## --Jvm

2019/4/10

按照jvm管理规范，Jvm运行时会把内存划分为不同的数据区域：

**堆**：被所有线程共享的一块内存区域。内部存储几乎所有的对象实例以及数组。可以不是物理上连续的，只要逻辑连续即可。

**栈**：为JVM执行java方法服务，每当有一个方法执行，就会有一个该方法自己的栈帧，底层上讲，一个方法从开始到结束，就是一个栈针从入栈到出栈的过程。该内存中的数据是线程私有的。并且与线程的生命周期相同。**栈帧**包含局部变量表（包含各种基本数据类型，对象引用，和returnAddress），操作数栈，动态链接，方法出口等。

**程序计数器**：这块内存也是线程私有的，用于表明该线程执行到的字节码的位置。因为多个线程之间是相互使用cpu时间片的，所以在来回切换线程的时候就可以通过程序计数器来找到当钱线程执行到的位置。在执行java访法的时候指定的是正在执行的jvm字节码指令的地址，而在执行本地访法的时候，则为null。

**方法区**：class文件在jvm加载后，该文件所表达的信息就会以另一种形式存在与方法区中，也就是说，该内存区中存储的是“类对象”。此处是所有线程共享的。**运行时常量池归属于方法区。**

**本地方法栈**：该区域是与虚拟栈发挥的作用是非常相似的，只不过该区域是给本地方法服务，而虚拟机站是给java方法服务。

2019/4/11

**虚拟机栈中的栈元素**：虚拟机栈帧

**虚拟机栈帧**：栈针是一个支持虚拟机进行方法调用和方法执行的数据结构。对于执行引擎来说，只有在虚拟机栈顶部的栈帧才是有效的，它只会对顶部的栈帧进行操作。

**局部变量表**: \*存储着方法参数和方法中定义的局部变量。包含基本数据类型 和reference，returnAddress这三大类。其中，由于局部变量表的最小单位slot（变量槽）可以存放32位以内的数据类型，因此long，double类型是进行分割存储。\*如果当前正在执行的方法是实例方法，那么局部变量表的第0索引位的是一个指向对象实例的引用，在方法中可以通过”this”关键字来访问。然后再按照方法参数变量表的顺序依次从第1为开始存放。接下来再按照方法体内部定义的变量表的变量顺序分配其余的slot。\*Slot是可以复用的。

**操作数栈：**操作数栈是一个后进先出的栈，jvm虚拟机是从操作数栈中获取数据的，而不是寄存器。因此，JVM是“基于栈的执行引擎”。虚拟机把操作数栈作为它的工作区——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。**基于代码的执行过程以后做笔记。**

**动态链接：\*其实就是一个对方法的引用。**栈帧中存储着当前栈帧所属方法区中**方法的引用**（注意，是“引用”，该“引用就是为了支持方法调用过程中的动态链接**。\***Class文件中有个常量池，这个常量池里面有一堆符号引用，这些符号引用在程序每一次运行期间转化为直接饮用，就叫做动态链接。。

**方法返回地址：** 指的就是当前方法执行完毕后（不管是正常完成还是异常完成），JVM需要返回到该方法被调用的位置，这个位置所处的地址就是方法返回地址。

**附加信息：**虚拟机规范允许具体的虚拟机实现可以增加一些别的信息到栈帧中。

2019/4/16

**方法调用：**方法调用不是方法执行，在方法调用阶段唯一的任务就是确定方法的版本，也就是调用哪个方法，不涉及方法具体的执行过程。

**方法调用**按分析过程可以分为**解析调用**和**分配调用，**但是**二者不是排他的，比如静态重载方法，在编译的时候就需要通过静态解析+静态分配的过程才嫩确定唯一。**

**解析调用：\***解析调用是指调用那些在编译时就可以唯一确定得方法。这类方法都是“编译器可知，运行期不变”。\*因为在编译器就完全确定，因此可以说解析调用是个静态的过程，因此，**解析调用也可以成为静态解析，或者静态连接**。\*由于编译器就确定，所以对应到的字节码指令就是invokestatic、invokespecial。对应的方法就是类方法、私有方法、实例构造器、父类方法4类。

