

操作系统

2018年2月28日 22:05

目录

[进程、线程概论区别](#)
[进程、线程详细区别](#)
[寻址空间](#)

[进程间通信（进程间同步机制）](#)
[进程间通信—应用场景](#)

[进程互斥](#)
[进程互斥的具体方法](#)

[操作系统层面上的线程同步](#) [用户态和内核态](#)

[BIO、NIO、AIO](#)
[I/O操作](#)
[I/O多路复用](#)
[select\poll\epoll](#)
[epoll概述](#)
[epoll底层实现原理](#)
[epoll的水平触发和边缘触发](#)

什么情况下选多线程、什么情况下选多进程
IP路由转发

进程、线程概论区别

一句概况的总论：进程和线程都是一个时间段的描述，是CPU工作时间段的描述。

1. CPU执行任务的过程：先加载程序A的上下文，然后开始执行A，保存程序A的上下文，调入下一个要执行的程序B的程序上下文，然后开始执行B，保存程序B的上下文。 . . .
2. 所以**进程就是包换上下文切换的程序执行时间总和 = CPU加载上下文+CPU执行+CPU保存上下文**
3. 那么线程是什么呢？程序A分为a、b、c等多个块组合，CPU加载程序A上下文，开始执行程序A的a小段，然后执行A的b小段，然后再执行A的c小段，最后CPU保存A的上下文。
4. 这里a、b、c的执行是共享了A的上下文，CPU在执行的时候没有进行上下文切换的。这里的a、b、c就是线程，也就是说**线程是共享了进程的上下文环境的更为细小的CPU时间段。**

这里的上下文指什么？

一句话概括就是**描述进程的信息**。是当进程要切换时关于当前进程的寄存器内容以及内存页表的详细信息等内容，因为内核切换一个进程时，为了保证下次切回来能回到原状态，必须保存当前进程的所有状态，这里的所有状态就是上下文。

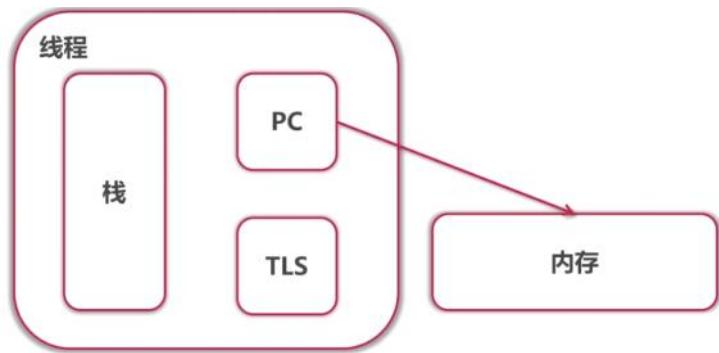
进程、线程系统层面的区别

进程和线程的主要差别在于它们是**不同的操作系统资源管理方式**。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其他进程产生影响。而线程只是一个进程中的不同执行路径，是一种轻量级的进程，线程没有地址空间，线程包含在进程的地址空间中，一个线程死掉就等于整个进程死掉。

进程、线程各自的描述

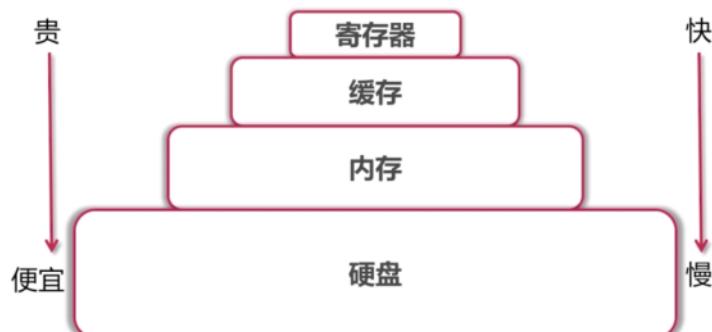


- 1、**线程是进程里面的一部分**，一个操作系统可能同时有几十到几百个进程在运行，每个进程里面可能又有几个到甚至上百个线程在运行。
- 2、32位操作系统，进程总共有4G的内存空间可供寻址，即单个进程可占用的理论最大内存。
- 3、**每个进程自己的内存都是互相独立的**。比如微信不可能读我网银的内存，不然钱都跑了。
- 4、文件/网络句柄都是进程共享的，即不同的进程都能访问同一个文件或者网络端口。
- 5、因为进程不共享内存，所以进程之间的交互，比较常见的就是通过TCP/IP端口来实现。
- 6、进程因为要分配这么多内存，所以开销大。
- 7、**绝大多数操作系统调度单位是线程、不是进程**



