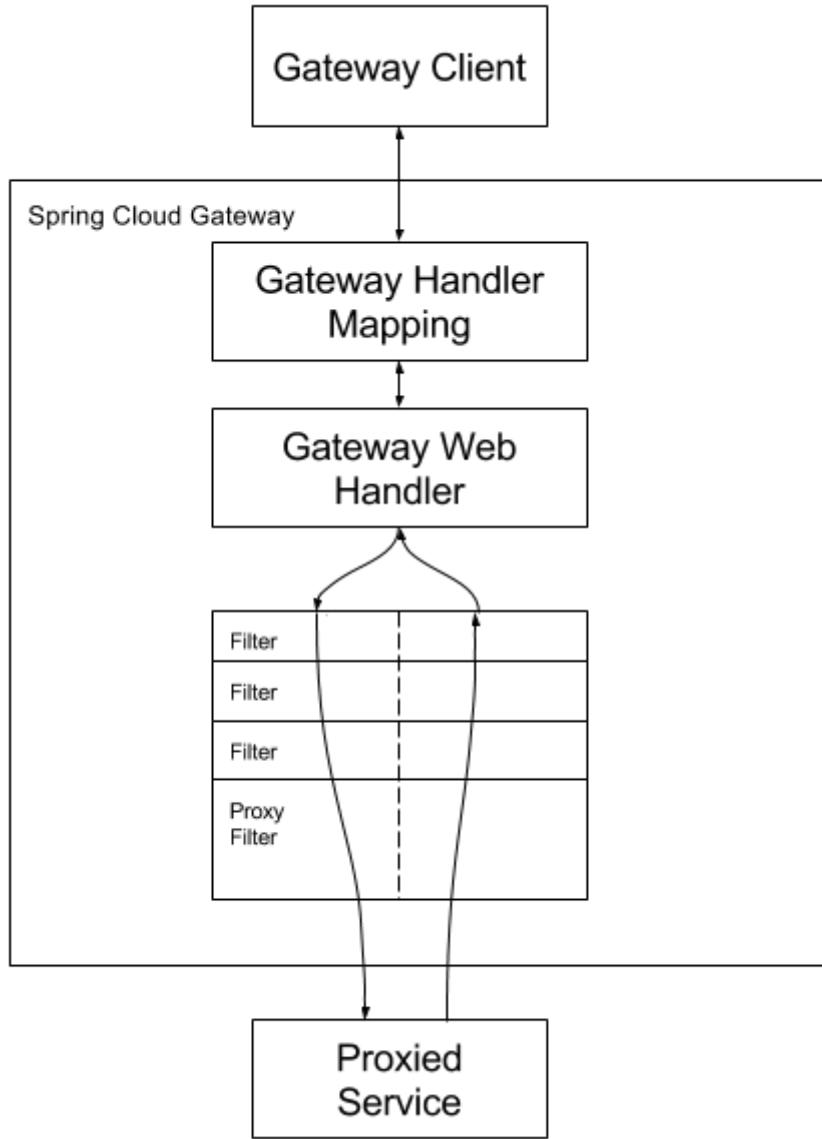
目的是

1. 提供一种简单有效的API路由管理方式
2. 安全校验：例如是否是在黑名单中，是否是合法用户，是否具备访问资源的权限
3. 监控请求
4. 限流

SpringCloud-Gateway是基于WebFlux框架来实现的，底层使用了高性能的Reactor模式通信的Netty框架。

处理流程

1. 客户端发起请求到Gateway
2. Gateway在Handle Mapping中找到与请求匹配的路由，然后发给WebHandler
3. WebHandler通过各种过滤器的preHandle后，发送到具体的服务上
4. 服务处理完成之后的响应信息会再走一遍过滤器的postHandle方法



组件

路由Route：由ID，目标URI，一组断言和一组过滤器组成

断言Predicates：路由转发条件，可以通过对HTTP请求进行匹配，例如请求头，请求参数，请求时间，请求路径等

过滤器Filter：对请求进行拦截，处理和修改

断言

SpringCloud-Gateway对于断言和过滤器的编写提供了两种方式：配置文件（一般是yml形式）和编码。

另外，路由匹配的前提是满足一系列的断言规则。

SpringCloud内置了很多断言规则可以使用。例如根据路径，时间，Cookie，请求头，查询参数，请求主机，请求方法等。可以参考官网文档part5: <https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#gateway-request-predicates-factories>

当然，也可以自定义断言，需要实现RoutePredicateFactory接口（也可以继承AbstractRoutePredicateFactory抽象类）并注入到Spring容器中。

过滤器

可以对传入的HTTP请求和响应进行修改。包含路由过滤器和全局过滤器两种。

- 路由过滤器：作用于特定的路由。官方内置了一些过滤器：对路径加前缀，改写路径，设置请求头等。也可以自定义路由过滤器，需要实现`GatewayFilterFactory`接口（也可以继承`AbstractGatewayFilterFactory`抽象类）并注入到容器中
- 全局过滤器：作用于全部的路由，也可以自定义全局过滤器，实现`GlobalFilter`接口。

Sleuth

概念

SpringCloud-Sleuth提供了一种分布式链路追踪的解决方案，也集成了Zipkin。分布式链路追踪就是将一次分布式请求还原成调用链路进行日志分析，比如各个节点的耗时，处理机器、处理状态等。

内部实现

1. Span：基本的工作单元，使用`SpanId`来标记。除了记录了在当前服务组件的开始、处理和结束时间外，还记录了时间的名称，请求信息等元数据。
2. Trace：表示一条请求链路，通常为树状结构，由一组Span串联而成。一条链路的`TraceId`是相同的。
3. Annotation/Event：用来记录一段时间内的事件，例如`Client Send / Server Received / Server Send / Client Received`

实现原理

1. 当一个请求在分布式系统中流转时，始终保持传递这一个标识`TraceId`。
2. 对于每一个服务组件，创建对应的Span，并使用`SpanId`来标识。此外，还会将上一个服务组件的`SpanId`标识为`parentSpanId`，这样一条链路便串联起来。
3. 具体实现是通过在传递的消息头上添加响应的`TraceID`和`SpanId`等信息，例如在HTTP头信息，JMS的头信息等消息上。

抽样收集

为了节约资源，Sleuth还支持抽样收集，同时消息是否被收集也是通过消息头上的`Sampled`来决定

Sleuth内置了一些抽样策略，例如通过百分比的方式收集，同时还支持自定义的抽样策略，需要实现`Sample`接口。

整合ELK

Sleuth在整合ELK中，主要负责日志收集，配合Logstash完成数据对接。由于SpringBoot和Logstash都支持logback方式记录收集日志，因此，只需要定义日志转化的格式即可。

例如Sleuth安装指定格式收集日志到JSON文件中，然后ELK可以负责展示管理。

整合Zipkin

Zipkin是推特下的一个开源项目，主要有以下功能

1. 收集请求链路上的各个服务器上的跟踪数据
2. 提供API接口来查询跟踪数据，从而找到系统性能瓶颈
3. 提供UI组件更直观地搜索跟踪信息和分析请求链路明细。
4. 默认情况下，Zipkin将跟踪信息存储在内存中，也可以配置Mysql持久化存储功能。

Contract

概念

契约测试：简单理解为提前模拟服务端的数据给调用者。

具体操作：

1. 服务端提前定义一个已知的结果，也叫契约，可以使用Groovy或者YML文件定义，并生成stub包提供给消费者
2. 当消费者调用的时候直接返回该结果，该结果并不是调用的真实服务，而是从stub包中获取到的结果
3. 根据提供的结果来判断消费者程序是否正确。

常用于测试方法中，介于单元测试和集成测试之间，既脱离了服务端的真实服务依赖，又使用了服务端的相对真实的数据。

底层实现

当服务端代码编写完毕进行打包时，除了打出来服务的jar包，还有一个stubs的jar包，在该jar包内包含了基本测试类，编写的契约文件和生成的Json文件，里面定义了测试的具体内容，例如调用的接口和应该返回的结果。

消费者进行测试时，当启动服务时，不会连接真实的服务端，因为Contract stub会启动mock服务器，并使用之前定义好的契约来返回数据。

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = ConsumerApplication.class, webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
//初始化测试配置，测试controller需要
@AutoConfigureMockMvc
@AutoConfigureJsonTesters
```

```
@AutoConfigureStubRunner(ids = {"com.xiaobai:producer:+:stubs:10001"},stubsMode =
StubRunnerProperties.StubsMode.LOCAL)
@Slf4j
public class ConsumerTest {
    @Autowired
    private RestTemplate restTemplate;

    @Test
    public void testMethod() throws Exception {
        ResponseEntity<JSONObject> response = restTemplate.exchange(
            "http://localhost:10001/user/1",
            HttpMethod.GET,
            null,
            JSONObject.class
        );
        log.info("测试数据:"+ response.getBody().toJSONString());
    }
}
```

中间件

Redis

概念

Redis是一种开源的数据存储系统，可以用作**数据库、缓存、消息中间件**。

- 数据结构：字符串String，列表List，集合Set，哈希表Hash，有序集合Zset等
- 内置功能：复制，LUA脚本，LRU驱动时间，事务和不同类型的持久化策略（RDB和AOF）
- 高可用：通过哨兵模式和自动分区提供高可用

持久化

RDB

全称：Redis Database Backup file

一定时间间隔内把内存中的数据写入磁盘，相当于打个快照，恢复时也是直接读文件到内存中。

该方式是默认开启的，在持久化过程中，会fork一个子进程。先写入临时文件，然后持久化结束后再把这个文件替换上次持久化的文件。

缺点是最后一次持久化操作后的数据可能丢失。

保存策略

- save 900 1 900秒内有1个key变化就保存
- save 300 10 300秒内有10个key变化就保存
- save 60 10000 60秒内有10000个key变化就保存
- save "" 禁用RDB模式

优点

文件类型：持久化文件是二进制文件，同时也记录了时间戳，可以快速定位数据版本

性能方面：在持久化过程中，只需要fork子进程就可以，持久化工作是由子进程来实现，不影响Redis主线程的正常运行

数据恢复：二进制文件数据恢复和传输比较方便

缺点

数据完整性：无法保证数据的完整性，在最后一次持久化后的数据将会丢失。

性能：如果Redis整体数据特别大，每次持久化会很耗时，持久化过程将占用大量系统资源，导致主线程无法及时响应

AOF

全称Append Only File

AOF是以日志文件保存每一次的增删改操作。当Redis重启时，会重新执行这些命令来恢复数据。

默认不开启，需要手动设置。

保存策略

- appendfsync always: 每当有新的修改数据命令时都会执行保存操作：效率低，但是安全
- appendfsync everysec: 每秒保存一次，如果断电，可能会丢失一秒的数据。
- appendfsync no: 从不保存，将数据交给操作系统来处理。

优点

1. 相对于RDB更安全
2. 以日志文件格式保存，包含的是各种redis命令，容易解读

缺点

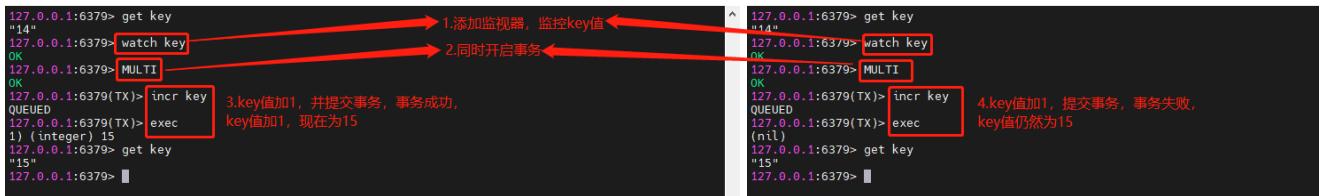
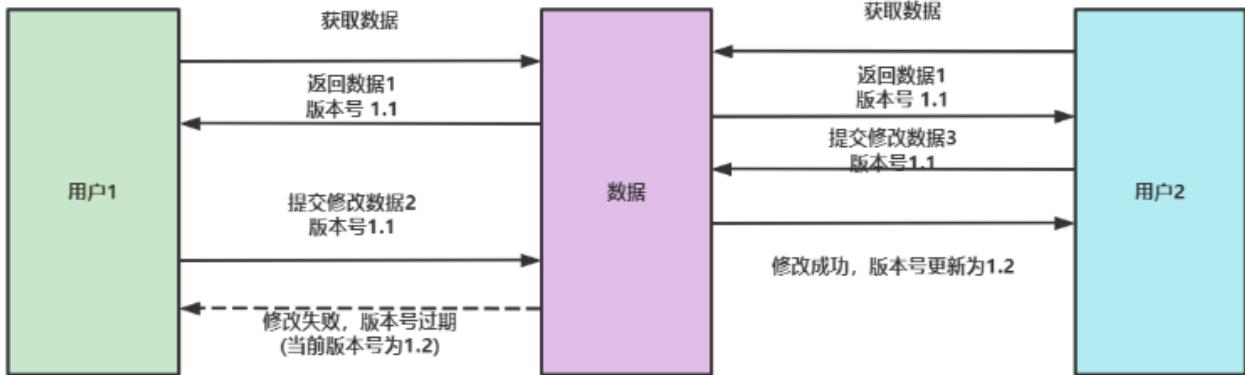
1. 比RDB更占用磁盘空间
2. 恢复速度慢
3. 每次读写都要同步的话，有一定的性能压力

事务

参考博客：https://blog.csdn.net/shark_chili3007/article/details/120884205

Redis中的事务只是一个单独的隔离操作，主要作用是串联多个命令防止其他命令插队。

Redis中的事务时基于乐观锁来实现的，底层实现是用CAS



常用命令：Multi开启事务，Exec执行事务，Discard取消事务

事务的错误和回滚：

- 如果语法层面上有报错，会遗弃所有的命令。
- 如果是运行中报错，则会执行所有的正确指令。

消息订阅

Redis是基于发布订阅的方式实现消息通信，即：发送者发送消息，订阅者接受消息。

- Subscribe [channel...] 订阅频道
- Publish [channel] [message] 发布消息到指定的频道

主从复制

当配置多态服务器时，主机和从机的身份是分开的，主机以写为主，从机以读为主。

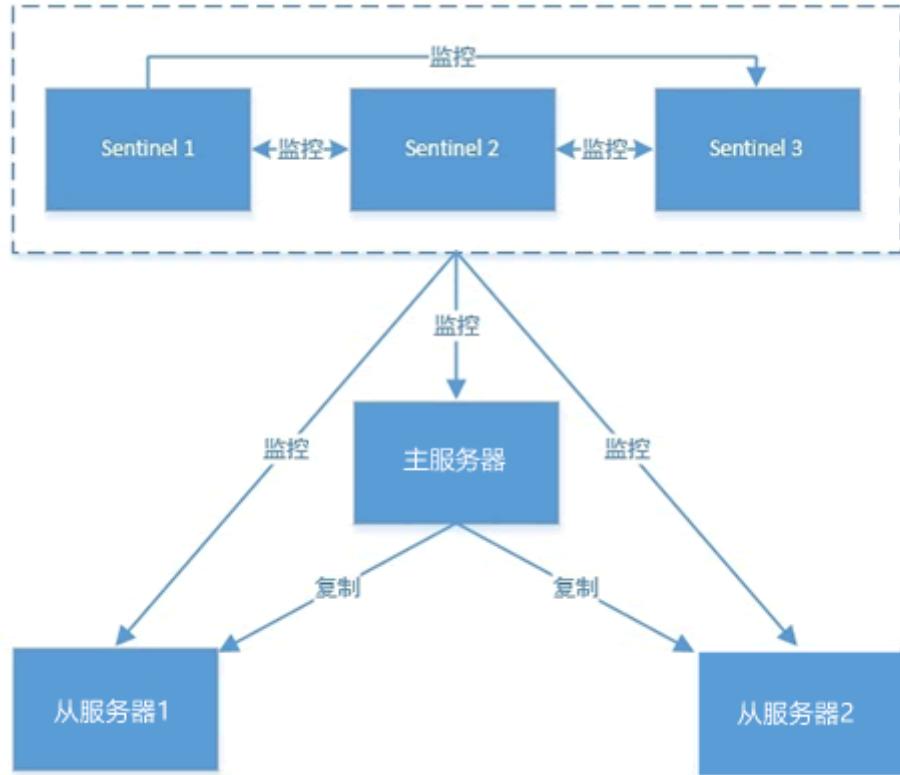
当从机联通后，会向主机发送sync指令来同步数据。当主机接收到写操作时，也会发送给从机。

从机是从头开始复制主机的信息，是完全负责。另外从机不能写，只能读

哨兵模式

在主从模式中，系统不具备自动修复功能，当主节点宕机时，需要手动切换服务器，安全性得不到保障。

Redis官方推荐一种高可用方案：哨兵模式。当主机发生故障时，会自动进行故障转移。



1. 主观下线：适用于主机和从机。在规定的时间内没用收到服务器的有效回复时，会被判定为主观下线。
2. 客观下线：只适用于主机。当哨兵发现主机出现故障时，会询问其他哨兵对主机的判断，如果超过半数以上的哨兵认为主机宕机了，就会被标记为客观下线。
3. 投票选举：所有的哨兵会通过投票机制，选举一个哨兵做故障转移操作。被选举的哨兵按照一定的规则从从服务中选举一个最优的服务器作为主服务器，并通过发布订阅方式通知其余从节点更改配置，跟随新上任的主服务器。至此完成了主从切换。

缓存问题

参考博客：<http://c.biancheng.net/redis/cache.html>

缓存穿透

数据在缓存中没有，在数据库中也没有

解决方案：缓存空对象和布隆过滤器（将所有存在的数据都哈希到一个足够大的map中，如果这个map里没有，则一定没有）

缓存击穿

数据在缓存中没有，但在数据库中存在。

一般出现在热点数据过期场景，大量并发访问该数据将会直接访问数据库。

解决方案：改变过期时间和分布式锁。

缓存雪崩

大量的key同时过期，导致大量请求同时访问数据库。

解决方案：对key设置随机不同的过期时间

缓存预热

在系统上线时，阿静热点数据加载到缓存系统中，这样当用户请求时，就可以直接查询缓存了。

Nginx

概念

1. 是一款高性能反向代理服务器。
2. 占用内存少，并发能力强。
3. 支持动静分离，即静态资源和动态资源来自于不同的服务器，加快解析速度。也可以作为静态页面的web服务器。
4. 具备负载均衡能力

Tips：正向代理是代理客户端，对服务器隐藏真实信息，如VPN。反向代理是代理服务器，隐藏真实访问的服务器。

配置文件

主要由三部分构成

1. 全局块：主要设置一些整体的配置，例如：用户组、日志文件路径、配置文件等
2. event块：主要是配置Nginx服务器与用户的网络连接。例如序列化，网络连接数，驱动模型等。
3. http块：包含http全局块和server块。
每个http块可以包含多个server块，每一个server块相当于一台虚拟主机。
每个server块可以包含多个location块，代表不同规则。

```
worker_processes 1;

# 定义负载均衡组，upstream可以放在全局中，也可以放在http块中
upstream backend {
    #ip_hash 使用hash算法库避免登录信息丢失
    #weight=3 添加服务器比重
    server 192.168.56.128:8080 max_conns=2 fail_timeout=15s weight=3;
    server 192.168.56.129:8080 weight=7;
    #fair; 按照响应时间来分派
}

events {
    worker_connections 1024;
}

http {
```

```

include      mime.types;
default_type application/octet-stream;
sendfile     on;
keepalive_timeout 65;

server {
    listen      80;
    server_name localhost;

#          location后面也有一个参数：等号=是精准匹配；波浪号~是正则匹配；波浪号加星~*是忽略大小写正则匹配；^~表示正常匹配后不再正则匹配
    location / {
        root   /home/ruoyi/projects/ruoyi-ui;
        try_files $uri $uri/ /index.html;
        index  index.html index.htm;
    }

    location /prod-api/ {
#          $http_host代表本机
        proxy_set_header Host $http_host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header REMOTE-HOST $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_pass http://ruoyi-gateway:8080/;
    }

