Java

hashcode()和equals()

equals()相等的两个对象，hashcode()一定相等，换句话说hashcode不相等，equals一定不相等
去重复判断的时候先根据hashcode进行的判断，相同的情况下再根据equals()方法进行判断
重写equals()方法同时重写hashcode()方法，就是为了保证当两个对象通过equals()方法比较相等时，在Set中认为他们的hashCode值也一定要保证相等，这样才能映射到同一个位置，对原来的对象进行覆盖。

HashSet去重原理

去重原理：当hashset add一个元素A的时候，首先获取这个元素的散列码（hashcode的方法），即获取元素的哈希值。

情况一：如果计算出的元素的存储位置目前没有任何元素存储，那么该元素可以直接存储在该位置上。

情况二：如果算出该元素的存储位置目前已经存在有其他元素了，那么会调用该元素的equals方法与该位置的元素再比较一次，如果equals返回的值是true，那么该元素与这个位置上的元素就视为重复元素，不允许添加，如果equals方法返回的是false，那么该元素允许添加。

为什么单继承多实现

- 单继承
  如果为多继承时，当多个父类有重复的方法或属性时，子类调用就会含糊不清
  保证所有的对象具备某些功能，所有的对象很容易在堆上创建，参数传递得到了极大的简化
- 多实现
  实现类中必须重写接口的方法，调用的还是调用实现类的方法

接口和抽象类

相同点

（1）都不能被实例化 （2）接口的实现类或抽象类的子类都只有实现了接口或抽象类中的方法后才能实例化。

不同点

（1）接口只有定义，不能有方法的实现，java 1.8中可以定义default方法体，而抽象类可以有定义与实现，方法可在抽象类中实现。

（2）实现接口的关键字为implements，继承抽象类的关键字为extends。一个类可以实现多个接口，但一个类只能继承一个抽象类。所以，使用接口可以间接地实现多重继承。

（3）接口强调特定功能的实现，而抽象类强调所属关系。

接口的设计目的，是对类的行为进行约束，也就是提供一种机制，可以强制要求不同的类具有相同的行为。因为只要实现这个接口就意味着要实现接口的所有函数。

而抽象类的设计目的，是代码复用。当不同的类具有某些相同的行为(记为行为集合A)，且其中一部分行为的实现方式一致时（A的非真子集，记为B），可以让这些类都派生于一个抽象类。在这个抽象类中实现了B，避免让所有的子类来实现B，这就达到了代码复用的目的。而A减B的部分，留给各个子类自己实现。正是因为A-B在这里没有实现，所以抽象类不允许实例化出来（否则当调用到A-B时，无法执行）。

面向对象的特性

1、封装，隐藏对象的属性和实现细节，仅对外公开接口，使用者不必了解具体的实现细节，而只是要通过外部接口，以特定的访问权限来使用类的成员。

2、继承，继承就是子类继承父类的特征和行为，使子类继承父类的属性和方法，提高代码复用率。

3、多态，多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。**抽象类、接口的使用体现了多态性**

多态存在的**三个必要条件**， 简言之，**多态其实是在继承的基础上的。**

- 继承
- 重写（子类继承父类后对父类方法进行重新定义）
- 父类引用指向子类对象

4、抽象（Java的特性）

抽象包括两个方面：数据抽象和过程抽象。数据抽象也就是对象的属性。过程抽象是对象的行为特征。

面向对象和面向过程

面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

面向过程：性能比面向对象高，因为类调用时需要实例化，开销比较大

面向对象：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

泛型

解释：代码可以被不同类型的对象所重用，也就是说操作的数据类型是一个参数

类型安全。编译阶段确定了类型，消除了强制类型转换，在编译的时候检查类型安全

> 因为 Map.get() 被定义为返回 Object，所以一般必须将 Map.get() 的结果强制类型转换为期望的类型，如下面的代码所
> Map m = new HashMap();
> m.put(“key”, “blarg”);
> String s = (String) m.get(“key”);
> 要让程序通过编译，必须将 get() 的结果强制类型转换为 String，并且希望结果真的是一个 String。但是有可能某人已经在该映射中保存了不是 String 的东西，这样的话，上面的代码将会抛出 ClassCastException。

集合

List和Set继承自Collection接口。
Set无序不允许元素重复。HashSet和TreeSet是两个主要的实现类。
List有序且允许元素重复。ArrayList、LinkedList和Vector是三个主要的实现类。
Map也属于集合系统，但和Collection接口没关系。Map是key对value的映射集合，其中key列就是一个

集合。key不能重复，但是value可以重复。HashMap、TreeMap和Hashtable是三个主要的实现类。

- **HashMap** 基于**哈希表**
- **LinkedHashMap** 扩展了 HashMap，维护了一个贯穿所有元素的**双向链表**，保证按插入顺序迭代 https://blog.csdn.net/qq_40050586/article/details/105851970
- **TreeMap** 基于**红黑树**，保证**键**的**有序性**，迭代时按**键**大小的排序顺序

HashMap原理

HashMap是一个在日常开发中经常用到的集合类，它是以以键值对的形式进行存储，在jdk1.7和jdk1.8之间，HashMap的实现略有区别，在jdk1.7的时候HashMap使用的是数组+链表的结构，在jdk1.8的时候改成了数组+链表+红黑树，红黑树的引入是为了提高它的查询效率，因为链表的查询时间复杂度是O(N)，而红黑树的查询时间复杂度是O(logN), 还有就是在jdk1.7之前遇到hash碰撞的时候采用的是头插法，但是到了jdk1.8之后改成了尾插法，因为头插法在多线程的情况下会遇到一些问题，比如循环链表的问题。当然可能有更多的优化细节，这里我记不太清了，我们可能要去源码里面才能看的更透。

接下来我就从jdk1.8的hashmap版本跟您聊聊它的基本原理，首先我们在创建hashmap的时候，阿里规约（包命名小写，常量命名大写下划线分割，只要重写 equals，就必须重写 hashCode，@Transactional注解使用的时候，如果catch的异常需要回滚，则需要手动回滚）里面是要求我们传入一个初始化容量，就是预估的数据量，这个最好是一个二的次幂，如果不是二的次幂它会找一个最近的而且大于这个数的二的次幂进行初始化，比如传入14，它实际的初始化容量其实是16。

为什么这个要是二的次幂是因为在往hashmap里面Put的时候要传入hash值，这个hash值是通过这个容量（就是我们的2的n次幂-1，低位都是1）与上key的hashcode，算出它的下标，因为这个容量减一之后的低位都是1，高位都是0，与运算之后的结果一定在容量范围之内，不会产生越界异常。刚刚说的与运算在计算机的底层实现效率极高，只需要一条汇编就能实现。这也就是为什么我们不用取模运算计算下标的原因。

添加数据的时候会遇到两个问题，一个是扩容的问题，另一个是树化的问题。关于扩容问题，hashmap里面有一个成员变量叫加载因子，默认应该是0.75，当插入的节点数量大于容量乘以加载因子，它就会进行一次扩容。当链表上悬挂的节点数量大于等于8的时候它还会进行树化，当然这还不够，在hashmap里面还有一个成员变量就是最小的树化容量，这个一般是64，意思就是数组的容量达不到64它会优先选择扩容，而不是对链表进行树化。所以说树化有两个条件，一个是链表长度大于等于8，还有就是数组容量大于等于64。

关于hashmap我暂时想到的就这么多。

HashMap的线程不安全主要体现在下面两个方面：

1.在JDK1.7中，当并发执行扩容操作时会造成环形链和数据丢失的情况。
2.在JDK1.8中，在并发执行put操作时会发生数据覆盖的情况。

ConcurrentHashMap底层原理

JDK1.7
ConcurrentHashMap 内部进行了 Segment 分段，Segment 继承了 ReentrantLock，可以理解为一把锁，各个 Segment 之间都是相互独立上锁的，互不影响。相比于之前的 Hashtable 每次操作都需要把整个对象锁住而言，大大提高了并发效率，因为hashtable的方法都有synchronized修饰。默认有 0~15 共 16 个 Segment，所以最多可以同时支持 16 个线程并发操作（操作分别分布在不同的 Segment 上）

ReentrantLock的基本实现可以概括为：先通过CAS尝试获取锁。如果此时已经有线程占据了锁，那就加入AQS队列并且被挂起。当锁被释放之后，排在AQS队列队首的线程会被唤醒，然后CAS再次尝试获取锁。在这个时候，如果：

- 非公平锁：如果同时还有另一个线程进来尝试获取，那么有可能会让这个线程抢先获取；

- 公平锁：如果同时还有另一个线程进来尝试获取，当它发现自己不是在队首的话，就会排到队尾，由队首的线程获取到锁。

ReentrantLock默认非公平锁，但是可以在初始化的时候传入boolean值，通过CAS判断state是否是0（表示当前锁未被占用），是0则置为1，并且设置当前线程为该锁的独占线程，表示获取锁成功。当多个线程同时尝试占用同一个锁时，CAS操作只能保证一个线程操作成功。

JDK1.8
Node中的value和next都用volatile修饰，保证并发的可见性。
内部大量采用CAS操作，CAS是compare and swap的缩写。CAS在操作系统中通过一条指令来实现，所以他能保证原子性。CAS的三个操作数：内存地址（V）、预期值(A)、替换值(B)，内存地址V上的值与A值相等则用B替换A值返回true，不相等返回false。put的时候如果数组该位置为空，即预期值是null，用 CAS 操作将这个新值放入这个位置，如果 CAS 失败，说明此内存地址的值不是null，那就是有并发操作，已经有线程在这里放值了，继续下一个循环，这时候就会发生hash冲突，通过加锁保证线程安全，进行插入。

synchronized原理

synchronized关键字在经过Javac编译之后，会在同步块的前后形成一个monitorenter和两个monitorexit两个字节码指令。（因为一个线程对一个对象上锁了，后续就一定要解锁，第二个`monitorexit`是为了保证在**线程异常**时，也能正常**解锁**，避免造成**死锁**。）

在 Java 早期版本中，`synchronized`属于 **重量级锁**，效率低下，monitorenter和monitorexit实际上是java线程对操作系统线程的映射，所以每当挂起或者唤醒一个线程都要从用户态切换到内核态，这种操作是比较重量级的，有时切换时间甚至超过线程本身的执行时间，所以synchronized本身是非常消耗性能的，但是从jdk1.6开始，java引入了偏向锁，轻量级锁，重量级锁，在markword中...

一个对象刚开始实例化的时候，没有任何线程来访问它的时候。markword中偏向锁位是0，当第一个线程来访问它的时候，它会偏向这个线程，此时对象持有偏向锁偏向第一个线程，这个线程在修改对象头成为偏向锁的时候使用CAS操作，并将对象头中的ThreadID改成自己的ID，一旦有第二个线程访问这个对象，第二个线程可以看到对象的markword中偏向锁位是1，这时表明在这个对象上已经存在竞争了。若是某一时刻又来了线程二、线程三也想竞争这把锁，此时是轻度的竞争，便升级为**轻量级锁**，于是这三个线程就开始竞争了，他们就会去判断锁是否由释放，若是没有释放，没有获得锁的线程就会自旋，这就是**自旋锁。**这个竞争的过程的实质就是看谁能把自己的`ThreadID`贴在对象的`markword`中，而这个过程就是**CAS操作**，但是当自旋的线程超过一定的次数，轻量级锁膨胀为重量级锁，重量级锁使除了拥有锁的线程以外的线程都阻塞，防止CPU空转。

谈谈synchronized与ReentrantLock的区别?

① **底层实现**上来说，synchronized 是**Java关键字**，monitorenter与monitorexit，对象只有在同步块或同步方法中才能调用wait/notify方法，ReentrantLock 是**API层面**的锁。synchronized 的实现涉及到锁的升级，具体为无锁、偏向锁、自旋锁、向OS申请重量级锁，ReentrantLock实现则是通过CAS尝试获取锁。如果此时已经有线程占据了锁，那就加入AQS队列并且被挂起。当锁被释放之后，排在AQS队列队首的线程会被唤醒，然后CAS再次尝试获取锁。

② **是否可手动释放**：synchronized 不需要用户去手动释放锁，synchronized 代码执行完后系统会自动让线程释放对锁的占用； ReentrantLock则需要用户去手动释放锁，如果没有手动释放锁，就可能导致死锁现象。一般通过lock()和unlock()方法配合try/finally语句块来完成，使用释放更加灵活。

③**是否公平锁**synchronized为非公平锁 ReentrantLock则即可以选公平锁也可以选非公平锁，通过构造方法new ReentrantLock时传入boolean值进行选择，为空默认false非公平锁，true为公平锁。

④**锁的对象**synchronzied锁的是对象，锁是保存在对象头里面的，根据对象头数据来标识是否有线程获得锁/争抢锁；ReentrantLock锁的是线程，根据进入的线程和int类型的state标识锁的获得/争抢。

HashMap和HashTable区别

1）.HashTable的方法前面都有synchronized来同步，是线程安全的；HashMap未经同步，是非线程安全的。
2）.HashTable不允许null值(key和value都不可以)；HashMap允许null值(key和value都可以)。
3）.哈希值的使用不同，HashTable直接使用对象的hashCode； HashMap重新计算hash值，而且用与代替求模

ArrayList和vector区别

ArrayList和Vector都实现了List接口，都是通过数组实现的。
Vector是线程安全的，而ArrayList是非线程安全的。
List第一次创建的时候，会有一个初始大小，随着不断向List中增加元素，当List 认为容量不够的时候就会进行扩容。Vector缺省情况下自动增长原来一倍的数组长度，ArrayList增长原来的50%。

ArrayList和LinkedList

区别
ArrayList底层是用数组实现的，可以认为ArrayList是一个可改变大小的数组。随着越来越多的元素被添加到ArrayList中，其规模是动态增加的。
LinkedList底层是通过双向链表实现的， LinkedList和ArrayList相比，增删的速度较快。但是查询和修改值的速度较慢。同时，LinkedList还实现了Queue接口，就是调用remove()可以移除头，调用add()可以添加到队列尾。
使用场景
LinkedList更适合从中间插入或者删除（链表的特性）。
ArrayList更适合检索和在末尾插入或删除（数组的特性）。

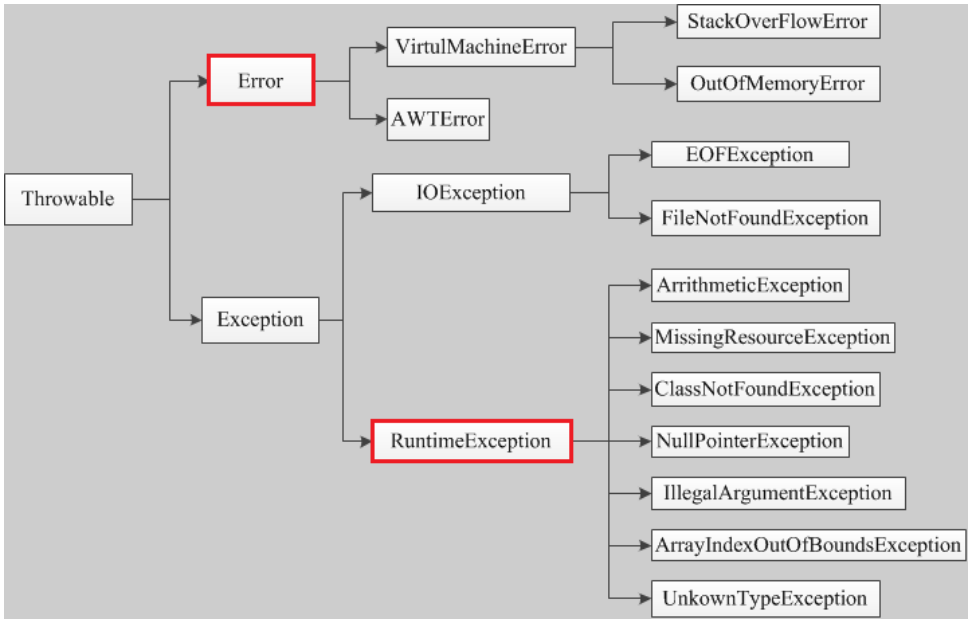
Error、Exception区别

Error类和Exception类的父类都是throwable类，他们的区别是：
Error类一般是指与虚拟机相关的问题，如系统崩溃，虚拟机错误，内存空间不足，方法调用栈溢等。对于这类错误的导致的应用程序中断，仅靠程序本身无法恢复和和预防，遇到这样的错误，建议让程序终止。
Exception类表示程序可以处理的异常，可以捕获且可能恢复。遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。