- 1、线程含有“栈”或者称“调用堆栈”，主线程入口的main函数，会不断的进行函数调用，每次调用会把所有的参数和返回地址一层层的压到“栈”里面去，包括每个函数内部的局部变量也会放到这个“栈”里面。
- 2、线程含有PC (ProgrammaCounter)，里面放当前或者下一条指令的地址。**所以我们操作系统真正运行的是一个一个线程，进程只是一个容器**。PC这个指针放在内存中，PC的指针就是指向内存的。因此经常听到一个漏洞叫**缓冲区溢出**，比如用户名的位置，黑客把用户名输的特别特别长，这个长度超出了我给用户分配的缓冲区，一直往后写写到了存储程序的那部分内存去了，黑客就通过这种方法植入代码，当然防止的方法是，一定要检查用户名长度。
- 3、TLS是各线程之间独立的内存空间，可存变量、数据，这些数据就是单个线程独有的。
- 4、**因为线程除了TLS其他是共享内存，因此线程间通信就是各自指针指向同一块内存。**
- 5、线程因为只需要分配一个“栈”，一个ProgrammaCounter，因此开销小。
- 6、系统进行资源分配和调度的时候同时考虑线程和进程
- 7、线程是程序的多个顺序的流动态执行

存储



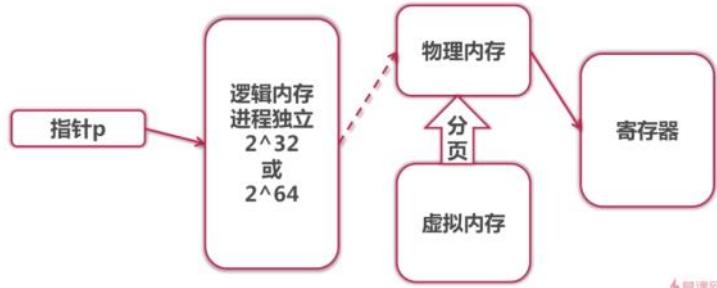
- 1、寄存器最快，就在CPU运算模块的旁边
- 2、CPU里面的缓存，有些缓存是各个CPU核心共有的，有些缓存是单个核心独有的。
- 3、为什么不用最快的内存呢？因为贵呀。土豪如Google就是把主要的数据存在内存里，所以我们用谷歌搜索很快。

寻址空间

操作系统怎么来寻址？就是给了一个地址，操作系统如何找到这个地址

寻址空间就是每一个进程里的指针可以取到的地址的范围。32位操作系统是4G

寻址 `int n = *p; → MOV EAX, [EBX]`



上图就是一个寻址过程

- 1、我们有一个指针p，他所指向的内存不是物理内存，而是逻辑内存，只是在逻辑上存在，是进程独立的，每个进程都有自己的逻辑内存，根据操作系统是32位还是64位有不同的大小。
- 2、接下来我们需要把逻辑内存和实际物理内存建立一个关系，我们就能通过逻辑内存找到物理内存了。
- 3、但是逻辑内存对应的地址可能在物理内存里，也可能不在物理内存里，而是在硬盘上的虚拟内存里，因为物理内存空间有限，操作系统会在硬盘上开辟一个空间作为虚拟内存。
- 4、如果在虚拟内存里，我们就要把虚拟内放到物理内存中去。放的时候，这里是int，我们不可能只把4字节放过去，这样开销太大。所以虚拟内存有一个“分页”（下面还有分段、分段页）的概念，我们不是把p所指向的内存放过去，而是把p所指向的内存所在的分页整个放过去。分页根据操作系统配置可能有几KB也可能几MB。
- 5、如果物理内存放不下分页，那么我们就要通过算法把物理内存中很久没有用的一块内存区域交换进虚拟内存
- 6、在物理内存确保了p所在的数据后，就取出来放入寄存器里，就好了。
- 7、因为存在“交换”，这就是为什么内存不够了，系统就很慢了，因为频繁交换分页。

分页、分段、分段页

调度方式（页式、段式、段页式）

有分页式、段式、段页式3种

页式：

页式调度是将逻辑和物理地址空间都分成固定大小的页。主存按页顺序编号，而每个独立编址的程序空间有自己的页号顺序，通过调度辅存中程序的各页可以离散装入主存中不同的页面位置，并可据表——对应检索