# 避免actuator暴露
if ($request_uri ~ "/actuator") {
    return 403;
}

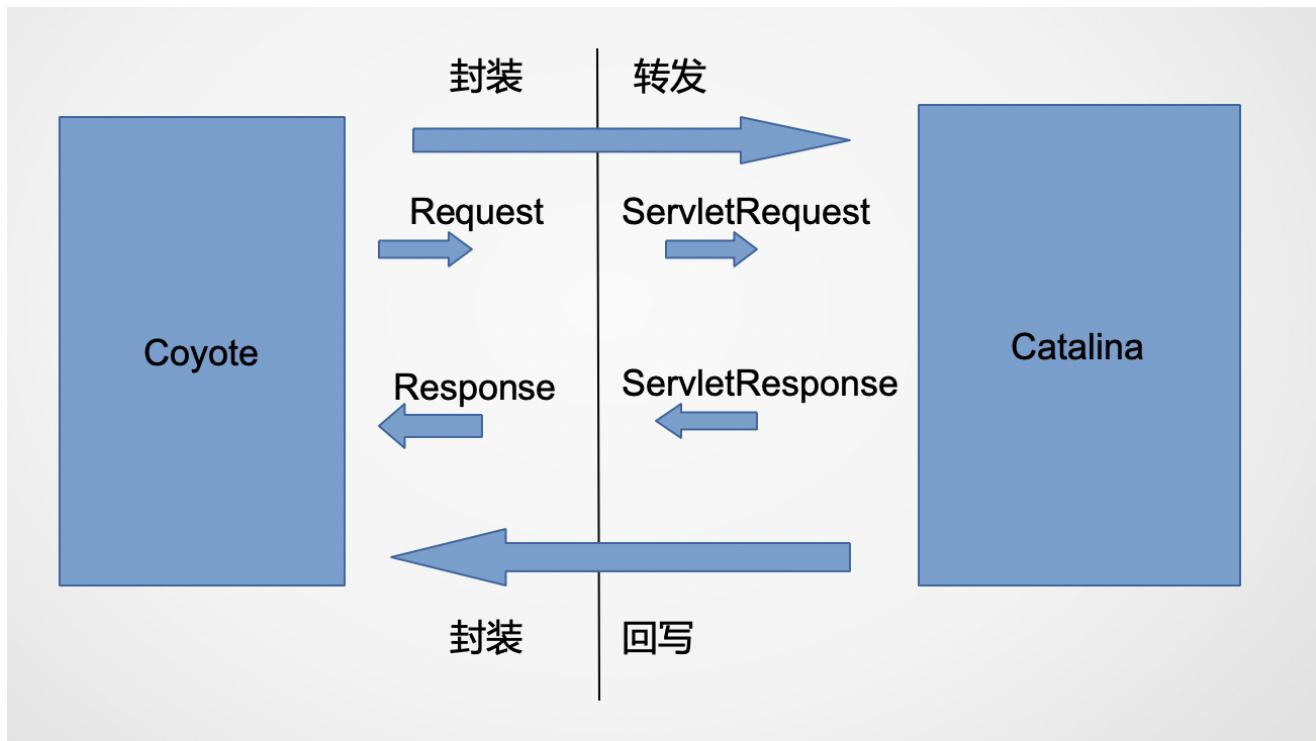
error_page 500 502 503 504  /50x.html;
location = /50x.html {
    root   html;
}
}
}
}

```

Tomcat

总体架构

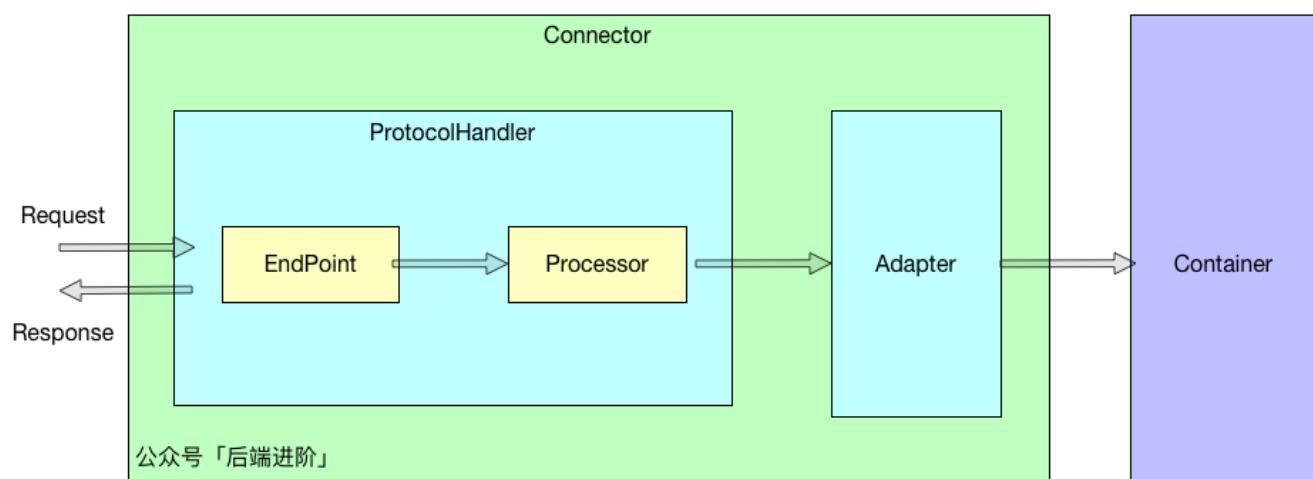
Tomcat服务器主要包含两部分，连接器Connector和容器Container，分别使用Coyote和Catalina实现的。



Connector(Coyote)

链接器主要作用和流程是是

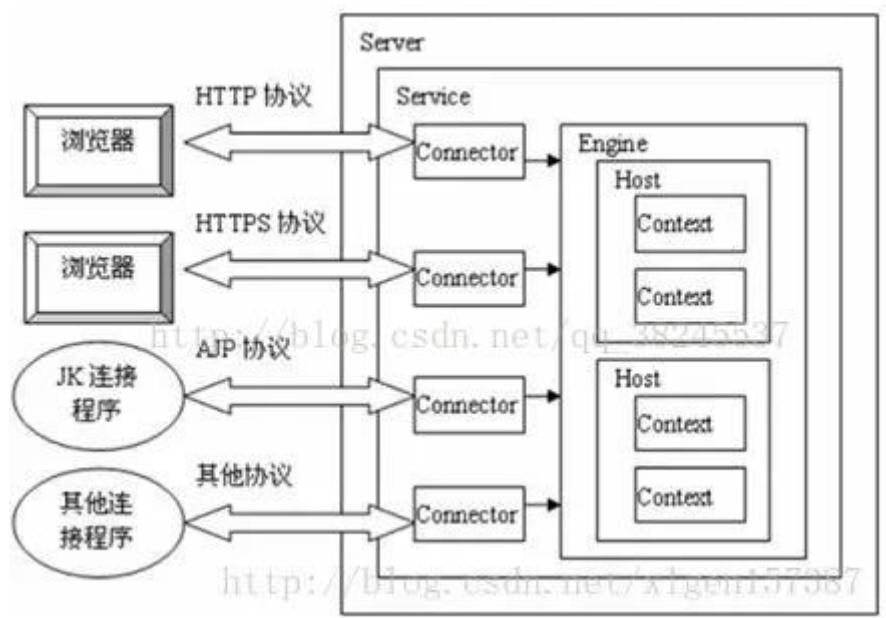
1. 监听：EndPoint接口类监听服务器端口，读取客户端请求
2. 解析：Processor将请求根据指定协议进行解析，例如HTTP/AJP
3. 映射：通过映射表并根据请求地址，来匹配正确的容器进行处理。底层使用了适配器模式解耦。
4. 响应：将响应返回给客户端



支持的协议：HTTP和AJP

支持的IO方式，BIO(8.5版本移除)，NIO，NIO2，APR

Tomcat针对不同的协议和IO方式，提供了不同的实现，他们都实现了ProtocolHandler接口,这里使用了**桥接模式**，因为处理器的种类有两个变化维度，在AbstractProtocol中包含了AbstractEndpoint来定义注入不同IP方式的实现。



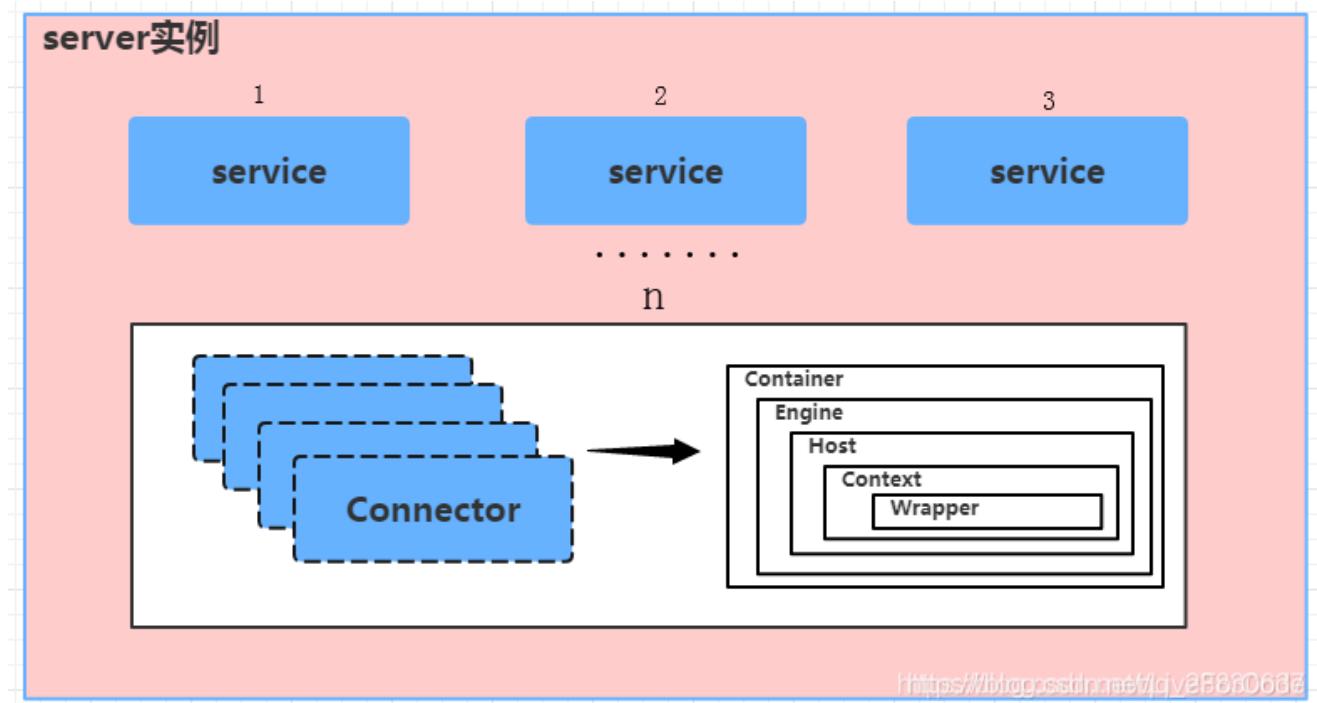
Container(Catalina)

Catalina是Servlet容器的具体实现。具体工作：

1. 解析：使用Digester解析XML文件
2. 创建容器：根据配置文件创建容器对象，在tomcat容器中，都实现了Container接口。
3. 初始化：初始化创建的对象
4. 启动容器：容器类都实现了Lifecycle接口，实现了例如init,start,stop等方法。

在容器的层级关系中，大概如下图的结构

1. 一个server实例包含很多service，一个service包含多个Connector连接器和Engine引擎
2. 引擎内部也是一对多的关系：分别是Host虚拟主机，Context上下文，Wrapper具体的Servlet包装类。



Kafka

概念

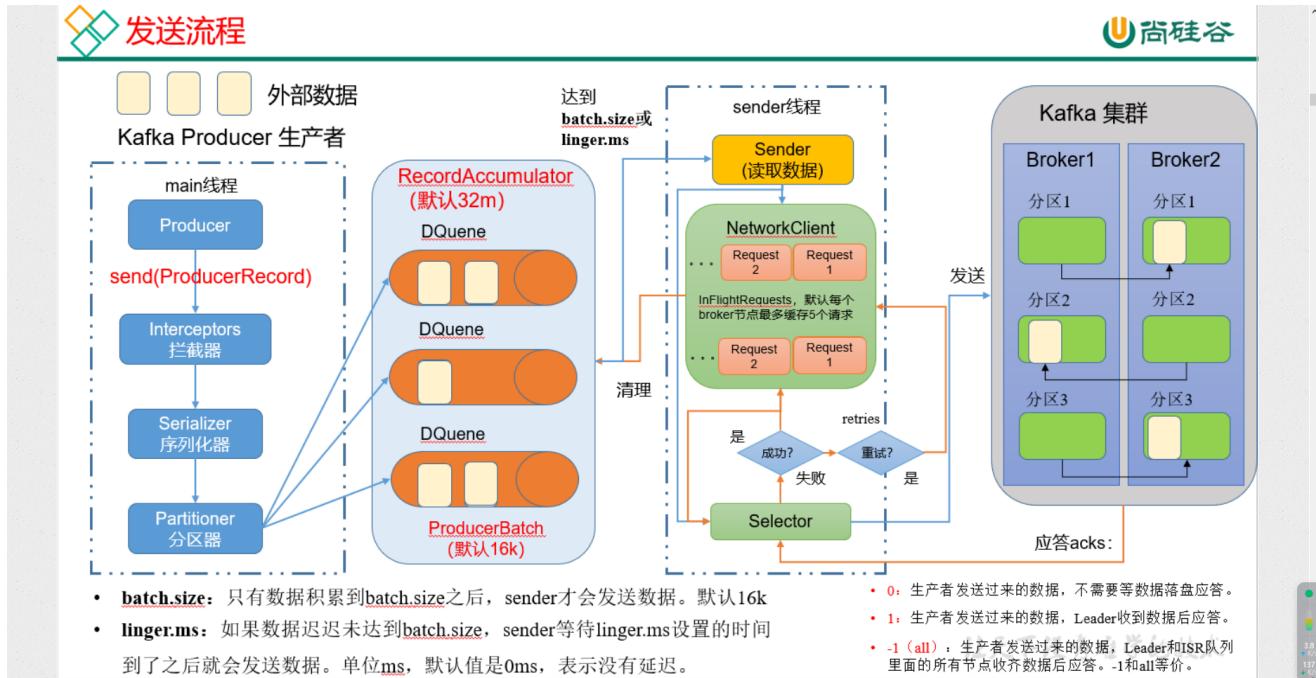
1. Kafka是一个**分布式的基于发布订阅模式的消息队列**，主要应用于大数据实时处理领域。
2. 发布订阅：分为多种类型，订阅者根据需求选择性的订阅
3. 优点：解耦，削峰，异步
4. 两种模式：

1. 点对点模式：一个生产者一个消费者，一个topic，消费完后删除数据。
2. 发布订阅模式：多个生产者和多个消费者相互独立，多个topic消费完不会删除数据

5. 架构

1. 生产者：
2. broker集群
 1. broker服务器
 2. topic主题
 3. partition分区
 4. replication副本
 5. leader&follower主从
3. 消费者
 1. 消费者之间相互独立
 2. 消费者组（某个分区只能有一个消费者消费）
4. zookeeper
 1. 在线的broker(broker.ids): /kafka/brokers/ids
 2. 分区leader: /kafka/brokers/topics/first/partitions/0/state

生产者



分区

分区的好处：

1. 合理的利用资源，如果数据量很大，把数据放在一个很大的区域内，不方便操作
2. 提高并行度：生产者可以以分区为单位发送数据，消费者可以以分区为单位消费数据。

默认分区规则：

- 如果指定分区，则按照分区规则执行
- 如果没有指定分区，但是设置的有key，则根据key的哈希值与分区数取模定位分区
- 如果没有指定分区也没有key，使用粘性策略，第一次随机挑选，到达一定时间，重新随机

自定义分区规则：

- 定义类实现Partitioner接口，
- 重写partition
- 在配置中引用该自定义分区规则

提高吞吐量

1. 修改批次大小(batch.size): 16k -> 32k
2. 修改等待时间(linger.ms) : 0-5ms
3. 压缩
4. 缓冲区大小: 32MB ->64MB

可靠性

acks:

- 0: 不需要等待服务器响应，可靠性差，但效率高，会丢失数据
- 1: 需要leader应答后继续发，也可能会丢失数据，适合传输普通日志

- 1: 需要leader和ISR队列中的所有的follower应答，可靠性高，但是效率差
- 完全可靠: ack = -1 & 副本大于2 & isr >= 2

数据重复

幂等

- <pid, 分区号, 序列号> 默认打开
- 幂等性只能保证单分区会话不重复

事务

- 底层基于幂等
- 5个API

数据有序

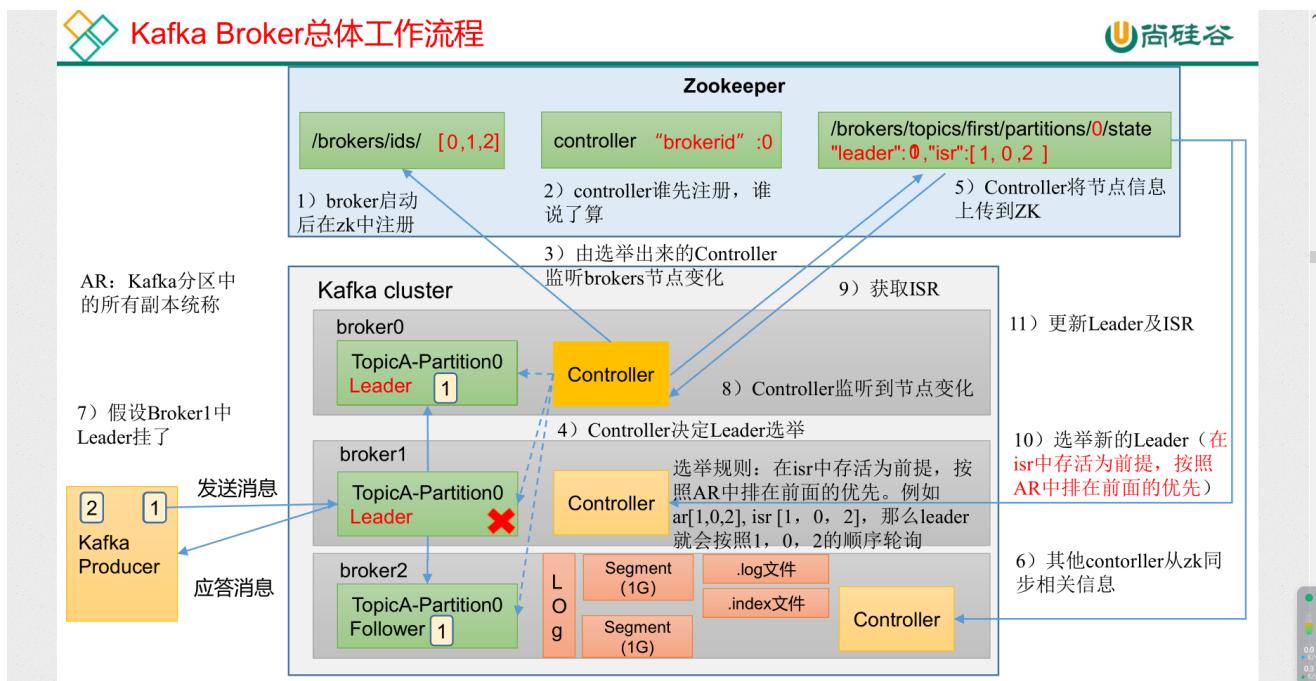
单分区内有序:

多分区有序

乱序: 1.x版本

Broker集群

zookeeper



1. 当前所有的节点broker.ids

2. leader

3. 辅助选举controller

副本

- 作用：提高数据可靠性，默认是一个，生产环境一般配置两个
- 分区统称为AR。 AR=ISR(活跃的Follow集合)+OSR
- controller：前提是ISR上存活，第一抢到的broker设置controller leader，如果leader挂了轮询选一个
- leader挂了（选举一个leader然后同步follower）和follower挂了（同步最高水位线HW后的数据）
- 副本分配，默认是均匀的分配到每个机器上，防止过度集中和依赖某个机器。也可以手动配置副本的分配。

存储机制

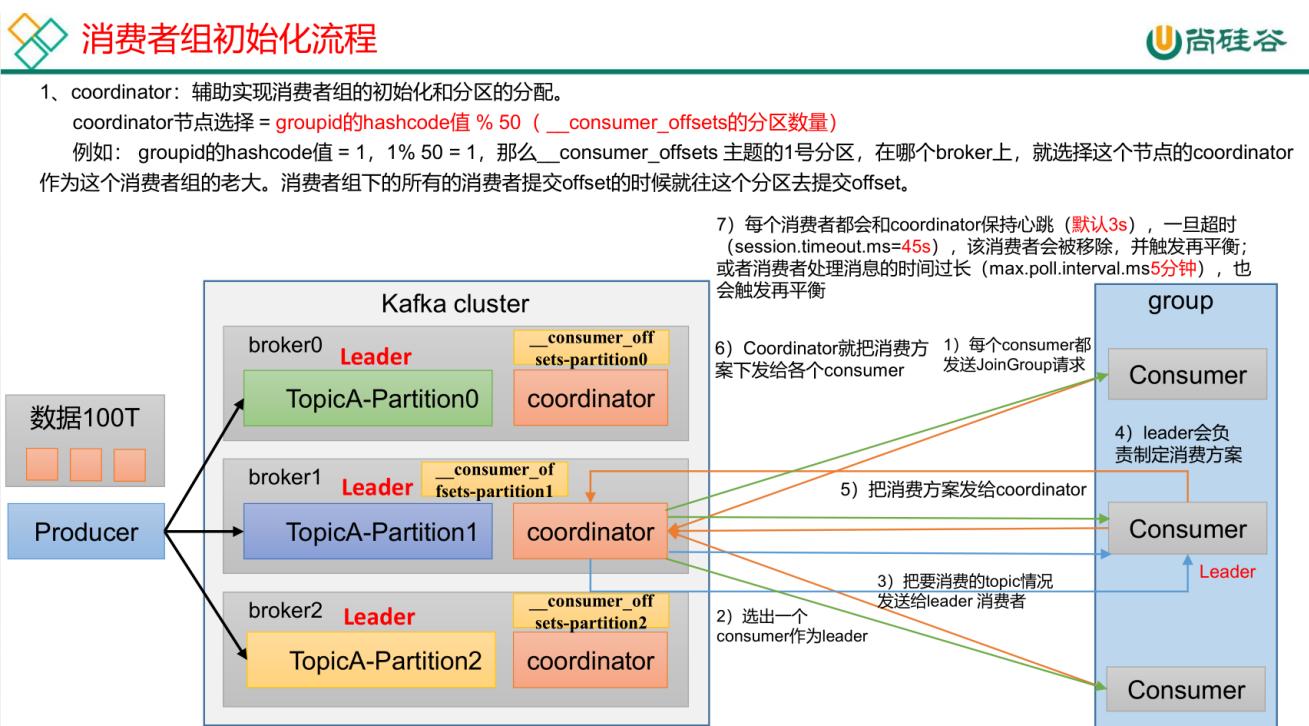
1. Topic是逻辑概念，partition是物理概念，每个分区对应一个log文件。
2. procedure生产出来的数据会被不断追加到log文件末端
3. kafka采用分片和索引机制，将每个partition分为多个segment，每个segment包含索引文件，数据文件和时间戳。
4. segment默认是1GB，index默认是4kb。所以采用的是稀疏索引。每当写入4KB的数据时，插入一个索引。
5. 数据默认保存时间为7天， 默认直接删除，页可以选择压缩保存

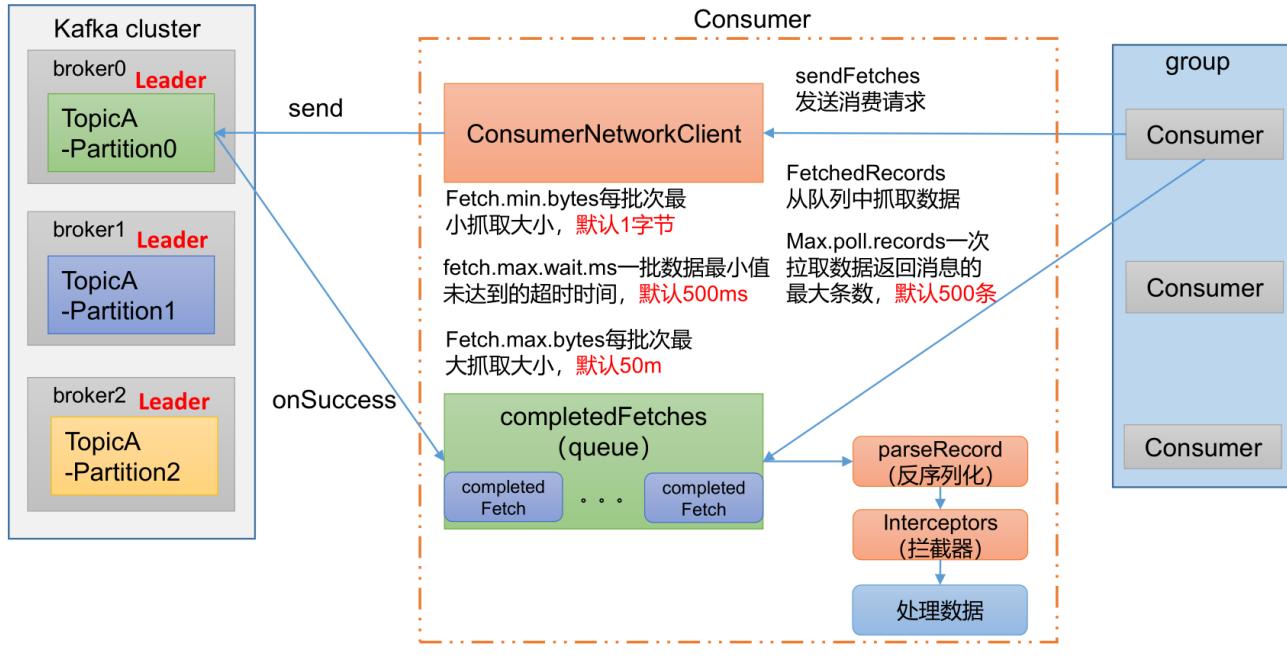
高效读写

kafka为什么运行那么快，为什么可以高效读写

- kafka本身是分布式集群，可以采用分区技术，并行度高
- 读数据采用稀疏索引，可以快速定位消费的数据
- 顺序写磁盘
- 页缓存和零拷贝技术

消费者





消费方式

kafka默认是消费者**主动拉取数据**的策略。因为如果是推送模式的话，如果每个消费者有不同的消费速率，很难决定推送的速度。

拉模式也有确定，如果broker集群没有数据，将一直空转。

消费者组

1. 由多个消费者组成，具备相同的groupid
2. 消费者组内每个消费者负责不同分区的数据，一个分区只能由一个消费者消费
3. 消费者组可以理解为一个独立的消费者，消费者组之间互不影响
4. 消费者组里消费者应该消费哪个分区的数据？主流的分区分配策略是：RoundRobin, Range, Sticky（粘性）CooperativeSticky（合作者粘性）。可以同时使用多个分区分配策略，默认是Range+CooperativeSticky

Offset

在0.9前存储在zookeeper中，0.9版本后为了建设过多的通信，放入到kafka内置的__consumer_offset主题中

- 默认存储在topic中
- 自动提交 5S
- 手动提交，同步或者异步
- 也可以按照offset或者时间消费
- 自动提交可能会出现，漏消费或者重复消费，可以采用事务来解决

数据积压

消费者如何提高吞吐量？

1. 可以适当增加分区数和消费者组中的消费者
2. 提高每批次拉取的数量

Tibco EMS

JMS

全称Java message Service， Java消息服务， 定义应用之间消息传递的Java框架规范。是接口规范，而不是具体的实现。

JMS组成：生产者，服务器和消费者。其中生产者和消费者统称为client

JMS模型：点对点和发布订阅

TIBCO

TIBCO公司的产品 TIBCO Enterprise Message Service也是JMS标准规范的实现，除了作为消息的中介，还提供了企业级的特性，如容错、消息路由等，同时还支持和TIBCO Rendezvous等产品组进行消息通信。

Tibco EMS创建消息时的增强包括：

- 错误恢复：备份消息到磁盘防止服务器宕机是丢失信息
- 权限控制：在队列或者主题上定义权限
- 消息路由：消息可以路由到其他服务器，也就是说服务器可以作为一个消费者接受其他server的消息
- 加密连接：client和服务器间可以设置SSL加密连接
- 容错、负载均衡、流量控制、高扩展和高可靠性

Gemfire

概念

Geode/Gemfire 是Pivotal公司开发的一款开源的、分布式NoSql内存数据库，可用来进行完成分布式缓存，数据持久化，分布式事务、动态扩展等功能。

参考博客：<https://www.php.cn/faq/562818.html>

术语

- 数据网格：是一个高扩展、分布式的内存缓存系统。
- 区域（Region）：网格中的区域，是数据管理中心。在gemfire中，每个区域对应一个缓存，每个缓存都提供了一个独立的数据区域，可以访问、查询和更新数据。区域可以视为多个缓存的组合。
- 节点：网格中一个实际运行的服务和集群中的一个成员。节点提供了可靠的通信机制，支持在整个网格中进行数据访问和存储。它还包含了一些核心的可配置的参数，例如IP地址和端口。

三种连接结构

P2P

将Gemfire嵌入到应用中充当缓存使用。

```
public static void main(String[] args) {
    // 创建Cache实例
    Cache cache = new CacheFactory().create();

    // 创建RegionShortcut并指定为PARTITION
    RegionShortcut shortcut = RegionShortcut.PARTITION;

    // 创建Region实例
    Region<Object, Object> region =
    cache.createRegionFactory(shortcut).create("exampleRegion");

    // 在Region中添加数据
    region.put("key1", "value1");
    region.put("key2", "value2");

    // 从Region中获取数据
    String value1 = (String) region.get("key1");
    String value2 = (String) region.get("key2");