异常有哪几类

非检查异常（unckecked exception）：Error 和 RuntimeException 以及他们的子类。javac在编译时，不会提示和发现这样的异常，不要求在程序处理这些异常。数组索引越界异常，空指针异常等。

检查异常（checked exception）：除了Error 和 RuntimeException的其它异常。javac强制要求程序员为这样的异常做预备处理工作（使用try...catch...finally或者throws）否则编译不会通过，IOException,ClassNotFoundException 等



Collection 和 Collections

java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。
Collections则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

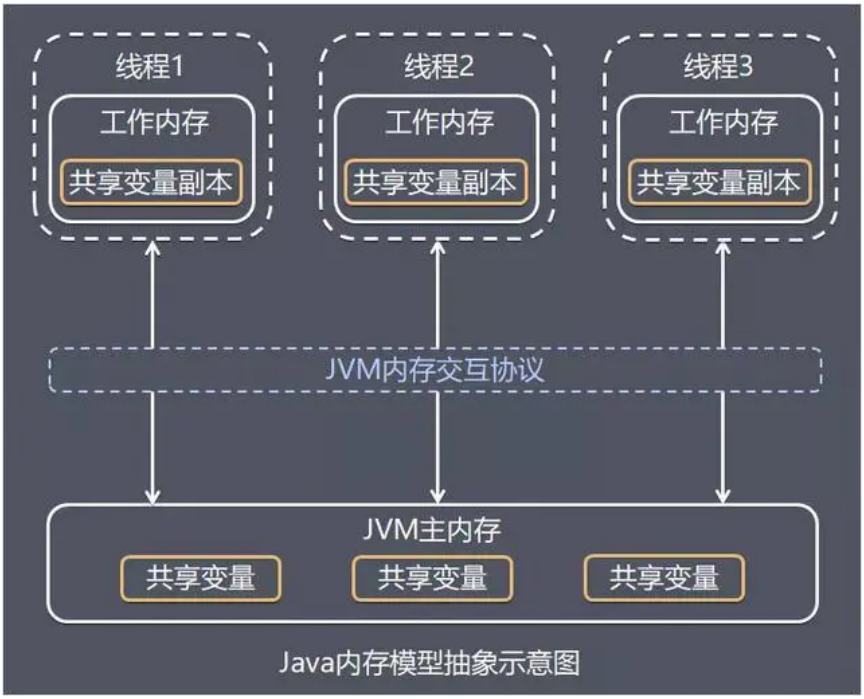
序列化与反序列化

把对象转换为字节序列的过程称为对象的序列化。
把字节序列恢复为对象的过程称为对象的反序列化。
对象的序列化主要有两种用途：

1. 把对象的字节序列永久地保存到硬盘上，通常存放在一个文件中；
2. 在网络上传送对象的字节序列，RPC。

Java内存模型（JMM）

JMM定义了主内存和线程之间抽象关系，线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本，如果说共享变量不加volatile关键字修饰，那么变量在线程之间是互相不可见的



关键字 volatile

1. 关键字 volatile 解决的是变量在多个线程之间的可见性，本质上就是采用了缓存一致性协议。通过CPU的嗅探机制，也就是监听到线程工作内存引用的变量值发生了改变，就会让缓存失效，也就是线程下一次需要重新从主存中读取，保证了主存和工作内存中变量的一致性。
2. 关键字 volatile 还有一个作用是确保代码的执行顺序不变。为了提高执行程序时的性能，编译器和处理器会对指令进行重排序优化，因此代码的执行顺序和编写代码的顺序可能不一致。添加关键字 volatile 可以禁止指令进行重排序优化。读前加读屏障，写后加写屏障

- 在每个volatile写操作前插入StoreStore屏障，这样就能让其他线程修改A变量后，把修改的值对当前线程可见，在写操作后插入StoreLoad屏障，这样就能让其他线程获取A变量的时候，能够获取到已经被当前线程修改的值

  在每个volatile读操作前插入LoadLoad屏障，这样就能让当前线程获取A变量的时候，保证其他线程也都能获取到相同的值，这样所有的线程读取的数据就一样了，在读操作后插入LoadStore屏障；这样就能让当前线程在其他线程修改A变量的值之前，获取到主内存里面A变量的的值。

NIO，BIO，AIO

BIO: 同步阻塞I/O，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器就需要启动一个线程进行处理

NIO: 同步非阻塞I/O，服务器实现模式为一个请求一个线程

AIO: 异步非阻塞I/O，服务器实现模式为一个有效请求一个线程，jdk1.7之后支持

同步：发送一个请求，等待返回，再发送下一个请求，同步可以避免出现死锁，脏读的发生
异步：发送一个请求，不等待返回，随时可以再发送下一个请求，可以提高效率，保证并发

线程阻塞状态

- 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。
- 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。
- 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

对线程安全的理解

在网上有另一种比较通俗的解释，如果多线程的程序的运行过程和单线程运行的结果一致，则认为它是线程安全的。 线程安全问题，通常都是由于对共享资源的竞争导致的，如果所有的线程都是读操作，而没有写操作，那么我们可以认为也是线程安全的，如果多个线程读写操作都有的话才会容易产生线程安全问题。这个时候我们就需要采取加锁，同步等手段来解决线程安全问题。

sleep()、wait()、join()、yield()的区别

锁池：所有需要竞争同步锁的线程都会放在锁池当中，比如当前对象的锁已经被其中一个线程得到，则其他线程需要在这个锁池进行等待当某个线程得到后会进入就绪队列进行等待cpu资源分配。

wait()当我们调用wait（）方法后，线程会放到等待池当中，等待池的线程是不会去竞争同步锁。只有调用了notify（）或notifyAll()后等待池的线程才会开始去竞争锁，notify（）是随机从等待池选出一个线程放到锁池，而notifyAll()是将等待池的所有线程放到锁池当中

wait()和sleep()

1、sleep 是 Thread 类的方法，wait 则是 Object 类的方法。（线程为了进入同步代码块内（临界区），需要获得锁，它们不需要知道哪些线程持有锁，它们只需要知道当前资源是否被占用，是否可以获得锁，所以锁的持有状态应该由同步监视器来获取，而不是线程本身）
2、sleep方法不会释放锁，但是wait会释放，而且会加入到等待队列中。
3、sleep不需要被唤醒（休眠之后推出阻塞），但是wait需要（不指定时间需要被别人中断）

yield（）执行后线程直接进入就绪状态，马上释放了cpu的执行权，但是依然保留了cpu的执行资格，所以有可能cpu下次进行线程调度还会让这个线程获取到执行权继续执行
join（）执行后线程进入阻塞状态，例如在线程B中调用线程A的join（），那线程B会进入到阻塞队列，直到线程A结束或中断线程

Java8的新特性

Lambda表达式和函数式接口

完善了Stream的API，流式计算，.map, .filter

Base64加入到了java官方库中

并行数组

java深拷贝与浅拷贝

浅拷贝是指：复制一个对象，如果属性是基本类型，拷贝的就是基本类型的值；如果属性是内存地址（引用类型），拷贝的就是内存地址 ，因此如果其中一个对象改变了这个地址，就会影响到另一个对象。
深拷贝是指：在拷贝引用类型成员变量时，为引用类型的数据成员另辟了一个独立的内存空间，实现真正内容上的拷贝。

java线程池

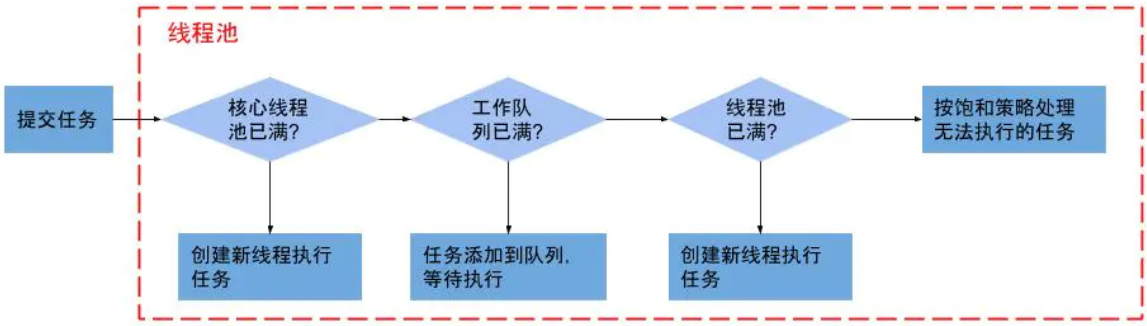
优势

- **降低资源消耗。**通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度。**当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- **提高线程的可管理性。**线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

参数

corePoolSize（线程池基本大小）
maximumPoolSize（线程池最大大小）
keepAliveTime（线程存活保持时间）当线程池中线程数大于核心线程数时，线程的空闲时间如果超过线程存活时间，那么这个线程就会被销毁，直到线程池中的线程数小于等于核心线程数
workQueue（任务队列）：用于传输和保存等待执行任务的阻塞队列
handler： 线程池对拒绝任务的处理策略

线程创建



1、判断核心线程池是否已满，没满则创建一个新的工作线程来执行任务。已满则。
2、判断任务队列是否已满，没满则将新提交的任务添加在工作队列，已满则。
3、判断整个线程池是否已满，没满则创建一个新的工作线程来执行任务，已满则执行饱和策略。

阻塞队列

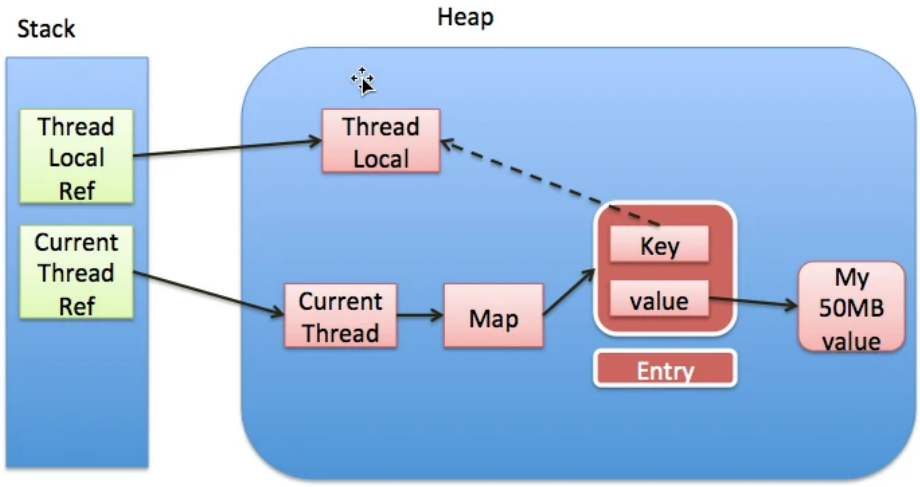
阻塞队列可以保证任务队列中没有任务时阻塞获取任务的线程，使得线程进入wait状态，释放cpu资源。当队列中有任务时才唤醒对应线程从队列中取出消息进行执行。使得在线程不至于一直占用cpu资源。

处理策略

ThreadPoolExecutor.AbortPolicy()　　---　抛出java.util.concurrent.RejectedExecutionException异常
ThreadPoolExecutor.CallerRunsPolicy()　　---　重试添加当前的任务，他会自动重复调用execute()方法
ThreadPoolExecutor.DiscardOldestPolicy()　---　抛弃旧的任务
ThreadPoolExecutor.DiscardPolicy()　　---　抛弃当前的任务

ThreadLocal

threadlocal可以理解为线程本地变量，每个线程的内部都维护了一个 ThreadLocalMap，这个map是一个数组，每个元素都是一个key-value的entry，这个key 是一个弱引用，也就是 ThreadLocal 本身，而 value 存的是线程变量的值，通过线性探测解决hash冲突。

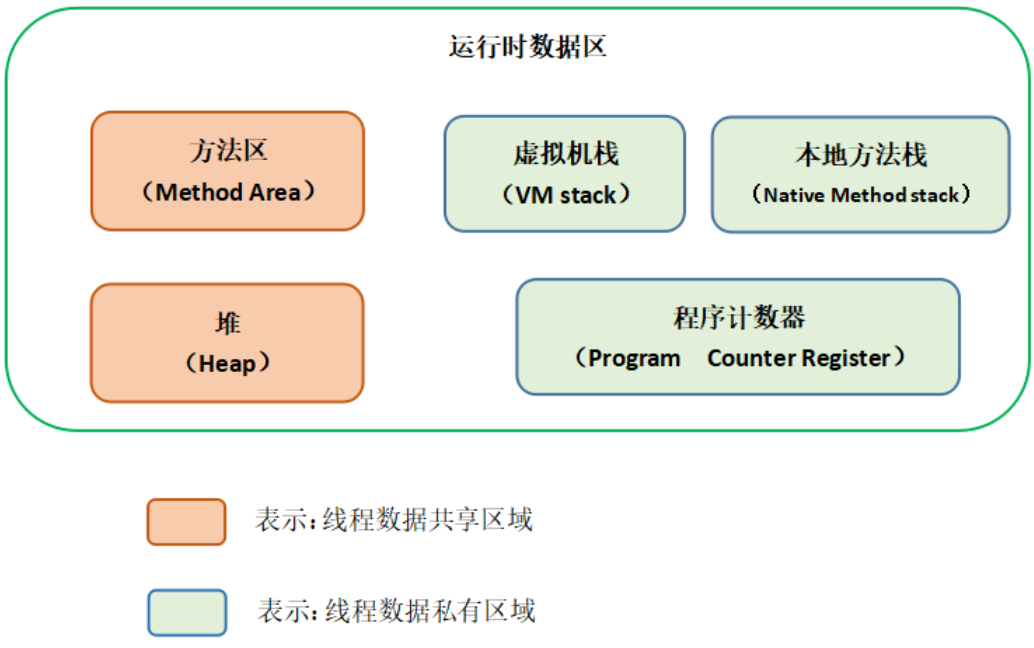


内存泄漏问题：key为弱引用，如果threadlocal被回收，那么key就变成了null，value就不会被获取到。可以调用remove()方法删除key为null的entry。（当一个对象只被弱引用实例引用（持有）时，这个对象就会被GC回收）

为什么要用弱引用：在方法中创建了threadlocal对象，当方法执行完之后，栈帧销毁，对threadlocal的引用没有了，但是此时我们的threadlocalmap的entry的key还指向这threadlocal对象，就不能被gc回收，但是这个时候threadlocal已经不用了。