段式：

按程序的逻辑结构划分地址空间，段的长度是随意的，并且允许伸长，它的优点是消除了内存零头，易于实现存储保护，便于程序动态装配；缺点是调入操作复杂。

段页式：

把物理空间分成页，程序按模块分段，每个段再分成与物理空间页同样小的页面

分页和分段有什么区别？

段是信息的逻辑单位，它是根据用户的需要划分的，因此段对用户是可见的；页是信息的物理单位，是为了管理主存的方便而划分的，对用户是透明的。

段的大小不固定，由它所完成的功能决定；页大小固定，由系统决定

段向用户提供二维地址空间；页向用户提供的是一维地址空间

段是信息的逻辑单位，便于存储保护和信息的共享，页的保护和共享受到限制。

进程间通信

能通信就一定是一种同步机制，这是显而易见的。

- 1、文件。我写一个文件，你打开这个文件

- 2、Signal Signal是一个进程给另一个进程发的信号，是一个数字，代表一些特殊的含义。比如Linux中通过kill命令向另一个进程发Signal

```
1      HUP (hang up)
2      INT (interrupt)
3      QUIT (quit)
6      ABRT (abort)
9      KILL (non-catchable, non-ignorable kill)
14     ALRM (alarm clock)
15     TERM (software termination signal)

58649 ttys001    0:00.06 python
hegzdeMacBook-Pro:tmp huj$ kill -l
 1) SIGHUP        2) SIGINT        3) SIGQUIT
 5) SIGTRAP       6) SIGABRT       7) SIGEMT
 9) SIGKILL       10) SIGBUS       11) SIGSEGV
13) SIGPIPE       14) SIGALRM       15) SIGTERM
17) SIGSTOP       18) SIGTSTP       19) SIGCONT
21) SIGTTIN       22) SIGTTOU       23) SIGIO
25) SIGXFSZ       26) SIGVTALRM    27) SIGPROF
29) SIGINFO       30) SIGUSR1      31) SIGUSR2
hegzdeMacBook-Pro:tmp huj$ kill -2 58649

https://coding.imooc.com/lesson/132.html#mid=8905 5分钟
```

最常用的是kill -9 pid

为什么可以断掉程序

处理器，最终会把中断作为一个exception送到用户执行的代码中来。
中断是个什么东西？中断的概念和流程！！！查《操作系统》教科书

当启动列 消启动列 就是 一个进程向另一个进程发送消息

管道 (PIPE) 、命名管道 (NAMED PIPE) 、管道PIPE、命名管道NAMED PIPE

4、管道 (PIPE)、命名管道 (NAMED PIPE) 管道PIPE 命名管道NAMEDPIPE 即命名管道通常都是单向的，只能用于有亲缘关系的进程通信，命名管道可能是单向的也可能是双向的，可以用于非亲缘关系进程通信。

比如这里，`cat`命令是打开一个文件，`grep`是找到“ERROR”，`color`是变色。

比如这里，cat命令是打开一个文件，grep -e 是找到 ERROR --color 是支持高亮输出的命令，从而输出带有颜色的文本。

中间的|就是管道，这里就是打开adobegc.log文件找到ERROR关键字升高为

<https://coding.imooc.com/lesson/132.html#mid=8905> 10分钟

```
cat adobegc.log | grep -e "ERROR" --color | wc
```

wc命令是统计

5、**共享内存** **共享内存** 两个进程都同意我们共享某块内存，当然就可以共享。

6、信号量 信号量

7、最重要的！！！Socket Socket。在机器上开一个端口作为服务器让客户连接，走TCP/UDP协议。这就是不同机器之间进程通信了。

各通信方式应用场景

1. **管道、命名管道**: 适合非常短小、频率很高的消息，而且只能是两个进程之间
 2. **共享内存**: 适合非常庞大的、读写操作频率很高的数据（配合信号量使用），常见于多进程之间。
 3. **消息队列**: 不建议使用
 4. **其他的考虑用Socket**, 在多进程、多线程、多模块所构成的今天最常见的分布式系统开发中, **Socket是第一选择**。