    System.out.println(value1);
    System.out.println(value2);
}
```

C/S结构

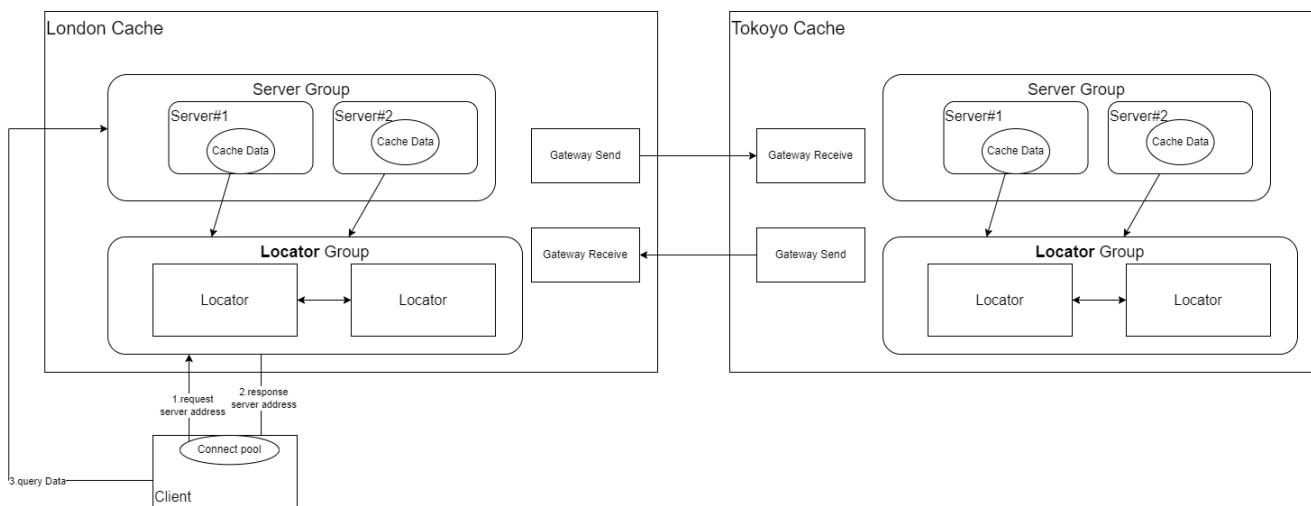
Gemfire的C/S结构是指客户端/服务器结构，它是一种分布式系统的基本架构。在这种结构中，客户端应用程序通过网络与Gemfire服务器进行通信，使用Gemfire提供的分布式缓存和数据管理功能。

Gemfire的C/S结构包括以下几个主要组件：

- 客户端（Client）**：客户端应用程序通过与Gemfire服务器建立连接，可以访问缓存中的数据。客户端应用程序可以使用Gemfire提供的API来操作缓存中的数据，例如put、get等操作。
- 服务器（Server）**：Gemfire服务器是分布式系统的中心组件，它提供了分布式缓存和数据管理功能。服务器将数据存储在内存中，并可以与其他的服务器进行数据同步。客户端通过与服务器建立连接，可以访问和操作缓存中的数据。
- 守护进程（Daemon）**：守护进程是Gemfire服务器的一种特殊形式，它是一个长时间运行的进程，可以接收客户端的连接请求并处理它们。守护进程还可以在后台执行一些任务，例如统计系统信息、管理日志等。
- 通信协议（Protocol）**：客户端和服务器之间的通信协议是用于传输数据和命令的标准协议。Gemfire支持多种通信协议，例如GF可靠协议和TCP套接字协议等。
- 数据存储（Data Store）**：Gemfire可以将数据存储在磁盘上，作为内存数据的补充。数据存储可以提供持久化的数据存储功能，并且可以与服务器进行数据同步。

通过这种C/S结构，Gemfire可以实现分布式缓存和数据管理，提高系统的性能和可靠性。客户端和服务器之间的通信可以通过网络进行传输，因此可以支持大规模的分布式系统。

多地 / 多数据中心WAN部署



WAN 交互是通过网络复制进行的。每个数据中心都有自己的 Gemfire 集群，数据中心之间通过 WAN 连接。当一个数据中心中的数据更新时，该数据中心的 Gemfire 集群会通过网络复制将这些更新发送到其他数据中心的 Gemfire 集群。

这种网络复制的方式可以确保数据的一致性和可靠性，同时还可以提高系统的可扩展性和容错性。如果一个数据中心发生故障，其他数据中心的 Gemfire 集群仍然可以提供服务，并且数据也可以通过网络复制进行恢复。

需要注意的是，Gemfire 的多数据中心 WAN 交互可能会对系统的性能和网络带宽产生一定的影响。因此，在设计和配置 Gemfire 的多数据中心 WAN 交互时，需要考虑到这些因素，并采取适当的优化措施来确保系统的性能和稳定性。

区别

和Redis相比的区别：

1. 数据存储：Redis将数据存储在内存中，可以通过持久化机制将数据定期写入磁盘，但是磁盘IO会影响性能；而GemFire可以将数据存储在内存中也可以存储在磁盘中，可以通过缓存数据到磁盘来避免内存不足的问题。
2. 数据模型：Redis使用键值存储模型，支持丰富的数据类型，如字符串、列表、哈希表、集合和有序集合等；而GemFire使用内存对象模型，可以存储Java对象、JSON对象、XML文档等。
3. 数据分布：Redis使用一致性哈希算法将数据分布在多个节点上，每个节点负责部分数据的存储和查询；而GemFire支持多种分布策略，如哈希分区、范围分区、复制和备份等。
4. 事务支持：Redis支持简单的事务，可以将多个操作封装在一个事务中，但是不支持复杂的事务和回滚操作；而GemFire支持分布式事务，可以在多个节点上执行复杂的事务，并支持回滚操作。
5. 应用场景：Redis适合存储小型数据，如缓存、会话数据、计数器、排行榜等；而GemFire适合存储大型数据和复杂对象，如金融交易数据、传感器数据、分布式会话等。

	Redis	Gemfire
数据存储	只能在内存中，有持久化机制	既可以存储在内存中，也可以存储在磁盘中
数据模型	string,list,set,hashmap,zset	Java对象，Json对象，XML文档等
数据分布	使用哈希算法计算要存储的节点	哈希，范围分区、复制、备份
事务支持	只支持简单的事务，不支持回滚	支持分布式事务，支持回滚

	Redis	Gemfire
应用场景	小型数据，例如缓存，排行榜等	大型数据和复杂对象

Mysql

博客：<https://dhc.pythonanywhere.com/article/public/1/>

B站：<https://www.bilibili.com/video/BV1Kr4y17ru>

Github: https://github.com/Buildings-Lei/mysql_note/blob/main/README.md#%E9%94%81

事务

四大特性ACID

- 原子性(Atomicity)：事务是不可分割的最小操作单元，要么全部成功，要么全部失败
- 一致性(Consistency)：事务完成时，必须使所有数据都保持一致状态
- 隔离性(Isolation)：数据库系统提供的隔离机制，保证事务在不受外部并发操作影响的独立环境下运行
- 持久性(Durability)：事务一旦提交或回滚，它对数据库中的数据的改变就是永久的

事务问题

问题	描述
脏读	一个事务读到另一个事务还没提交的数据
不可重复读	一个事务先后读取同一条记录，但两次读取的数据不同
幻读	一个事务按照条件查询数据时，没有对应的数据行，但是再插入数据时，又发现这行数据已经存在

隔离级别	脏读	不可重复读	幻读
Read uncommitted (读未提交)	√	√	√
Read committed (读已提交)	✗	√	√
Repeatable Read(可重复读)	✗	✗	√
Serializable (序列化)	✗	✗	✗

可重复读

MySQL session demonstrating repeatable read isolation:

1. 开启事务
2. 会话1查询id=5的数据
3. 更新id=5的数据name='test'
4. 提交会话2
5. 查询会话1 id=5的数据，仍然是修改前的数据，说明可以重读
6. 更新会话1的数据为name = 'test2'
7. 查询会话1更新后的数据，更新成功
8. 提交事务
9. 会话1事务生效

Session 2 (right) shows an error due to transaction syntax.

序列化

MySQL session demonstrating serialization isolation:

1. 查看事务级别
2. 开启事务
3. 查询id=5的数据
4. 会话2更新数据时会一直阻塞，直到会话1提交或者回滚事务，才会继续执行，否则会一直等待直到超时
5. 会话1查询id=5的数据
6. 提交会话2
7. 执行超时
8. 重新执行
9. 会话1重新查询时仍然可以查询到数据
10. 提交会话1，会话2的更新操作会立刻执行成功

Session 2 (right) shows an error due to lock wait timeout.

Mysql默认是可重复读， Oracle默认为读已提交

查看事务隔离级别：

```
SELECT @@TRANSACTION_ISOLATION;
```

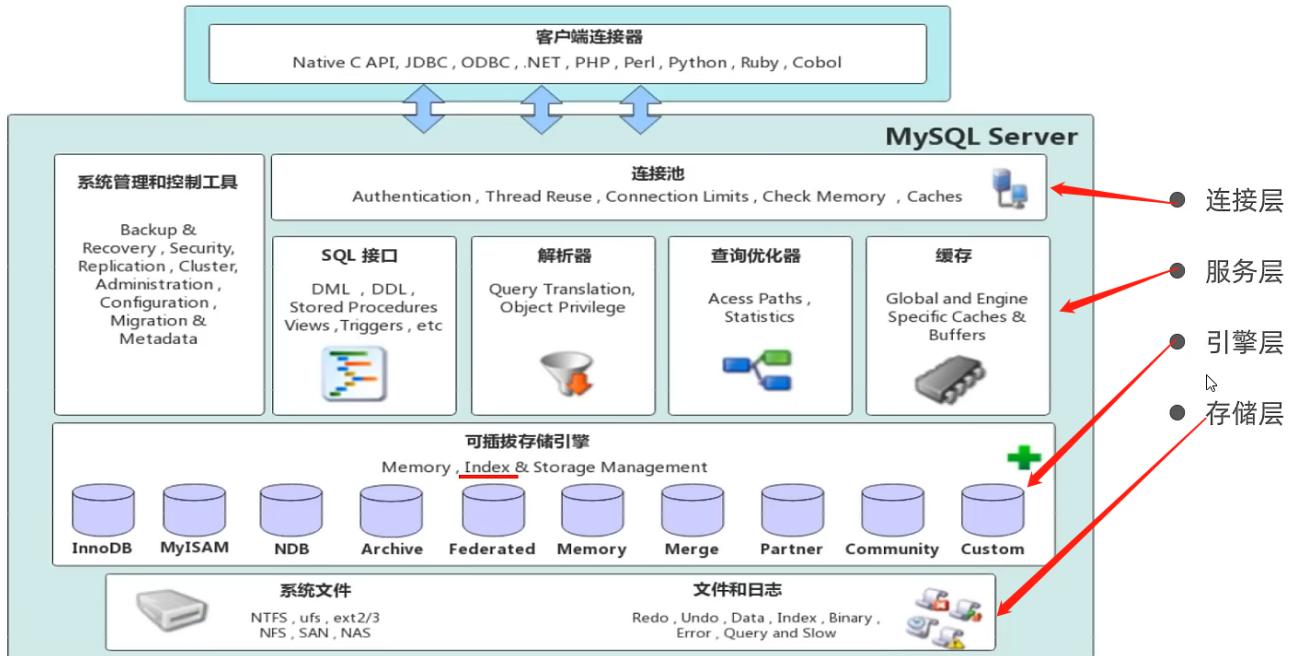
设置事务隔离级别：

```
SET [ SESSION | GLOBAL ] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED |
REPEATABLE READ | SERIALIZABLE };
```

SESSION 是会话级别，表示只针对当前会话有效， GLOBAL 表示对所有会话有效

存储引擎

MySQL体系结构:



存储引擎就是存储数据、建立索引、更新/查询数据等技术的实现方式。存储引擎是基于表而不是基于库的，所以存储引擎可以被称为表引擎。

```
# 查看表的存储引擎
show create table student;
# CREATE TABLE `student` (XXX) ENGINE=InnoDB

# 查看支持的所有的引擎
SHOW ENGINES;
```

InnoDB

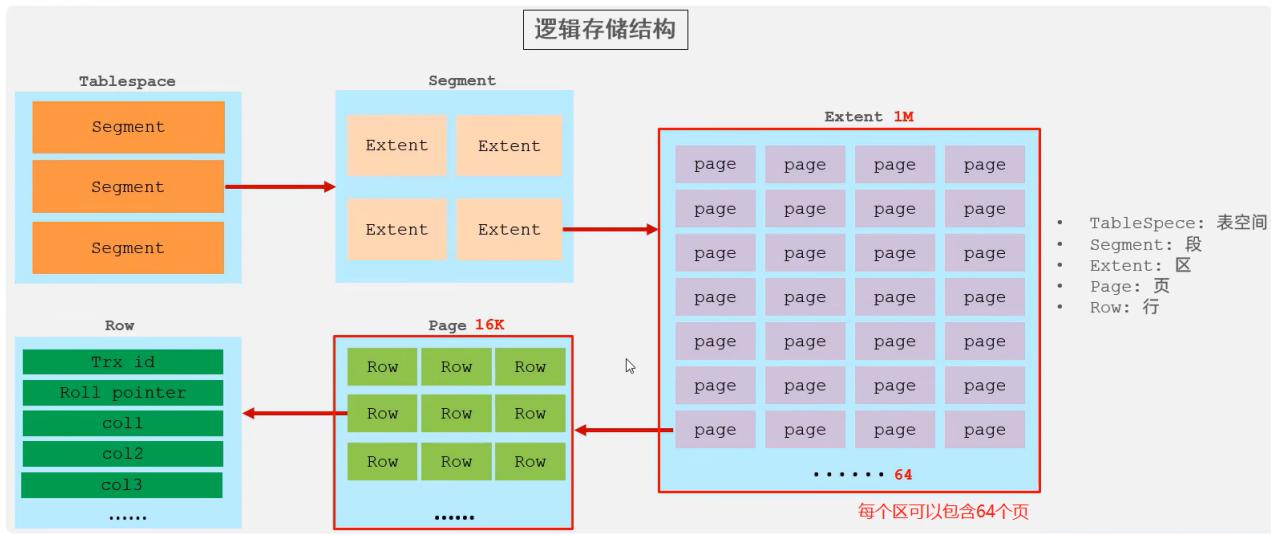
Mysql 5.5版本之后，默认存储引擎是InnoDB，有以下特点

- 支持事务
- 行级锁
- 支持外键约束

文件: xxx.ibd对应一个表空间，存储数据的表结构、数据和索引。使用ibd2sdi xxx,ibd可以查看idb文件

InnoDB逻辑存储结构

1. ● InnoDB



MyISAM

MyISAM 是 MySQL 早期的默认存储引擎。

特点：

- 不支持事务，不支持外键
- 支持表锁，不支持行锁
- 访问速度快

文件：

- xxx.sdi: 存储表结构信息
- xxx.MYD: 存储数据
- xxx.MYI: 存储索引

```
class_353.sdi  class.MYD  class.MYI  student.ibd
[root@hadoop1 test]# ll
total 144
-rw-r-----. 1 mysql mysql    4967 Jul 29 18:30 class_353.sdi → 表信息
-rw-r-----. 1 mysql mysql     108 Jul 29 18:30 class.MYD → 数据
-rw-r-----. 1 mysql mysql   3072 Jul 29 18:30 class.MYI → 索引
-rw-r-----. 1 mysql mysql 131072 Jul 30 22:56 student.ibd
[root@hadoop1 test]#
```

Memory

Memory引擎的数据存储在内存中，断电会丢失数据，只能作为临时表或者缓存使用

内部使用的是内存存放，基于hash索引，速度较快

只存储sdi文件，也就是只存储表结构

```

class_353.sdi class.MYD class.MYI part_358.sdi student.tbd
[root@hadoop1 test]# ls -altr
total 156
-rw-r----. 1 mysql mysql    3072 Jul 29 18:30 class.MYI
-rw-r----. 1 mysql mysql     108 Jul 29 18:30 class.MYD
-rw-r----. 1 mysql mysql   4967 Jul 29 18:30 class_353.sdi
-rw-r----. 1 mysql mysql 131072 Jul 30 22:56 student.ibd
drwxr-xr-x. 7 mysql mysql   4096 Jul 31 21:32 .
-rw-r----. 1 mysql mysql   4825 Jul 31 21:46 part_358.sdi 表结构信息
drwxr-x---. 2 mysql mysql    100 Jul 31 21:45 .
[root@hadoop1 test]#

```

区别

其中InnoDB和MyISAM的三大区别：

- InnoDB支持事务， MyISAM不支持
- InnoDB是行级锁， MyISAM是表级锁
- InnoDB支持外键， MyISAM不支持

Engine	Support	Comment	Transactions	XA	Savepoints
FEDERATED	NO	Federated MySQL storage engine	(Null)	(Null)	(Null)
MEMORY	YES	Hash based, stored in memory, useful for temporary tables	NO	NO	NO
InnoDB	DEFAULT	Supports transactions, row-level locking, and foreign keys	YES	YES	YES
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
MRG_MyISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
BLACKHOLE	YES	/dev/null storage engine (anything you write to it disappears)	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
ARCHIVE	YES	Archive storage engine	NO	NO	NO

特点	InnoDB	MyISAM	Memory
存储限制	64TB	有	有
事务安全	支持	-	-
锁机制	行锁	表锁	表锁
B+tree索引	支持	支持	支持
Hash索引	-	-	支持
全文索引	支持 (5.6版本之后)	支持	-
空间使用	高	低	N/A
内存使用	高	低	中等
批量插入速度	低	高	高
支持外键	支持	-	-

存储引擎的选择

- InnoDB: 是默认的存储引擎，支持事务、外键。如果系统对事务的完整性要求比较高，在并发情况下进行增删改查操作，InnoDB是唯一的选择。
- MyISAM: 由于不支持事务操作，适合以读和写为主，很少的删除和更新的场景。例如评论，足迹，日志等。
- Memory: 将所有数据保存在内存中，访问速度快，可以作为临时表或者缓存使用。

索引

概述

索引是一种提高查找效率的一种数据结构。

索引是在存储引擎层实现的，不同的存储引擎有不同的结构。

优点：

- 提高查找效率
- 可以通过索引对数据进行排序
- 方便范围查找

缺点：

- 索引页会占用磁盘空间（但是可以几乎忽略不计）
- 增删改操作时，需要维护索引表，降低更新速度

索引结构

主要的索引结构为：B+树索引和哈希索引

索引结构	描述
B+Tree	最常见的索引类型，大部分引擎都支持B+树索引
Hash	底层数据结构是用哈希表实现，只有精确匹配索引列的查询才有效，不支持范围查询
R-Tree(空间索引)	空间索引是 MyISAM 引擎的一个特殊索引类型，主要用于地理空间数据类型，通常使用较少
Full-Text(全文索引)	是一种通过建立倒排索引，快速匹配文档的方式，类似于 Lucene, Solr, ES

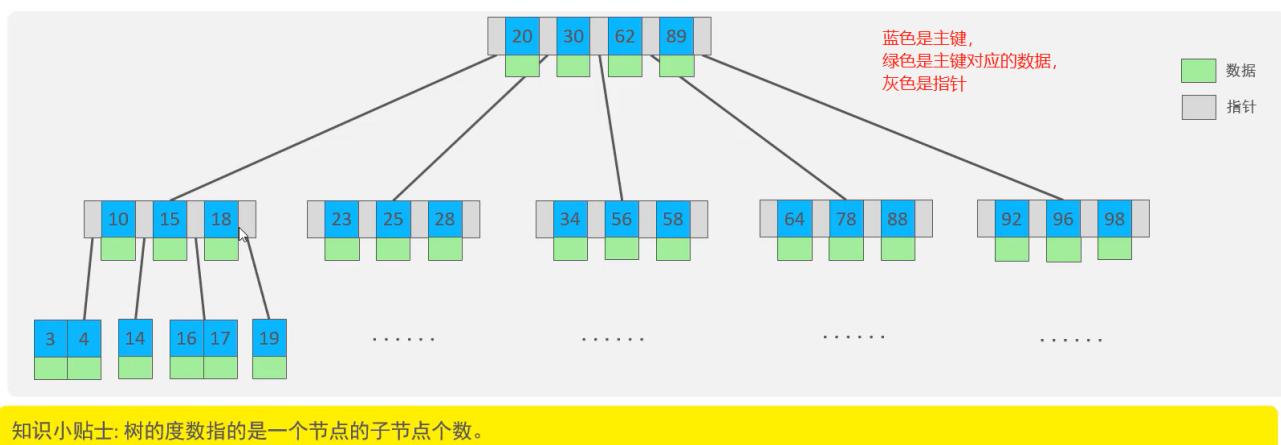
索引	InnoDB	MyISAM	Memory
B+Tree索引	支持	支持	支持
Hash索引	不支持	不支持	支持

索引	InnoDB	MyISAM	Memory
R-Tree索引	不支持	支持	不支持
Full-text	5.6版本后支持	支持	不支持

B树

一个m阶B数，一个节点最多存储m-1个key，该节点的出度为m，代表m个指针。

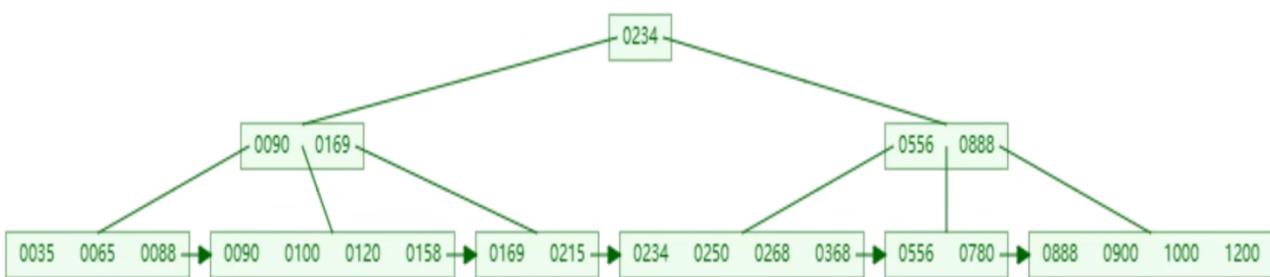
可视化页面：<https://www.cs.usfca.edu/~galles/visualization/BTree.html>



B+树

- B+Tree

插入 100 65 169 368 900 556 780 35 215 1200 234 888 158 90 1000 88 120 268 250 数据为例。



相对于B-Tree区别：

- ①. 所有的数据都会出现在叶子节点 ↓
- ②. 叶子节点形成一个单向链表

Mysql的索引结构对B+树做了优化，在原有的B+树基础上增加一个指向前面页的指针，从而在页的级别上形成了双向循环链表。

Hash索引

用哈希算法计算对象的位置，并获取到对应的数据。如果遇到了hash冲突，可以使用链表来解决。

特点

- 只能用于对等比较，不支持范围查询

- 无法利用索引来完成排序操作
- 查找效率高，一般情况下一次检索就可以获取到数据，效率要高于B+树索引。

存储引擎

在MySQL中，Memory引擎支持hash索引，而且InnoDB中也有自适应hash功能，hash索引是存储引擎根据B+树索引在指定条件下自动构建的。

面试题

为什么InnoDB存储引擎选择B+树作为索引结构？

1. 如果使用**二叉树**，一个节点下最多挂两个子节点。相同数据下，层级太深，查询效率低
2. 如果使用**B树**，因为InnoDB是聚集索引，索引和数据是在一起的，如果非叶子节点既存储索引又存储数据，那么一页中存储的索引数将大大减少，最终导致层级同样很深。相反，如果使用B+树，非叶子节点只存储索引的话，比B树要多很多。
3. 如果采用hash索引，不支持范围匹配和排序操作。

索引的分类

索引类型

分类	含义	特点	关键字
主键索引	针对表中主键创建的索引	默认创建，只能有一个	PRIMARY
唯一索引	避免同一个表中的某个列值重复，当加唯一约束会自动创建该索引	可以有多个	UNIQUE
常规索引	快速定位特定的数据	可以有多个	
全文索引	查找的是文本中的关键词，不是比较索引中的值	可以有多个	FULLTEXT

聚集和非聚集索引

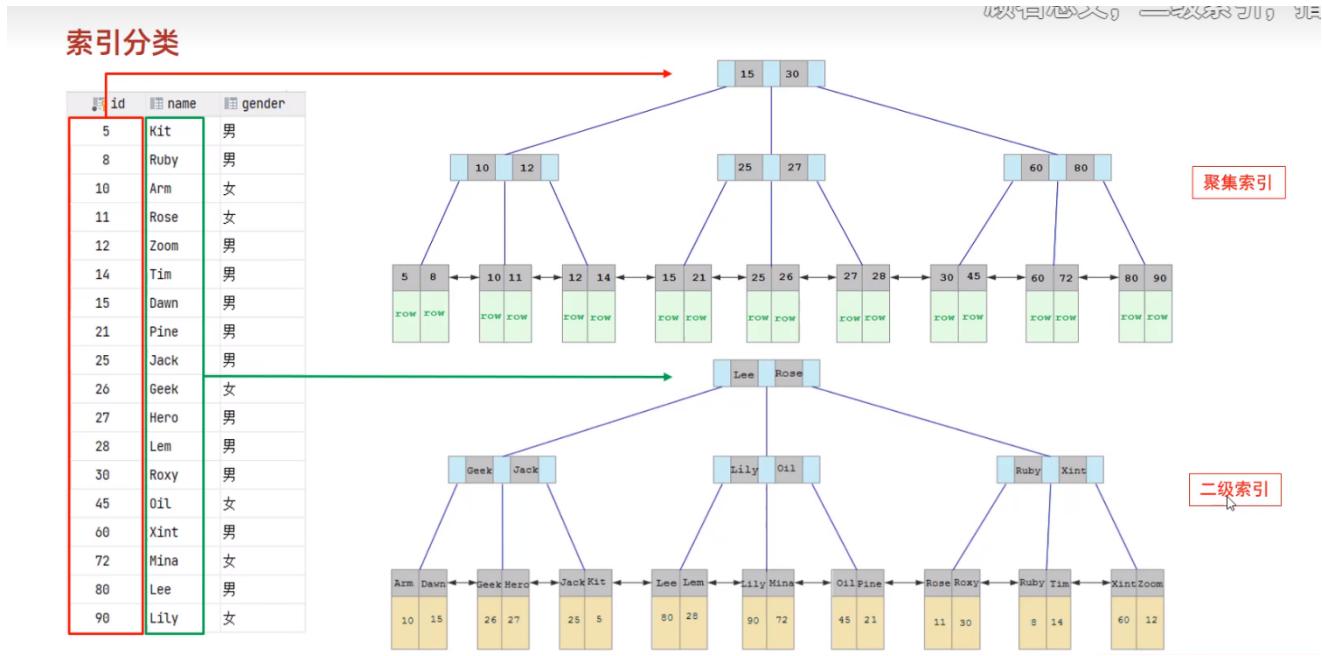
根据存储的存储形式，分为两种

分类	含义	特点
聚集索引 (Clustered Index)	将数据存储在索引和索引放在一起，索引结构的也只节点保存了行数据	有且只有一个
二级索引 (Secondary Index)	又叫辅助索引或者非聚集索引，叶子节点关联的是对应的主键，需要回表查询	可以存着多个

聚集索引默认是主键索引。如果没有设置主键，将依照下面的选举规则

- 如果有主键，主键索引就是聚集索引
- 如果没有主键，将使用第一个唯一 (UNIQUE) 索引作为聚集索引

- 如果没有主键，也没有唯一索引，InnoDB会自动生成一个rowid作为隐藏的聚集索引。



B+树数据量

回顾：正如上面所说，一个区是1MB，一页是16kb，一个区包含64个页。

假设：

- 一行数据的平均大小为1kb，那么一页可以包含 $16\text{kb}/1\text{kb}=16$ 条数据。
- 使用bigInt作为主键占用8b，指针占用6b。则一页可以存储 $16\text{kb}/(8\text{b}+6\text{b})=1171$ 个索引。
- 如果是三层B+树，则一层和二层可以存储 $1171 * 1171 = 1,371,241$ 条索引。
- 三层是数据层，一页存储16条数据。索引可以存储 $16 * 1171 * 1171=21,939,856$ 条数据。

总之：对于平均大小为1kb，使用bigint为主键的情况下，三层B+树可以存储两千多万条数据，数据量约为20GB。

索引语法

- 创建索引：create [unique | fulltext] index index_name on table (index_col_name,...)
- 查看索引：show index from table_name
- 删除索引：drop index index_name on table_name

使用索引

最左前缀法则（联合索引）

如果创建的是联合索引，要遵循最左前缀法则。

例如创建(A,B,C)的联合索引，则当where条件中出现 (A)(A,B)(A,B,C)三种情况时才会使用索引。