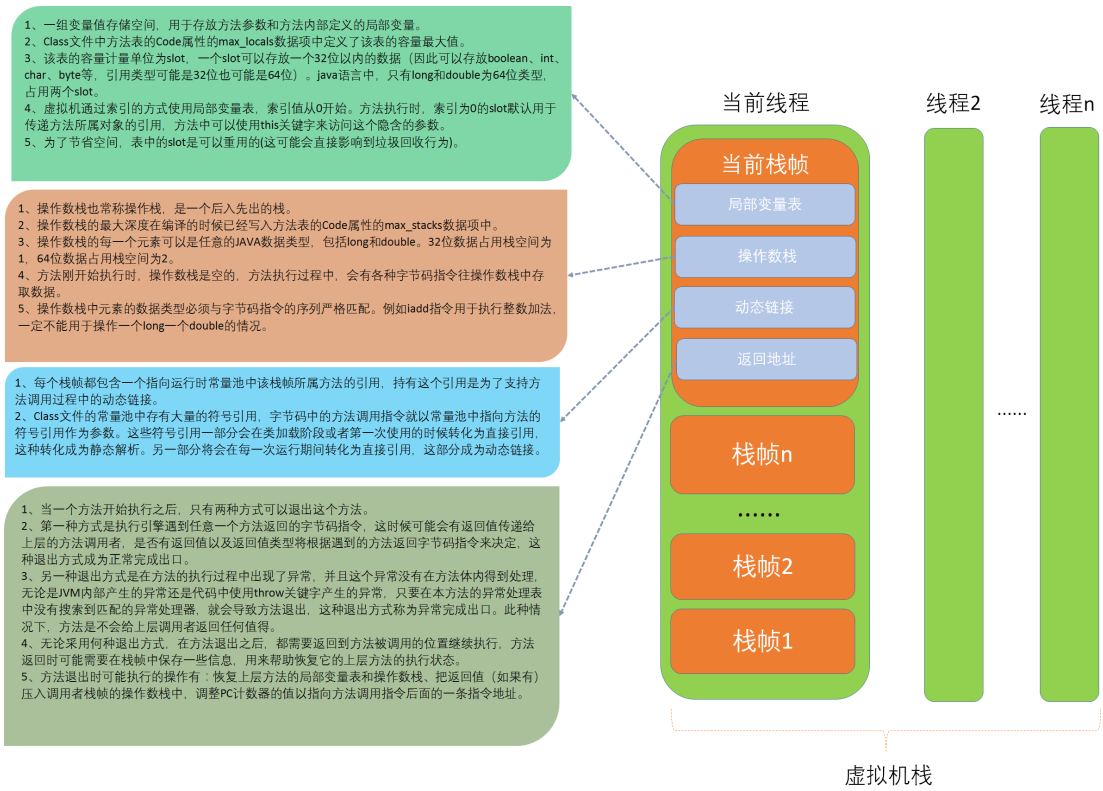
JVM

组成



栈帧包含：

- 局部变量表
- 操作数栈
- 返回地址
- 动态链接，指向该方法在运行时常量池中的位置



类的生命周期

- 加载，查找并加载类的二进制数据，在Java堆中也创建一个java.lang.Class类的对象
- 连接，连接又包含三块内容：验证、准备、初始化。 1）验证，文件格式、元数据、字节码、符号引用验证； 2）准备，为类的静态变量分配内存，并将其初始化为默认值； 3）解析，把类中的符号引用转换为直接引用
- 初始化，为类的静态变量赋值，执行static代码块
- 使用，new出对象程序中使用
- 卸载，执行垃圾回收

new一个对象的过程

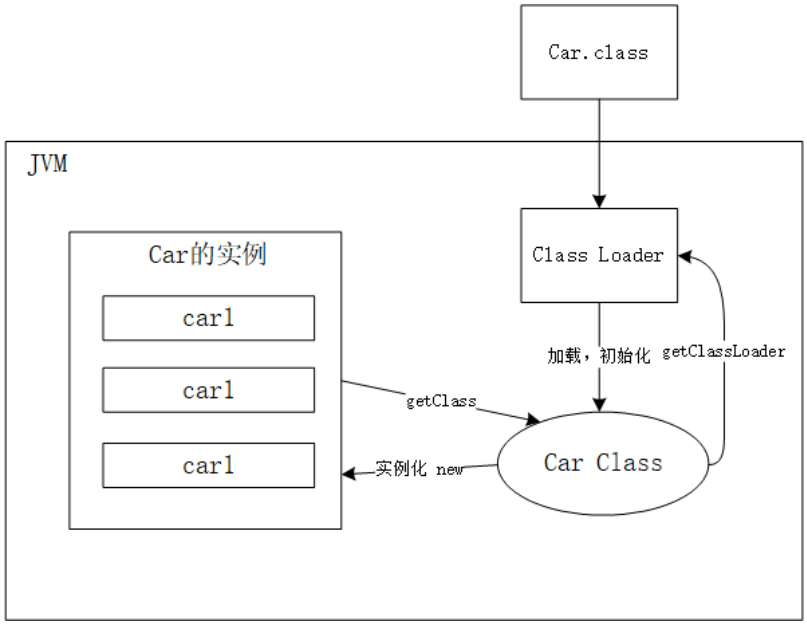
加载-连接-初始化

（1）通过双亲委派机制进行类加载，在Java堆中也创建一个Class类的对象

（2）验证字节码是符合规范的，初始化静态变量的初始值

（3）类初始化：为静态变量赋值，执行static代码块

再实例化：分配内存，执行实例初始化代码（先初始父类，再初始子类，执行构造方法），如果有类似于Child c = new Child()形式的c引用的话，在栈区定义Child类型引用变量c，然后将堆区对象的地址赋值给它



双亲委派机制

JVM中提供了三层的ClassLoader：

Bootstrap classLoader:主要负责加载核心的类库(java.lang.*等)

ExtClassLoader：主要负责加载jre/lib/ext目录下的一些扩展的jar。

AppClassLoader：主要负责加载应用程序的主函数类

当一个Hello.class这样的文件要被加载时。不考虑我们自定义类加载器，首先会在AppClassLoader中检查是否加载过，如果有那就无需再加载了。如果没有，那么会拿到父加载器ExtClassLoader，也会先检查自己是否已经加载过，如果没有再往上到达Bootstrap classLoader，检查是否加载过。此时已经没有父加载器了，这时候开始考虑自己是否能加载了，如果自己无法加载，会下沉到子加载器去加载，一直到最底层，如果没有任何加载器能加载，就会抛出ClassNotFoundException

为什么需要双亲委派机制?

①双亲委派机制使得类加载出现层级，父类加载器加载过的类，子类加载器不会重复加载，可以**防止类重复加载**；

②使得类的加载出现优先级，**防止了核心API被篡改**，提升了安全，所以越基础的类就会越上层进行加载，反而一般自己的写的类，就会在应用程序加载器（Application）直接加载。

本地方法栈

Java在内存区域中专门开辟了一块标记区域——本地方法栈，用来登记native方法，凡是带了native关键字的，会进入到本地方法栈中，调用本地方法接口（JNI），在最终执行的时候，加载本地方法库中的方法通过JNI

方法区

静态变量、常量、类信息(构造方法、接口定义)、运行时的常量池存在方法区中，但是实例变量存在堆内存中，和方法区无关。简单的来说就是：==static、final、Class、常量池==

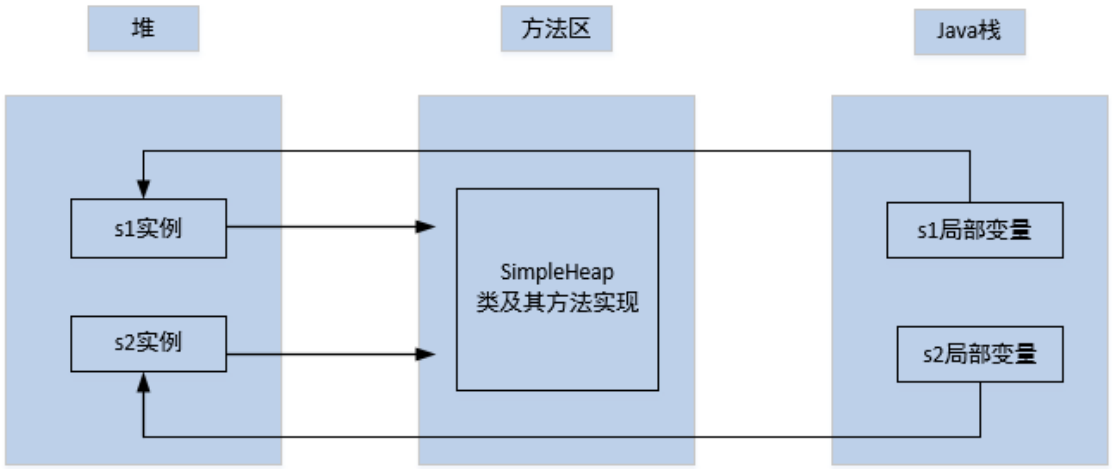
举例

```
public class SimpleHeap {
    private int id;
    public SimpleHeap(int id){
        this.id = id;
```

```
    }
    public void show(){
        System.out.println("My id is "+id);
    }

    public static void main(String[] args) {
        SimpleHeap s1 = new SimpleHeap(1);
        SimpleHeap s2 = new SimpleHeap(2);
        s1.show();
        s2.show();
    }
}
```

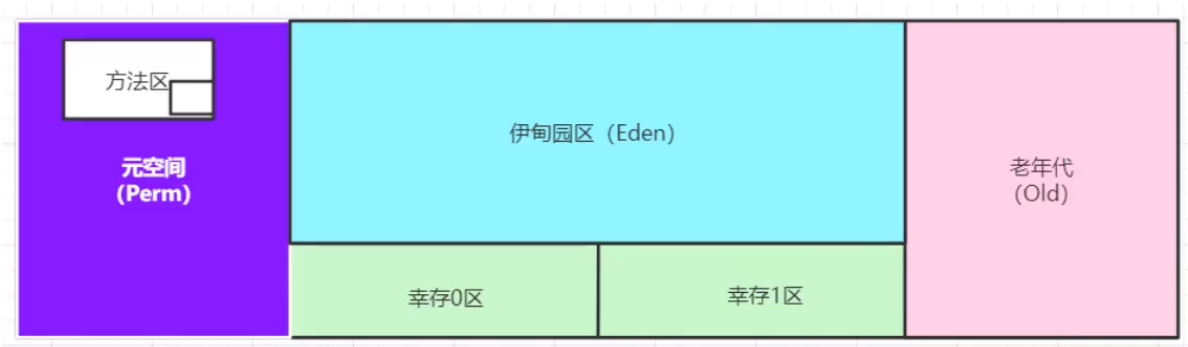
堆、方法区、java栈的关系



堆

堆内存中还要细分为三个区域：

- 新生区
- 养老区
- 元空间（方法区在元空间里）



内存分配策略

- 1. 对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 上分配，当 Eden 空间不够时，发起 Minor GC。

- 2. 大对象直接进入老年代

大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）

- 3. 长期存活的对象进入老年代

为对象定义年龄计数器，对象在 Eden 出生并经过 Minor GC 依然存活，将移动到 幸存区 中，年龄就增加 1 岁，增加到一定年龄则移动到老年代中。

- 4. 空间分配担保

在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的。

GC

GC垃圾回收，主要在**新生区**和**养老区**

Minor GC 和 Full GC

- Minor GC：回收新生代，因为新生代对象存活时间很短，因此 Minor GC 会频繁执行，执行的速度一般也会比较快。
- Full GC：回收老年代和新生代，老年代对象其存活时间长，因此 Full GC 很少执行，执行速度会比 Minor GC 慢很多。

Full GC 的触发条件

对于 Minor GC，其触发条件非常简单，当 Eden 空间满时，就将触发一次 Minor GC。而 Full GC 则相对复杂，有以下条件：

- 1. 调用 System.gc()

只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。不建议使用这种方式，而是让虚拟机管理内存。

- 2. 老年代空间不足

老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等。为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。

- 3. 空间分配担保失败

使用复制算法的 Minor GC 需要老年代的内存空间作担保，如果担保失败会执行一次 Full GC

GC算法

- 复制算法（新生代GC）

  它可以将内存分为大小相同的两块，每次只使用一块。当一块的内存使用完后，触发GC，将存活的对象复制到另一块内存中，然后把原来内存空间中的垃圾对象清理掉。问题：内存浪费

- 标记清除法（老年代GC）

  分为标记阶段和清除阶段；标记阶段使用可达性分析算法，从GCRoot触发遍历标记存活的对象。清除阶段回收所有未被标记的对象。问题：标记设计对象遍历，速度慢；标记清除之后产生内存碎片

- 标记整理算法

  分为标记阶段和清除阶段；标记阶段使用可达性分析算法，从GCRoot触发遍历标记存活的对象。整理阶段将存活对象向一端移动，然后清理掉端边界以外的内存

判断对象是否需要回收

- 可达性分析，通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连时，就认为不可达，对象需要回收

  GC Roots包含的对象：

  - 栈（栈帧中的本地变量表）中引用的对象
  - 方法区中的静态变量、常量所引用的对象；
  - 本地方法栈中Native方法引用的对象

  Why?

  - 栈是线程私有，其最小单位是栈帧，在其中引用的对象即当前线程正在使用的对象
  - 经过类的加载阶段，会在方法区中形成关于该类的数据结构，包括类的静态属性引用及常量引用的对象，他们都是线程共享的，在当前线程访问他们引用的对象时，就必须存在

垃圾收集器

- Serial
  - 单线程的垃圾收集器
  - 垃圾收集过程中STW，停止其他全部线程，直到垃圾收回收结束
  - 年轻代使用复制算法、老年代（Serial Old）使用标记整理算法
- CMS
  - 回收老年代，使用标记清除算法
  - 1.初始标记只标记GCROOTs，需要STW，这个过程很快 2.遍历对象图，比较耗时，但与用户线程并行 3.重新标记，因为并发标记是会出现漏标的问题，CMS使用增量更新，再以新增的黑色对象为跟扫描一次。4.并发清除，与用户线程并行
  - CMS不会等到老年代被填满再被收集，因为CMS无法处理浮动垃圾，浮动垃圾是因为之前兵法标记阶段标为黑色的对象本轮GC不会再被回收，只能留在下一次GC再清除，但垃圾收集阶段用户线程还在进行，要预留内存空间提供给用户线程。它会设定一个阈值，jdk6的默认阈值是92%，超过这个阈值会执行gc
- G1
  - G1将内存划分成了多个大小相等的Region（默认是512K），物理内存地址不连续，Eden、Survivor、Old、Humongous，**当分配的对象大于等于Region大小的一半**的时候就会被认为是Humongous，默认分配在老年代，可以防止GC的时候大对象的内存拷贝
  - 通过引入 Region 的概念，从而将原来的一整块内存空间划分成多个的小空间，使得每个小空间可以单独进行垃圾回收。通过记录每个 Region 垃圾回收时间以及回收所获得的空间（这两个值是通过过去回收的经验获得），并维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。
  - 每个 Region 都有一个 Remembered Set，用来记录该 Region 对象的引用对象所在的 Region。通过使用 Remembered Set，在做可达性分析的时候就可以避免全堆扫描
  - 并发标记阶段，当灰色对象要删除指向白色对象的引用关系时，会把这种删除记录并保存在一个队列里，在重新标记阶段会扫描这个队列，通过这种方式，旧的引用所指向的对象就会被标记上，当然这时可能有浮动垃圾，允许其存在

三色标记问题

- 白色：未被扫描过，一开始全是白色，如果最后还是白色，证明没被访问过
- 黑色：已经被访问过，所有的引用都扫描过
- 灰色：至少有一个引用的对象没被扫描过

浮动垃圾：在并发标记过程，如果方法运行结束，栈帧销毁，部分局部变量（GCRoot）被销毁，但这个对象之前已经被标记为黑色，黑色又不会被二次扫描，本轮GC不会回收。不会影响垃圾回收的正确性，只是需要等到下一轮垃圾回收中才被清除。

漏标：会导致被引用的对象被当做垃圾误删除，这是严重bug,必须解决。有两种方案：**增量更新、原始快照（SATB）**

- 增量更新就是当黑色对象插入新的指向白色对象的引用关系，就将这个新插入的引用记录下来。等并发扫描结束之后，再将这些记录过的引用关系中的黑色对象为根，重新扫描一次。CMS
- 原始快照（SATB）就是当灰色对象要删除指向白色对象的引用关系时，将这个要删除的引用记录下来，让它活过这次GC，当然它可能确实已经死了。在并发扫描结束后，在将这些记录过的引用关系中的灰色对象为根，重新扫描一次。G1

为什么G1用SATB？CMS用增量更新?