实际使用中！！！！！！！！！！！！！！！！！！！！！！！！

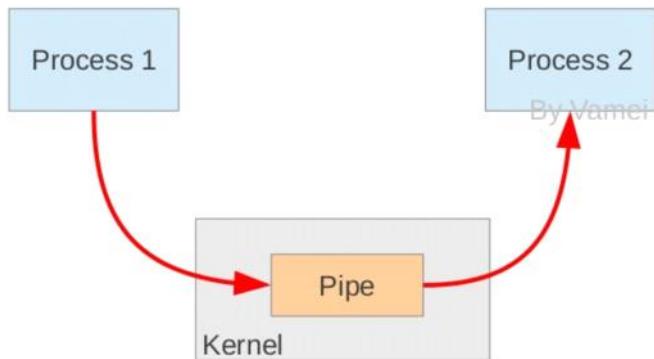
实际使用中，除非你有非常有说服力的理由，否则请用Socket。原因如下：

1. 虽然PIPE\FIFO理论上比Socket快，但现在计算机上真心不计较这么一点点速度损失的。你费劲纠结半天，不如我把socket设计好了，多插一块CPU来得更划算。
 2. 另外，进程间通信的数据一般我们都会存入数据库，这样万一哪个进程挂掉了，也不至于丢失数据，从这个角度考虑，适用共享内存的场景就更少了。
 3. 而且，PIPE和共享内存是不能跨网LAN的，虽然FIFO可以跨网，但代码可读性、易操作性、可移植性远不如Socket。
 4. 最后，信号也如Socket，信号不能跨LAN，信息量极其有限。

管道如何通信

管道是由内核管理的一个缓冲区，相当于我们放入内存中的一个纸条。管道的一端连接一个进程的输出，另一端连接一个进程的输入。

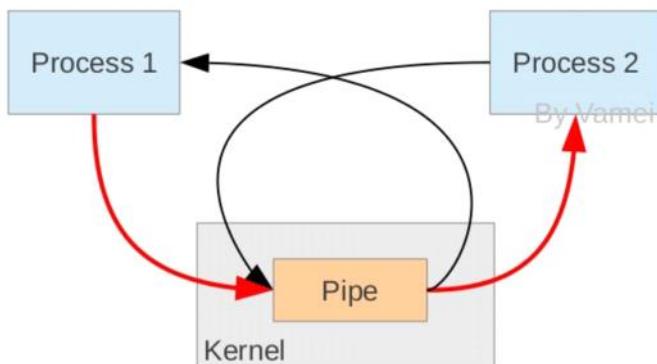
管道这个缓冲区不需要很大，它被设计成为环形的数据结构，以便管道可以被循环利用。当管道中没有信息的话，从管道中读取的进程会等待，直到另一端的进程放入信息。当管道被放满信息的时候，尝试放入信息的进程会等待，直到另一端的进程取出信息。**当两个进程都终结的时候，管道也自动消失。**



管道如何创建？

管道利用**fork机制**（不同进程中的同一个虚拟地址被映射到不同的物理地址）建立，从而让两个进程可以连接到同一个PIPE上。

如下图，最开始的时候，左边一红一黑都连接在同一个进程Process 1上(连接在Process 1上的一红一黑)。当fork复制进程的时候，会将这两个连接也复制到新的进程(Process 2)。随后，每个进程关闭自己不需要的一个连接，两个黑色的箭头被关闭，这样，剩下的红色连接就构成了如上图的PIPE。



命名管道如何通信

管道只能用于父进程和子进程之间，或者拥有相同祖先的两个子进程之间，而命名管道可以让没有亲缘关系的进程之间通信。

命名管道又叫做**FIFO** (First in, First out)为一种特殊的文件类型，**本质是文件**。以FIFO的文件形式存储于文件系统中。命名管道是一个设备文件，有名字，因此，**不相关的进程可以通过打开命名管道进行通信**，即使进程与创建FIFO的进程不存在亲缘关系，只要可以访问该路径，就能够通过FIFO相互通信。值得注意的是，FIFO(first input first output)总是按照**先进先出**的原则工作，第一个被写入的数据将先从管道中读出。

信号量

定义：信号量是一个特殊的变量，用来协调进程对共享资源的访问，确保一个**临界区**同一时间只有一个线程在访问。

- 最简单的信号量是只能取0和1的变量，这也是信号量最常见的一种形式，叫做二进制信号量。而可以取多个正整数的信号量被称为通用信号量。
- 信号量只能进行两种操作：等待和发送信号
- 举个例子，就是两个进程共享信号量sv，sv=1，第一个进程得到了这个信号量，可以进入临界区域，并使sv减1变为0。第二个进程一看信号量sv=0，就会被挂起以等待第一个进程离开临界区，第一个进程离开临界区的时候会把sv+1。