```

mysql> explain select * from sys_config where config_key = 'sys.index.skinName' and config_value = 'skin-blue' and config_type='Y';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | sys_config | NULL | ref | idx_key_value_type | 2411 | const,const,const | const,const,const |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> explain select * from sys_config where config_key = 'sys.index.skinName' and config_value = 'skin-blue';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | sys_config | NULL | ref | idx_key_value_type | 2406 | const,const | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> explain select * from sys_config where config_key = 'sys.index.skinName';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | sys_config | NULL | ref | idx_key_value_type | 403 | const | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql>

```

- 当where条件中没有索引第一列时，索引失效；

-- 没有第一列，索引失效
WHERE B=xx and C=xx;

```

mysql> explain select * from sys_config where config_type='Y';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | sys_config | NULL | ALL | NULL | NULL | NULL | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

```

- 当where条件中缺少索引中的一个，则缺少后的索引失效。

-- A索引起作用，C没有用
WHERE A=xx abd C=xx

```

mysql> explain select * from sys_config where config_key = 'sys.index.skinName' and config_type='Y';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | sys_config | NULL | ref | idx_key_value_type | 403 | const | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

```

- where条件中的列是顺序不必和创建索引时的顺序相同，只要出现即可。

-- 仍然会走联合索引。因为会对下面的条件优化
WHERE C=XX AND B=XX AND A=XX`同样有效

```

explain select * from sys_config where config_type='Y' and config_value = 'skin-blue' and config_key = 'sys.index.skinName';

Where C=xx and B=xx and A=xx仍然会走全部索引，因为底层会做优化为A,B,C

```

总之，以上三种情况均会导致联合索引失效问题。原理是因为底层B+树联合索引是使用A-B-C作为key，如果缺少

索引失效

- 在索引列上进行运算操作，索引将失效。如：explain select * from tb_user where substring(phone, 10, 2) = '15'；
- 字符串类型字段使用时，不加引号，索引将失效。如：explain select * from tb_user where phone = 17799990015；，此处phone的值没有加引号
- 模糊查询中，如果仅仅是尾部模糊匹配，索引不会失效；如果是左模糊匹配，索引失效。如：explain select * from tb_user where profession like '%工程'；全模糊也会失效。阿里巴巴规范如下：

4. 【强制】页面搜索严禁左模糊或者全模糊，如果需要请走搜索引擎来解决。

说明：索引文件具有 B-Tree 的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索引。

- 用 or 分割开的条件，如果 or 其中一个条件的列没有索引，那么涉及的索引都不会被用到。

5. 如果 MySQL 评估使用索引比全表更慢，则不使用索引。

SQL提示

是优化数据库的一个重要手段，简单来说，就是在SQL语句中加入一些人为的提示来达到优化操作的目的。

例如，建议使用指定索引，实际使用哪个索引 MySQL 还会自己权衡运行速度去更改

```
explain select * from tb_user use index(idx_user_pro) where profession="软件工程";
```

不使用哪个索引：

```
explain select * from tb_user ignore index(idx_user_pro) where profession="软件工程";
```

必须使用哪个索引，force就是无论如何都强制使用该索引

```
explain select * from tb_user force index(idx_user_pro) where profession="软件工程";
```

覆盖索引&回表查询

覆盖索引：当使用索引查询，需要返回的字段只包含索引中的字段和id时，则为覆盖索引，避免了回表查询。

```
-- 联合索引是(A,B,C)  
select id,B,C from table where A=xx;
```

如果是覆盖索引，extra字段里是Using where; Using index;

如果没有使用覆盖索引，extra字段里是Using index condition

前缀索引

如果字段类型为字符串或者大文本时，如果创建索引会很占用空间，查询时会有大量的磁盘IO。这时候可以考虑使用前缀索引。

```
create index idx_xxxx on table_name(columnn(n));
```

前缀长度：可以根据索引的选择性来决定，而选择性是指不重复的索引值（基数）和数据表的记录总数的比值，索引选择性越高则查询效率越高，唯一索引的选择性是1，这是最好的索引选择性，性能也是最好的。

求选择性公式：

```
-- 查询email字段的非重复率  
select count(distinct email) / count(*) from tb_user;  
-- 查询email前五个字段的非重复率，非重复率越高，查询效率越好  
select count(distinct substring(email, 1, 5)) / count(*) from tb_user;
```

show index 里面的sub_part字段可以看到截取的长度。

单列索引&联合索引

单列索引：即一个索引只包含单个列

联合索引：即一个索引包含了多个列

在业务场景中，如果存在多个查询条件，考虑针对于查询字段建立索引时，建议建立联合索引，而非单列索引。

如果同一个字段既有联合索引又有单列索引。MySQL优化器会评估哪个字段的索引效率更高，会选择该索引完成本次查询

设计原则

1. 当**数据量比较大**，**查询频繁**的表建立索引
2. 经常作为**where条件，排序，分组**操作字段需要建立索引。
3. 选择**区分度高**的列作为索引，区分度越高，使用索引效率越高（反例：性别，标志位），尽量建立唯一索引。
4. 如果字符串长度较长，可以创建**前缀索引**
5. 尽量创建**联合索引**，减少单列索引，联合索引很多时候可以覆盖索引，避免回表
6. 要控制**索引的数量**，索引并不是多多益善，索引越多维护索引结构的代价就越大，影响增删改的效率
7. 如果索引不能存储NULL值，使用NOT NULL**约束它**。mysql优化器会更好的使用索引。

性能分析

查看执行次数

```
show [global|session] status like 'Com_____';
```

	Variable_name	Value
1	Com_binlog	0
2	Com_commit	0
3	Com_delete	0
4	Com_import	0
5	Com_insert	242
6	Com_repair	0
7	Com_revoke	0
8	Com_select	1679
9	Com_signal	0
10	Com_update	0
11	Com_xa_end	0

慢查询日志

查询是否开启慢查询日志记录

```
show variables like 'slow_query_log';
```

开启慢查询并且设置慢查询时间为2秒，在/etc/my.cnf文件中追加下面代码

```
# 开启慢查询日志开关
slow_query_log=1
# 设置慢查询日志的时间为2秒，SQL语句执行时间超过2秒，就会视为慢查询，记录慢查询日志
long_query_time=2
```

当执行到慢查询时，会默认在\$MYSQL_HOME/data/host-name-slow.log中追加查询。

```
/opt/module/mysql/bin/mysqld, Version: 8.0.17 (MySQL Community Server - GPL). started with:
Tcp port: 0 Unix socket: (null)
Time           Id  Command   Argument
# Time: 2023-08-03T12:22:43.737321Z
# User@Host: root[root] @ [192.168.56.1]  Id:      9
# Query_time: 7.250119  Lock_time: 0.000000  Rows_sent: 1  Rows_examined: 0
use test;
SET timestamp=1691065356;
/* ApplicationName=IntelliJ IDEA 2023.1.4 */ select benchmark(1000000000,1*2);
# Time: 2023-08-03T12:25:09.700676Z
# User@Host: root[root] @ [192.168.56.1]  Id:      9
# Query_time: 25.520087  Lock_time: 0.000000  Rows_sent: 1  Rows_examined: 0
SET timestamp=1691065484;
/* ApplicationName=IntelliJ IDEA 2023.1.4 */ select benchmark(3500000000,1*2);
# Time: 2023-08-03T12:25:59.550726Z
# User@Host: root[root] @ [192.168.56.1]  Id:      9
# Query_time: 7.277640  Lock_time: 0.000000  Rows_sent: 1  Rows_examined: 0
SET timestamp=1691065552;
/* ApplicationName=IntelliJ IDEA 2023.1.4 */ select benchmark(1000000000,99*99);
hadoop1-slow.log (END)
```

SQL性能分析

```
-- 查看是否支持监控
select @@have_profiling
-- 查看是否开启监控
select @@profiling
show variables like 'profiling'
-- 打开监控
set profiling=1;
-- 查询所有的执行语句耗时
show profiles;
-- 查询指定query的具体耗时
show profile for query #{query_id}
-- 查询CPU占用情况
show profile cpu for query #{query_id}
```

Explain

Explain是mysql的关键字，用来显示SQL查询的执行计划，更好的查看SQL查询的瓶颈。

Explain并不会执行SQL，只是分析SQL的执行计划。

具体使用：在查询语句前加上explain关键字即可。

```
explain select * from sys_config where config_value='123456';
explain insert into class(class_name) value ('一年级150班');
```

Explain的各个字段的含义：

- **id**: select 查询的序列号，表示查询中执行 select 子句或者操作表的顺序 (id相同，执行顺序从上到下； id不同，值越大越先执行)
- **select_type**: 表示 SELECT 的类型，常见取值有 SIMPLE (简单表，即不适用表连接或者子查询) 、 PRIMARY (主查询，即外层的查询) 、 UNION (UNION 中的第二个或者后面的查询语句) 、 SUBQUERY (SELECT/WHERE之后包含了子查询) 等
- **type**: 表示连接类型，性能由好到差的连接类型为 NULL、 system、 const、 eq_ref、 ref、 range、 index、 all
 - 1. null: 一般是没有表查询时： explain select 1+1 from dual;
 - 2. system: 一般是一个表里只有一条数据时。

```
delete from class where id!=1;
explain select * from test.class where id=1;
```

3. const: 使用聚集索引，也就是主键索引查询时，对应的级别。

```
explain select * from class where id=1;
```

4. eq_ref: 使用普通索引时的查询级别：

```
create index idx_class_name on class(class_name);
explain select * from class where class_name='test';
```

5. range: 当查询in、 between、 like左匹配时，使用该级别

```
explain select * from class where id in (1,2,3,4,5);
explain select * from class where id between 1 and 10;
```

6. index: 查询全部索引，一般常见于order by，阿里巴巴规范中约定要避免该级别。

8. 【推荐】SQL 性能优化的目标：至少要达到 range 级别，要求是 ref 级别，如果可以是 consts 最好。

说明：

1) consts 单表中最多只有一个匹配行 (主键或者唯一索引)，在优化阶段即可读取到数据。

2) ref 指的是使用普通的索引 (normal index)。

3) range 对索引进行范围检索。

反例： explain 表的结果， type=index，索引物理文件全扫描，速度非常慢，这个 index 级别比较 range 还低，与全表扫描是小巫见大巫。

```
explain select config_id from sys_config order by config_key;
```

7. All: 查询全表，一般用于查询全表，或者where条件没有索引，或者左模糊全模糊匹配。

- possible_key: 可能应用在这张表上的索引，一个或多个
- Key: 实际使用的索引，如果为 NULL，则没有使用索引
- Key_len: 表示索引中使用的字节数，该值为索引字段最大可能长度，并非实际使用长度，在不损失精确性的前提下，长度越短越好
- rows: MySQL认为必须要执行的行数，在InnoDB引擎的表中，是一个估计值，可能并不总是准确的
- filtered: 表示返回结果的行数占需读取行数的百分比，filtered的值越大越好

SQL优化

批量插入

- 一次插入不要超过1000条
- 手动提交事务，因为自动提交事务每次执行sql都会自动提交，增加资源开销。
- 主键顺序插入

对于百万级别数据的大批量插入，可以使用load指令，将文件load到数据库，效率提升10倍以上。

主键优化

页分裂：当插入一行数据到指定页时，如果页内的数据无法满足插入数据的大小要求，将会发生页分裂，会将要插入的页中的一半数据和需要插入的数据统一插入到新的页中，然后改变链表指针。

页合并：当删除一定量的数据使得页中的数据量达到阈值时，InnoDB会寻找前后两页是否可以合并，从而优化空间。

合并阈值：MERGE_THRESHOLD

主键设计原则

- 满足业务需求的情况下，尽量降低主键的长度
- 插入数据时，尽量选择顺序插入，选择使用 AUTO_INCREMENT 自增主键
- 尽量不要使用 UUID 做主键或者是其他的自然主键，如身份证号
- 业务操作时，避免对主键的修改

Order by

1. Using filesort：通过表的索引或全表扫描，读取满足条件的数据行，然后在排序缓冲区 sort buffer 中完成排序操作，所有不是通过索引直接返回排序结果的排序都叫 FileSort 排序
2. Using index：通过有序索引顺序扫描直接返回有序数据，这种情况即为 using index，不需要额外排序，操作效率高

如果order by字段全部使用升序排序或者降序排序，则都会走索引，但是如果一个字段升序排序，另一个字段降序排序，则不会走索引，explain的extra信息显示的是Using index, Using filesort，如果要优化掉Using filesort，则需要另外再创建一个索引，如：

```
create index idx_user_age_phone_ad on tb_user(age asc, phone desc);
```

此时使用select id, age, phone from tb_user order by age asc, phone desc;会全部走索引

总结：

- 根据排序字段建立合适的索引，多字段排序时，也遵循最左前缀法则
- 尽量使用覆盖索引
- 多字段排序，一个升序一个降序，此时需要注意联合索引在创建时的规则（ASC/DESC）
- 如果不可避免出现filesort，大数据量排序时，可以适当增大排序缓冲区大小sort_buffer_size（默认256k）

Group by

- 在分组操作时，可以通过索引来提高效率
- 分组操作时，索引的使用也是满足最左前缀法则的

如索引为idx_user_pro_age_stat，则句式可以是select ... where profession order by age，这样也符合最左前缀法则

Limit优化

常见的问题如limit 2000000, 10，此时需要MySQL排序前2000000条记录，但仅仅返回2000000 - 2000010的记录，其他记录丢弃，查询排序的代价非常大。

优化方案：一般分页查询时，通过创建覆盖索引能够比较好地提高性能，可以通过覆盖索引加子查询形式进行优化

例如：

```
-- 此语句耗时很长
select * from tb_sku limit 9000000, 10;
-- 通过覆盖索引加快速度，直接通过主键索引进行排序及查询
select id from tb_sku order by id limit 9000000, 10;
-- 下面的语句是错误的，因为 MySQL 不支持 in 里面使用 limit--
select * from tb_sku where id in (select id from tb_sku order by id limit 9000000, 10);
-- 通过连表查询即可实现第一句的效果，并且能达到第二句的速度
select * from tb_sku as s, (select id from tb_sku order by id limit 9000000, 10) as a where s.id
= a.id;
```

Count优化

MyISAM 引擎把一个表的总行数存在了磁盘上，因此执行 count() 的时候会直接返回这个数，效率很高（前提是不适用 where）；

InnoDB 在执行 count() 时，需要把数据一行一行地从引擎里面读出来，然后累计计数。

优化方案：自己计数，如创建key-value表存储在内存或硬盘，或者是用redis

count的几种用法：

- 如果count函数的参数（count里面写的那个字段）不是NULL（字段值不为NULL），累计值就加一，最后返回累计值
- 用法：count(*)、count(主键)、count(字段)、count(1)
- count(主键)跟count()一样，因为主键不能为空；count(字段)只计算字段值不为NULL的行；count(1)引擎会为每行添加一个1，然后就count这个1，返回结果也跟count()一样；count(null)返回0

各种用法的性能：

- count(主键)：InnoDB引擎会遍历整张表，把每行的主键id值都取出来，返回给服务层，服务层拿到主键后，直接按行进行累加（主键不可能为空）
- count(字段)：没有not null约束的话，InnoDB引擎会遍历整张表把每一行的字段值都取出来，返回给服务层，服务层判断是否为null，不为null，计数累加；有not null约束的话，InnoDB引擎会遍历整张表把每一行的字段值都取出来，返回给服务层，直接按行进行累加
- count(1)：InnoDB引擎遍历整张表，但不取值。服务层对于返回的每一层，放一个数字1进去，直接按行进行累加
- count(*)：InnoDB引擎并不会把全部字段取出来，而是专门做了优化，不取值，服务层直接按行进行累加

按效率排序：count(字段) < count(主键) < count(1) < count()，所以尽量使用 count()

Update优化

针对索引字段进行更新，避免行锁升级为表锁

InnoDB 的行锁是针对索引加的锁，不是针对记录加的锁，并且该索引不能失效，否则会从行锁升级为表锁。

如以下两条语句：

```
-- 这句由于id有主键索引，所以只会锁这一行
update student set no = '123' where id = 1;
-- 这句由于name没有索引，所以会把整张表都锁住进行数据更新，解决方法是给name字段添加索引
update student set no = '123' where name = 'test';
```

总结



1. 插入数据
`insert: 批量插入、手动控制事务、主键顺序插入`
`批量插入: load data local infile`
2. 主键优化
`主键长度尽量短、顺序插入 AUTO_INCREMENT UUID`
3. order by优化
`using index: 直接通过索引返回数据，性能高`
`using filesort: 需要将返回的结果在排序缓冲区排序`
4. group by优化
`索引，多字段分组满足最左前缀法则`
5. limit优化
`覆盖索引 + 子查询`
6. count优化
`性能: count(字段) < count(主键 id) < count(1) ≈ count(*)`
7. update优化
`尽量根据主键/索引字段进行数据更新`

锁

概述

锁是计算机协调多个进程或线程并发访问某一资源的机制。在数据库中，除传统的计算资源（CPU、RAM、I/O）的争用以外，数据也是一种供许多用户共享的资源。如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。从这个角度来说，锁对数据库而言显得尤其重要，也更加复杂。

NOTE : 针对事物才有加锁的意义。

分类：MySQL中的锁，按照锁的粒度分，分为以下三类：

1. 全局锁：锁定数据库中的所有表。
2. 表级锁：每次操作锁住整张表。
3. 行级锁：每次操作锁住对应的行数据。

全局锁

全局锁就是对整个数据库实例加锁，加锁后整个实例就处于只读状态，后续的DML的写语句，DDL语句，已经更新操作的事务提交语句都将被阻塞。

其典型的使用场景是做**全库的逻辑备份**，对所有的表进行锁定，从而获取一致性视图，保证数据的完整性。