增量更新在进行并发标记的时候，记录的是新增关系，如果新增记录的黑色节点是一个很发散的树的树根，遍历时间就会很长。G1的作用范围是整个堆空间，CMS只是老年代，所以增量更新更适合CMS。SATB记录的是删除关系，是把删除记录中的灰色再重新扫描，不太会遍历很长时间。当然这时可能有浮动垃圾，G1允许存在，下次GC再清除

操作系统

fork()发生了什么

fork 函数调用之后，就会为子进程创建一个新的 pcb(process control block)，但这时**还不会复制父进程的代码段和数据段**，只是为子进程复制一份父进程的页表，而且这个页表是只读的。如果父子进程都对内存进行只读操作，那么这个状态就可以一直保持下去，但一旦有一方想要对内存进行写操作，就会引发**缺页异常**，这是就会进行写时复制，内核会为子进程申请一个新的物理页，并将原物理页中的内容真正的复制到新的物理页中，让父子进程真正拥有各自的物理内存页，此时他们各自内存中的内容就是可写的了。
fork()返回值：在父进程中，fork返回新创建子进程的进程ID，在子进程中，fork返回0，如果出现错误，fork返回一个负值

逻辑地址->物理地址

逻辑地址是页号+页内偏移，逻辑地址除以pageSize，商就是页号，余数就是页内偏移。每个进程都有一个页表，记录了页号与物理块号的映射，这时我们可以根据页号找到物理块号，后面几位再补上之前计算的页内偏移就能得到物理地址

线程和进程的区别

1. 进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位
2. 一个进程可以包含多个线程
3. 线程之间共享进程的堆中的信息，每个线程有独立的线程 ID，程序计数器，寄存器组和堆栈。进程之间则相互独立

4. 为什么进程切换比线程切换更慢？进程切换导致页表和TLB需要刷新，进而导致程序执行时的缺页异常，频繁陷入内核由page_fault处理函数去处理，处理完回到用户态继续执行。

进程间通信（IPC）

共享内存，管道（半双工），SIGNAL(kill -9 pid)，信号量，socket

线程间通信

1. volatile
2. wait(), notify()机制
3. join()方式

软连接和硬链接

硬链接文件是具有相同inode节点号的不同文件，相当于复制了一份源文件，更新任何一份，另一份都会更新
软连接和源文件是不同类型的文件，也是不同的文件，inode号不同，相当于创建了快捷方式，删除源文件，软连接文件依然存在，但是无法访问指向的路径内容

死锁产生的必要条件

互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。（占有的可以释放可以解决这个问题）

循环等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，…，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。（对申请资源进行编号，递增变换，那么一定没有环）

虚拟内存

操作系统会给每个进程4G的虚拟内存，进程会认为自己有4G的连续地址空间，当然实际上这4G有些在主存里面，有些在磁盘上。而且它也不是连续的，我们需要通过页表才能找到对应的物理地址。虚拟内存有一个demand paging的机制，简单来说就是按需加载，当程序执行过程中访问的信息不在内存时，触发缺页中断，把需要的数据从磁盘读入主存，还可以根据替换策略进行替换，常用的替换策略时LRU，FIFO

内核态和用户态切换

a. 系统调用

这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中fork()实际上就是执行了一个创建新进程的系统调用。

b. 异常

当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

c. 外围设备的中断

当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等（DMA）。

僵尸进程和孤儿进程

僵尸进程是当子进程比父进程先结束，而父进程没有并未调用 wait()回收子进程，此时子进程将成为一个僵尸进程。在每个进程退出的时候，内核释放该进程所有的资源。但是仍然保留了一些信息（如进程号 pid等）。这些保留的信息直到进程通过调用 wait()放。这样就导致了一个问题，如果没有调用 wait()的话，那么保留的信息就不会释放。比如进程号就会被一直占用了。

孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程(进程号为1)管理。

Linux命令

```
netstat -nap | grep 进程id # 通过进程id查看占用的端口
netstat -a       # 列出所有端口
netstat -at      # 列出所有TCP端口
# 怎么在某个目录下找到包含 txt 的文件
find /home -name "*.txt"
# 监控Linux的系统状况，比如CPU、内存的使用
top
# 判断一个主机是不是开放某个端口
telnet 127.0.0.1 3389
#列出所有打开的文件:
lsof
#递归查看某个目录的文件信息
lsof +D /filepath/filepath2/
```

定位排查CPU占用率过高的问题

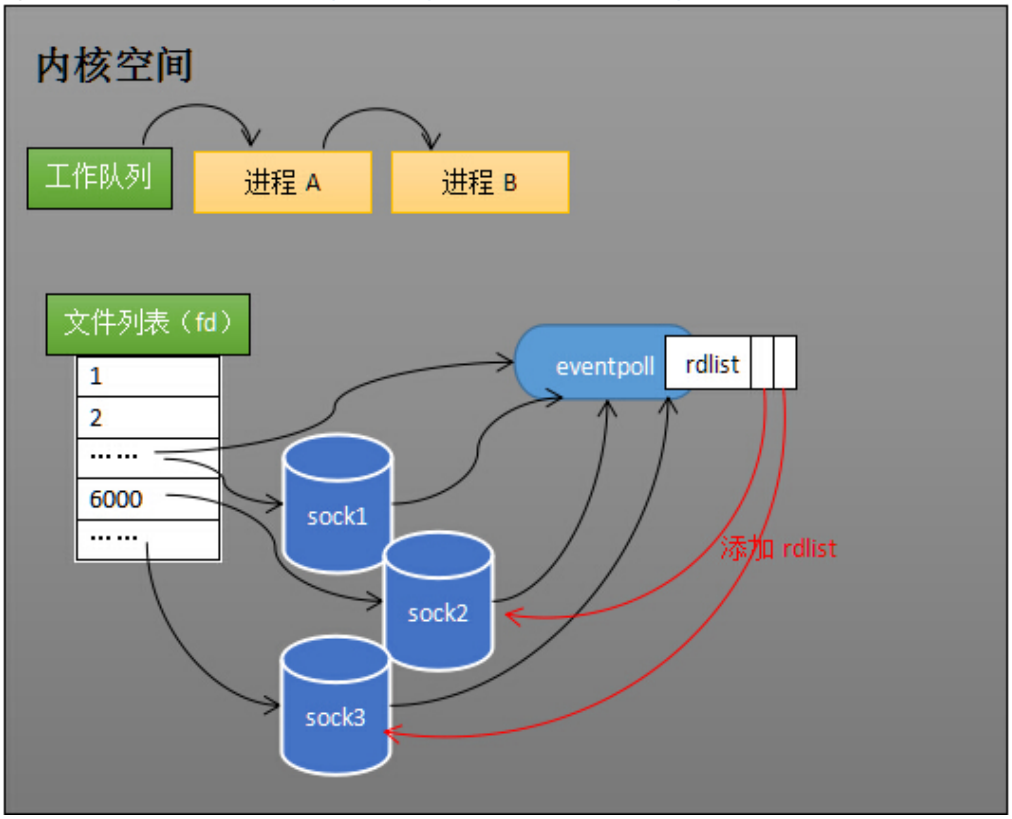
1. 使用top命令，找出占用cpu比例较高的进程
2. 使用top -Hp pid，找出pid进程中占用率较高的线程
3. jstack命令可以查看线程栈，查看哪个方法出了问题

硬中断和软中断

硬中断是由硬件产生的，比如，像磁盘，网卡，键盘，时钟等。每个设备或设备集都有它自己的中断请求
软中断是由由当前正在运行的进程所产生的，比如I/O请求

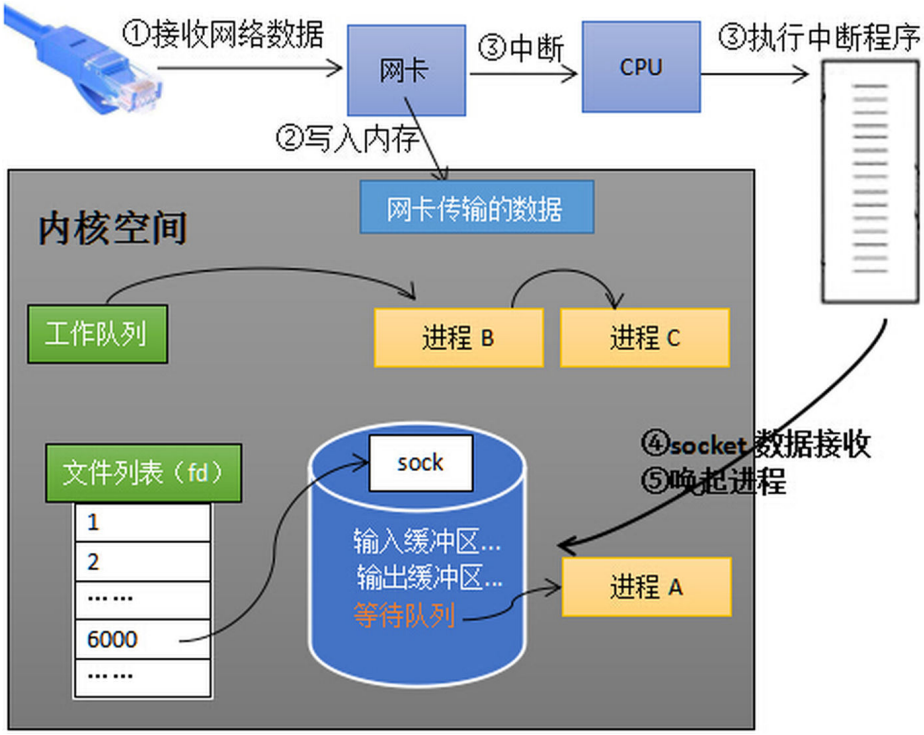
IO模型

- 阻塞IO：服务器调用recvfrom读取数据时，内核直到数据包到达并将数据复制到应用进程的缓冲区之后才返回，在此期间一直等待，进程从调用到返回这段时间内都是被阻塞的成为阻塞IO
- 非阻塞IO：服务器调用recvfrom读取数据时，如果socket接收缓冲区没有数据的话，就会直接返回一个错误标识，不会让应用一直等待中。服务器隔一段时间再调用recvfrom请求，直到读取到它数据要的数据为止。
- 多路复用IO：通过select/poll/epoll 来监控多fd，阻塞在选择器上，选择器帮我们侦测多个fd是否准备就绪，当有fd准备就绪时，选择器返回数据可读状态，应用程序再调用recvfrom读取数据。

- select: 有一个long类型的数组fd_set，fd_set存放着所有需要监视的socket文件描述符。然后调用select，如果fd_set中的所有socket都没有数据，select会阻塞，直到有一个socket接收到数据，select返回，唤醒进程，把进程从等待队列中移除，转移到工作队列里面。缺点：每次调用select，都需要把`fd_set`数组从用户区拷贝到内核区，这个开销也很大；同时每次调用select都需要在内核遍历fd_set，这个开销也很大；对数组大小做了限制。
- poll: 跟select大致相似，底层把数组变成了链表，没有了大小限制
- epoll: 当进程调用epoll_create方法时，内核创建一个eventpoll对象，通过epoll_ctl()添加或删除所要监听的socket（红黑树），当监听的socket收到数据，触发中断给eventpoll对象的就绪列表添加socket引用（双向链表），当程序执行epoll_wait的时候，如果eventpoll的就绪列表里面有socket引用就可以直接返回了，因为这个就绪队列的存在，进程知道哪些socket发生了变化。
  - epoll有哪两种触发方式：https://blog.csdn.net/sunshixingh/article/details/50988109
  - 

- 异步IO：应用只需要向内核发送一个read 请求，内核收到请求后会建立一个信号联系，等所有操作都完成之后，内核会发起一个通知告诉应用

进程在recv阻塞期间，计算机收到了对端传送的数据（步骤①）。数据经由网卡传送到内存（步骤②），然后网卡通过中断信号通知cpu有数据到达，cpu执行中断程序（步骤③）。此处的中断程序主要有两项功能，先将网络数据写入到对应socket的接收缓冲区里面（步骤④），再唤醒进程A（步骤⑤），重新将进程A放入工作队列中



协程

协程运行在线程之上，当一个协程执行完成后，可以选择主动让出，让另一个协程运行在当前线程之上。**协程并没有增加线程数量，只是在线程的基础之上通过分时复用的方式运行多个协程**，而且协程的切换在用户态完成，切换的代价比线程从用户态到内核态的代价小很多。

上下文切换

- 进程的上下文切换
  就是先把前一个任务的 CPU 上下文（也就是 CPU 寄存器和程序计数器）保存起来，然后加载新任务的上下文到这些寄存器和程序计数器，最后再跳转到程序计数器所指的新位置，运行新任务。而这些保存下来的上下文，会存储在系统内核中，并在任务重新调度执行时再次加载进来。这样就能保证任务原来的状态不受影响，让任务看起来还是连续运行。
- 线程的上下文切换

  线程共享进程的虚拟内存，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据
- 区别：进程切换涉及到虚拟地址空间的切换而线程切换则不会。因为每个进程都有自己的虚拟地址空间，而线程是共享所在进程的虚拟地址空间的，因此同一个进程中的线程进行线程切换时不涉及虚拟地址空间的转换。

Linux

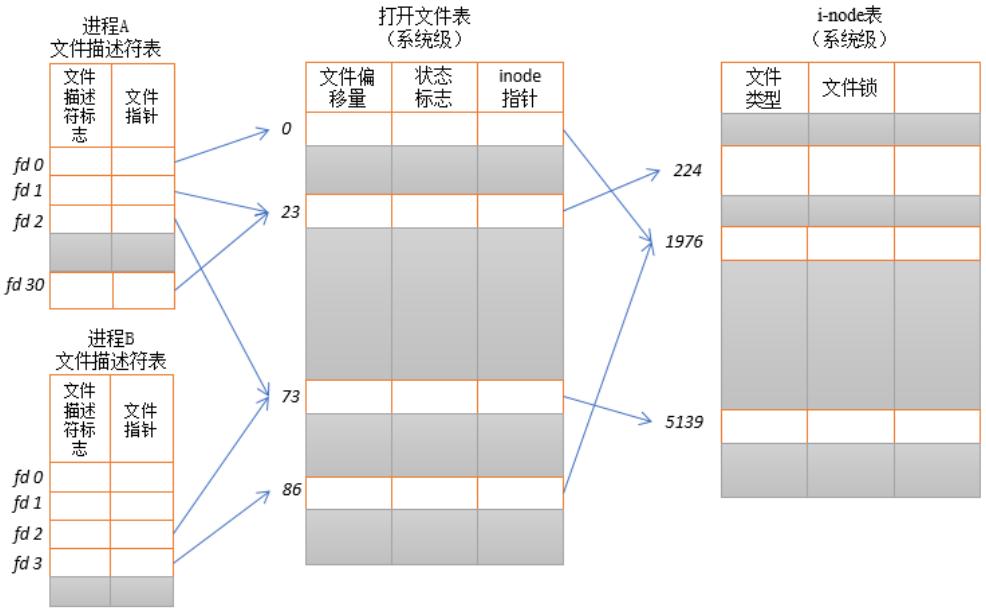
文件系统

- 进程级的文件描述符表
- 系统级的打开文件描述符表
- 文件系统的i-node表

文件描述符表的文件指针指向打开文件表的打开文件句柄，句柄包含文件的inode指针，通过这个指针就可以找到数据区的文件数据

文件描述符（file descriptor）是内核为了高效管理已被打开的文件所创建的索引，每一个文件描述符会与一个打开文件相对应，不同的文件描述符也会指向同一个文件。系统为每一个进程维护了一个文件描述符表。