**分派调用：**\*分派调用可能是静态的，也可能是动态的，并且依据总量的数量是否多于一个来分为单分派和多分派。\*

**静态分派：**\*重载是其一种典型的应用，或者说是jvm对重载的实现。\*在invokevirtual字节码指令后面写该方法的符号引用\*依赖静态类型来定位方法执行版本的分派动作称为静态分派。\*在代码中，对于一段“对象实例调用自己的一个函数”的代码，可以直接确定接受者，同时在编译期就可以确定方法参数的类型和数量，因此，编译器在编译的时候就能确定这个被调用的实例的方法是具体的哪一个重载方法，所以静态分配是在编译器中完成的，而不是jvm虚拟机完成的。\*由于字面量可以在不声明定义下直接使用，所以会没有显示的静态类型，这种情况下，编译器会寻找“比较合适”的方法版本。

**动态分派：**\*jvm对重写的实现。\*在invokevirtal指令中调用。\*编译时候不能确定方法的具体位置，只能通过编译时判断的接受者的类型和参数类型及参数数量来定位一个位置，并且在invokevirtal指令后面标有DynamicDispacter的分配标识。然后在jvm执行的时候，会动态的来判断对象的实际类型，如果找到与常量中的描述符和简单名称都相符的方法，则校验访问权限，通过了就执行，否则抛异常；如果在类型中没有找到，就寻找该类型的父类型，依次类推，直到找到后执行；如果还是没找到，抛出异常。\*由于动态调用需要花时间去分析类的来源书来找到目标方法，比较繁琐，所以jvm（注意，是jvm！不是编译器！）会有个优化的手段，比如在方法区中加一个虚拟方法表，利用查表的索引过程取代对元数据的查询过程。

2019/4/18

操作数栈的代码执行细节：

**提问：**

1. **在实例方法执行的时候，开始计算实例的成员变量，那么这个变量的值在参与运算的时候，是存放在哪里的呢？**

Java堆的分类：

新生代

老年代

## --多线程

2019/4/10

**线程的生命周期**

**创建**：刚刚被创建的时候，如new Thread()。

**就绪**：在执行了run()后，线程等待分配时间片。

**执行**：线程已经开始执行自己的内容了。除非自己放弃时间片或者有更高优先级的线程进来，线程将一直执行到结束。

**死亡**：线程执行完毕（run方法结束）或者被别的线程杀死（被调用了自己的stop()方法）。

**阻塞**：不能执行当前的任务，暂停执行，让出cpu时间片。

睡眠：sleep()方法被调用，让出cpu时间片，但是如果他是包含在synchronized里面的话，锁是不会被释放的。

等待：线程在执行的过程中，执行了某个object的wait()方法（由于调用wait方法的对象必须是被锁住的，所以该方法必须是在同步块或方法中使用），就会让出时间碎片，释放锁。