```

-- 1.开启全局锁（执行SQL）
flush tables with read lock;
-- 2.在${MYSQL_HOME}/bin下执行备份
./mysqldump -uroot -p123456 test > test_backup.sql
-- 3.解锁
unlock tables;

```

存着两个问题：

- 将会锁库，如果只有一个数据库，那么备份期间，业务基本全部停止。
- 如果是主从结构，对从库备份，将无法及时更新来自主库同步的二进制文件（binlog），导致同步延迟。

表级锁

表级锁，每次操作锁住整张表。锁定粒度大，发生锁冲突的概率最高，并发度最低。应用在MyISAM、InnoDB、BDB等存储引擎中。

主要分为三类：

1. 表锁：对于表锁，分为两类：

1. 表共享读锁（read lock）所有的事物都只能读（当前加锁的客户端也只能读，不能写），不能写
 2. 表独占写锁（write lock），对当前加锁的客户端，可读可写，对于其他的客户端，不可读也不可写。
 3. 读锁不会阻塞其他客户端的读，但是会阻塞写。写锁既会阻塞其他客户端的读，又会阻塞其他客户端的写。
2. 元数据锁（meta data lock，MDL），MDL加锁过程是系统自动控制，无需显式使用，在访问一张表的时候会自动加上。MDL锁主要作用是维护表元数据的数据一致性，在表上有活动事务的时候，不可以对元数据进行写入操作。在MySQL5.5中引入了MDL，当对一张表进行增删改查的时候，加MDL读锁（共享）；当对表结构进行变更操作的时候，加MDL写锁（排他）。

3. 意向锁：为了避免DML在执行时，加的行锁与表锁的冲突，在InnoDB中引入了意向锁，使得表锁不用检查每行数据是否加锁，使用意向锁来减少表锁的检查。

一个客户端对某一行加上了行锁，那么系统也会对其加上一个意向锁，当别的客户端来想要对其加上表锁时，便会检查意向锁是否兼容，若是不兼容，便会阻塞直到意向锁释放。

意向锁兼容性：

1. 意向共享锁（IS）：与表锁共享锁（read）兼容，与表锁排它锁（write）互斥。
2. 意向排他锁（IX）：与表锁共享锁（read）及排它锁（write）都互斥。意向锁之间不会互斥。

行级锁

行级锁，每次操作锁住对应的行数据。锁定粒度最小，发生锁冲突的概率最低，并发度最高。应用在InnoDB存储引擎中。

InnoDB的数据是基于索引组织的，行锁是通过对索引上的索引项加锁来实现的，而不是对记录加的锁。对于行级锁，主要分为以下三类：

1. 行锁（Record Lock）：锁定单个行记录的锁，防止其他事务对此行进行update和delete。在RC（read commit）、RR（repeat read）隔离级别下都支持。

- 间隙锁 (GapLock) : 锁定索引记录间隙 (不含该记录) , 确保索引记录间隙不变, 防止其他事务在这个间隙进行insert, 产生幻读。在RR隔离级别下都支持。比如说 两个临近叶子节点为 15 23, 那么间隙就是指 [15, 23], 锁的是这个间隙。
- 临键锁 (Next-Key Lock) : 行锁和间隙锁组合, 同时锁住数据, 并锁住数据前面的间隙Gap。在RR隔离级别下支持。

InnoDB实现了以下两种类型的行锁:

- 共享锁 (S) : 允许一个事务去读一行, 阻止其他事务获得相同数据集的排它锁。
- 排他锁 (X) : 允许获取排他锁的事务更新数据, 阻止其他事务获得相同数据集的共享锁和排他锁。

行锁 - 演示

默认情况下, InnoDB在REPEATABLE READ事务隔离级别运行, InnoDB使用next-key 锁进行搜索和索引扫描, 以防
止幻读。

- 针对唯一索引进行检索时, 对已存在的记录进行等值匹配时, 将会自动优化为行锁。
- InnoDB的行锁是针对于索引加的锁, 不通过索引条件检索数据, 那么InnoDB将对表中的所有记录加锁, 此时就
会升级为表锁。

间隙锁/临键锁-演示

默认情况下, InnoDB在REPEATABLE READ事务隔离级别运行, InnoDB使用next-key 锁进行搜索和索引扫描, 以防
止幻读。

- 索引上的等值查询 (唯一索引) , 给不存在的记录加锁时, 优化为间隙锁。
- 索引上的等值查询 (普通索引) , 向右遍历时最后一个值不满足查询需求时, next-key lock 退化为间隙锁。
- 索引上的范围查询 (唯一索引) --会访问到不满足条件的第一个值为止。

注意: 间隙锁唯一目的是防止其他事务插入间隙。间隙锁可以共存, 一个事务采用的间隙锁不会阻止另一个事务在
同一间隙上采用间隙锁。

总结



1. 概述

- 在并发访问时, 解决数据访问的一致性、有效性问题
- 全局锁、表级锁、行级锁

2. 全局锁

- 对整个数据库实例加锁, 加锁后整个实例就处于只读状态
- 性能较差, 数据逻辑备份时使用

3. 表级锁

- 操作锁住整张表, 锁定粒度大, 发生锁冲突的概率高
- 表锁、元数据锁、意向锁

4. 行级锁

- 操作锁住对应的行数据, 锁定粒度最小, 发生锁冲突的概率最低
- 行锁、间隙锁、临键锁

逻辑存储结构

表空间（ibd文件），一个mysql实例可以对应多个表空间，用于存储记录、索引等数据。

段，分为数据段（Leaf node segment）、索引段（Non-leaf node segment）、回滚段（Rollback segment），InnoDB是索引组织表，数据段就是B+树的叶子节点，索引段即为B+树的非叶子节点。段用来管理多个Extent（区）。

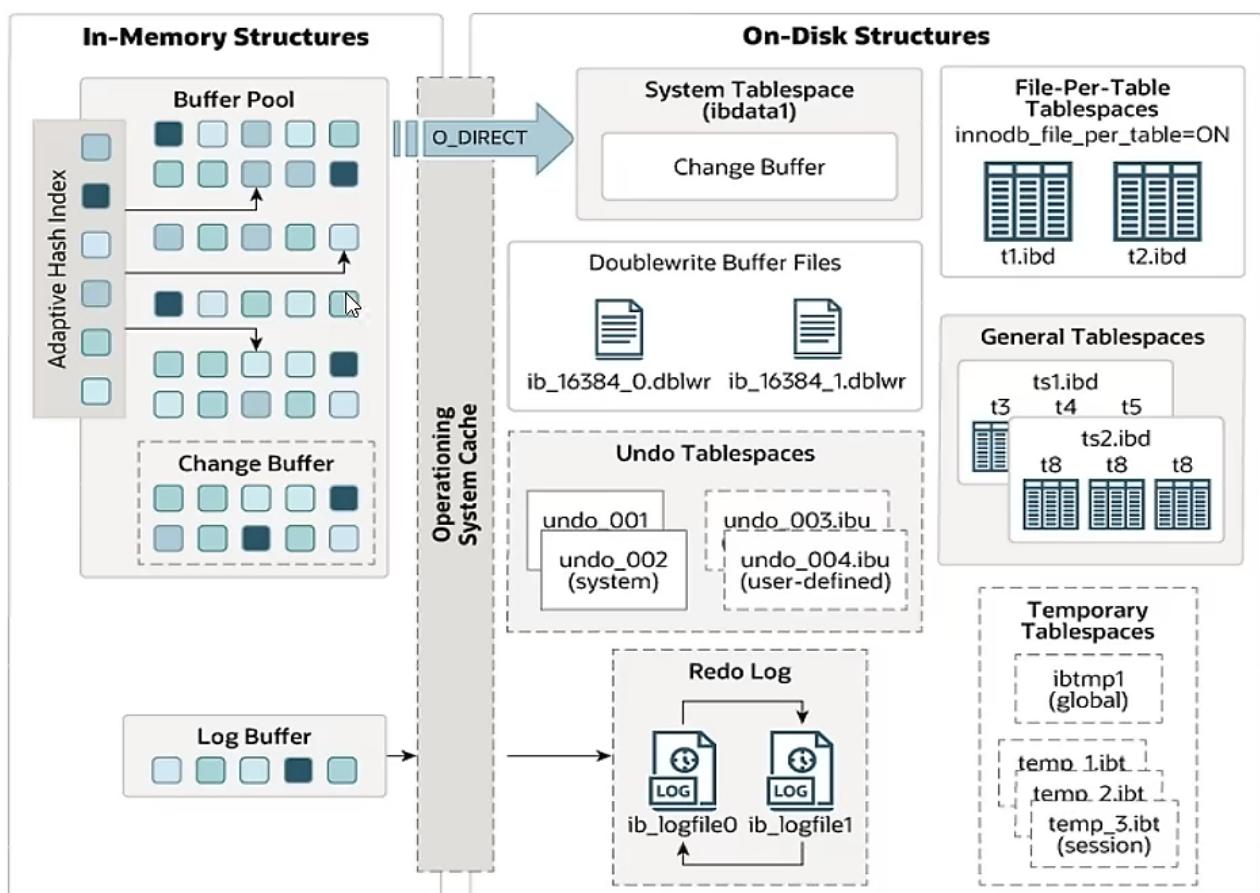
区，表空间的单元结构，每个区的大小为1M。默认情况下，InnoDB存储引擎页大小为16K，即一个区中一共有64个连续的页。

页，是InnoDB存储引擎磁盘管理的最小单元，每个页的大小默认为16KB。为了保证页的连续性，InnoDB存储引擎每从磁盘申请4-5个页。一页包含若干行。

行，InnoDB存储引擎数据是按行进行存放的。

架构

包含内存架构、磁盘架构和同步线程三部分



内存架构

1. 缓冲池：包含三部分，空闲页，被使用的页和数据被修改过的脏页。
2. 更改缓冲区：对非聚集索引进行合并后刷入缓冲池中，减少磁盘IO
3. 自适应哈希索引：优化数据查询，无需人工干预，系统根据情况自动完成。参数adaptive_hash_index
4. 日志缓冲区：包含 redo log 和 undo log。默认是 16MB。两个参数innodb_log_buffer_size 缓冲区大小和 innodb_flush_log_at_trx_commit 日志刷新到磁盘时机。

磁盘架构

1. 系统表空间：如果没有对应表空间文件，将会存储在该区域内。
2. 独立表空间：每一个表对应的表空间文件，默认开启。
3. 通用表空间：在创建表结构的时候，手动指定关联的表空间。
4. 撤销表空间：自动创建的两个默认的undo表空间，默认大小为16MB，用于存储undo log日志。
5. 临时表空间：用来存储用户创建的临时表数据
6. 双写缓冲区：在缓冲池刷入磁盘前，会先将数据页写入到双重缓冲区文件中，方便系统异常时恢复数据。文件为ibd_xxx_0 dblwr
7. 重做日志（Redo Log）：用来实现事务的持久性。日志文件包含两部分，重写日志缓冲和重做日志文件，前者在内存中，后者在磁盘中。当事务提交后会吧所有修改新存到该日志中，用来刷新脏页到错误发生时，进行数据恢复使用。ib_logfile0和ib_logfile1

同步数据线程

主要分为以下四中线程：

1. Master线程：核心后台线程，辅助调度其他线程，负责将缓冲池中的数据异步刷新到磁盘中，保持数据的一致性。还包括脏页刷新，合并插入缓存，undo页的回收。
2. IO线程：大量使用了AIO来处理请求，包括四个读线程，四个写线程、一个日志线程和一个缓冲区线程刷新到磁盘。
3. Purge线程：回收提交完的事务的undo log。
4. Page cleaner线程：协助Master线程刷新脏页到磁盘，减少阻塞。

```
I/O thread 0 state: waiting for completed aio requests (insert buffer thread)
I/O thread 1 state: waiting for completed aio requests (log thread)
I/O thread 2 state: waiting for completed aio requests (read thread)
I/O thread 3 state: waiting for completed aio requests (read thread)
I/O thread 4 state: waiting for completed aio requests (read thread)
I/O thread 5 state: waiting for completed aio requests (read thread)
I/O thread 6 state: waiting for completed aio requests (write thread)
I/O thread 7 state: waiting for completed aio requests (write thread)
I/O thread 8 state: waiting for completed aio requests (write thread)
I/O thread 9 state: waiting for completed aio requests (write thread)
```

事务原理

事务的原子性、一致性和持久性是通过redo log和undo log来实现的，隔离性是通过锁和MVCC来实现的。

- 持久性是根据redo log（提交日志）来实现的，redo log是用来在刷新内存脏页到磁盘发生错误时，进行数据恢复使用。

- 原子性根据undo log（回滚日志）来实现的，undo log内容记录的是逻辑日志，可以理解为记录的是与操作相反的DML语句。例如，事务中执行insert那么undo log记录delete，事务中执行update，那么undo log中记录与之相反的记录。当执行rollback的时候就可以读取该内容并回滚。
- 一致性是由undo log和redo log实现的。
- 隔离性是由锁加MVCC实现

MVCC

全称Multi-Version Concurrency Control，多版本并发控制。指维护一个数据的多个版本，使得读写操作没有冲突，快照读为MySQL实现MVCC提供了一个非阻塞读功能。

当前读

读取的是记录的最新版本，读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁。对于我们日常的操作，如：

- select...lock in share mode（共享锁）。
- select....for update、update、insert、delete（排他锁）都是一种当前读。

快照读

简单的select（不加锁）就是快照读，快照读，读取的是记录数据的可见版本，有可能是历史数据，不加锁，是非阻塞读。

- Read Committed：每次select，都生成一个快照读。
- Repeatable Read：开启事务后第一个select语句才是快照读的地方。
- Serializable：快照读会退化为当前读。

具体实现

MVCC的具体实现，还需要依赖于数据库记录中的**三个隐式字段**、**undo log日志**、**readView**。

三个隐式字段：undo log, readView

Mysql管理

系统数据库

1. mysql：存储mysql服务器正常运行所需要的各种信息，例如时区，主从，用户、权限等。
2. information_schema：提供了访问数据库元数据的各种表和视图，包含数据库、表、字段类型以及访问权限等。
3. performance_schema：为Mysql服务器运行时状态提供了一个底层监控功能，主要用于收集数据库服务器性能参数。
4. sys：包含了一系列方便DBA和开发人员利用performance_schema性能数据库进行性能调优和诊断的视图。

常用命令

mysql

- -h:host
- -P:port
- -u:user
- -p:password
- -e:execute

mysql可以直接执行语句，不用登录

```
mysql -uroot -p123456 -e "select * from xxx";
```

mysqladmin

用来管理mysql，可以用来创建数据库等操作。也可以用来检查服务器的配置和当前状态、创建并删除数据库等。

一般在脚本中使用

```
-----  
bind-address          (No default value)  
count                0  
force               FALSE  
compress             FALSE  
character-sets-dir   (No default value)  
default-character-set auto  
host                 (No default value)  
no-beep              FALSE  
port                 0  
relative             FALSE  
socket               (No default value)  
sleep                0  
ssl-ca               (No default value)  
ssl-capath            (No default value)  
ssl-cert              (No default value)  
ssl-cipher             (No default value)  
ssl-key               (No default value)  
ssl-crl               (No default value)  
ssl-crlpath            (No default value)  
tls-version            (No default value)  
tls-ciphersuites      (No default value)  
server-public-key-path (No default value)  
get-server-public-key FALSE  
user                 (No default value)  
verbose              FALSE  
vertical              FALSE  
connect-timeout       43200  
shutdown-timeout      3600  
plugin-dir            (No default value)  
default-auth           (No default value)  
enable-cleartext-plugin FALSE  
show-warnings         FALSE
```

mysqlbinlog

由于服务器生成的二进制日志文件是以二进制格式保存，因此如果想要检查这些文本的具体内容，则需要使用到该命令。

语法：mysqlbinlog [optional] log-file1 log-file2

mysqlshow

用来很快地查找存在哪些数据库、数据库中的表、表中的列或者索引。

语法：mysqlshow [options] [db_name[table-name[col_name]]]

```
[root@hadoop1 data]# ./bin/mysqlshow -u root -p123456 --count
mysqlshow: [Warning] Using a password on the command line interface can be insecure.
+-----+-----+-----+
| Databases | Tables | Total Rows |
+-----+-----+-----+
| information_schema | 67 | 24251 |
| mysql | 33 | 3368 |
| performance_schema | 103 | 475283 |
| sys | 101 | 5647 |
| test | 45 | 256 |
+-----+-----+-----+
5 rows in set.
[root@hadoop1 data]#
```

mysqlDump

mysqldump客户端工具用来备份数据库或者在不同数据库之间进行数据迁移。备份内容包含创建表，以及插入表的SQL语句。

语法：mysqldump [options] db_name [tables]

```
mysqldump -uroot -p1234 test > test.sql
```

mysqlimport/source

mysqlimport是客户端数据导入工具用来导入mysqldump加-T参数导出的文本文件

语法：mysqlimport [options] db_name textfile [textfile]'

如果需要导入sql文件，可以使用mysql的source指令

```
source /xxx/xxx.sql
```

Shell

B站：<https://www.bilibili.com/video/BV1st411N7WS>

小工具

Grep

grep是行过滤工具；用于根据关键字进行行过滤

语法：# grep [选项] '关键字' 文件名

OPTIONS:

- i: 不区分**大小写**(ignore)
- v: 查找不包含指定内容的行,反向选择(invert-match)
- c: 统计匹配到的行数
- n: 显示行号
- r: 递归逐层遍历目录查找
- A: 显示匹配行及后面多少行(After)
- B: 显示匹配行及前面多少行(Before)
- C: 显示匹配行前后多少行(Context)
- l: 只列出匹配的文件名
- e: 使用**正则**匹配
- ^\$: 匹配空行

举例：

查找/root/logs文件夹下所有文件包含SpringApplicationShutdownHook字符串的文件名

```
grep -rl SpringApplicationShutdownHook /root/logs
```

Cut

cut是列截取工具，用于列的截取

语法：

```
# cut 选项 文件名
```

常见选项：

- c: 以字符为单位进行分割,截取
- d: 自定义分隔符，默认为制表符\t
- f: 与-d一起使用，指定截取哪个区域

举例说明：

# cut -d: -f1 1.txt	以:冒号分割，截取第1列内容
# cut -d: -f1,6,7 1.txt	以:冒号分割，截取第1,6,7列内容
# cut -c4 1.txt	截取文件中每行第4个字符
# cut -c1-4 1.txt	截取文件中每行的1-4个字符
# cut -c4-10 1.txt	截取文件中每行的4-10个字符
# cut -c5- 1.txt	从第5个字符开始截取后面所有字符

Sort

sort工具用于排序;它将文件的每一行作为一个单位,从首字符向后,依次按ASCII码值进行比较,最后将他们按升序输出。

语法和选项

```
-u : 去除重复行  
-r : 降序排列, 默认是升序  
-o : 将排序结果输出到文件中,类似重定向符号>  
-n : 以数字排序, 默认是按字符排序  
-t : 分隔符  
-k : 以第N列作为排序条件  
-b : 忽略前导空格。  
-R : 随机排序, 每次运行的结果均不同
```

举例说明

```
# sort -n -t: -k3 1.txt          按照用户的uid进行升序排列  
# sort -nr -t: -k3 1.txt         按照用户的uid进行降序排列  
# sort -n 2.txt                  按照数字排序  
# sort -nu 2.txt                 按照数字排序并且去重  
# sort -nr 2.txt  
# sort -nru 2.txt  
# sort -nru 2.txt  
# sort -n 2.txt -o 3.txt         按照数字排序并将结果重定向到文件  
# sort -R 2.txt  
# sort -u 2.txt
```

Uniq

用来去除连续的重复行

常见选项:
-i: 忽略大小写
-c: 统计重复行次数
-d: 只显示重复行

举例说明:
uniq 2.txt
uniq -d 2.txt
uniq -dc 2.txt

Tee

将标准输入读取并写入到标准输出和文件

语法: 屏幕输出 | 文本输入

```
echo hello world | tee test.txt
```

Diff

比较文件的不同

diff描述两个文件的不同方式。也就是说搞事我们怎样改变第一个文件之后与第二个文件匹配。

语法： diff [选项] 文件1 文件2

选项

- -b: 不检查空格
- -B: 不检查空白行
- -i: 不检查大小写
- -w: 忽略所有的空格
- --normal: 正常格式显示 (默认)
- -c: 上下文格式显示
- -u: 合并格式显示

前言

bash中的引号

- 双引号：会转义，例如echo "\$(hostname)"，打印主机名
- 单引号：不会转移，例如echo '\$(hostname)'，打印\$(hostname)
- 反撇号`：等价于\$(), 但是内部不能再嵌套

```
[root@MissHou dir1]# echo $(echo `date +%F`)  
2018-11-22  
[root@MissHou dir1]# echo `echo `date +%F``  
date +%F  
-- 第一个反撇号起作用了，但是内部的反撇号没有起作用
```

\$符号

- 如果\$内部包含的是命令，使用括号。例如pwd,ls,grep等，例如\$(hostname)
- 如果\$内部包含的是变量，使用大括号。例如系统变量PATH，echo \${PATH}；自定义变量A,B,C,D，echo \${A}，当然也可以省略大括号但是不能省略\$，echo \$A

变量

概念

定义：

- 变量名=变量值
- 可以将命令的执行结果值赋给变量。例如file_list=`ls`
- 也可以读取用户输入作为变量read [选项] 变量名

使用：echo \$变量名 或者echo \${变量名}

规则：

- 变量区分大小写
- 定义变量时**等号两边不能有空格**
- 使用变量时，也可以只使用变量的值的一部分，例如：echo \${A:2:4}，表示从A变量的第二个字符开始截取，截取四个字符。

变量类型

- 用户变量：一般定义的变量都是临时变量。可以使用export升级为临时环境变量，升级的变量只对该进程有用。
- 环境变量：定义在/etc/profile
- 全局变量：所有用户都能调用的变量，例如HOME变量
- 系统变量：也叫内置bash的变量。
 - \$?:上一条命令的执行状态。0表示正常结束。非0表示异常
 - \$0: 当前执行的shell脚本名
 - \$#: 执行脚本的参数个数
 - \$*: 脚本的所有参数，是个整体
 - \$@: 脚本的所有参数，是独立的
 - \$1~9: 脚本后的第几个参数
 - \${10}-\${n}: 脚本后的第几个参数
 - \$\$: 当前进程号

数组

定义：

- 数组名[下标]=值 例如arr[5]=5
- 一次赋值，数组名=(v1 v2 v3 ...)。例如：arr=(cat /etc/passwd)

读取：

- \${array[0]}: 获取第一个元素

- \${array[*]}: 获取所有元素
- \${#array[*]}: 获取所有元素个数
- \${!array[@]}: 获取元素的下标
- \${array[@]:1:2}: 访问从第一个元素, 取两个元素

流程控制

条件判断

语法

- 格式1: test 条件表达式
- 格式2: [条件表达式]
- 格式3: [[条件表达式]] 支持正则 =~,

单个 [] 使用字符串对字符串必须加双引号
 两个 [[]] 不用对字符串变量加双引号
 两个 [[]] 里面可以使用 &&, ||, 而单个不行

注意上面的空格

参数

判断参数	含义	说明
-e	文件是否 存在 (link文件指向的也必须存在)	exists
-f/d/L	文件是否存在并且是一个 普通文件/目录/软连接	file
-s	文件是否存在并且是一个 非空文件 (有内容)	is not empty
-r/w/x	当前用户是否可读/写/执行文件	
-eq/nq/gt/lt/ge/lt	等于/不等于/大于/小于/大于等于/小于等于	
-z/n	字符串是否 为空/非空	
-a/o	多个判断判断条件的逻辑与/或	

If...Else

```

if [ condition ];then
    command1
else
    command2
fi

[ 条件 ] && command1 || command2

```

Case

```
case var in
  pattern 1)
    command1
    ;;
  pattern 2)
    command2
    ;;
  pattern 3)
    command3
    ;;
  *)
    default, 不满足以上模式, 默认执行*)下面的语句
  command4
  ;;
esac
```

定义变量;var代表是变量名
模式1;用 | 分割多个模式, 相当于or
需要执行的语句
两个分号代表命令结束
*)
esac表示case语句结束

循环

For

```
for variable in {list}
do
  command
  command
...
done
```

While

```
while 表达式
do
  command...
done
```

例子:

```
#!/bin/env bash

i=1
while [ $i -le 5 ]
do
  echo $i
  let i++
done
```

函数

语法

```
函数名()
{
    函数体（一堆命令的集合，来实现某个功能）
}

function 函数名()
{
    函数体（一堆命令的集合，来实现某个功能）
}
```

例子

```
#!/bin/bash

# 定义一个函数，返回一个数值
function add_numbers() {
    local a=$1
    local b=$2
    local sum=$((a + b))
    return $sum
}

# 调用函数，并将结果存储在变量中
result=$(add_numbers 2 3)

# 打印结果
echo "函数的返回值为: $result" # 输出: 函数的返回值为: 5
```

正则表达式

字符

元字符: 在正则中，具有特殊意义的专用字符，如: 星号(*)、加号(+)等

前导字符: 元字符前面的字符叫前导字符

元字符	功能	示例
*	前导字符出现0次或者连续多次	ab* abbbb
.	除了换行符以外，任意单个字符	ab. ab8 abu
.*	任意长度的字符	ab.* adfdfdf
[]	括号里的任意单个字符或一组单个字符	[abc][0-9][a-z]

元字符	功能	示例
[^]	不匹配括号里的任意单个字符或一组单个字符	[^abc]
^[]	匹配以括号里的任意单个字符开头	[4]
^[^]	不匹配以括号里的任意单个字符开头	
^	行的开头	^root
\$	行的结尾	bash\$
^\$	空行	
{n}和{n}	前导字符连续出现n次	[0-9]{3}
{n,}和{n,}	前导字符至少出现n次	[a-z]{4,}
{n,m}和{n,m}	前导字符连续出现n-m次	go{2,4}
<>	精确匹配单词	<hello>
(保留匹配到的字符	(hello)
+	前导字符出现1次或者多次 (最少一个)	[0-9]+
?	前导字符出现0次或者1次 (==最多一个==)	go?
	或	^root ^ftp
()	组字符	(hello world)123
\d	perl内置正则	grep -P \d+
\w	匹配字母数字下划线	

使用

1. 要找什么？数字or字母or标点？
2. 如何找？以什么开头？结尾？包含什么？不包含什么？
3. 找多少？0-n次？任意一次？出现几次？

例子

```
# 1、查找不以大写字母开头的行（三种写法）。
grep '^[^A-Z]' 2.txt
```

```
# 2、查找有数字的行（两种写法）
grep '[0-9]' 2.txt
grep -P '\d' 2.txt
```

```

# 3、查找一个数字和一个字母连起来的
grep -E '[0-9][a-zA-Z]|[a-zA-Z][0-9]' 2.txt

# 4、查找不以r开头的行,第一个^指的是以什么什么开头, 第二个^指的是非
grep '^[^r]' 2.txt

# 5、查找以数字开头的
grep '^[0-9]' 2.txt

# 6、查找以大写字母开头的
grep '^[A-Z]' 2.txt

# 7、查找以小写字母开头的
grep '^[a-z]' 2.txt

# 8、查找以点结束的, \是转移, 表示.这个符号, 而不是正则表达式中的元符号
grep '\.$' 2.txt

# 9、去掉空行
grep -v '^$' 2.txt

# 10、查找完全匹配abc的行
grep '\<abc\>' 2.txt

# 11、查找A后有三个数字的行
grep -E 'A[0-9]{3}' 2.txt

# 12、查找ip地址
grep -o -E '([0-9]{1,3}\.){3}[0-9]{1,3}' 1.txt

# 16、找出全部是数字的行
grep -E '^[0-9]+$' test

```

sed

概念

sed是一行一行读取文件内容并按照要求进行处理，把处理后的结果输出到屏幕。

语法： sed [options] '处理动作' 文件名

■ 常用选项

选项	说明	备注
-e	进行多项(多次)编辑	
-n	取消默认输出	不自动打印模式空间
-r	使用扩展正则表达式	
-i	原地编辑 (修改源文件)	

选项	说明	备注
-f	指定sed脚本的文件名	

■ 常见处理动作

丑话说在前面：以下所有的动作都要在单引号里，你敢出轨，回家跪搓衣板

动作	说明	备注
'p'	打印	
'i'	在指定行之前插入内容	类似vim里的大写O
'a'	在指定行之后插入内容	类似vim里的小写o
'c'	替换指定行所有内容	
'd'	删除指定行	
's'	搜索并替换	

例子

打印(p)

[root@server ~]# sed -n '1p' a.txt	打印第1行
[root@server ~]# sed -n '1,5p' a.txt	打印1到5行
[root@server ~]# sed -n '\$p' a.txt	打印最后1行

增加内容(a/i)

[root@server ~]# sed '\$a99999' a.txt	文件最后一行下面增加内容99999
[root@server ~]# sed 'a99999' a.txt	文件每行下面增加内容
[root@server ~]# sed '5a99999' a.txt	文件第5行下面增加内容
[root@server ~]# sed '\$i99999' a.txt	文件最后一行上一行增加内容
[root@server ~]# sed 'i99999' a.txt	文件每行上一行增加内容
[root@server ~]# sed '6i99999' a.txt	文件第6行上一行增加内容
# /^uucp/指的是查询条件，	
# i指的是在查到的所有结果上面插入，	
# hello代表插入的内容	
[root@server ~]# sed '/^uucp/ihello'	以uucp开头行的上一行插入内容

修改内容(c)

[root@server ~]# sed '5chello world' a.txt	替换文件第5行内容为hello world
[root@server ~]# sed 'chello world' a.txt	替换文件所有内容
[root@server ~]# sed '1,5chello world' a.txt	替换文件1到5号内容为hello world
[root@server ~]# sed '/^user01/c888888' a.txt	替换以user01开头的行为888888

删除内容(d)

```
[root@server ~]# sed '1d' a.txt          删除文件第1行  
[root@server ~]# sed '1,5d' a.txt        删除文件1到5行  
[root@server ~]# sed '$d' a.txt         删除文件最后一行
```

查找并替换(s)

默认只替换一个，如果替换一行中的所有，需要加参数g

```
[root@server ~]# sed -n 's/root/ROOT/p' 1.txt  
  
# 默认只替换一个，如果替换行中的所有，需要加参数g  
[root@server ~]# sed -n 's/root/ROOT/gp' 1.txt  
[root@server ~]# sed -n 's/^#/gp' 1.txt  
  
# 注意：搜索替换中的分隔符/可以自己指定为@  
[root@server ~]# sed -n 's@/sbin/nologin@itcast@gp' a.txt  
[root@server ~]# sed -n '10s#/sbin/nologin#itcast#p' a.txt  
[root@server ~]# sed -n 's@/sbin/nologin@itcastheima@p' 2.txt  
  
# 注释掉文件的1-5行内容  
[root@server ~]# sed -n '1,5s/^/#/p' a.txt
```

awk

语法

语法：awk 选项 '命令部分' 文件名

选项：

- -F 定义字段分割符号，默认的分隔符是空格
- -v 定义变量并赋值

命令部分

- 正则表达式，地址定位

```
'/root/{awk语句}'           sed中: '/root/p'  
'NR==1,NR==5{awk语句}'     sed中: '1,5p'  
'/^root/,/^ftp/{awk语句}'  sed中: '/^root/,/^ftp/p'
```

- {awk语句1;awk语句2;...}

```
'{print $0;print $1}'      sed中: 'p'  
'NR==5{print $0}'         sed中: '5p'  
注: awk命令语句间用分号间隔
```

- BEGIN...END....