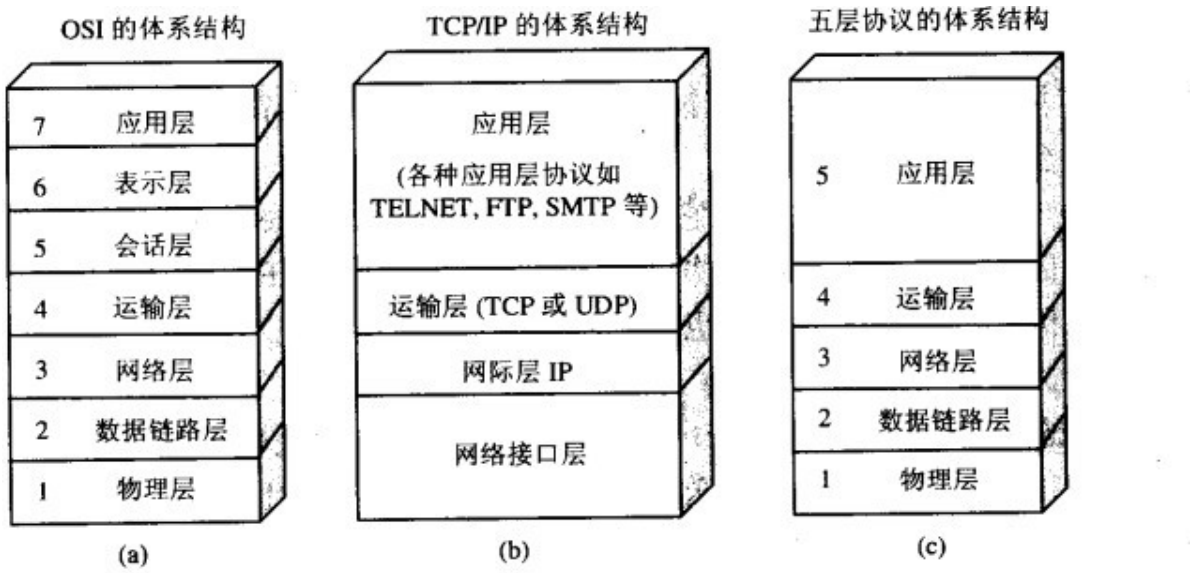
内核对所有打开的文件的文件维护有一个系统级的open file table，包括文件偏移量，inode指针，文件权限，文件大小



文件描述符、打开文件句柄和i-node之间的关系图

计算机网络

体系结构



计算机网络体系结构：(a) OSI 的七层协议；(b) TCP/IP 的四层协议；(c) 五层协议

TCP三次握手

TCP是面向连接的可靠传输，三次握手的目的是建立可靠的通信信道，确认**双方发送和接收信号都是正常的**

- 客户端–发送带有 SYN 标志的数据包–一次握手–服务端
- 服务端–发送带有 SYN/ACK 标志的数据包–二次握手–客户端
- 客户端–发送带有带有 ACK 标志的数据包–三次握手–服务端

衍生：传了 SYN,为啥还要传 ACK？答：传了 SYN，证明发送方到接收方的通道没有问题，但是接收方到发送方的通道还需要 ACK 信号来进行验证。

TCP四次挥手

- 客户端-发送一个 FIN，用来关闭客户端到服务器的数据传送
- 服务器-收到这个 FIN，它发回一 个 ACK，确认序号为收到的序号加1 。和 SYN 一样，一个 FIN 将占用一个序号
- 服务器-关闭与客户端的连接，发送一个FIN给客户端
- 客户端-发回 ACK 报文确认，并将确认序号设置为收到序号加1

为什么要四次挥手？答：任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。不确定另一方是不是还要发送数据。

TCP和UDP

UDP 在传送数据之前不需要先建立连接，远地主机在收到 UDP 报文后，不需要给出任何确认。虽然 UDP 不提供可靠交付，但在某些情况下 UDP 确是一种最有效的工作方式（一般用于即时通信），比如： QQ 语音、 QQ 视频

TCP 提供面向连接的服务。在传送数据之前必须先建立连接，数据传送结束后要释放连接。于 TCP 要提供可靠的，面向连接的传输服务（TCP的可靠体现在TCP在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用来节约系统资源），这一难以避免增加了许多开销。TCP 一般用于文件传输、发送和接收邮件、远程登录等场景。

TCP如何保证可靠传输

1. 三次握手保证面向连接，保证了客户端和服务端都能正常的接收和发送数据
2. TCP 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。如果编号不连续则不接收，导致超时重传
3. 校验和： TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。
4. 拥塞控制： 当网络拥塞时，减少数据的发送。**拥塞控制就是为了防止过多的数据注入到网络中**，Reno算法：发送方要维持一个拥塞窗口(cwnd) 的状态变量，拥塞控制窗口的大小取决于网络的拥塞程度，并且动态变化。一开始是采用慢开始，指数形式增大拥塞窗口，到达阈值之后线性增加，每次拥塞窗口+1，如果发送端收到连续三个相同的ACK，证明信道太拥塞，发生了丢包，这时候就把拥塞窗口减半，继续发送，每次拥塞窗口+1。但其实发送数据包的数量不只取决于拥塞窗口的大小，还取决于接收窗口的大小，两个窗口取最小的那个。
5. ARQ协议： 也是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。
6. 超时重传： 当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

从输入URL到网页呈现的过程

1、域名解析
当我们在浏览器中输入一个URL，例如"www.google.com"时，这个地址并不是谷歌网站真正意义上的地址。互联网上每一台计算机的唯一标识是它的IP地址，因此我们输入的网址首先需要先解析为IP地址，这一过程叫做DNS解析。

浏览器缓存->操作系统缓存->本地host文件->(操作系统发送域名到本地域名服务器->根域名服务器->顶级域名服务器)->存入缓存

2、TCP连接
HTTP协议是使用TCP协议作为其传输层协议的，在拿到服务器的IP地址后，浏览器客户端会与服务器建立TCP连接。该过程包括三次握手：

第一次握手：建立连接时，客户端向服务端发送请求报文
第二次握手：服务器收到请求报文后，如同意连接，则向客户端发送确认报文
第三次握手，客户端收到服务器的确认后，再次向服务器给出确认报文，完成连接。
三次握手主要是为了防止已经失效的请求报文字段发送给服务器，浪费资源。

3、浏览器发送HTTP请求
浏览器构建http请求报文，并通过TCP协议传送到服务器的指定端口。http请求报文一共包括三个部分：

请求行：指定http请求的方法、url、http协议版本等
请求头：描述浏览器的相关信息，语言、编码等。如下

请求正文：当发送POST, PUT等请求时，通常需要向服务器传递数据。这些数据就储存在请求正文中。

4、服务器处理HTTP请求
服务器处理http请求，并返回响应报文。响应报文包括三个部分：

- 状态码
- 响应头：包含了响应的相关信息，如日期等
- 响应正文：服务器返回给浏览器的文本信息，通常的html、js、css、图片等就包含在这一部分里面。

5、浏览器页面渲染
浏览器接受到http服务器发送过来的响应报文，并开始解析html文档，渲染页面。具体的渲染过程包括：构建DOM树、构建渲染树、定位页面元素、绘制页面元素等。具体可参看：浏览器时如何渲染页面的

6、断开TCP连接
客户端与服务器四次挥手，断开tcp连接。

第一次挥手：客户端想分手，发送消息给服务器
第二次挥手：服务器通知客户端已经接受到分手请求，但还没做好分手准备
第三次回收：服务器已经做好分手准备，通知客户端
第四次挥手：客户端发送消息给服务器，确定分手，服务器关闭连接

Cookie和Session

Session是在服务器保存的一个数据结构，用来记录用户的状态，这个数据可以保存在数据库、文件中；Cookie保存在客户端，是浏览器（客户端）保存用户信息的一种机制，用来记录用户的一些信息（偏好设置，用户名密码等）。

服务端如何识别特定的客户：

服务器第一次接收到请求时，开辟了一块Session空间（创建了Session对象），同时生成一个Session id，并通过响应头的Set-Cookie：“JSESSIONID=XXXXXXX”命令，向客户端发送要求设置cookie的响应； 客户端收到响应后，在本机客户端设置了一个JSESSIONID=XXXXXXX的cookie信息，该cookie的过期时间为浏览器会话结束；

接下来客户端每次向同一个网站发送请求时，请求头都会带上该cookie信息（包含Session id）；然后，服务器通过读取请求头中的Cookie信息，获取名称为JSESSIONID的值，得到此次请求的Session id；Session id 相同即认为是同一个会话。当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。

Appendix：
Session占用服务器性能，Session过多，增加服务器压力，所以服务器往往设置一个生命周期减小压力，当登录成功后xx小时之内没有对该服务器进行任何HTTP请求，删除session，之后提示重新登陆，再重新创建session

Cookie 被禁用怎么办?

最常用的就是利用 URL 重写把 Session ID 直接附加在URL路径的后面。

HTTP长连接,短连接

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个HTML或其他类型的Web页中包含有其他的Web资源（如JavaScript文件、图像文件、CSS文件等），每遇到这样一个Web资源，浏览器就会重新建立一个HTTP会话。

而从HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头加入这行代码：

```
Connection:keep-alive
```

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

URI和URL的区别是什么

- URI(Uniform Resource Identifier) 是统一资源标志符，可以唯一标识一个资源。
- URL(Uniform Resource Location) 是统一资源定位符，可以提供该资源的路径。它是一种具体的 URI，即 URL 可以用来标识一个资源，而且还指明了如何 locate 这个资源。

URI的作用像身份证号一样，URL的作用更像家庭住址一样。URL是一种具体的URI，它不仅唯一标识资源，而且还提供了定位该资源的信息。

HTTP 和 HTTPS 的区别

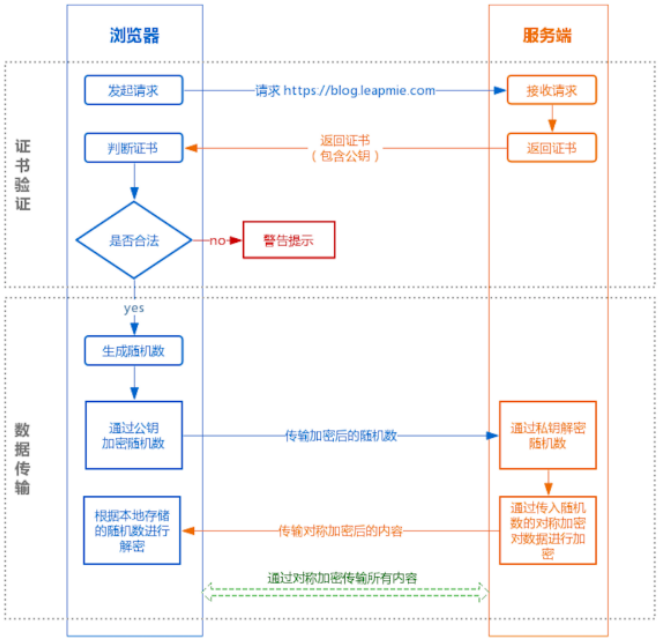
1. 端口： HTTP的URL由“http://”起始且默认使用端口80，而HTTPS的URL由“https://”起始且默认使用端口443。
2. 安全性和资源消耗： HTTP协议运行在TCP之上，所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。HTTPS是运行在**SSL/TLS**之上的HTTP协议，SSL/TLS 运行在TCP之上。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。

HTTPS

1. 非对称加密获取密钥
2. 利用这个密钥进行对称加密的数据传输

如果我们想要安全地创建一个对称加密的密钥，可以让客户端这边来随机生成，但是生成出来的密钥不能直接在网络上传输，而是要用网站提供的公钥对其进行非对称加密。而网站在收到消息之后，只需要使用私钥对其解密，就获取到浏览器生成的密钥了。

只有在浏览器和网站首次商定密钥的时候需要使用非对称加密，一旦网站收到了浏览器随机生成的密钥之后，双方就可以都使用对称加密来进行通信了，因此工作效率是非常高的。



① 证书验证阶段

(1)浏览器发起HTTPS请求，要求与Web服务器建立SSL连接

(2)服务端返回HTTPS证书：Web服务器收到客户端请求后，会生成一对公钥和私钥，并把公钥放在证书中发给客户端浏览器

(3)客户端验证证书是否合法，如果不合法则提示告警

② 数据传输阶段

(1)当证书验证合法后，在本地生成随机数

(2)通过公钥加密随机数，并把加密后的随机数传输到服务端

(3)服务端通过私钥对随机数进行解密

(4)服务端通过客户端传入的随机数构造对称加密算法，对返回结果内容进行加密后传输

Docker 2376

配置TLS认证的远程端口的证书

在远程服务器生成 CA 证书，服务器证书，服务器密钥，然后自签名，再颁发给需要连接远程 docker 容器的服务器

TIME_WAIT, CLOSE_WAIT

- TIME_WAIT：主动关闭连接的一方发完ACK之后进入的状态，保持这个状态两个MSL(max segment lifetime)
  - 维持的原因：1、发完最后一个ack，接收方可以没有收到，会再发一个FIN；2、让重复的包都消失
  - 表示开启TCP连接中TIME-WAIT sockets的快速回收，默认关闭　net.ipv4.tcp_tw_recycle = 1
  - 表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认关闭 net.ipv4.tcp_tw_reuse = 1
- CLOSE_WAIT：在被动关闭连接情况下，在已经接收到FIN，但是还没有发送自己的FIN的时刻
  - 在对方连接关闭之后，程序里没有检测到，或者程序压根就忘记了这个时候需要关闭连接
  - 需要改程序代码

粘包、半包

原因：

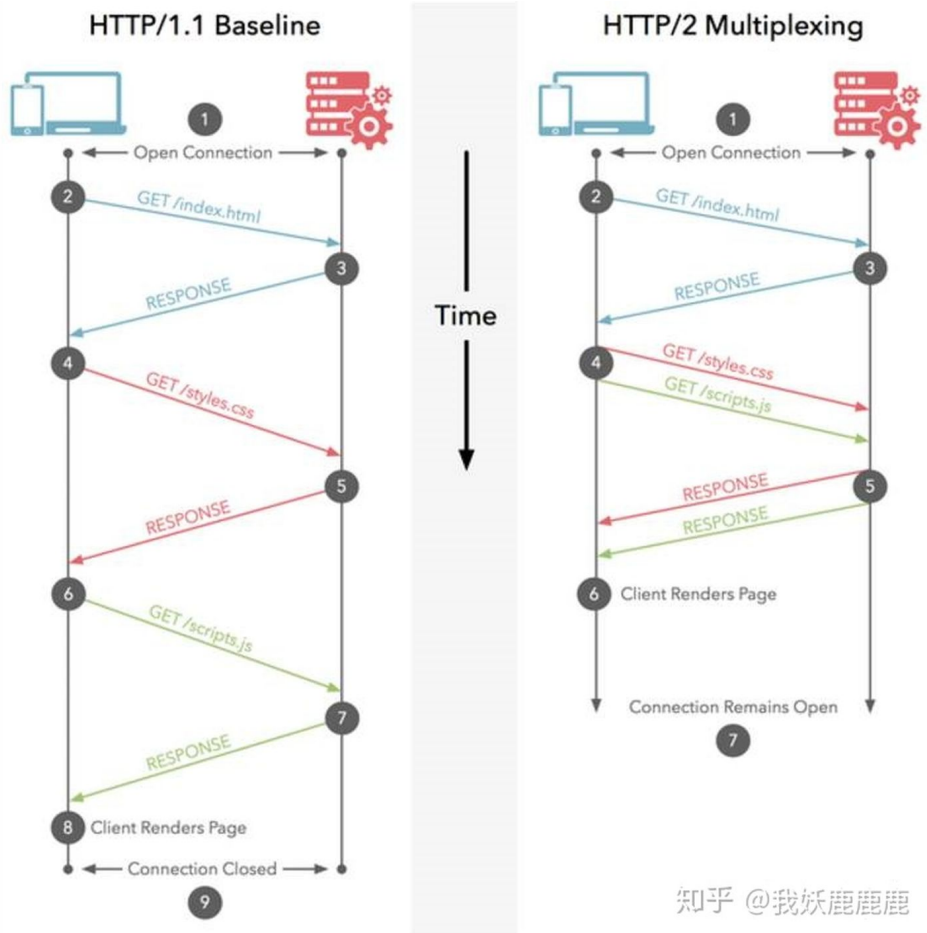
TCP 传输的数据是以字节流的形式，字节流没有明确的边界，所以 TCP 也没办法判断哪一段流属于一个消息

解决：

1. 在 TCP 协议的基础上封装一层数据请求协议，既将数据包封装成数据头（存储数据正文大小）+ 数据正文的形式，这样在服务端就可以知道每个数据包的具体长度了，知道了发送数据的具体边界就可以解决半包和粘包的问题
2. 以特殊的字符结尾，比如以"\n"结尾，这样我们就知道结束字符，从而避免了半包和粘包问题

HTTP 2.0

1. http1的解析是基于文本协议的各式解析，而http2.0的协议解析是二进制格式,更加的强大
2. **多路复用(Mutiplexing) :** 在`HTTP1.1`的协议中，我们传输的`request`和`response`都是基本于文本的，这样就会引发一个问题：所有的数据必须按顺序传输，因为接收端并不知道这些字符的顺序，所以并行传输在`HTTP1.1`是不能实现的。`HTTP/2`引入`二进制数据帧`和`流`的概念，其中帧对数据进行顺序标识，这样浏览器收到数据之后，就可以按照序列对数据进行合并。
3. header压缩: http1.x中的header需要携带大量信息.而且每次都要重复发送.http2.0使用encode来减少传输的header大小.而且客户端和服务端可以各自缓存(cache)一份header filed表,避免了header的重复传输,还可以减少传输的大小.
4. 服务端推送(server push): 可以通过解析html中的依赖,只能的返回所需的其他文件(css或者js等),而不用再发起一次请求.