名词解释：

某些资源同一时间只能被一个进程占用，这些资源叫**临界资源**。进程内访问临界资源的代码被称为**临界区**。

消息队列

消息队列是链表队列

消息队列跟命名管道有不少的相同之处

1. 与命名管道一样，消息队列进行通信的进程可以是不相关、无亲缘关系的进程

- 同时它们都是通过发送和接收的方式来传递数据的。
- 而且它们对每个数据都有一个最大长度的限制。

消息队列优势：

- 消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- 除开发送、接收进程，消息队列可以独立存在。管道是发送、接收进程关闭了，管道就自动关闭了。
- 因为消息队列，避免了命名管道同步和阻塞问题
- 接收可以通过消息类型有选择地接收数据，而不是像命名管道中那样，只能默认地接收。

Signal信号

信号是进程间通信机制中唯一的异步通信机制，一个进程不必通过任何操作来等待信号的到达。

信号的种类

可以从两个不同的分类角度对信号进行分类：

一、可靠性方面：可靠信号与不可靠信号；

- 信号分为可靠和不可靠的，早期的信号比较原始，容易丢失，因此不可靠。后期新增了一些信号，这些信号支持排队，不会丢失，直接定义为可靠信号。
- 信号值小于SIGRTMIN=32都是不可靠信号，信号值位于SIGRTMIN=32和SIGRTMAX=63之间的信号都是可靠的。信号的可靠与不可靠只与信号值有关，与信号的发送及安装函数无关。

二、与时间的关系上：实时信号与非实时信号。

非实时信号都不支持排队，都是不可靠信号，编号是1-31,0是空信号；实时信号都支持排队，都是可靠信号。

信号的发送和安装

- 发送信号的主要函数有：kill()、raise()、sigqueue()、alarm()、setitimer()以及abort()。
- 如果进程要处理某一信号，那么就要在进程中安装该信号。安装信号主要用来确定信号值及进程针对该信号值的动作之间的映射关系，即进程将要处理哪个信号；该信号被传递给进程时，将执行何种操作。

共享内存

共享内存是最有用的、最快的进程间通信方式。是针对其他通信机制运行效率较低而设计的。两个不同进程A、B共享内存的意思是，同一块物理内存被映射到进程A、B各自的进程地址空间。进程A可以即时看到进程B对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。

套接字Socket

网络上的两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个Socket。Socket通常也称作“套接字”，用于描述IP地址和端口，是一个通信链的句柄，可以用来实现不同虚拟机或不同计算机之间的通信。

分为两种

- 流式Socket(STREAM)：是一种面向连接的Socekt，针对面向连接的TCP服务应用，安全，但是效率低；
- 数据报式Socket(DATAGRAM)：是一种无连接的Socket，对应于无连接的UDP服务应用。不安全(丢失，顺序混乱，在接受端要分析重排及要求重发)，但效率高。

多个进程处理一个Socket（端口）？

计算机上不同服务调用不同的端口（Tomcat和Nginx端口就不一样），同一种服务的不同进程也要用不同的端口（Tomcat集群端口不一样），即一个服务只能用一个端口。但一个端口可以同时连接N多个用户请求。

多个线程处理一个Socket（端口）？？？

- 对于UDP，多个线程读写一个Socket不用加锁，当然最好的做法是每个线程有自己的Socket。
- 对于TCP，多个线程处理一个Socket是错误的设计。
- 总结：对于UDP，加锁是多余的，对于TCP，加锁是错误的。

进程间互斥

当一个进程进入临界区使用临界资源时，另一个进程必须等待。只有当使用临界资源的进程退出临界区后，这个进程才会解除阻塞状态。比如进程B需要访问打印机，但此时进程A占有了打印机，进程B会被阻塞，直到进程A释放了打印机资源，进程B才可以继续执行。

竞争条件: 即两个或者多个进程读写某些共享数据，而最后的结果取决于各个进程运行的精确时序，称为竞争条件。

一组并发进程互斥执行时必须满足如下准则：

1. **平等竞争**: 不能假设各并发进程的相对执行速度。即各并发进程享有平等地、独立地竞争共有资源的权利，且在不采取任何措施的条件下，在临界区内任意指令结束时，其他并发进程可以进入临界区。
2. **不可独占**: 并发进程中的某个进程不在临界区时，它不能阻止其他进程进入临界区。
3. **互斥使用**: 并发进程中的若干个进程申请进入临界区时，只能允许一个进程进入。
4. **有限等待**: 并发进程中的某个进程从申请进入临界区时开始，应在有限时间内得以进入临界区。