```
'BEGIN{awk语句};{处理中};END{awk语句}'  
'BEGIN{awk语句};{处理中}'  
'{处理中};END{awk语句}'
```

内部变量

变量	变量说明	备注
\$0	当前处理行的所有记录	
\$1,\$2,\$3...\$n	文件中每行以间隔符号分割的不同字段	awk -F: '{print \$1,\$3}'
NF	当前记录的字段数 (列数)	awk -F: '{print NF}'
\$NF	最后一列	\$(NF-1)表示倒数第二列
FNR/NR	行号	
FS	定义间隔符	'BEGIN {FS=":"};{print \$1,\$3}'
OFS	定义输出字段分隔符, 默认空格	'BEGIN {OFS="\t"};print \$1,\$3'
RS	输入记录分割符, 默认换行	'BEGIN {RS="\t"};{print \$0}'
ORS	输出记录分割符, 默认换行	'BEGIN {ORS="\n\n"};{print \$1,\$3}'
FILENAME	当前输入的文件名	

例子

```
# 1. 打印用:分割后的文件中的第一列, 倒数第二列和最后一列,  
awk -F: '{print $1,$(NF-1),$NF}' /etc/passwd  
  
# 2. 打印第1行到第五行, 并且以root或lp开头以:分割的行的第一列  
awk -F: 'NR==1,NR==5;/^root/,/^lp/{print $1}' a.txt  
  
# 3. 以:作为列分割条件;输出列之间用\t\t; 每行的输入分隔符是\n, 每行的输出分隔符为\n\n  
# 打印以root或者以lp开头的第1列和最后一列  
awk 'BEGIN{FS=":";OFS="\t\t";RS="\n";ORS="\n\n"};/^root/,/^lp/{print $1,$NF}' a.txt  
  
# 4. 格式化输入, 中间间隔分别是1,5,15个字符  
awk 'BEGIN{FS=":"};{printf "%-1s %-5s %-15s\n",$1,$2,$NF}' a.txt  
  
# 5. 定义变量num, 使用变量不用加$  
awk -v num=1 'BEGIN{print num}'  
  
# 6.BEGIN{};{};END{}语法  
awk 'BEGIN{ FS=":";print "Login_shell\tLogin_home\n*****\n*****"};  
{print $(NF-1),$(NF)}';
```

```
END{print "*****"}' a.txt  
  
# 7. 打印大于等于1, 小于等于10行的, 并且以bash结尾的行  
awk 'NR>=1 && NR<=10 && /bash$/' a.txt
```

Mybatis

基础

概念

是一个半自动ORM框架，封装了JDBC的操作，专注于SQL语句的编写，提高开发效率

优缺点

优点

- 基于SQL语句编程，开发相对灵活
- 封装了各种JDBC操作，和原生JDBC相比，减少了一半的代码量
- 兼容各种数据库
- 和Spring很好的兼容
- 提供很多灵活的标签，支持动态SQL

缺点

- 需要手动写SQL，需要一定的SQL基础
- SQL语句依赖具体的数据库，移植性较差

#{} 和 \${}

#{}是预编译，会调用PreparedStatement的set方法赋值

```
// 原生SQL  
PreparedStatement ps = connection.prepareStatement("select * from user where id=?");  
ps.setInt(1, 123);  
ResultSet rs = ps.executeQuery();
```

\${}是直接替换，有SQL注入的风险

```
// 原生SQL  
int sid=123;  
Statement stmt = conn.createStatement();  
ResultSet rs = stmt.executeQuery("select * from user where id="+sid);
```

字段名不同

1. 通过SQL别名解决
2. 通过resultMap设置字段和属性的映射关系

```
<resultMap type="me.gacl.domain.order" id="orderresultmap">  
    <!--用 id 属性来映射主键字段-->  
    <id property="id" column="order_id">  
        <!--用 result 属性来映射非主键字段, property 为实体类属性名, column  
        为数据表中的属性-->  
        <result property = "orderno" column ="order_no"/>  
        <result property="price" column="order_price" />  
</resultMap>
```

插入获取主键

获取到insert插入后的主键id，可以在insert标签中添加useGeneratedKeys参数即可

```
<insert id="insertAuthor" useGeneratedKeys="true" keyProperty="id">  
    insert into Author (username,password,email,bio)  
    values (#{username},#{password},#{email},#{bio})  
</insert>
```

进阶

设计模式

缓存模块：装饰器模式

日志模块：适配器模式

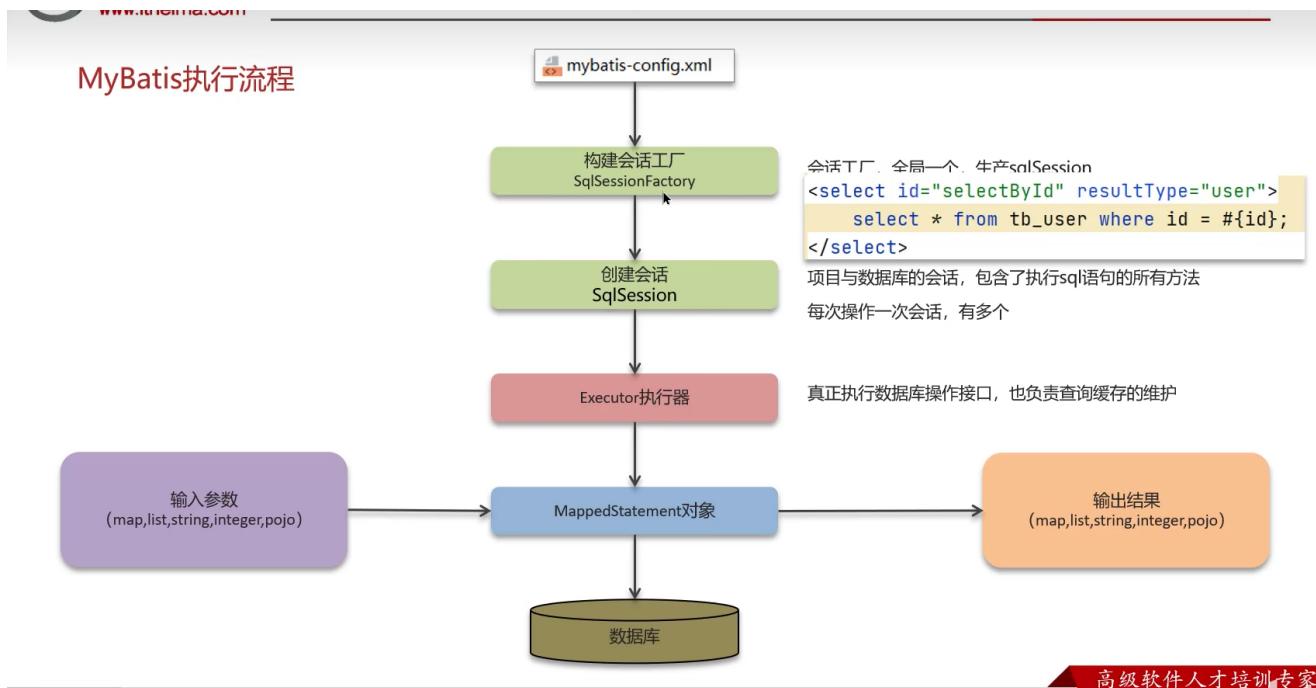
SqlSessionFactory: 工厂模式

SqlSessionFactoryBuilder: 建造者模式

Mapper接口：代理接口，JDK动态代理

执行流程

1. 读取Mybatis配置文件，加载运行环境和Mapper映射文件
2. 构建会话工厂SqlSessionFactory
3. 会话工厂创建SqlSession对象，包含了SQL语句增删改查的所有方法
4. 操作Executor执行器，同时负责缓存的维护
5. Executor接口方法中有一个MappedStatement类型的参数，封装了映射的信息
6. Executor执行前，进行输入参数映射，执行SQL
7. 输出结果映射



执行器

1. SimpleExecutor: 简单的执行器，每次执行操作都会开启一个新的Statement对象，用完就会立刻关闭。
2. ReuseExecutor: 重复使用执行器，实现了对Statement对象的复用
3. BatchExecutor: 批处理任务执行器。

所有的执行器都和SqlSession生命周期保持一致

延迟加载

当遇到1对多的情况下，可以手动配置延迟加载：即当用到的时候才加载，默认是false，立即加载。

延迟加载的底层是使用CGLIB代理来实现的

具体做法是在中设置fetchType="lazy"或者在全局配置中设置lazyLoadingEnabled=true

```
public class User{  
    private int id;  
    private String name;  
    private List<Order> orderList;  
}
```

```
<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper  
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="org.mybatis.example.mapper.StudentMapper">  
    <resultMap id="studentMap" type="org.mybatis.example.bean.Student" autoMapping="true">  
        <id property="id" column="id" />  
        <result property="name" column="name"/>  
        <!-- 懒加载，只有当用到blogList的时候才会加载-->  
        <collection property="blogList" ofType="Blog" column="id"  
            select="org.mybatis.example.mapper.BlogMapper.selectBlogBySid"  
fetchType="lazy"/>  
    </resultMap>  
    <select id="selectStudent" resultMap="studentMap">  
        select * from student where id = #{id}  
    </select>  
</mapper>
```

约定

1. Mapper接口和XML映射文件名称一致
2. Mapper接口的方法名要和XML映射文件中的id一致
3. Mapper接口的方法返回类型和XML定义的标签返回类型一致
4. Mapper接口的全类名和XML全路径一致

高级

SqlSession线程安全问题

SqlSession是线程不安全的，我们在工作中不会单独使用DefaultSqlSession，而是整合了Spring框架来使用。

Spring框架整合Mybatis是如何解决线程安全问题的？

首先，线程不安全的原因是多个线程同时操作同一个成员变量，如果将该成员编程局部变量，将不存在线程安全的问题。

Spring框架的做法是创建一个SqlSessionTemplate模板对象，定义了数据库操作的相关方法，本质上是通过代理对象来获取DefaultSqlSession对象后来执行的，而且吧DefaultSqlSession对象声明在了局部变量中，从而解决了线程不安全的问题。

```
// MybatisAutoConfiguration, 注入SqlSessionTemplate
@Bean
@ConditionalOnMissingBean
public SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory sqlSessionFactory) {
    ExecutorType executorType = this.properties.getExecutorType();
    if (executorType != null) {
        return new SqlSessionTemplate(sqlSessionFactory, executorType);
    } else {
        return new SqlSessionTemplate(sqlSessionFactory);
    }
}
```

```
// org.mybatis.spring.SqlSessionTemplate, 构造器
public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory, ExecutorType executorType,
    PersistenceExceptionTranslator exceptionTranslator) {
    // 创建代理对象
    this.sqlSessionProxy = (SqlSession)
        newProxyInstance(sqlSessionFactory.class.getClassLoader(),
            new Class[] { SqlSession.class }, new SqlSessionInterceptor());
}
```

```
Constructs a Spring managed SqlSession with the given SqlSessionFactory and ExecutorType. A custom SQLExceptionTranslator can be provided as an argument so any PersistenceException thrown by MyBatis can be custom translated to a RuntimeException. The SQLExceptionTranslator can also be null and thus no exception translation will be done and MyBatis exceptions will be thrown
形参: sqlSessionFactory - a factory of SqlSession
      executorType - an executor type on session
      exceptionTranslator - a translator of exception

public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory, ExecutorType executorType,
    PersistenceExceptionTranslator exceptionTranslator) {
    notNull(sqlSessionFactory, "Property 'sqlSessionFactory' is required");
    notNull(executorType, "Property 'executorType' is required");
    this.sqlSessionFactory = sqlSessionFactory;
    this.executorType = executorType;
    this.exceptionTranslator = exceptionTranslator;
    this.sqlSessionProxy = (SqlSession)
        newProxyInstance(sqlSessionFactory.class.getClassLoader(),
            new Class[] { SqlSession.class }, new SqlSessionInterceptor());
}

// 创建SqlSession的代理对象
public SqlSessionFactory getSqlSessionFactory() { return this.sqlSessionFactory; }

public ExecutorType getExecutorType() { return this.executorType; }

public PersistenceExceptionTranslator getPersistenceExceptionTranslator() { return this.exceptionTranslator; }

// Retrieve a single row mapped from the statement key.
@Override
public <T> T selectOne(String statement) {
    return this.sqlSessionProxy.selectOne(statement);
}
```

// UnsupportedOperationException

Proxy needed to route MyBatis method calls to the proper SqlSession got from Spring's Transaction Manager. It also unwraps exceptions thrown by Method.invoke(Object, Method, Object[]) to pass a PersistenceException to the PersistenceExceptionTranslator.

private class SqlSessionInterceptor implements InvocationHandler {
 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 SqlSession sqlSession = getSqlSession(sqlSessionTemplate.this.sqlSessionFactory,
 SqlSessionTemplate.this.executorType, SqlSessionTemplate.this.exceptionTranslator);
 try {
 Object result = method.invoke(sqlSession, args);
 if (!isSqlSessionTransactional(sqlSession, SqlSessionTemplate.this.sqlSessionFactory)) {
 // force commit even on non-dirty sessions because some databases require
 // a commit/rollback before calling close()
 sqlSession.commit(force: true);
 }
 return result;
 } catch (Throwable t) {
 Throwable unwrapped = unwrapThrowable(t);
 if (SqlSessionTemplate.this.exceptionTranslator != null && unwrapped instanceof PersistenceException)
 // release the connection to avoid a deadlock if the translator is not loaded. See issue #22
 closeSqlSession(sqlSession, SqlSessionTemplate.this.sqlSessionFactory);
 sqlSession = null;
 Throwable translated = SqlSessionTemplate.this.exceptionTranslator
 .translateExceptionIfPossible((PersistenceException) unwrapped);
 if (translated != null) {
 unwrapped = translated;
 }
 }
 }
}

缓存

mybatis的一级缓存和二级缓存，默认开启一级缓存，关闭二级缓存，如果需要开启二级缓存，只需要再映射文件中添加标签即可

缓存更新机制：当某一个作用于进行增删改操作后，默认会清空改作用于下的所有查询的缓存。

一级缓存：基于本地的HashMap本地缓存，其作用域是session，当session进行刷新或者关闭后，缓存就会被清空。

```
public class TransactionalCacheManager {
    Map<Cache, TransactionalCache> transactionalCaches = new HashMap<>();
}
```

前提

1. 使用相同的session
2. 调用方法和参数相同

```

51      1个用法 新 *
52
53      public void selectStudentUseOneCache() {
54          try (SqlSession session = sqlSessionFactory.openSession()) {
55              BlogMapper mapper = session.getMapper(BlogMapper.class);
56              Blog blog = mapper.selectBlog( id: 1);
57              System.out.println(blog);
58              System.out.println("-----");
59              BlogMapper mapper2 = session.getMapper(BlogMapper.class);
60              Blog blog2 = mapper2.selectBlog( id: 1);
61              System.out.println(blog2);
62          }
63      }

```

测试已通过: 1共 1 个测试 - 747毫秒

```

47毫秒 "D:\Program Files\JAVA\bin\java.exe" ...
47毫秒 DEBUG [main] - ==> Preparing: select id, msg, sid from blog where id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <== Columns: id, msg, sid
TRACE [main] - <== Row: 1, hello world, 2
DEBUG [main] - <== Total: 1
Blog(id=1, msg=hello world, sId=2)
-----
Blog(id=1, msg=hello world, sId=2)

```

进程已结束, 退出代码0

两次相同的调用, 实际上只调用一次数据库, 第二次是使用缓存

二级缓存: 作用域是namespace和mapper的作用于, 不依赖session。当某一个作用域发生了增删改操作后, 会清空二级缓存数据。

只有当会话提交或者关闭之后, 一级缓存数据才会转移到二级缓存

1. 在映射文件中开启二级缓存

2. 二级缓存的实现类

3. 只有当会话提交或者关闭之后, 一级缓存数据才会转移到二级缓存

4. 缓存命中, 第二次查询从缓存中取数据

数据结构

可视化动态界面：<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

散列表

1. 概念：又名哈希表，是一种根据给定的key直接方法内存中的值的数据结构，由数组演化而来，利用数组根据下标可以快速定位的特性
2. 底层实现：是数组加链表或者数组加红黑树
3. 复杂度：平均情况下O(1)，如果发生哈希冲突，则查询复杂度：链表O(n)，红黑树O(logN)
4. hash函数：如果两个对象值相等，则hashcode一定相等；反之不成立
5. 发生哈希冲突后，可以使用拉链法（链表）

树

二叉搜索树

Binary Search Tree, BST

1. 概念：左边的值 < 中间的值 < 右边的值
2. 复杂度：查找，插入和删除的时间复杂度为O(logn)
3. 极坏情况下，例如有序递增数组，生成的二叉搜索树会退化为链表

平衡二叉树 (AVL)

1. 概念：左右子树的高度差的绝对值不超过1
2. 复杂度：查找时间复杂度最坏和平均都是O(nlogn)
3. 添加删除后需要判断是否失衡，如果失衡则需要进行调整，总共分为四种

B树

B-tree, B代表balance，又叫平衡多路查找树。

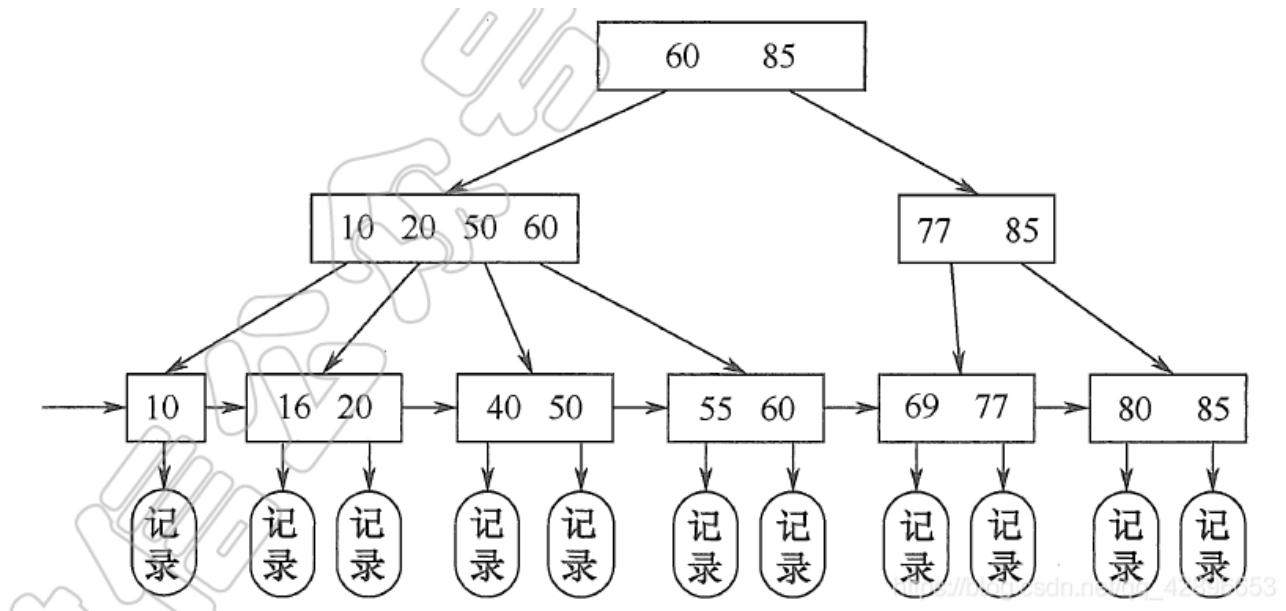
一颗m阶的B树应该满足：

1. 每个节点最多只有m个子节点。
2. 每个非叶子节点（除了根）具有至少 $\lceil m/2 \rceil$ 个子节点。
3. 如果根不是叶节点，则根至少有两个子节点。
4. 具有k个子节点的非叶节点包含k - 1个键。

5. 所有叶子都出现在同一水平，没有任何信息（高度一致）。

B+树

- 有m个子树的中间节点包含有m个元素（B树中是k-1个元素），每个元素不保存数据，只用来索引；
- 所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。（而B树的叶子节点并没有包括全部需要查找的信息）；
- 所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。（而B树的非终节点也包含需要查找的有效信息）；



B/B+树区别

B树	B+树
B树的每个节点，有m个key，m+1个指针，每个指针分别是区间，代表大于前面的key，小于后面的key	B+树的每个节点，有m+1个key，m+1个指针，每个指针与一个key对应，代表子节点中的数全部大于当前key。同时，因此每个节点的key值更多，所以整个树的高度更低。
B树中每个节点的每个key都有数据信息	B+树中只有叶子节点有数据信息，非叶子节点没有。所以B+树的非叶子节点大小更小
B树的所有数据是一个整体：非叶子节点也是数据的一部分，可能还没到叶子节点就已经找到，返回了。B树的所有节点都是数据	B+树的非叶子节点就是单纯的索引，所有实际的数据都存储在叶子节点中，所以每次查询，都必须查询到叶子节点，所以每次查询的速度就十分的稳定
B树不可以进行叶子节点间的顺序查找，同时若是可以也没意义，因为是中序遍历	B+树的叶子节点有指针连着，可以范围查找，即循着范围起点的叶子节点进行顺序遍历

红黑树

性质

1. 节点是红色或者黑色
2. 根节点是黑色
3. 叶子节点都是黑色的空节点
4. 红色节点的子节点都是黑色
5. 从任意节点到叶子节点的所有路径包含相同数目的黑色节点

概念

- 概念：红黑树是一种自平衡二叉查找树，需要满足五个性质来保证平衡
- 复杂度：查找、添加和删除都是 $O(\log N)$
- 添加删除后需要进行旋转调整操作，但是复杂度是 $O(1)$ ，因此对于添加和删除的整体复杂度仍然是 $O(\log N)$

算法

遍历方式

先序遍历：根 - 左 - 右

中序遍历：左 - 根 - 右

后序遍历：左 - 右 - 根

层次遍历

队列版

```
private void levelTraverse(Queue<TreeNode> queue, List<Integer> res) {  
    TreeNode root = queue.poll();  
    if (root == null)  
        return;  
    res.add(root.val);  
    if (root.left != null)  
        queue.add(root.left);  
    if (root.right != null)  
        queue.add(root.right);  
    levelTraverse(queue, res);  
}
```

列表版