HTTP 3.0

基于QUIC（Quick UDP Internet Connections）

无队头阻塞，QUIC 连接上的多个 Stream 之间并没有依赖，都是独立的，也不会有底层协议限制，某个流发生丢包了，只会影响该流，其他流不受影响；

建立连接速度快，因为 QUIC 内部包含 TLS1.3，因此仅需 1 个 RTT 就可以「同时」完成建立连接与 TLS 密钥协商，甚至在第二次连接的时候，应用数据包可以和 QUIC 握手信息（连接信息 + TLS 信息）一起发送，达到 0-RTT 的效果。

连接迁移，QUIC 协议没有用四元组的方式来“绑定”连接，而是通过「连接 ID 」来标记通信的两个端点，客户端和服务器可以各自选择一组 ID 来标记自己，因此即使移动设备的网络变化后，导致 IP 地址变化了，只要仍保有上下文信息（比如连接 ID、TLS 密钥等），就可以“无缝”地复用原连接，消除重连的成本；

KeepAlive

当一个 TCP 连接建立之后，启用 TCP Keepalive 的一端便会启动一个计时器，tcp_keep-alive_time，这个值默认是7200s（2h），当这个计时器数值到达 0 之后，会发送一个不携带数据的tcp包，如果对方没有回应，如果重复几次之后依然没有回应，服务器会断开tcp连接。

状态码

https://blog.csdn.net/weixin_43551152/article/details/85041399

状态码　含义　常见示例
1** 服务器已经接受到请求，客户端可继续发送请求
2** 请求成功 200：请求已成功，请求所希望的响应头或数据体将随此响应返回。
3** 重定向

301表示永久重定向（301 moved permanently），表示请求的资源分配了新url，以后应使用新url。
302表示临时性重定向（302 found），请求的资源临时分配了新url，本次请求暂且使用新url。302与301的区别是，302表示临时性重定向，重定向的url还有可能还会改变。
303 表示请求的资源路径发生改变，使用GET方法请求新url。她与302的功能一样，但是明确指出使用GET方法请求新url。
4** 客户端错误 404：请求的网页不存在
5** 服务器错误 503： 服务器超时

命令相关

- ping
  ICMP协议，属于网络层，ICMP协议规定：目的主机必须返回ICMP回送应答消息给源主机。如果源主机在一定时间内收到应答，则认为主机可达。
- traceroute
  traceroute的原理是试图以最小的TTL（存活时间）发出探测包来跟踪数据包到达目标主机所经过的网关，然后监听一个来自网关ICMP的应答。首先它发送一份TTL字段为1的IP数据包给目的主机，处理这个数据包的第一个路由器将TTL值减1，然后丢弃该数据报，并给源主机发送一个ICMP报文（“超时”信息，这个报文包含了路由器的IP地址，这样就得到了第一个路由器的地址），然后traceroute发送一个TTL为2的数据报来得到第二个路由器的IP地址，继续这个过程，直至这个数据报到达目的主机。

数据库

事务四大特性（ACID）

原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

一致性（Consistency）

事务开始前和结束后，数据库的完整性约束没有被破坏。比如A向B转账，不可能A扣了钱，B却没收到。

隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

并发带来的问题

- **脏读（Dirty read）：**当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
- **丢失修改（Lost to modify）：**指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。 例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。
- **不可重复读（Unrepeatableread）：**指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。
- **幻读（Phantom read）：**幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

事务的隔离级别

SQL 标准定义了四个隔离级别：

- **READ-UNCOMMITTED(读取未提交)：** 最低的隔离级别，允许读取尚未提交的数据变更，**可能会导致脏读、幻读或不可重复读。**
- **READ-COMMITTED(读取已提交)：** 允许读取并发事务已经提交的数据，**可以阻止脏读，但是幻读或不可重复读仍有可能发生。**
- **REPEATABLE-READ(可重复读)：** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，**可以阻止脏读和不可重复读，但幻读仍有可能发生。**
- **SERIALIZABLE(可串行化)：** 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，**该级别可以防止脏读、不可重复读以及幻读。**

隔离级别	脏读	不可重复读	幻影读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	×	√	√
REPEATABLE-READ	×	×	√
SERIALIZABLE	×	×	×

MySQL InnoDB 存储引擎的默认支持的隔离级别是 **REPEATABLE-READ（可重读）**

MYSQL的两种存储引擎区别

MYISAM：管理非事务表，粒度是表级，查询效率比较高，如果有大量的SELECT，适合用这个

InnoDB：管理事务表，支持事务ACID，支持行级锁定，适用于用户并发的场景

B+索引和hash索引

hash索引，等值查询效率高，不能排序,不能进行范围查询，不能进行模糊查询，也不支持联合索引

引申：为啥B+?

B+树比较矮比较宽，需要较少的I/O次数，相比于红黑树，不需要进行复杂的左旋右旋，相比于Hash索引，支持范围查询，联合索引

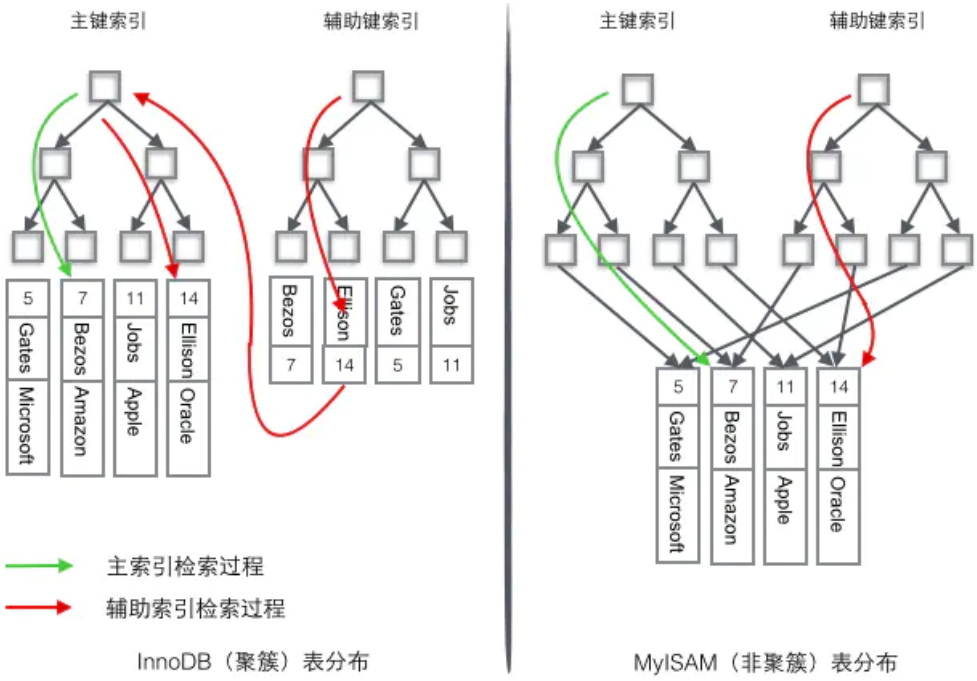
- 磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入
- **为什么 B+ 树比 B 树更适合应用于数据库索引?**

    1. B+ 树的非叶子结点只存关键字不存数据，因而单个页可以存储更多的关键字，即一次性读入内存的需要查找的关键字也就越多，磁盘的随机 I/O 读取次数相对就减少了。
    2. B+ 树的查询效率相比B树更加稳定，由于数据只存在在叶子结点上，所以查找效率固定为 O(log n)。
    3. B+ 树叶子结点之间用链表有序连接，所以扫描全部数据只需扫描一遍叶子结点，利于扫库和范围查询；B 树由于非叶子结点也存数据，所以只能通过中序遍历按序来扫。也就是说，对于范围查询和有序遍历而言，B+ 树的效率更高。

聚簇索引和非聚簇索引

https://blog.csdn.net/cy973071263/article/details/104513633

- 聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据，主文件按照对应字段排序存储，索引文件无重复排序存储。
- 非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，主文件并没有按照对应字段排序存储，索引文件有重复排序存储。



为什么主键通常建议使用自增id?
答：**聚簇索引的数据的物理存放顺序与索引顺序是一致的**，即：**只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上的。**如果主键不是自增id，对于已经填满的页面，因为无序很可能多次命中，产生分裂，进而产生更多不被填满页面。还有分裂本身的成本

mysql中聚簇索引的设定?
答：聚簇索引**默认是主键**，如果表中没有定义主键，InnoDB 会选择一个**唯一的非空索引**代替。如果没有这样的索引，InnoDB 会**隐式定义一个主键**来作为聚簇索引。

索引的优缺点

索引最大的好处是提高查询速度， 缺点是更新数据时效率低，因为要同时更新索引 对数据进行频繁查询进建立索引
如果要频繁更改数据不建议使用索引

索引优化

- 使用那些基数比较大的字段
- 使用短索引
- 主键一定是自增的，别用UUID之类
- 经常修改的字段不要建索引

sql优化

- **利用最左前缀**,顾名思义，就是最左优先，就是先要看第一列，在第一列满足的条件下再看左边第二列，以此类推
- 在索引使用NOT ,<>,会导致索引失效,比如a不等于0 a<>0可以修改为 a>0 or a<0 ,NOT修改为a>0 或者a>"",避免全表扫描
- 尽量避免使用前导模糊查询,因为前导模糊查询由%,不能利用索引,影响查询效率

数据库连接池

连接池是维护的数据库连接的缓存，以便将来需要对数据库的请求时可以重用这些连接。为每个用户打开和维护数据库连接，尤其是对动态数据库驱动的网站应用程序的请求，既昂贵又浪费资源。**在连接池中，创建连接后，将其放置在池中，并再次使用它，因此不必建立新的连接。如果使用了所有连接，则会建立一个新连接并将其添加到池中。**连接池还减少了用户必须等待建立与数据库的连接的时间。

乐观锁和悲观锁

悲观锁：对数据的修改抱有悲观态度的并发控制方式，一般认为数据被并发修改的概率比较大，所以需要在修改之前先加锁。但是在效率方面，处理加锁的机制会**让数据库产生额外的开销**，还有增加**产生死锁**的机会；另外，还会**降低并行性**，一个事务如果锁定了某行数据，其他事务就必须等待该事务处理完才可以处理那行数据。

乐观锁：假设数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做

数据库范式

第一范式就是属性不可分割，每个字段都应该是不可再拆分的。比如一个字段是姓名（NAME），在国内的话通常理解都是姓名是一个不可再拆分的单位，这时候就符合第一范式；但是在国外的话还要分为FIRST NAME和LAST NAME，这时候姓名这个字段就是还可以拆分为更小的单位的字段，就不符合第一范式了。

第二范式就是要求表中要有主键，表中其他其他字段都依赖于主键，因此第二范式只要记住主键约束就好了。比如说有一个表是学生表，学生表中有一个值唯一的字段学号，那么学生表中的其他所有字段都可以根据这个学号字段去获取，依赖主键的意思也就是相关的意思，因为学号的值是唯一的，因此就不会造成存储的信息对不上的问题，即学生001的姓名不会存到学生002那里去。

第三范式就是要求表中不能有其他表中存在的、存储相同信息的字段，通常实现是在通过外键去建立关联，因此第三范式只要记住外键约束就好了，消除传递依赖。

BCNF范式：关系模式中每一个决定因素都包含候选键，关系模式中的非候选键不能决定任何一个属性。

Mysql 事务是如何实现的

原子性：通过**undo log**实现的。每条数据变更都伴随一条undo log日志的生成，当系统发生错误或执行回滚根据undo log做逆向操作
持久性：通过**redo log**实现的。redo log记录了数据的修改日志。数据持久化到磁盘，先是储存到缓冲池里，然后缓冲池中的数据定期同步到磁盘中，如果系统宕机，可能会丢失数据，系统重启后会读取redo log恢复数据

隔离性：MVCC（多版本并发控制协议），在记录中有两个隐藏列：**trx_id**，这个id用来存储的每次对某条记录进行修改的时候的事务id，修改之后写入undo日志中。还有**roll_pointer**，就是存了一个指针，它指向这条记录的上一个版本的位置，通过它来获得上一个版本的记录信息。如果说多个事务正在并行，此时有一条查询语句，那么就会生成一个read view，这个view包含了正在活动的事务列表，也就是还没有commit的，如果说我们此时的事务id<最小的活动事务id，那么说明这个事务已经提交了，可以读，如果我们此时的事务id>最大的事务id，说明建立read view的时候这个事务还没开始，不能读，如果此时事务id在最大和最小之间，而且在我们当前的事务活动列表里面，说明事务还没有commit，不能读，就要通过那个roll_pointer继续找下一个事务。已提交读隔离级别下的事务在每次查询的开始都会生成一个独立的ReadView,而可重复读隔离级别则在第一次读的时候生成一个ReadView，之后的读都复用之前的ReadView

一千万条数据有几层

假如我们的主键id为bigint类型，长度为8字节，而指针大小在InnoDB源码中设置为6字节。一个页等同于一个节点大小，是16KB，这样算下来就是 16384 / 14 = 1170，就是说一个页上可以存放1000个指针。那三层的话就是16*1000*、1000也就是两千万

MySQL中的锁

兼容性角度

- 共享锁（Shared Lock）：允许读，只有共享锁是兼容的
- 排他锁（Exclusive Lock）： 允许更新和删除，与共享锁和排他锁都不兼容

锁模式

- 行锁（Record Lock）
  - 锁住一行的数据，行锁加在索引上
- 间隙锁（Gap Lock）
  - 当发生update record set age = 10 where user_id>10;时候，加锁如下：索引user_id上会加上gap锁，锁住user_id (10,+无穷大）这个范围
  - 可以解决幻读
- Next-key lock （组合）
- 意向锁（Intension Lock）
  - 意向锁产生的主要目的是为了处理行锁和表锁之间的冲突
  - 比如，表中的某一行上加了X锁，就不能对这张表加X锁，如果不在表上加意向锁，对表加锁的时候，都要去检查表中的某一行上是否加有行锁。

判断索引是否失效

联合索引

select 语句前面加 explain

type: const(通过索引一次就找到，一般是primary key, unique key), range, all

key: 实际使用的索引

rows: 估算找所需记录需要读取的行数

一条sql语句在mysql中的执行流程

https://blog.csdn.net/weter_drop/article/details/93386581

SELECT: 连接器(身份验证)->分析器(语法分析)->优化器(sql语句优化)->执行器(权限校验)->调引擎接口

https://www.cnblogs.com/cdf-opensource-007/p/6502556.html

```
SELECT `name`, COUNT(`name`) AS num
FROM student
WHERE grade < 60
GROUP BY `name`
HAVING num >= 2
ORDER BY num DESC
LIMIT 0, 2;
```

where生成临时表，经过where筛选后的表T1

group by切分临时表t1, t2, t3...

select对每张临时表进行属性筛选，重命名生成T2

having对T2进行操作，所以只能是select后的字段

一条SQL执行很慢的原因

- 数据库在刷新脏页，例如 redo log 写满了需要同步到磁盘
  当我们要往数据库插入一条数据、或者要更新一条数据的时候，我们知道数据库会在**内存**中把对应字段的数据更新了，但是更新之后，这些更新的字段并不会马上同步持久化到**磁盘**中去，而是把这些更新的记录写入到 redo log 日记中去，等到空闲的时候，在通过 redo log 里的日记把最新的数据同步到**磁盘**中去。
- 执行的时候，遇到锁，如表锁、行锁
- 没有用上索引

RPC

RPC 的全称是 Remote Procedure Call，是一种进程间通信方式。它允许程序调用另一个地址空间的函数，建立在TCP/IP协议之上。

当应用开始调用PRC的方式时，就会去容器中去取Bean对象，所以我们应该首先注册Bean对象到容器中，再调用接口，这时候请求参数会被序列化成二进制封装到请求报文里，发送请求的时候还会设置请求端口（即服务器的微服务端口），这个一般会写在接口文档里面，然后发送给服务端。服务端就会监听这个端口，如果有请求进来会把二进制反序列化成对象，执行完方法之后再把返回结果序列化成二进制传输给客户端。

rpc跟http不是同等级的概念吧，http是协议，rpc指的是远程调用，rpc可以用各种协议实现，可以使用http协议。这个是我个人的理解。

Redis

什么是Redis

Redis是一个使用 C 语言编写的，高性能非关系型（NoSQL）的键值对数据库

Redis 可以存储键和五种不同类型的值之间的映射。键的类型只能为字符串，值支持五种数据类型：字符串、列表、集合、散列表、有序集合

redis读写速度非常快，被广泛应用于缓存

为什么要用 Redis 作为缓存

1. 高性能：假如用户第一次访问数据库中的某些数据。这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据存在数缓存中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可
2. redis能够承受的请求是远远大于直接访问数据库的，可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库，承受更大量的并发。

为什么要用 Redis 而不用 map/guava 做缓存

使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

使用 redis之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。

Redis为什么很快?