被另一个线程阻塞-调用suspend方法。

2019/4/11

**线程池**

线程池通过复用线程，避免线程频繁创建和销毁。

Java的Executors工具类中，提供了5种类型线程池的创建方法，它们的特点和适用场景如下：

第1种是：固定大小线程池，特点是线程数固定，使用无界队列，适用于任务数量不均匀的场景、对内存压力不敏感，但系统负载比较敏感的场景；

第2种是：Cached线程池，特点是不限制线程数，适用于要求低延迟的短期任务场景；

第3种是：单线程线程池，也就是一个线程的固定线程池，适用于需要异步执行但需要保证任务顺序的场景；

第4种是：Scheduled线程池，适用于定期执行任务场景，支持按固定频率定期执行和按固定延时定期执行两种方式；

第5种是：工作窃取线程池，使用的ForkJoinPool，是固定并行度的多任务队列，适合任务执行时长不均匀的场景。

Tomcat中servlet是多线程？

## --NIO

**阻塞非阻塞**：在数据准备好之前，可以做别的事情就是非阻塞，否则就是阻塞。

**同步异步**：主动的来判断数据是否准备好，就是同步；如果是被动通知，则是异步。

以烧开水为例：

什么都不做，一直盯着水开，就是同步阻塞。

什么都不做，等着水壶通知，就是异步阻塞。

做别的，偶尔来看水开了没，同步非阻塞。

做别的，等水壶通知我开，异步非阻塞。

**Java NIO（java new IO， Non-block IO）**，一个新的java IO API：一个线程管理多个链接。

**BIO（blocking IO）**：一个线程管理一个链接。

**Java NIO与IO的最大区别就是**，IO面向流，而NIO面向缓冲区。

在Java1.4之前的I/O系统中，提供的都是面向流的I/O系统，系统一次一个字节地处理数据，一个输入流产生一个字节的数据，一个输出流消费一个字节的数据，面向流的I/O速度非常慢，而在Java 1.4中推出了NIO，这是一个面向块的I/O系统，系统以块的方式处理处理，每一个操作在一步中产生或者消费一个数据库，按块处理要比按字节处理数据快的多。