```
public void levelOrder(TreeNode root, List<List<Integer>> list, int level){  
    if(root==null)  
        return;  
    if(list.size()<=level)  
        list.add(new ArrayList<>());  
    list.get(level).add(root.val);  
    levelOrder(root.left,list,level+1);  
    levelOrder(root.right,list,level+1);  
}
```

设计模式

23种设计模式

- 创建型：单例模式，工厂模式，抽象工厂模式，建造者模式，原型模式
- 结构型：适配器模式，桥接模式，组合模式，装饰模式，外观模式，享元模式，代理模式
- 行为型：责任链模式，命令者模式，解释器模式，迭代器模式，中介者模式，备忘录模式，状态模式，策略模式，模板方法模式，观察者模式，访问者模式

项目中使用到的设计模式

模式：策略模式+责任链模式+命令者模式+适配器

1. 工厂模式：工厂会根据不同的key创建不同的commander
2. 策略模式：根据不同的应用场景，采用不同的策略，可以使用if/else来实现（只要代码中有冗长的if-else或者switch判断都可以使用策略模式优化），但是不利于维护，可以考虑使用策略模式。在本项目中，针对不同的业务流，采用不同的命令链模型，底层是使用Map实现的，key是一个字符串表示是什么样的业务流，例如新建，更新，更换交易对手方，行权等业务流，映射的value是一个命令链，都实现了apache下的commons-chains的Chain方法，然后执行与之对应的命令链。key的具体值是通过flowKeyHelper来确定，大致流程是JMS接受到消息后，根据消息内容来判断key是什么。
3. 责任链模式：为了避免请求发送者与多个请求耦合在一起，将所有请求的处理着根据前一个对象记住下一个对象的引用而形成一条链路，当请求发生时，可以沿着这条链传递，直至处理结束。例如，报销流程审批系统，拦截器，过滤器。首先由策略模式确定了使用哪个命令链CommandChain，然后对于命令链的实现首先是责任链，在某一个命令链中一般有几个命令对象组成：例如存储数据命令，格式转化命令和发布命令等。这些命令都实现了Command接口，重写execute方法，如果返回Procecssing则继续执行下一个命令，如果返回Complete则结束命令链路，后面的命令不再执行。
4. 命令者模式：命令发起人将命令封装给一个对象发给接受方执行的一种模式，同时也将两者解耦。接收方可以从命令中获取到命令的主要执行内容和需要的数据。在项目中，主要是定义了很多命令类，都实现了Command接口并加入到上述的命令链中。Command接口中的execute方法有一个上下文Context参数，可以获取到之前传入的各种数据，可能会在命令代码中用到。

5. 适配器模式：将一个类的接口通过适配器转化为另外一种期待的接口，从而使得两个系统都连在一起正常工作。在项目中，有两个个适配器的模块，他的主要功能就是将外部的系统发过来的数据转化为内部系统可以识别的一种数据格式。具体实现是创建一个Mapper接口，该接口有两个泛型分别是源数据类型和目标数据类型，有一个map方法，传入的是源数据和上下文，返回值是目标数据。在该接口的具体实现类中经常遇到层级调用情况，对于每一个实现类都可以认为是一种适配器。

计算机网络

网络分层

OSI七层网络模型	TCP/IP四层概念模型	对应网络协议
应用层 (Application)	应用层	HTTP、TFTP, FTP , NFS, WAIS、SMTP
表示层 (Presentation)		Telnet, Rlogin, SNMP, Gopher
会话层 (Session)		SMTP, DNS
传输层 (Transport)	传输层	TCP, UDP
网络层 (Network)	网络层	IP, ICMP, ARP, RARP, AKP, UUCP
数据链路层 (Data Link)	数据链路层	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP
物理层 (Physical)		IEEE 802.1A, IEEE 802.2到IEEE 802.11 https://blog.csdn.net/WangJinruo

TCP建立和关闭连接

三次握手

- ① 握手过程其实是发送的TCP报文，在这里面有两个字段，SYN 和 seq
- ② 当服务器接受到我们的握手请求时，会回复一个确认报文
- ③ 当客户端收到确认报文的时候，客户端需要对这个确认报文进行回复

经过了这三次握手，两者就进入了连接状态

三次握手



成都成都
四川的8633 SYN
(Seq = 1000)

Synchronization: 同步
Sequence: 序列

SYN + ACK 请讲
(Seq = 2000, Ack = 1001)

Acknowledgment: 答复

现在…有点故障
我申请下高度

连接已建立

(Seq = 1001, Ack = 2001)

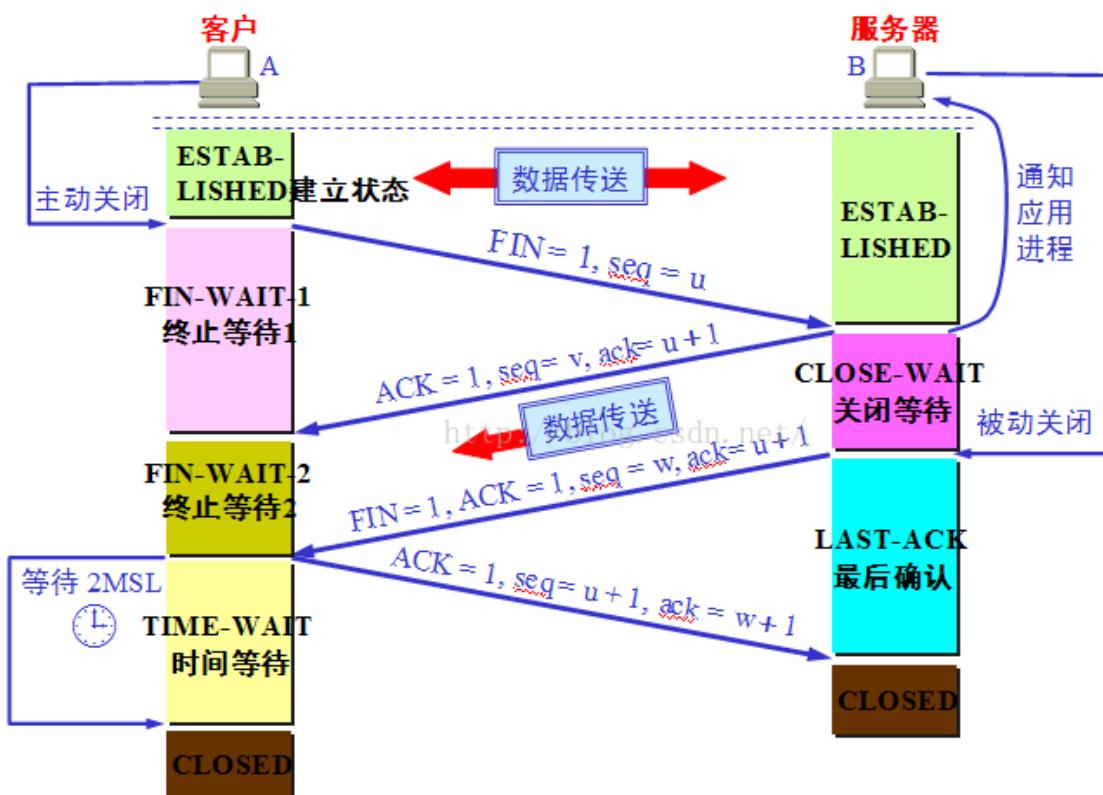
技术蛋老师



四次挥手

所谓的四次挥手，就是关闭TCP连接的过程，指的是断开一个TCP连接，需要客户端和服务端总共发送4个包，以确定双方连接的断开。

主要目的：保证TCP连接的全双工连接



https://blog.csdn.net/weixin_40030258

过程

- **第一次挥手**: 客户端发送一个FIN包 ($\text{FIN}=1, \text{seq}=U$) 给服务器, 用来关闭客户端到服务器端的数据传输, 客户端进入`FIN_WAIT_1`状态 (终止等待)
- **第二次挥手**: 服务器端收到FIN包后, 发送一个ACK包 ($\text{ACK}=1, \text{ack}=u+1$, 在随机产生一个值v给seq) 给客户端, 服务器进入了`CLOSE_WAIT`状态 (关闭等待)
- **第三次挥手**: 服务器端发送一个FIN包 ($\text{FIN}=1, \text{ACK}=1, \text{ack}=u+1$, 在随机产生一个w值给seq) 给客户端, 用来关闭服务器到客户端的数据传输, 服务端进入了`LAST_ACK` (最后确定) 状态
- **第四次挥手**: 客户端接收FIN包, 然后进入`TIME_WAIT`状态, 接着发送一个ACK包 ($\text{ACK}=1, \text{seq}=u+1, \text{ack}=w+1$) 给服务端, 服务端确定序号, 进入`CLOSE`状态, 完成了四次挥手。

TCP流量控制和拥塞控制

博客: <https://zhuanlan.zhihu.com/p/37379780>

区别

流量控制: 流量控制是作用于接收者的, 它是控制发送者的发送速度从而使接收者来得及接收, 防止分组丢失的。

拥塞控制: 拥塞控制是作用于网络的, 它是防止过多的数据注入到网络中, 避免出现网络负载过大的情况;

流量控制

如果发送者发送数据过快, 接收者来不及接收, 那么就会出现分组丢失, 为了避免分组丢失, 控制发送者的发送速度, 使得接收者来得及接收, 这就是流量控制。

流量控制的目的是: 防止分组丢失, 是构成TCP可靠性的一方面。

解决方案:

由滑动窗口协议 (连续ARQ协议) 实现, 滑动窗口协议即保证了分组无差错, 有序接收, 也实现了流量控制。主要的方式就是接收方返回的ACK会包含自己的接受窗口大小, 并利用大小来控制发送方的数据发送。

具体做法是:

1. 当接收方的缓存空间足够大时, 接收方会告诉发送方自己能够接收的最大数据量, 即通告窗口大小。
2. 当接收方的缓存空间不足时, 接收方会减小通告窗口大小并通知发送方。
3. 发送方根据接收方通告的窗口大小控制发送数据的速率, 以确保接收方能够接收。
4. 如果发送方发送的数据量超过了接收方的缓存容量, 则接收方会丢弃部分数据, 并通知发送方减小发送速率。
5. 如果发送方继续发送超过接收方缓存容量的数据, 则接收方将丢弃所有收到的数据, 并关闭连接。

拥塞控制

常用的方法就是：（1）慢开始、拥塞避免（2）快重传、快恢复。

TCP的拥塞控制是通过一系列算法来实现的，主要的拥塞控制算法有慢开始、拥塞避免、快重传和快恢复。

1. 慢开始算法：在连接建立时，发送方维持一个叫做**拥塞窗口**的变量，它的大小可以动态变化。发送方开始时，拥塞窗口大小设置为1，发送方每次收到一个 ACK，就将拥塞窗口的大小加1，直达到慢开始门限值。当超过慢开始门限值时，切换到拥塞避免算法。
2. 拥塞避免算法：拥塞避免算法通过维持一个叫做**慢开始门限**的变量，使得窗口大小以指数级别增长。当拥塞窗口大于慢开始门限时，切换到快重传算法。
3. 快重传算法：当发送方收到连续的3个重复的 ACK 时，就重新发送数据包。
4. 快恢复算法：当发送方收到连续的3个重复的 ACK 时，将慢开始门限设置为当前拥塞窗口的一半，然后切换到慢开始算法。

这些算法的主要目的是为了防止过多的数据包在网络中造成拥塞，从而导致网络性能下降。

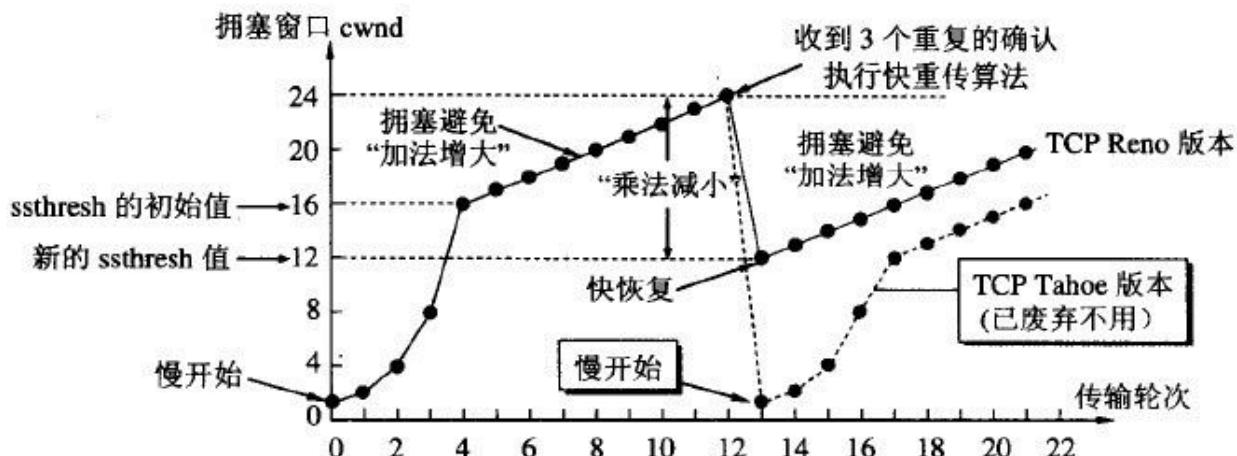


图 5-27 从连续收到三个重复的确认转入拥塞避免

http 和 https

http

http是一种无状态协议。无状态是指客户机和服务器之间不需要建立持久连接，这意味着当一个客户端向服务器发出请求，然后服务器返回响应(response)，连接就被关闭了，在服务器端不保留连接的有关信息，HTTP遵循请求/应答模型。客户机向服务器发送请求，服务器处理请求并返回适当的应答。所有HTTP连接都构成一套请求和应答。

https

HTTPS是以安全为目标的HTTP通道，简单来说就是HTTP的安全版。即HTTP下加入SSL层，HTTPS的安全基础是SSL。其所用的端口是443，过程大致如下：

1. 请求证书：浏览器向服务器的443端口发出请求，请求携带加密算法和哈希算法。
2. 响应证书：服务器收到请求后，会选择浏览器支持的加密方式，把数字证书发送给浏览器
3. 验证证书：浏览器拿到证书后，在内置的TLS中开始认证环节。首先在内部证书列表中查找，找到服务器下发证书对应的机构，没找到的话，提示用户该证书不是权威机构下发的不可信任。如果找到对应的机构，则取出该机构颁发的公钥。使用证书的公钥解密后，得到该证书的内容和签名，证书内容包括证书的网址、证书的公钥、证书有效期。浏览器会验证证书的网址是否一致，证书是否到期，如果不通过则提示用户，通过则表示可以安全使用网站公钥。
4. 公钥加密：如果确认证书有效，浏览器生成一个随机数，使用公钥加密，并将加密后的随机数发送给服务器
5. 数据加密和传输：服务器使用自己的私钥解密后得到随机数。服务器以随机数为密钥使用对称加密算法加密网页内容，并发送给浏览器
6. 解密：浏览器以随机数为密钥使用之前约定好的解密算法获取网页内容

http状态码

类别	原因
1XX	信息性状态码 接收到的请求正在处理中
2XX	成功状态码 请求正常处理完毕
3XX	重定向状态码 需要附加操作以完成请求
4XX	客户端错误码 服务器无法处理请求
5XX	服务器错误码 服务器处理请求出错

3XX重定向

301 Moved Permanently

永久性重定向，该状态码表示请求的资源已经被分配了新的URI，以后应使用资源现在所指的URI

302 Found

临时重定向，该状态码表示请求的资源已经分配了新的URI，希望用户本次使用新的URI访问。和301类似，但是该状态码表示资源不是永久性被移动，只是短暂的

303 See other

该状态码表示由于请求对应的资源存在另一个URI，应使用GET方法定向获取请求的资源

304 Not Modified

该状态码表示客户端发送附带条件的请求时，服务端允许访问资源，但未满足条件的情况。虽然被划分在3XX中，但是和重定向没有任何关系。

4XX客户端错误

400 Bad Request

该状态码表示请求报文中存在语法错误，需要修改请求的内容后再次发送请求

401 Unauthorized

该状态码表示发送的请求需要HTTP认证信息或者用户认证失败。客户端与服务器访问时，用户没有通过身份验证，需要再次认证。401着重于**认证**

403 Forbidden

该状态码表示请求资源的访问被服务器拒绝了。授权失败，是客户端已经通过身份验证，但没有权限访问要求的资源。403着重于**授权**

404 Not Found

该状态码表示服务器上无法找到请求的资源

405 Method Not Allowed

该状态码表示请求的方法不被允许

5XX服务器错误

500 Internal Server Error

该状态码表示服务器在执行请求时发生了错误，也可能是web应用存在BUG和某些临时的故障

503 Service Unavailable

该状态码表示服务器暂时处于超负荷或正在停机维护，现在无法处理请求。

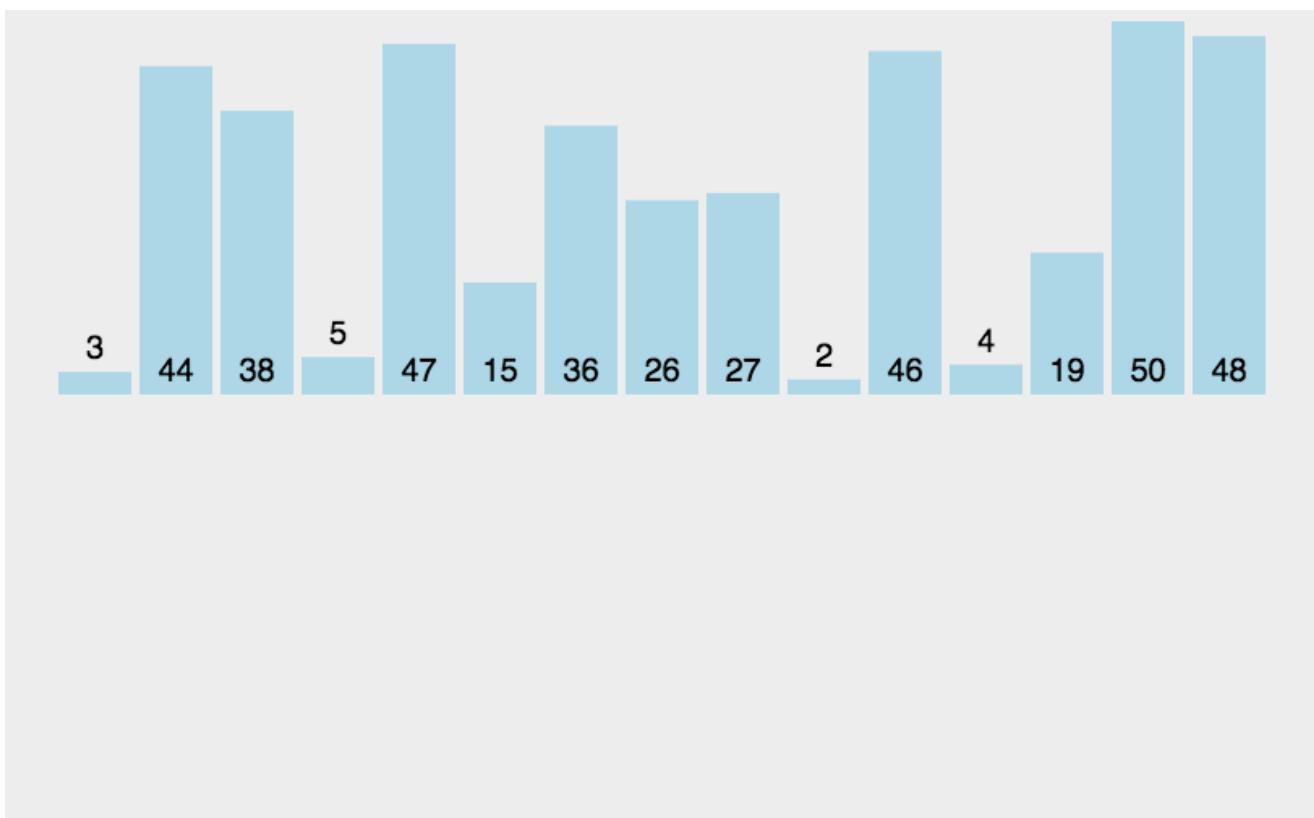
算法

排序算法

参考博客：https://blog.csdn.net/weixin_50886514/article/details/119045154

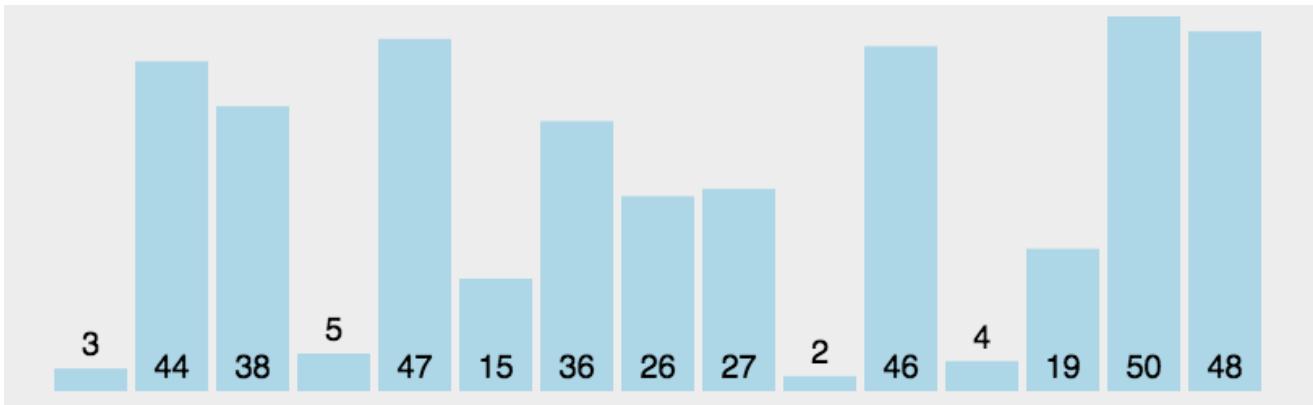
	平均复杂度	最好情况	最坏情况	空间复杂度	稳定性	比较算法
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	是
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	是
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	是
希尔排序	不确定	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定	是
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定	是
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	不稳定	是
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	不稳定	是
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$	稳定	否
桶排序	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n+k)$	稳定	否
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定	否

插入排序



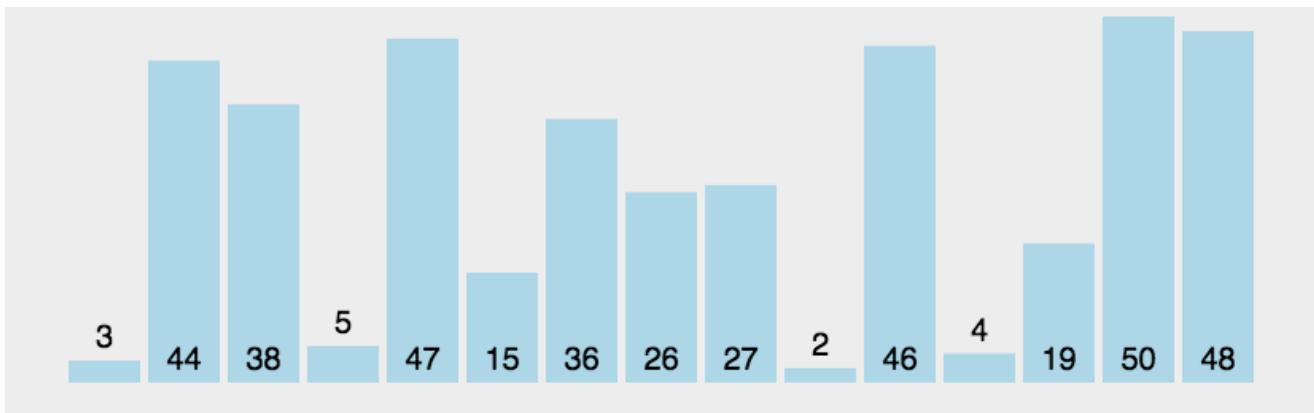
```
private static void insertSort(int[] arr, int n) {
    for (int i = 1; i < n; i++) {
        int temp = arr[i];
        for (int j = i - 1; j >= 0; j--) {
            if (temp > arr[j]) break;
            arr[j + 1] = arr[j];
            arr[j] = temp;
        }
    }
}
```

选择排序



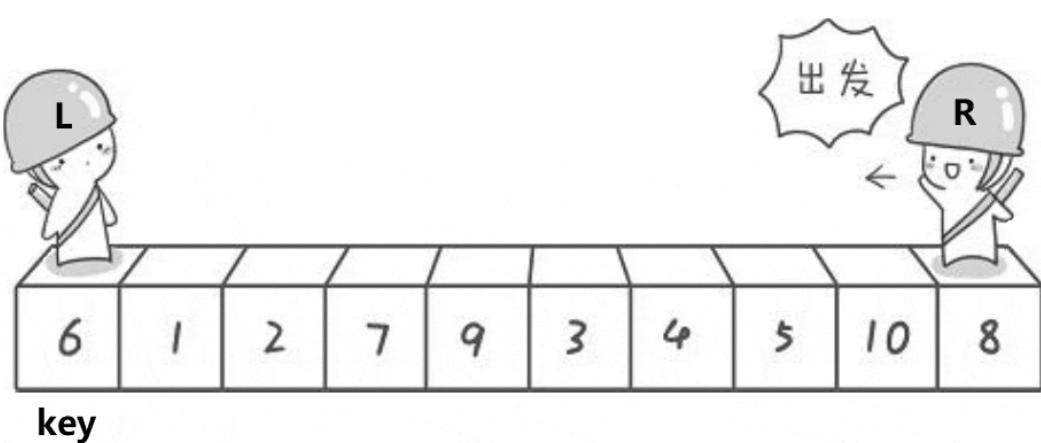
```
private static void selectSort(int[] arr, int n) {
    for (int i = 0; i < n; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
        // swap
        if (minIndex != i) {
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }
}
```

冒泡排序



```
private static void bubbleSort(int[] arr, int n) {
    int temp;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            // swap
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

快速排序



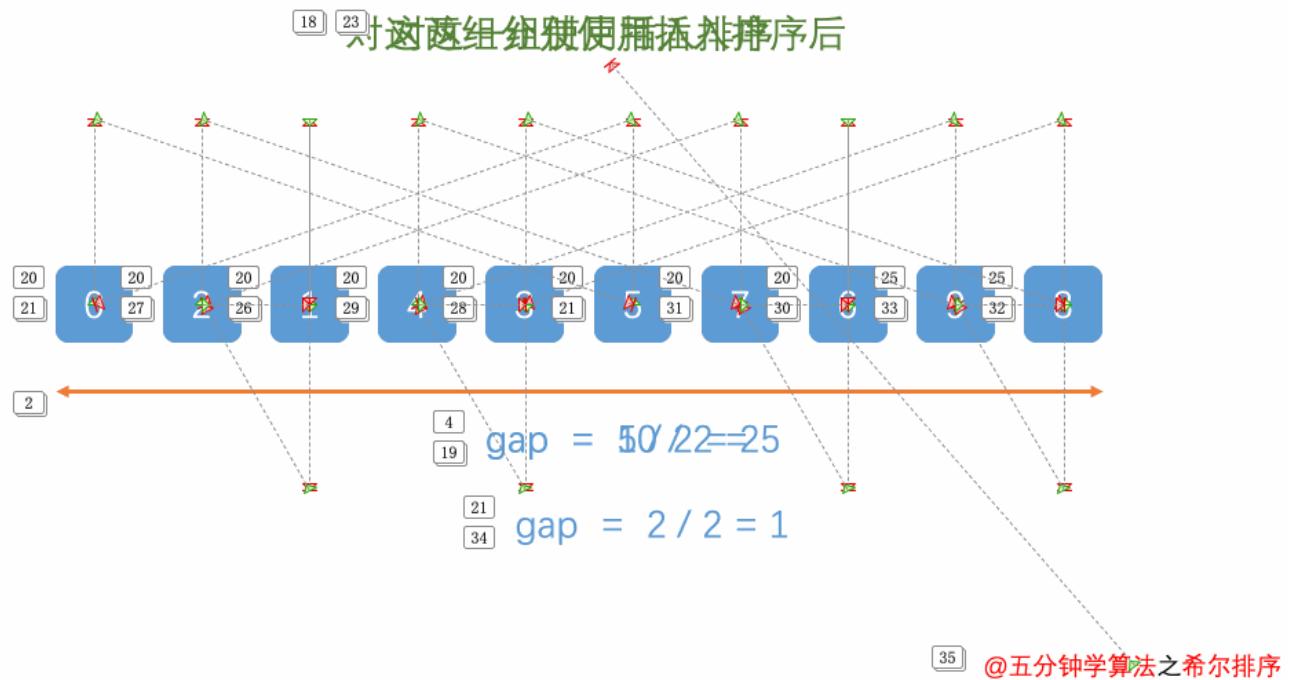
```
private static void quickSort(int[] arr, int left, int right) {
    if (left >= right)
        return;
    int leftTemp = left;
    int rightTemp = right;
```

```

int key = arr[left];
while (left < right) {
    while (left < right && arr[right] >= key) {
        right--;
    }
    while (left < right && arr[left] <= key) {
        left++;
    }
    if (left < right) {
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
    }
}
if (left == right) {
    arr[leftTemp] = arr[left];
    arr[left] = key;
}
quickSort(arr, leftTemp, left - 1);
quickSort(arr, right + 1, rightTemp);
}