- Redis是基于内存存储的，而mysql是基于磁盘存储的，内存的交互肯定是要比磁盘的交互快的
- Redis内数据存储格式是KV形式查找数据时的时间复杂度是O(1)，Mysql底层是B+树，查找数据时的时间复杂度是O(logN)所以查找数据时Redis是比Mysql要快的
- Redis是单线程多路复用IO，单线程的切换的话他避免的线程切换消耗的时间，多路复用IO避免了IO等待的开销

Redis支持哪些数据类型?

Redis主要有5种数据类型：包括String，List，Set（无序集合），Zset（有序集合），Hash（可以存结构化数据）

Redis过期时间

通过expire来设置key 的过期时间

缓存雪崩

缓存雪崩是指缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决：缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。

缓存穿透

缓存穿透是指缓存和数据库中都没有的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉

1. 接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截
2. 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击
3. 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力

缓存击穿

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。和缓存雪崩不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决：提前设置热点数据（轮播图举例）

zset结构

1. ziplist（双向链表）数据量较少的时候使用
2. skiplist（跳表）为什么用跳表不用红黑树：范围搜索直接往后遍历，红黑树还要回溯寻找，最差可能要回溯到根节点

如何保证redis数据不丢失

- RDB
  - 介绍：产生一个数据快照文件，通过save或bgsave命令，让Redis在本地生成RDB快照文件，可以定时进行备份
  - 优点：宕机恢复时，加载RDB文件的速度很快，能够在很短时间内迅速恢复文件中的数据
  - 缺点：生成rdb文件的代价比较大，消耗很多内存资源；数据不全，比起AOF，写入磁盘的频率比较低，15分钟至少一次写入才触发
  - 原理：bgsave命令，fork系统调用。fork子进程时，操作系统需要拷贝父进程的内存页表给子进程，如果整个Redis实例内存占用很大，那么它的内存页表也会很大，在拷贝时就会比较耗时。之后父进程在处理写命令时，采用了操作系统提供的Copy On Write技术，父进程会重新分配新的内存地址空间，从操作系统申请新的内存使用，不再与子进程共享。这样父子进程的内存就会逐渐分离。
  - 使用场景：适合数据备份，数据丢失不敏感业务
- AOF
  - 介绍：Append Only File，实时追加命令的日志文件
  - 通过配置文件开启AOF，把appendonly设置成yes
  - 优点：AOF数据文件更新比较及时，比RDB保存更完整的数据；不会fork子进程，不会在主存中复制页表，没有CPU消耗
  - 缺点：每秒刷磁盘，增加磁盘IO的负担；随着时间增长，aof文件越来越大
  - 使用场景：对丢失数据比较敏感的业务，比如转账

redis事务

使用watch命令对库存字段进行监视，实际上是保存了版本号，然后在事务提交的时候利用了乐观锁，也就CAS操作，比较版本号，如果不一致说明有事务对这个字段进行了修改，那么这是我们的事务执行完会返回null，如果返回null那我们重新进行watch，保存下一个版本号即可。

过期策略和内存淘汰机制

- 惰性删除

  惰性删除是指 Redis 服务器不主动删除过期的键值，而是当访问键值时，再检查当前的键值是否过期，如果过期则执行删除并返回 null 给客户端；如果没过期则正常返回值信息给客户端。
  它的优点是不会浪费太多的系统资源，只是在每次访问时才检查键值是否过期。缺点是删除过期键不及时，造成了一定的空间浪费。

- 定期删除

  定期删除是指 Redis 服务器每隔一段时间会检查一下数据库，看看是否有过期键可以被清除。
  定期删除的扫描并不是遍历所有的键值对，这样的话比较费时且太消耗系统资源。Redis 服务器采用的是随机抽取形式，每次从过期字典中，取出 20 个键进行过期检测，过期字典中存储的是所有设置了过期时间的键值对。如果这批随机检查的数据中有 25% 的比例过期，那么会再抽取 20 个随机键值进行检测和删除，并且会循环执行这个流程，直到抽取的这批数据中过期键值小于 25%，此次检测才算完成。

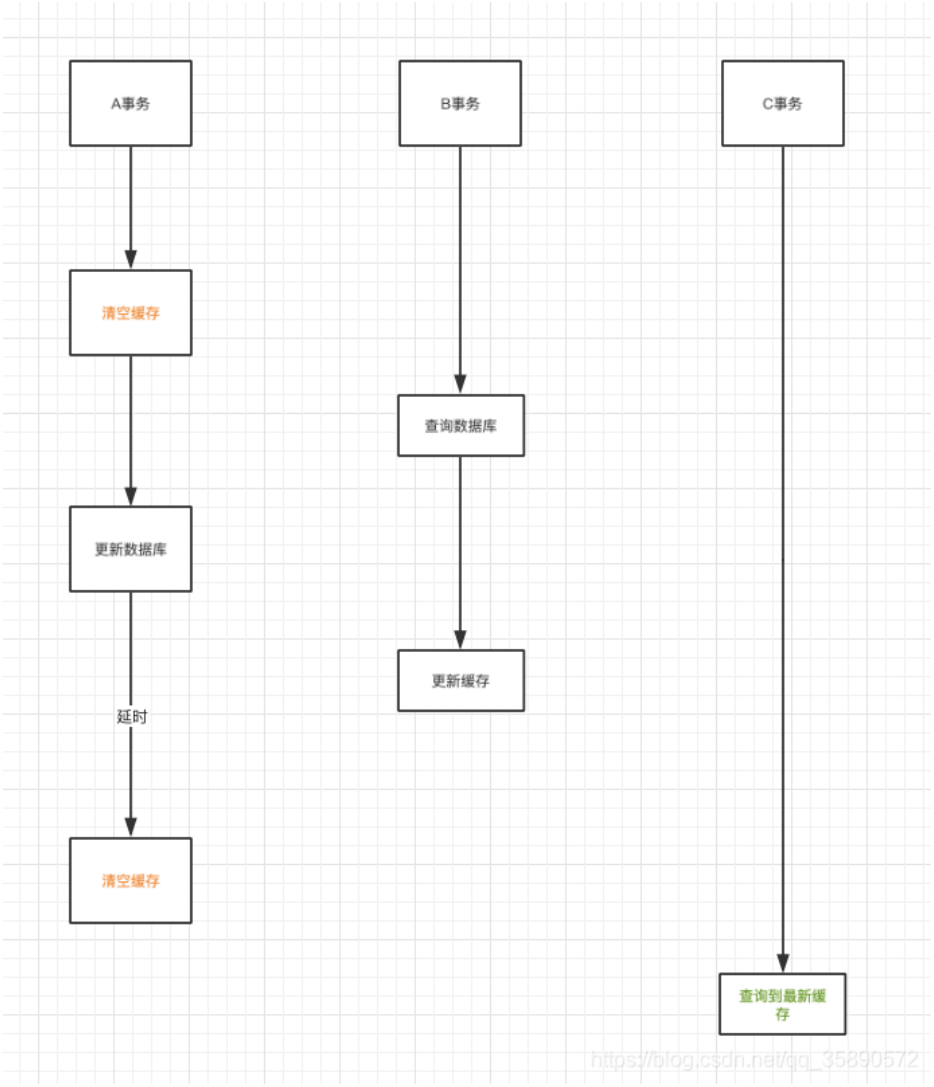
内存淘汰是内存不足时会触发的机制，Redis 4.0 之后提供了 8 种内存淘汰策略，默认是noeviction：不淘汰任何数据，当内存不足时，执行缓存新增操作会报错。还有比较经典的lru：淘汰整个键值中最久未使用的键值，还有lfu、random

数据一致性

https://blog.csdn.net/qq_35890572/article/details/108538712

只删一次：先删缓存，在改库前，其他事务又把旧数据放到缓存里去了

延时是确保 **修改数据库 -> 清空缓存前，其他事务的更改缓存操作已经执行完。**



redis源码

rdb.c

```
rdbSave()
rdbSaveBackground() //调用redisFork()
```

server.c

```
redisFork(); //系统调用fork
```

server.h

```
typedef struct redisDb {
    dict *dict;                 // 数据库键空间，保存着数据库中的所有键值对
    dict *expires;              // 键的过期时间
    dict *blocking_keys;        // 正处于阻塞状态的键
    dict *ready_keys;           // 可以解除阻塞的键
    dict *watched_keys;         // 正在被 WATCH 命令监视的键
    int id;                     // 数据库编号
} redisDb;
```

数据结构

红黑树和AVL树的区别

1、红黑树放弃了追求完全平衡，追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单。

2、平衡二叉树追求绝对平衡，条件比较苛刻，实现起来比较麻烦，每次插入新节点之后需要旋转的次数不能预知。

并查集

并查集是一种树型的数据结构，用于处理一些不相交集合（Disjoint Sets）的合并及查询问题

排序算法

稳定的：冒泡，插入，归并； 不稳定：快排，选排，堆排序

Arrays.sort()并不是单一的排序，而是插入排序，快速排序，归并排序三种排序的组合，数量非常小的情况下（少于47），插入排序等可能会比快速排序更快。 所以数组少于47的会进入插入排序，大于47小于286是快速排序，大于286是归并排序。但也不一定，这里会有个小动作：“// Check if the array is nearly sorted”就是在降序组太多的时候（被判断为没有结构的数据，要转回快速排序。

快排存在的问题与优化

时间复杂度最理想的就是取到中位数情况，那么递归树就是一个完全二叉树，那么树的深度也就是最低为logN

Mybatis

什么是Mybatis

1. Mybatis是一个对象关系映射框架，它内部封装了JDBC，开发者开发时只需要关注如何编写SQL语句
2. MyBatis 可以使用 XML 或注解来配置，将pojo与数据库表一一对应，方便CRUD（举例子）
3. 可以引入mybatis-generator依赖，可以自动生成pojo和对应的基本的增删改查语句，非常方便

#{}和${}的区别是什么?

${}是字符串替换，#{}是将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值
使用#{}可以有效的防止SQL注入，提高系统安全性

个Dao接口的工作原理是什么?

Mapper 接口的工作原理是JDK动态代理，Mybatis运行时会使用JDK动态代理为Mapper接口生成代理对象proxy，代理对象会拦截接口方法，根据类的全限定名+方法名，唯一定位到一个MapperStatement并调用执行器执行所代表的sql，然后将sql执行结果返回。

Mapper接口里的方法，是不能重载的，因为是使用 全限名+方法名 的保存和寻找策略。

当体类中属性名和字段不一样

通过来映射字段名和实体类属性名的一一对应的关系。

```
<id property="id" column="order_id">
```

如何获取自动生成的(主)键值

如果采用自增长策略，自动生成的键值在 insert 方法执行完后可以被设置到传入的参数对象中

```
<insert id="insertname" usegeneratedkeys="true" keyproperty="id">
     insert into names (name) values (#{name})
</insert>
```

mapper接口调用时有哪些要求?

Mapper接口方法名和mapper.xml中定义的每个sql的id相同;
Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同;
Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同;

模糊查询like语句该怎么写?

因为#{...}解析成sql语句时候，会在变量外侧自动加单引号' '，所以这里 % 需要使用双引号" "，不能使用单引号' '，不然会查不到任何结果

```
string wildcardname = “smi”;
list<name> names = mapper.selectlike(wildcardname);
    <select id='selectlike'>
        select * from foo where bar like "%"#{userName,jdbc=xxx}"%"
    </select>
```

Spring

Spring是什么

Spring是一个轻量级的控制反转（IoC)和面向切面（AOP）的容器框架，目的是用于简化企业应用程序的开发

- 通过IOC达到松耦合的目的
- 通过AOP减少系统中的重复代码，提高系统的可维护性。比如权限认证、日志等

至于说它是一个容器框架:

- 包含并管理应用对象(Bean)的配置和生命周期，这个意义上是一个容器
- 将简单的组件配置、组合成为复杂的应用，这个意义上是一个框架

IOC

IOC就是控制反转，之所以成为反转是因为传统的程序需要我们手动new创建对象，控制反转的话就是不需要用户手动去new，而是把创建对象的控制权交给了容器，由容器进行依赖注入，减少了对象之间的耦合。

没有引入IOC容器之前，对象A依赖于对象B，那么对象A在初始化或者运行到某一点的时候，自己必须主动去创建对象B或者使用已经创建的对象B。无论是创建还是使用对象B，控制权都在自己手上。引入IOC容器之后，对象A与对象B之间失去了直接联系，当对象A运行到需要对象B的时候，IOC容器会主动创建一个对象B注入到对象A需要的地方。通过前后的对比，不难看出来：对象A获得依赖对象B的过程，由主动行为变为了被动行为，控制权颠倒过来了，这就是“控制反转”这个名称的由来。

AOP

AOP是面向切面编程，将一些可重用的代码块抽离出来，这个模块称为切面，可以加在某些方法之前，也可以加在某些方法之后，比如权限认证。Spring AOP是基于动态代理实现的，使用AOP可以减少系统中的重复代码，提高可维护性。

Spring AOP动态代理

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理

使用代理对象, 是为了在不修改目标对象的基础上, 增强目标对象的业务逻辑

比如转账业务，在转账之前要进行身份验证，创建一个代理类对转账业务进行增强，调用身份验证函数，JDK动态代理需要代理类实现InvocationHandler接口，实现invoke方法，这样每次调用代理类的转账方法都会来执行这个invoke方法

(1) 转账业务

```
public interface IAccountService {
    //主业务逻辑: 转账
    void transfer();
}
public class AccountServiceImpl implements IAccountService {
    @Override
    public void transfer() {
        System.out.println("调用dao层,完成转账主业务.");
    }
}
```

(2) 增强

```
public class AccountAdvice implements InvocationHandler {
    //目标对象
    private IAccountService target;

    public AccountAdvice(IAccountService target) {
        this.target = target;
    }

    /**
     * 代理方法, 每次调用目标方法时都会进到这里
     */
```

```
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        before();
        return method.invoke(target, args);
    }

    /**
     * 前置增强
     */
    private void before() {
        System.out.println("对转账人身份进行验证.");
    }
}
```