实现进程间（临界区）互斥的基本方法

1. **屏蔽中断**: 进程进入临界区后立即屏蔽所有中断，离开后打开中断。**缺点**是多核系统无效，并且用户程序控制中断很危险（试想一下，一个进程屏蔽中断后不再打开中断，那你的系统就GG了）。
2. **互斥加锁**: 对临界区加锁以实现互斥。当某个进程进入临界区后，它将锁上临界区，直到它退出临界区为止。**加锁的缺点**: 1、有可能出现其中一个进程的执行，导致另一个进程长期得不到处理机资源，而处于永久饥饿状态（starvation）。2、如果一个进程进入临界区，但是在它把锁变量置1之前被中断，另一个进程进入临界区后将0置1，这样，当前一个进程再次运行时它也将锁变量置1，这样临界区内依然存在两个进程。**如何解决**? 我们可以为临界区设置一个管理员，由这个管理员来管理相应临界区的公共资源，这个管理员就是信号量。
3. **信号量**: 信号量sem是一个整数，信号量的数值仅能由P、V原语操作改变。原语操作都是原子性的。

设置信号量初始值sem=1;

P操作（down）：申请进入临界区

sem=sem-1

判断sem≥0

TRUE: 继续（执行临界区命令）

FALSE: 阻塞进入队列挂起

V操作（up）：退出临界区

sem=sem+1

判断sem > 0

TRUE: 继续（执行退出临界区之后的命令）

FALSE: 唤醒队列上阻塞的进程，再执行退出临界区之后的命令

被唤醒的进程直接进入临界区，要退出的时候再执行V操作。

实例:

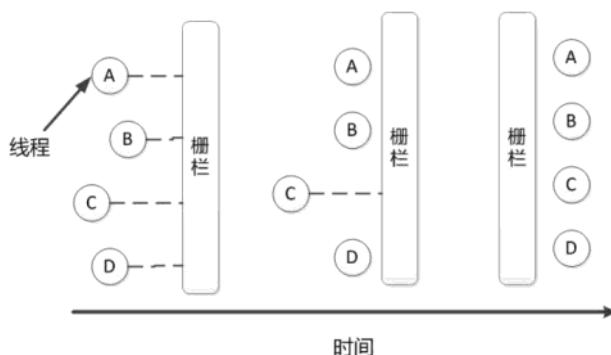
- i. 初始sem=1，有进程A\B\C。
- ii. A进来P操作，sem=0，进入临界区。B进来P操作，sem=-1，挂起；C进来P操作，sem=-2，挂起。
- iii. A出临界区V操作，sem=-1，唤醒挂起的进程，比如B。B被唤醒了直接进入临界区，退出时V操作，sem=0，唤醒挂起的进程C。C被唤醒了直接进入临界区，退出时V操作，sem=1。回到初始状态

操作系统层面上的线程同步

1. **锁**: 锁有两个基本操作，闭锁和开锁。而闭锁有两个步骤：一是等待锁达到打开状态，二是获得锁并锁上。显然，闭锁的两个操作应该是原子操作，不能分开。当对方持有锁时，你就不需要等待锁变为打开状态，而是去睡觉，锁打开后对方再来把你叫醒。
2. **信号量**: 和进程间同步一个概念，sem、P、V原语
3. **管程（Monitor）即监视器的意思**:
 - a. 它监视的就是进程或线程的同步操作。具体来说，管程就是一组子程序、变量和数据结构的组合。言下之意，把需要同步的代码用一个管程的构造框起来，即将需要保护的代码置于begin monitor和end monitor之间，即可获得同步保护，也就是任何时候只能有一个线程活跃在管程里面。
 - b. 在管程中使用两种同步机制：锁用来进行互斥，条件变量用来控制执行顺序。从某种意义上来说，**管程就是锁+条件变量**。
 - c. 条件变量就是线程可以在上面等待的东西，而另外一个线程则可以通过发送信号将在条件变量上的线程叫醒。因此，条件变量有点像信号量，但又不是信号量，因为不能对其进行up和down操作。
 - d. 管程只能在单台计算机上发挥作用，如果想在多计算机环境下进行同步，那就需要其他机制了，而这种其他机制就是消息传递。
4. **消息传递**:
 - a. 消息传递是通过同步双方经过互相收发消息来实现，它有两个基本操作：发送send和接收receive。他们均是操作系统的系统调用，而且既可以是阻塞调用，也可以是非阻塞调用。而同步需要的是阻塞调用，即如果一个线程执行receive操作，就必须等待受到消息