```

希尔排序



```

private static void shellSort(int[] arr, int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = 0; i < gap; i++) {
            for (int j = i + gap; j < n; j += gap) {
                if (arr[j - gap] > arr[j]) {
                    int temp = arr[j - gap];
                    arr[j - gap] = arr[j];
                    arr[j] = temp;
                }
            }
        }
    }
}

```

归并排序

分解 - 排序 - 合并

6 5 3 1 8 7 2 4

```

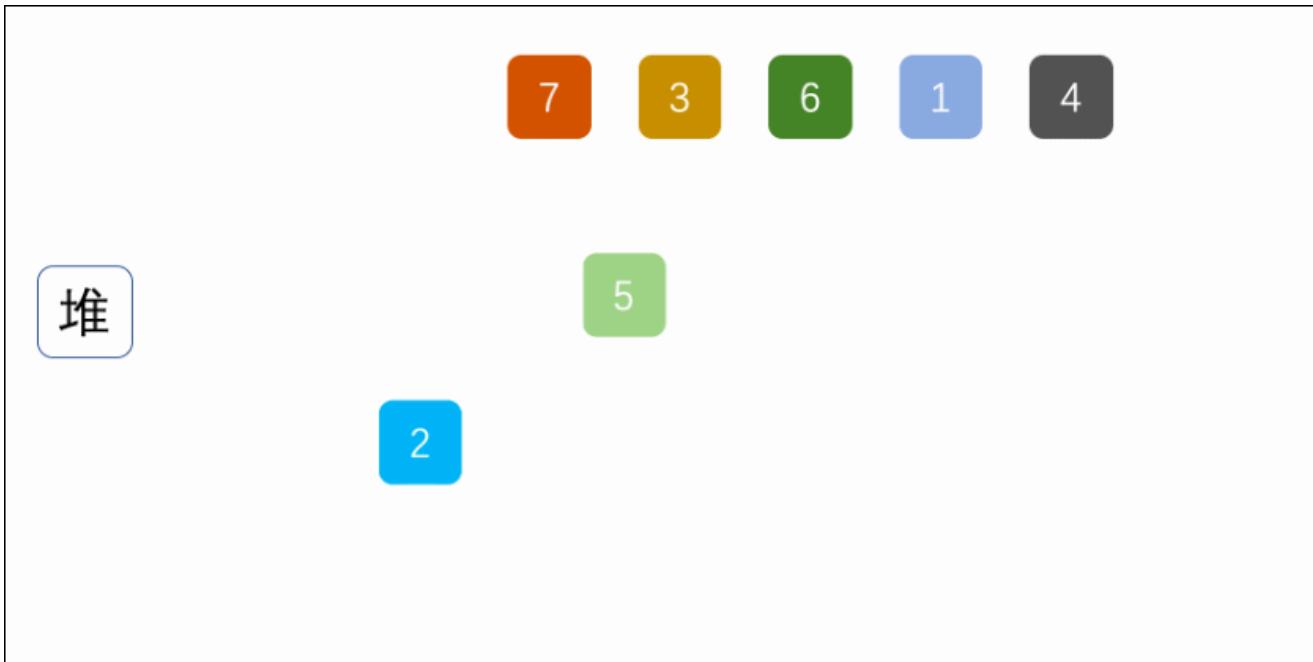
private static int[] mergeSort(int[] arr, int left, int right) {
    if (left >= right)
        return new int[]{arr[left]};
    // 分解
    int mid = (left + right) / 2;
    // 排序
    int[] leftArr = mergeSort(arr, left, mid);
    int[] rightArr = mergeSort(arr, mid + 1, right);
    // 合并
    int[] newArr = new int[leftArr.length + rightArr.length];
    int l = 0, r = 0, m=0;
    while (l < leftArr.length && r < rightArr.length) {
        newArr[m++] = leftArr[l] < rightArr[r] ? leftArr[l++] : rightArr[r++];
    }
    while (l < leftArr.length) {
        newArr[m++] = leftArr[l++];
    }
    while (r < rightArr.length) {
        newArr[m++] = rightArr[r++];
    }
    return newArr;
}

```

堆排序

总体流程：

1. 构建大顶堆
2. 最后一个节点和第一个节点交换
3. 重新堆化
4. 循环2,3步骤



1. 先构建一个堆，升序排序构建大顶堆，降序构建小顶堆
 1. 当前节点的左子节点下标为： $2i+1$ ，右子节点为 $2i+2$
 2. 当前节点的父节点下标为： $(i-1)/2$
2. 比较左右子节点和根点，如果发生交换，则需要向下递归根节点，对子树堆化
3. 交换，砍树，重新堆化

```
private static void heapSort(int[] arr, int n) {  
    buildMaxHeap(arr, n);  
    for (int i = n - 1; i >= 0; i--) {  
        swap(arr, i, 0);  
        // 注意最后一个参数不是数组长度，而是当前索引  
        // 因为要把排序过的数字排除出去  
        heapify(arr, 0, i);  
    }  
}  
  
// 创建大顶堆  
private static void buildMaxHeap(int[] arr, int length) {  
    int last = length - 1;  
    int lastParent = (last - 1) / 2;  
    // 从最后一个一个节点的父节点开始创建  
    for (int parent = lastParent; parent >= 0; parent--) {  
        heapify(arr, parent, length);  
    }  
}
```

```

// 堆化
// 可以以一个简单的三个节点的二叉树作为例子开始
// 在最后排序完成后再对子节点进行堆化
private static void heapify(int[] arr, int currentIndex, int length) {
    int left = currentIndex * 2 + 1;
    int right = currentIndex * 2 + 2;
    int maxIndex = currentIndex;
    if (left < length && arr[left] > arr[maxIndex])
        maxIndex = left;
    if (right < length && arr[right] > arr[maxIndex])
        maxIndex = right;
    if (maxIndex != currentIndex) {
        swap(arr, currentIndex, maxIndex);
        // 对从上面传下来的节点，重新堆化操作
        heapify(arr, maxIndex, length);
    }
}
private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

```

桶排序/计数排序

一般用于差距不是特别大的数组排序，否则会出现大量的空间浪费的情况

```

public static void bucketSort(int[] arr) {
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < arr.length; i++) {
        max = Math.max(max, arr[i]);
        min = Math.min(min, arr[i]);
    }
    // 创建桶
    int bucketNum = (max - min) / arr.length + 1;
    ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketNum);
    for (int i = 0; i < bucketNum; i++) {
        bucketArr.add(new ArrayList<Integer>());
    }
    // 将每个元素放入桶
    for (int i = 0; i < arr.length; i++) {
        int num = (arr[i] - min) / (arr.length);
        bucketArr.get(num).add(arr[i]);
    }
    // 对每个桶进行排序
    for (int i = 0; i < bucketArr.size(); i++) {
        Collections.sort(bucketArr.get(i));
    }
    // 赋值回数组
    int num = 0;
    for (ArrayList<Integer> integers : bucketArr) {
        for (Integer integer : integers) {

```

```
        arr[num++] = integer;
    }
}
}
```

基数排序

1. 首先取最大值，和最大值的位数
2. 根据个位的数字排序
3. 再根据十位，百位...排序
4. 最后合并所有位数的list

```
public void sort(int[] arr, int n){
    // 最大值
    int maxNum = theMaxNum();
    // 最大位数
    int maxTime = theMaxTime();
    // 创建十个list
    //建立 10 个队列;
    List<ArrayList> queue=newArrayList<ArrayList>();
    for(int i=0;i<10;i++){
        queue.add(new ArrayList<Integer>());
    }
}
```

常用方法

Map 常用方法

1. computeIfAbsent(K, Function<key, new Value>): 当不存在时，执行function获取到value并插入到map中
2. computeIfPresent(K, BiFunction<key, oldValue,newValue>): 当存在时，执行BiFunction获取到newValue并替换oldValue
3. computeIfPresent(K, BiFunction<key, oldValue,newValue>): 存在则替换，不存在插入
4. getOrDefault(key,defaultValue): 如果有则取。如果没有则用默认值

数组常用方法

- 字符串转化为数字数组

```
String s = "123456789";
int[] array = s.chars().map(i -> i - 48).toArray();
```

- 基本类型数组转化为链表

```
List<Integer> collect = Arrays.stream(arr).boxed().collect(Collectors.toList());
```

- 基本类型链表转数组

```
list.stream().mapToInt(Integer::intValue).toArray();
```

- 求整数数组的最小值

```
int min = Arrays.stream(num).min().orElse(0);
```

- XX

链表常用方法

- 基本类型链表转数组

```
list.stream().mapToInt(Integer::intValue).toArray();
```

剑指Offer

3数组中重复的数字

```
Set<Integer> set = new HashSet<>();

public int findRepeatNumber(int[] nums) {
    int n = nums.length;
    for(int i = 0; i < n; i++) {
        if(set.contains(nums[i]))
            return nums[i];
        set.add(nums[i]);
    }
    return 0;
}
```

4.二维数组查找

从左下到右上，当前值小则向右，当前值大向上

```
public boolean findNumberIn2DArray(int[][] matrix, int target) {
    int m = matrix.length;
    if (m == 0) return false;
    int n = matrix[0].length;
    if (n == 0) return false;
    int i = m - 1;
    int j = 0;
    while (i >= 0 && j < n) {
        if (matrix[i][j] > target) i--;
        else if (matrix[i][j] < target) j++;
        else return true;
    }
    return false;
}
```

```
        else return true;
    }
    return false;
}
```

5. 替换空格

```
public String replaceSpace(String s) {
    return s.replaceAll(" ","%20");
}
```

6. 反向打印链表

递归版

```
public int[] reversePrint(ListNode head) {
    List<Integer> list = new ArrayList<>();
    reversePrint(head, list);
    return list.stream().mapToInt(Integer::intValue).toArray();
}

private void reversePrint(ListNode head, List<Integer> list) {
    if (head == null)
        return;
    int val = head.val;
    reversePrint(head.next, list);
    list.add(val);
}
```

链表版

```
public int[] reversePrint(ListNode head) {
    ListNode temp = null;
    int n=0;
    while(head!=null){
        ListNode linked = new ListNode();
        linked.val=head.val;
        linked.next=temp;
        temp=linked;
        head = head.next;
        n++;
    }
    int[] res = new int[n];
    for(int i=0;i<n;i++){
        res[i]=temp.val;
        temp=temp.next;
    }
    return res;
}
```

7. 重建二叉树

力扣: <https://leetcode.cn/problems/zhong-jian-er-cha-shu-lcof/>

```
Map<Integer, Integer> map = new HashMap<>();  
  
public TreeNode buildTree(int[] preorder, int[] inorder) {  
    Map<Integer, Integer> map = new HashMap<>();  
    if (preorder.length == 0) return null;  
    for (int i = 0; i < preorder.length; i++) {  
        map.put(inorder[i], i);  
    }  
    return buildTree(preorder, 0, preorder.length - 1, map);  
}  
  
private TreeNode buildTree(int[] preorder, int root, int right, Map<Integer, Integer> map) {  
    int val = preorder[root];  
    TreeNode treeNode = new TreeNode(val);  
    Integer inorderIndex = map.get(val);  
    treeNode.left = root + 1 > inorderIndex ? null : buildTree(preorder, root + 1, inorderIndex  
- 1, map);  
    treeNode.right = inorderIndex + 1 > right ? null : buildTree(preorder, inorderIndex + 1,  
right, map);  
    return treeNode;  
}
```

9. 两个栈实现队列

```
class CQueue {  
  
    private final Stack<Integer> stack1 = new Stack<>();  
    private final Stack<Integer> stack2 = new Stack<>();  
  
    public CQueue() {}  
  
    public void appendTail(int value) {  
        stack1.push(value);  
    }  
  
    public int deleteHead() {  
        if (stack2.isEmpty()) {  
            if (stack1.isEmpty())  
                return -1;  
            else {  
                while (!stack1.isEmpty()) {  
                    stack2.push(stack1.pop());  
                }  
            }  
        }  
        return stack2.pop();  
    }  
}
```

```
    }
}
```

10斐波那契数列和青蛙跳台阶

```
// 斐波那契数列
public int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    int[] res = new int[n + 1];
    res[0] = 0;
    res[1] = 1;
    for (int i = 2; i <= n; i++) {
        int resNum = res[i - 1] + res[i - 2];
        res[i] = resNum > max ? resNum % max : resNum;
    }
    return res[n];
}

// 青蛙跳台阶
public int numWays(int n) {
    if(n==0) return 1;
    if(n==1) return 1;
    int[] res=new int[n+1];
    res[0]=1;
    res[1]=1;
    for(int i=2;i<=n;i++){
        res[i]=(res[i-1]+res[i-2])%1000000007;
    }
    return res[n];
}
```

11最小数字

力扣: <https://leetcode.cn/problems/xuan-zhuan-shu-zu-de-zui-xiao-shu-zi-lcof/>

```
public int minArray(int[] numbers) {
    return Arrays.stream(numbers).min().orElse(0);
}
```

12矩阵中的数列

力扣: <https://leetcode.cn/problems/ju-zhen-zhong-de-lu-jing-lcof/>

动态规划

两步：

第一步：列出转移表达式

第二步骤：列出初始条件

斐波那序列

```
private static int fib(int n) {  
    if (n <= 0)  
        return 0;  
    if (n == 1 || n == 2)  
        return 1;  
    return fib(n - 1) + fib(n - 2);  
}  
  
//改进为记忆化搜索  
private static int fib2(int n) {  
    Map<Integer, Integer> map = new HashMap<>();  
    map.put(1, 1);map.put(2, 1);  
    for (int i = 3; i <= n; i++) {  
        map.put(i, map.get(i - 1) + map.get(i - 2));  
    }  
    return map.get(n);  
}
```

爬楼梯

力扣70题：<https://leetcode.cn/problems/climbing-stairs/>

```
private static int climbStairs(int n) {  
    if (n == 1)  
        return 1;  
    if (n == 2)  
        return 2;  
    return climbStairs(n - 1) + climbStairs(n - 2);  
}  
  
private static int climbStairs2(int n) {  
    if (n == 1)  
        return 1;  
    if (n == 2)  
        return 2;  
    Map<Integer, Integer> map = new HashMap<>();  
    map.put(1, 1);  
    map.put(2, 2);  
    for (int i = 3; i <= n; i++) {  
        map.put(i, map.get(i - 1) + map.get(i - 2));  
    }  
}
```

```
    return map.get(n);
}
```

三角型最小路径

力扣120: <https://leetcode.cn/problems/triangle/>

```
public static int minimumTotal(List<List<Integer>> triangle) {
    for (int i = 1; i < triangle.size(); i++) {
        List<Integer> list = triangle.get(i);
        for (int j = 0; j < list.size(); j++) {
            list.set(j, list.get(j) + min(triangle, i, j));
        }
    }
    return Collections.min(triangle.get(triangle.size() - 1));
}

private static int min(List<List<Integer>> triangle, int i, int j) {
    List<Integer> lastRowList = triangle.get(i - 1);
    // 最左侧的，直接取第一个
    if (j == 0) return lastRowList.get(0);
    // 最右侧的，直接取最后一个
    if (j == lastRowList.size()) return lastRowList.get(lastRowList.size() - 1);
    return Math.min(lastRowList.get(j - 1), lastRowList.get(j));
}
```

最小路径和

力扣64: <https://leetcode.cn/problems/minimum-path-sum/>

```
public int minPathSum(int[][] grid) {
    for (int i = 0; i < grid.length; i++) {
        int[] row = grid[i];
        for (int j = 0; j < row.length; j++) {
            grid[i][j] = grid[i][j] + min(grid, i, j);
        }
    }
    return grid[grid.length - 1][grid[0].length - 1];
}

public int min(int[][] grid, int row, int column) {
    if (row == 0 && column == 0) return 0;
    if (row == 0) return grid[row][column - 1];
    if (column == 0) return grid[row - 1][column];
    return Math.min(grid[row - 1][column], grid[row][column - 1]);
}
```

整数拆分

力扣343:<https://leetcode.cn/problems/integer-break/>

```
// 取F(i)*F(n-i)的最大值,
// 例如F(63)=Max(F(2)*F(62),F(3)*F(61)....F(31)*F(32))
public int integerBreak(int n) {
    if (n == 1) return 1;
    if (n == 2) return 1;
    if (n == 3) return 2;
    Map<Integer, Integer> map = new HashMap<>();
    map.put(1, 1);
    map.put(2, 2);
    map.put(3, 3);
    for (int i = 4; i <= n; i++) {
        int max = 0;
        // 这里循环的最大值是i, 不是n
        for (int j = 2; j <= i / 2; j++) {
            max = Math.max(max, map.get(j) * map.get(i - j));
        }
        map.put(i, max);
    }
    return map.get(n);
}
```

完全平方数

```
public int numSquares(int n) {
    int[] arr = new int[n + 1];
    arr[1] = 1;
    for (int i = 2; i <= n; i++) {
        int min = Integer.MAX_VALUE;
        // 例如F(12) = 1 + Min(F(12-1), F(12-4).F(12-9))
        for (int j = 1; j <= Math.sqrt(i); j++) {
            min = Math.min(min, arr[i - j * j]);
        }
        arr[i] = 1 + min;
    }
    return arr[n];
}
```

解码方法

力扣91: <https://leetcode.cn/problems/decode-ways/>

```
public int numDecodings(String s) {
    if (s == null || s.length() == 0) return 0;
    // 将字符串转化为数字数组
    int[] array = s.chars().map(i -> i - 48).toArray();
    int n = array.length;
```

```

        return numDecodings(array, n);
    }

private int numDecodings(int[] array, int n) {
    int[] res = new int[n];
    res[0] = array[0] == 0 ? 0 : 1;
    for (int i = 1; i < n; i++) {
        int temp = array[i - 1] * 10 + array[i];
        if (temp == 0 || (temp % 10 == 0) && temp >= 30)
            return 0;
        else if (temp == 10 || temp == 20)
            res[i] = tempValue(i, res);
        else
            res[i] = res[i - 1] + (temp >= 11 && temp <= 26 ? tempValue(i, res) : 0);
    }
    return res[n - 1];
}

private int tempValue(int i, int[] res) {
    if (i - 2 < 0) return 1;
    return res[i - 2];
}

```

不同路径

与最小路径相似

```

public int uniquePaths(int m, int n) {
    int[][] arr = new int[m][n];
    for (int i = 0; i < m; i++) {
        arr[i][0] = 1;
    }
    for (int j = 0; j < n; j++) {
        arr[0][j] = 1;
    }

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            arr[i][j] = arr[i - 1][j] + arr[i][j - 1];
        }
    }
    return arr[m - 1][n - 1];
}

```

打家劫舍

力扣198: <https://leetcode.cn/problems/house-robber/>

```
public int rob(int[] nums) {  
    int n = nums.length;  
    int[] res = new int[n + 1];  
    res[0] = nums[0];  
    for (int i = 1; i <= n; i++) {  
        // 如果新加第n户，那么有两种选择  
        // 第一种是选择前n-2户的最大值加上第n户的值；  
        // 第二种是选择前n-1户的最大值  
        res[i] = Math.max(res[i - 2] + nums[i - 1], res[i - 1]);  
    }  
    return res[n];  
}
```

打家劫舍II

力扣213: <https://leetcode.cn/problems/house-robber-ii/>

```
// 将0-n分为两部分，一部分是arr[0...n-1]另外一部分是arr[1...n]  
// 求两个数组的最大值  
public int rob(int[] nums) {  
    int n = nums.length;  
    if (n == 1)  
        return nums[0];  
    int[] res = new int[n];  
    res[0] = nums[0];  
    res[1] = Math.max(nums[0], nums[1]);  
    for (int i = 2; i < n - 1; i++) {  
        res[i] = Math.max(res[i - 1], res[i - 2] + nums[i]);  
    }  
  
    int[] res2 = new int[n];  
    res2[1] = nums[1];  
    for (int i = 2; i < n; i++) {  
        res2[i] = Math.max(res2[i - 1], res2[i - 2] + nums[i]);  
    }  
    return Math.max(res[n - 2], res2[n - 1]);  
}
```

打家劫舍III

力扣337: <https://leetcode.cn/problems/house-robber-iii/>

递归版本，超出时间限制

```
public int rob(TreeNode root) {
    return calc(root, true);
}

public int calc(TreeNode root, boolean flag) {
    if (root == null) return 0;
    int tempFalse = calc(root.left, true) + calc(root.right, true);
    int tempTrue = root.val + calc(root.left, false) + calc(root.right, false);
    return flag ? Math.max(tempTrue, tempFalse) : tempFalse;
}
```

循环版本

```
// f存储当前节点被选中的最大值
private final Map<TreeNode, Integer> f = new HashMap<>();
// g存储当前节点未选中的值
private final Map<TreeNode, Integer> g = new HashMap<>();

public int rob(TreeNode root) {
    dfs(root);
    return Math.max(f.get(root), g.get(root));
}

public void dfs(TreeNode root) {
    if (root == null) return;
    dfs(root.left);
    dfs(root.right);
    f.put(root, root.val + g.getOrDefault(root.left, 0) + g.getOrDefault(root.right, 0));
    g.put(root, Math.max(f.getOrDefault(root.left, 0), g.getOrDefault(root.left, 0)) +
        Math.max(f.getOrDefault(root.right, 0), g.getOrDefault(root.right, 0)));
}
```

最佳买卖股票时机含冷冻期

力扣: <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/>

```
public int maxProfit(int[] prices) {
    int n = prices.length;
    // 持有股票的最大收益
    int[] f = new int[n];
    // 持有股票冷冻期的最大收益
    int[] g = new int[n];
    // 持有股票非冷冻期的最大收益
    int[] h = new int[n];
    f[0] = -prices[0];
    for (int i = 1; i < n; i++) {
        f[i] = Math.max(f[i - 1], h[i - 1] - prices[i]);
```

```

        g[i] = f[i - 1] + prices[i];
        h[i] = Math.max(h[i - 1], g[i-1]);
    }
    return Math.max(g[n - 1], h[n - 1]);
}

```

0-1背包问题

递归实现

```

private static int maxValue(int[] weightArr, int[] valueArr, int[] flag, int num, int weight) {
    if (num < 0) return 0;
    // 不偷第n个东西, f(n,c) = f(n-1,c)
    int noStealNumIndexObj = maxValue(weightArr, valueArr, flag, num - 1, weight);
    if (weight < weightArr[num] || flag[num] == 1)
        return noStealNumIndexObj;
    flag[num] = 1;
    // 偷第n个东西, f(n,c) = f(n,c-w[n])+v[n]
    int stealNumIndexObj =
        maxValue(weightArr, valueArr, flag, num, weight - weightArr[num]) + valueArr[num];
    flag[num] = 0;
    return Math.max(noStealNumIndexObj, stealNumIndexObj);
}

```

最长递增子序列

力扣300: <https://leetcode.cn/problems/longest-increasing-subsequence/>

```

public int lengthOfLIS(int[] nums) {
    if (nums == null || nums.length == 0) return 0;
    int n = nums.length;
    int[] res = new int[n];
    res[0] = 1;
    for (int i = 1; i < n; i++) {
        int max = 0;
        for (int j = 0; j < i; j++) {
            if (nums[j] < nums[i])
                max = Math.max(max, res[j]);
        }
        res[i] = max + 1;
    }
    return IntStream.of(res).boxed().max(Integer::compareTo).get();
}

```

<https://leetcode-cn.com/problems/wiggle-subsequence/>)

最长公共子序列

leetcode.1143 最长公共子序列

校验

```
public int longestCommonSubsequence(String text1, String text2) {  
    char[] s1 = text1.toCharArray();  
    char[] s2 = text2.toCharArray();  
    int m = s1.length;  
    int n = s2.length;  
    if (m == 0 || n == 0)  
        return 0;  
    return f1recursion(s1, m - 1, s2, n - 1);  
    return f2Loop(s1, m, s2, n);  
}
```

递归版：超出时间限制

```
private int f1recursion(char[] s1, int m, char[] s2, int n) {  
    if (m < 0 || n < 0)  
        return 0;  
    if (s1[m] == s2[n])  
        return 1 + f1recursion(s1, m - 1, s2, n - 1);  
    return Math.max(f1recursion(s1, m - 1, s2, n), f1recursion(s1, m, s2, n - 1));  
}
```

循环版

```
public int longestCommonSubsequence(String text1, String text2) {  
    char[] s1 = text1.toCharArray();  
    char[] s2 = text2.toCharArray();  
    int m = s1.length;  
    int n = s2.length;  
    if (m == 0 || n == 0)  
        return 0;  
    return f2Loop(s1, m, s2, n);  
}
```