（3）测试

```
public class Client {
    public static void main(String[] args) {
        //创建目标对象
        IAccountService target = new AccountServiceImpl();
        //创建代理对象
        IAccountService proxy = (IAccountService)
Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                new AccountAdvice(target)
        );
        proxy.transfer();
    }
}
结果:
对转账人身份进行验证.
调用dao层,完成转账主业务.
```

CGLIB动态代理

如果目标对象没有实现接口，必须采用cglib库，Spring会自动在JDK动态代理和cglib之间转换

Bean的生命周期

1、解析类得到BeanDefinition（比如是否懒加载）
2、如果有多个构造方法，则要推断构造方法（根据类型或者名称）
3、确定好构造方法后，进行实例化得到一个对象，在堆中开辟一块空间，属性赋默认值（反射）
4、对对象中的加了@Autowired注解的属性进行属性填充
5、回调Aware方法，比如BeanNameAware
6、调用BeanPostProcessor的初始化前的方法
7、调用初始化方法（如果Bean实现了InitializingBean接口，执行afeterPropertiesSet()方法）
8、调用BeanPostProcessor的初始化后的方法，在这里会进行AOP
9、如果当前创建的bean是单例的则会把bean放入单例池
10、使用bean
11、Spring容器关闭时调用DisposableBean中destory()方法

如何避免循环依赖

在没有实现AOP的情况下，可以使用二级缓存来解决循环依赖，第一级缓存是**singletonObjects**，也叫单例池，存储已经创建完成的Bean，第二级缓存是**earlySingletonObjects**，存储正在创建的bean，也叫半成品池。假设A和B是两个Bean，A中注入了B，B中也注入了A，从Bean的生命周期而言，Bean在初始化之前要经过getBean，实例化，填充属性三个步骤。首先创建A，当A进行实例化的时候，就会把A放到半成品池里面，这就是第二级缓存，给A填充属性的时候，扫描到了B，那么进行B的创建，B经过getBean，实例化，走到填充属性的时候，需要填充A，首先先去一级缓存看看有没有A，因为A还没有完成初始化，所以单例池还没有A，然后B会去二级缓存看看半成品池有没有，发现有，进行填充，填充完了完成初始化，这时候B走完了初始化流程，进入了单例池，此时A就能完成属性的填充和初始化，进入单例池，再把半成品池里的A删除。

当实现了AOP，就需要三级缓存了，因为此时注入的不是原来的对象本身，而是代理对象，所以需要第三级缓存来存代理对象。

SpringBoot初始化流程（SpringBoot自动装配）

所有程序的入口都是main方法，main方法里调用SpringApplication.run()方法，这里面有一个核心的方法就是refreshContext()方法，它会刷新容器，刷新容器的时候它就会开始扫描启动类，解析注解，启动类上有一个SpringBootApplication注解，SpringBootApplication本身就是一个配置类，里面有一个EnableAutoConfiguration注解，开启自动装配。这个类里面有一些核心方法帮助我们找到路径下的MATA-INF目录下的spring.factories文件，这个文件里面记录了非常多自动配置类的全限定类名，有了全路径就可以通过反射创建Bean对象

```
Class class = Class.forName("test.Phone");
Class class = phone.getClass();
Phone instance1 = (Phone) class.newInstance();
```

Autowied

在启动spring IoC时，容器自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到@Autowied、@Resource时，就会在IoC容器自动查找需要的bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：

如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；

如果查询的结果不止一个，那么@Autowired会根据名称来查找；

BeanFactory和ApplicationContext有什么区别

ApplicationContext是BeanFactory的子接口

ApplicationContext提供了更完整的功能：支持国际化

BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。

Spring框架中的Bean是线程安全的么?

Spring容器本身并没有提供Bean的线程安全策略，因此可以说Spring容器中的Bean本身不具备线程安全的特性，但是具体情况还是要结合Bean的作用域来讨论。

（1）对于prototype作用域的Bean，每次都创建一个新对象，也就是线程之间不存在Bean共享，因此不会有线程安全问题。

（2）对于singleton作用域的Bean，所有的线程都共享一个单例实例的Bean，因此是存在线程安全问题的。但是如果单例Bean是一个无状态Bean，也就是线程中的操作不会对Bean的成员执行查询以外的操作，那么这个单例Bean是线程安全的。比如Controller类、Service类和Dao等，这些Bean大多是无状态的，只关注于方法本身。

Spring 框架中都用到了哪些设计模式?

1. 工厂模式：Spring使用工厂模式，通过BeanFactory和ApplicationContext来创建对象
2. 单例模式：Bean默认为单例模式
3. 代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术

SpringMVC流程

第一步：发起请求到前端控制器(DispatcherServlet)

第二步：前端控制器请求HandlerMapping查找 Handler （可以根据xml配置、注解进行查找）

第三步：处理器映射器HandlerMapping向前端控制器返回Handler，HandlerMapping会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象，多个HandlerInterceptor拦截器对象），通过这种策略模式，很容易添加新的映射策略

第四步：前端控制器调用处理器适配器去执行Handler

第五步：处理器适配器HandlerAdapter将会根据适配的结果去执行Handler

第六步：Handler执行完成给适配器返回ModelAndView

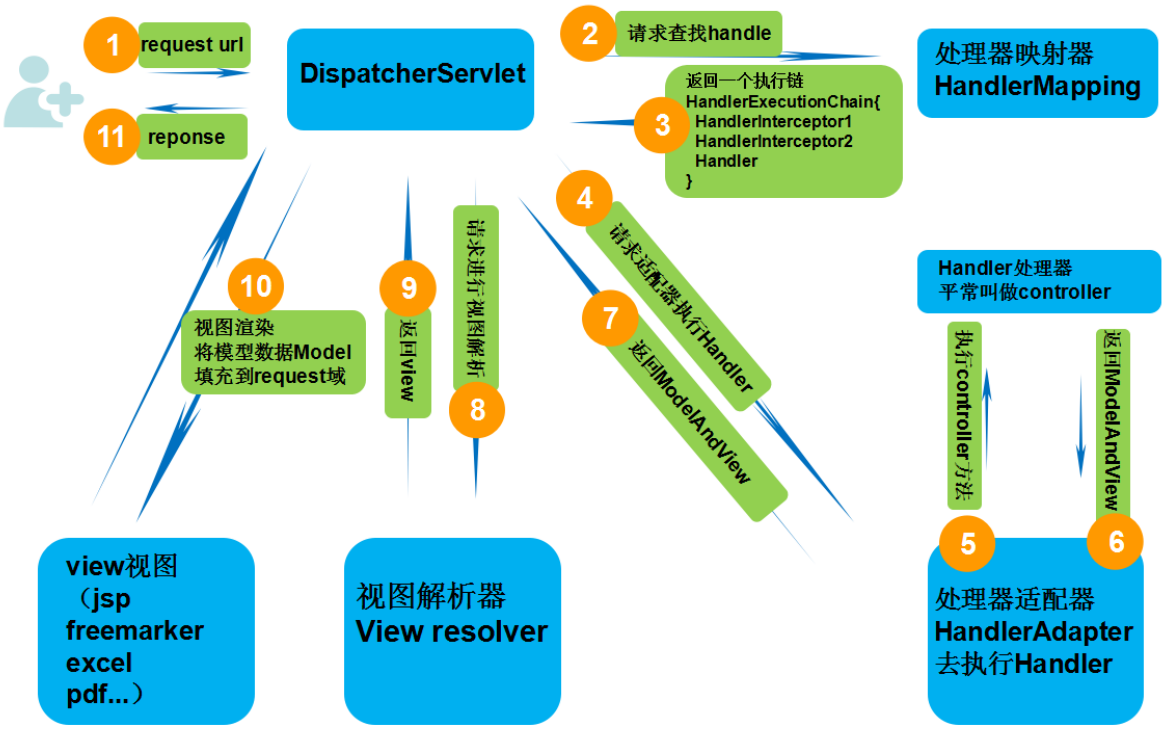
第七步：处理器适配器向前端控制器返回ModelAndView （ModelAndView是springmvc框架的一个底层对象，包括 Model和view）

第八步：前端控制器请求视图解析器去进行视图解析 （根据逻辑视图名解析成真正的视图(jsp)），通过这种策略很容易更换其他视图技术，只需要更改视图解析器即可

第九步：视图解析器向前端控制器返回View

第十步：前端控制器进行视图渲染 （视图渲染将模型数据(在ModelAndView对象中)填充到request域）

第十一步：前端控制器向用户响应结果



补充

多线程，项目优化优化方向，countdown锁，AQS怎么保证锁的可重入性，Atomic原子类

```
@Override
    public List<String> parallelUpload(List<FileModel> fileModelList) {
        List<String> urlList = new CopyOnWriteArrayList<>(); //保证并发插入安全
        ExecutorService pool = Executors.newCachedThreadPool();
        CountDownLatch countDownLatch = new
CountDownLatch(fileModelList.size());
        for (FileModel fileModel : fileModelList) {
            pool.execute(() -> {
                try {
                    String originPath = uploadPath(fileModel.getInputStream(),
fileModel.getFileName());
                    urlList.add(presignUrl(originPath));
                } finally {
                    countDownLatch.countDown();
                }
            });
        }
        try {
            countDownLatch.await();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        pool.shutdown();
        return urlList;
    }
```

1. jdk动态代理和cglib动态代理
2. - Guava RateLimiter限流原理：https://zhuanlan.zhihu.com/p/60979444
   - https://www.cnblogs.com/xrq730/p/11025029.html
3. - hashmap扩容时为什么线程不安全？当多个线程同时检测到总数量超过门限值的时候就会同时调用resize操作，各自生成新的数组并rehash后赋给该map底层的数组table，结果最终只有最后一个线程生成的新数组被赋给table变量，其他线程的均会丢失。
4. - Java不可重入锁和可重入锁理解：可重入就是说某个线程已经获得某个锁，可以再次获取锁而不会出现死锁。
   - AQS如何保证可重入？公平锁非公平锁？https://tech.meituan.com/2019/12/05/aqs-theory-and-apply.html
5. - 如何实现登陆拦截：自定义一个类，实现HandlerInterceptor，重写preHandle方法（接口中的默认方法体，默认返回true），在此做鉴权，未登录返回false。再自定义一个类，实现WebMvcConfigurer，重写addInterceptors方法，将自定义的实现HandlerInterceptor接口的类加入进去，然后添加路径，也就是访问这些路径的url要经过这个拦截器。

其他

- **实习项目遇到了什么技术问题，怎么解决的？**

1. SQL查询某个表的时候一直报错，后来发现是有个字段名叫condition，是个关键字
2. 写过一个方法，功能记不清了，有一步是通过反射获取一个类所有的fields，在本地跑着没有问题，一上测试环境发现类的字段多了一个，叫jacocoData。因为在项目发布的时候，maven继承了jacoco来统计代码覆盖率，是编译期间加上的字段，可以通过

```
if (field.isSynthetic()) {
    continue;
}
```

   来剔除不是原生的字段
3. BigDecimal在比较的时候，我都用equals比较值大小是否相等，但其实不能这样，BigDecimal的equals除了比较值的大小，还会比较精度，比如1.2和1.20就是不想等的，解决方法就是用compareTo，它会把两个数都转成高精度，在相同精度下进行比较。

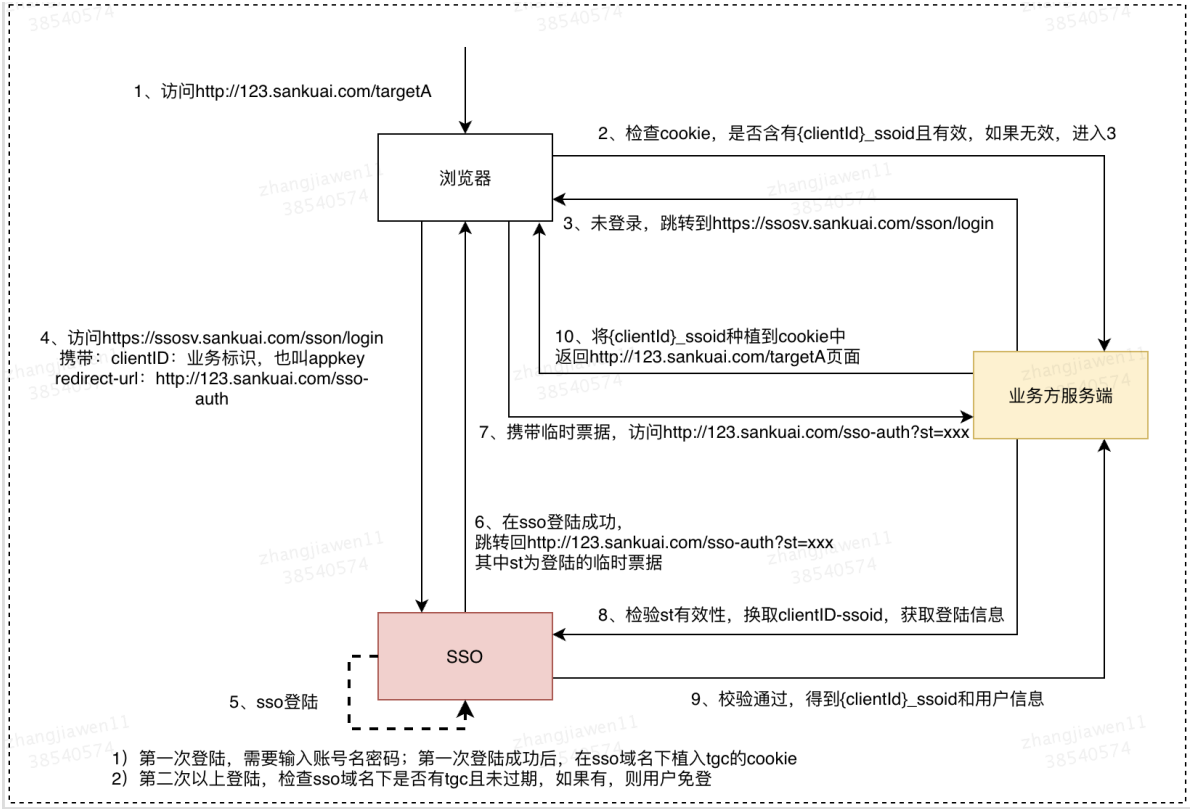
- **实习中学到了什么?**

- **你用过哪些学习网站呢？平时用他们干什么?**

- **sso**

https://developer.aliyun.com/article/636281

https://dinika-15.medium.com/single-sign-on-with-cas-central-authentication-service-fd60bae64fa7



无登录态或登录态失效，登录app1　需要用户登录

总结：首次访问SP（server provider）时，SP检查cookie是不是含有有效的token，无效则返回302，重定向到sso统一登录页面，用户扫码或者输入用户名密码之后，sso登录中心会做验证，如果登录成功会为在登录中心保存一个ticket，这个ticket就是登录的临时票据，默认会保存14天，这时候有了ticket，会携带着ticket去回调访问SP，SP再去SSO登录中心验证票据有效性，换取token，拿到用户信息，最后一步是将token放到客户端的cookie里面，下一次客户端来请求的时候只需要验证这个token就可以了。

1. 尚未登录过内网系统，用户访问app1，app1需要登录
2. 跳转到登录页
3. 用户登录
4. 登录成功，在SSO登录中心域名下种TGC的cookie
5. 携带code票据重定向回业务callback页面
6. 用code去SSO服务端换取token，在app1的域名下种下key为${clientid1}_ssoid的cookie，登录完成
7. 重定向回访问页，携带${clientid1}_ssoid的cookie访问app1

已登录app1，再次登录app1（ssoid有效） 无需用户登录

1. 再次访问app1
2. SSO服务端验证${clientid1}_ssoid有效性，有效则可直接访问

已登录app1，登录app2（有登录态且登录态有效） 无需用户登录

1. 访问app2
2. 重定向至登录页，TGC有效
3. 携带code票据重定向回业务callback页面
4. 用code去SSO换取token，在app2的域名下种key为${clientid2}_ssoid的cookie，登录完成
5. 重定向回访问页，携带${clientid2}_ssoid的cookie访问app2