后才能返回。也就是说，如果调用receive，则该线程将挂起，在收到消息后，才能转入就绪。

- b. 消息传递最大的问题就是消息丢失和身份识别。由于网络的不可靠性，消息在网络间传输时丢失的可能性较大。
 - c. 消息丢失可以通过使用TCP协议减少丢失，但也不是100%可靠。身份识别问题则可以使用诸如数字签名和加密技术来弥补。
5. **栅栏**: 栅栏顾名思义就是一个障碍，到达栅栏的线程必须停止下来，知道出去栅栏后才能往前推进。该院与主要用来对一组线程进行协调，因为有时候一组线程协同完成一个问题，所以需要所有线程都到同一个地方汇合之后一起再向前推进。例如，在并行计算时就会遇到这种需求，如下图所示：



I/O操作

unix(like)世界里，一切皆文件，而文件是什么呢？文件就是一串二进制流而已，不管socket,还是FIFO、管道、终端，对我们来说，一切都是文件，一切都是流。在信息交换的过程中，我们都是对这些流进行数据的收发操作，简称为**I/O操作**(input and output)

同步、异步

- **同步**: 进程执行一个操作之后，进程等待IO操作完成(也就是我们说的阻塞)或者轮询的去查看IO操作(也就是我们说的非阻塞)是否完成，等待结果，然后才继续执行后续的操作。
- **异步**: 进程执行一个操作后，可以去执行其他的操作，然后等待通知再回来执行刚才没执行完的操作。

阻塞、非阻塞

- **阻塞**: 进程给CPU传达一个任务之后，一直等待CPU处理完成，然后才执行后面的操作。
- **非阻塞**: 进程给CPU传达任务之后，继续处理后续的操作，隔断时间再来询问之前的操作是否完成。这样的过程其实也叫轮询。

TIPS:

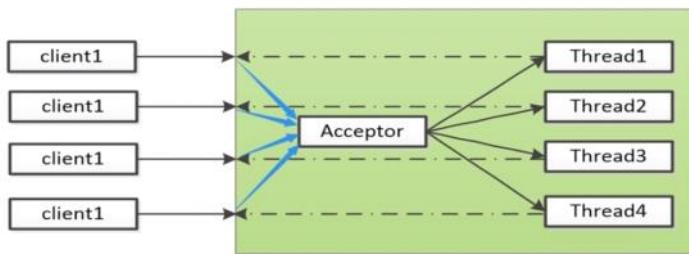
1. 同步有阻塞和非阻塞之分，异步没有，它一定是非阻塞的。
2. 阻塞、非阻塞、多路IO复用，都是同步IO，异步必定是非阻塞的。

BIO、NIO、AIO

- **同步阻塞IO (BIO)** :

我们熟知的Socket编程就是BIO，一个socket连接一个处理线程，典型的一请求一应答，缺点是缺乏弹性伸缩能力，线程可是很宝贵的资源。（这个线程负责这个Socket连接的一系列数据传输操作）。阻塞的原因在于：操作系统允许的线程数量是有限的，多个socket申请与服务端建立连接时，服务端不能提供相应数量的处理线程，没有分配到处理线程的连接就会阻塞等待或被拒绝。

BIO通信模型



- 同步非阻塞IO (NIO) : [NIO的底层实现原理](#)

一个socket连接只有在特点时候才会发生数据传输IO操作，大部分时间这个“数据通道”是空闲的，但还是占用着线程。[NIO作出的改进就是“一个请求一个线程”，在连接到服务端的众多socket中，只有需要进行IO操作的才能获取服务端的处理线程进行IO。这样就不会因为线程不够用而限制了socket的接入。](#) [epoll是NIO的具体实现。](#)

- 异步非阻塞IO (AIO) :