## --网络

<https://www.cnblogs.com/qishui/p/5428938.html>

2019/4/10

**协议栈/协议族：**为了完成通信，必须使用多层上的多种协议。这些协议按照层次顺序组合起来，构成协议栈，或者叫协议族。

**TCP/IP协议族的组成：**

**应用层**：如HTTP，FTP，SMTP，DNS，WebSocket等。

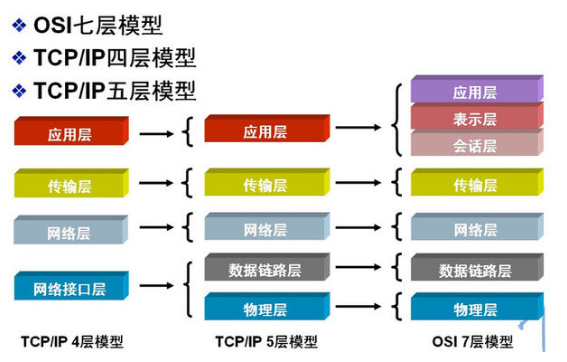
**传输层**：tcp ，UDP

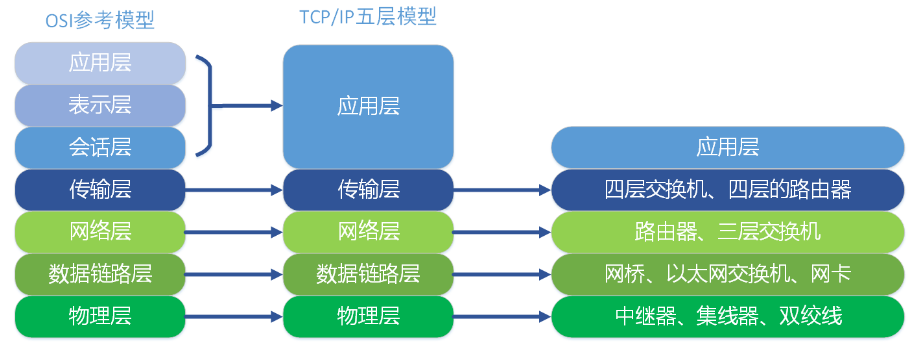
**网络层**：IP，ICMP

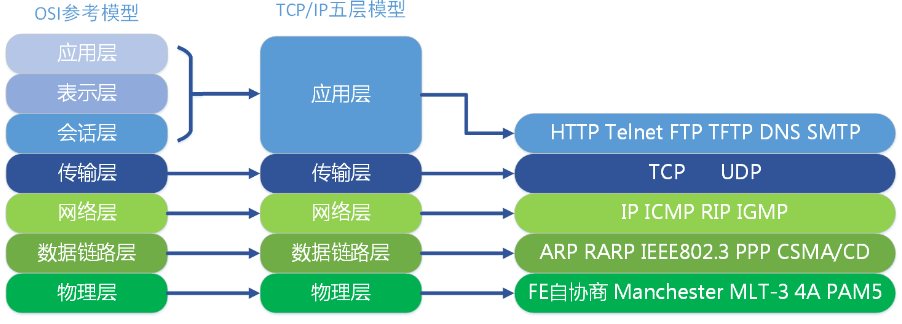
**链路层**：\

ARP ，RAR（这两个在tcp/ip中是网络层的，在osi中是链路层的）

**物理层**：/







**TCP协议**

是一个面向连接的可靠的基于字节流的传输层协议。用于**程序**之间（也就是进程到进程）的通信。

Tcp把应用层的传给它的字节流分解为多个报文段，然后把数据包传送给ip层，由ip层把数据包路由到指定的地方。

发送请求的过程是从最顶层（应用层）出发，每一层负责封装属于自己的信息到请求中，最后将一整个请求发送给对方。  
接收请求的过程是从最底层（网络接口层）开始，每一层的协议负责解析属于自己的东西，比如网际层（IP）处理ip信息，传输层（TCP）处理点对点的端口，应用层（HTTP）处理Request或Response的Line\Header\Body。

**Socket—不是协议，而是一组接口**

Socket是将应用层与传输层及以下分开的中间软件抽象层，它是一组接口，将tcp/ip的细节隐藏在背后，提供给用户接口。

**WebSocket协议—应用层协议**

Websocket是应用层协议；建立完成之前，需要借助http协议，请求者请求升级协议，响应者反馈可以升级（双方的header中都会出现：Connection：Upgrade 和 Upgrade： websocket），从此双方建立websocket协议，http就没事了。

客户端请求

GET / HTTP/1.1

Upgrade: websocket

Connection: Upgrade

Host: example.com

Origin: http://example.com

Sec-WebSocket-Key: sN9cRrP/n9NdMgdcy2VJFQ==

Sec-WebSocket-Version: 13

服务器回应

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: fFBooB7FAkLlXgRSz0BT3v4hq5s=

Sec-WebSocket-Location: ws://example.com/

**轮询，流：（好的实现js + java=>**<https://blog.csdn.net/z69183787/article/details/21315685>）

实现双方实时数据通信的方式：**ajax短轮询**（服务器立即响应，不管发送的数据是否有效。HTTP1.1之后支持了TCP长连接，也就是建立一个tcp链接后，可以发送多次http请求），**ajax长轮询【comet（服务器推送）的一种方式】**（等到服务端更新了有效数据后响应。服务器接受到请求后会把当前请求挂起一段时间，然后在有效期内有更新的数据就把数据返回，没有就直到超时断开链接），**HTTP流【comet（服务器推送）的另一种方式：分为streaming=>监听readystate==3的状态，随后就可以做后续处理；和iframe-streaming=> iframe中设置src的属性到为口地址，js的处理为定义一个处理数据的函数，然后这个数据被服务端返回的html元素中调用该函数，类似于jsonp的调用方式】**（与长轮询差不多，但是区别在于，长轮询在服务端发送数据给前端后，前端会发送下一次请求，而流则是服务端会连续发送会多个response后，前端才会发送下一次请求，由此可见，流适合用于大数据的传输。）

**长短轮序与长短链接的区别：长短链接是通过tcp协议的规范来实现的，需要在请求头的connection Header中设置keep-alive来决定的。而长短轮询是由服务端的编程手段，是根据服务器的处理手段：是否将当期请求挂起 来实现的。**

## --Spring

## --集合相关工具类的使用

## --并发包中类的原理

**HashMap**

1.7与1.8中的区别在于：

a. 1.,8中的是由数组+链表+红黑树组成。

b. 1.7是先扩容在加，1.8是先加再扩容

**concurrentHashMap**

## --数据结构

## --Sql练习题

## --正则表达式练习题

## --数据库知识（事务，一致性，分布式，主从复制等都可以）

## --一天一个算法学习或者习题联系

排序算法

查找算法

## --了解队列相关的使用（现在以rabbitMQ和kafka为主，了解其全部特性和使用方法）

## --springcloud

## --设计模式

## --OAuth