AIO是发出IO请求后，由操作系统自己去获取IO权限并进行IO操作，线程可以去做别的事情了，所以是异步的。NIO则是发出IO请求后，由线程不断尝试获取IO权限。

为什么NIO比BIO高效？

因为NIO一个请求一个线程，这样就不会因为线程不够用而限制了socket的接入。

NIO的底层实现原理

Java NIO的实现是这样的，NIO包中主要有Channel、Buffer、Selector这几种对象。

- **Channel**: NIO把它支持的I/O对象抽象为Channel，模拟了通信连接，用户可以通过它读取和写入数据。
- **Buffer**: Buffer是一块连续的内存区域，一般作为Channel收发数据的载体出现。所有数据都通过Buffer对象来处理。
- **Selector**: Selector类提供了监控一个和多个通道当前状态的机制。只要Channel向Selector注册了某种特定的事件，Selector就会监听这些事件是否会发生，一旦发生某个事件，便会通知对应的Channel。使用选择器，借助单一线程，就可对数量庞大的活动I/O通道实施监控和维护。

流程：

1. 向Selector对象注册感兴趣的事件
2. 从Selector中获取感兴趣的事件
3. 根据不同的事件进行相应的处理

举个例子：

服务端的selector上注册了读事件，某时刻客户端给服务端发送了一些数据，阻塞I/O这时会调用read()方法阻塞地读取数据，而NIO的服务端会在selector中添加一个读事件。服务端的处理线程会轮询地访问selector，如果访问selector时发现有感兴趣的事件到达，则处理这些事件，如果没有感兴趣的事件到达，则处理线程会一直阻塞直到感兴趣的事件到达为止。

I/O多路复用

一句话解释：单个线程，通过记录跟踪每个I/O流(socket)的状态，来同时管理多个I/O流

[select, poll, epoll 都是I/O多路复用的具体的实现。](#)

这些函数也会使进程阻塞，select先阻塞，有活动套接字才返回，但是和阻塞I/O不同的是，这两个函数可以同时阻塞多个I/O操作，而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写（就是监听多个socket）。

[正因为阻塞I/O只能阻塞一个I/O操作，而I/O复用模型能够阻塞多个I/O操作，所以才叫做多路复用。](#)

select\poll\epoll

[select, poll, epoll 都是I/O多路复用的具体的实现，epoll是NIO的具体实现](#)

[Nginx的高并发得益于其采用了epoll模型，Apache就没采用](#)

1. 因为一个线程只能处理一个套接字的I/O事件，如果想同时处理多个，可以利用非阻塞忙轮询的方式。
2. 我们只要把所有流从头到尾查询一遍，就可以处理多个流了，但这样做很不好，因为如果所有的流都没有I/O事件，白白浪费CPU时间片。
3. 正如有一位科学家所说，计算机所有的问题都可以增加一个中间层来解决，同样，为了避免这里cpu的空转，我们不让这个线程亲自去检查流中是否有事件，而是引进了一个代理(一开始是[select](#),后来是[poll](#))，[select只能监听1024个连接，而且会修改传入的参数数组，poll去](#)

- 掉了连接限制，而且不再修改传入的参数数组)，这个代理很牛，它可以同时观察许多流的I/O事件，如果没有事件，代理就阻塞，线程就不会挨个挨个去轮询了。很简单很感动后方可恢复到
4. 但是依然有个问题，我们从select\poll那里仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流。所以select具有O(n)的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。
 5. 因此有了epoll，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。复杂度降低到了O(1)。不能不说epoll跟select相比，是质的飞跃。

epoll概述

1. 设想一下如下场景：有100万个客户端同时与一个服务器进程保持着TCP连接。而每一时刻，通常只有几百上千个TCP连接是活跃的（事实上大部分场景都是这种情况）。如何实现这样的高并发？
2. 在select/poll时代，服务器进程每次都把这100万个连接告诉操作系统（从用户态复制句柄数据结构到内核态），让操作系统内核去查询这些套接字上是否有事件发生，轮询完后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，这一过程资源消耗较大，因此，select/poll一般只能处理几千的并发连接。
3. epoll的设计和实现与select完全不同。epoll通过在Linux内核中申请一个简易的文件系统，把原先的select/poll调用分成了3个部分：
 - 1) 调用epoll_create()建立一个epoll对象（在epoll文件系统中为这个句柄对象分配资源）
 - 2) 调用epoll_ctl向epoll对象中添加这100万个连接的套接字
 - 3) 调用epoll_wait收集发生的事件的连接
4. 如此一来，要实现上面说的场景，只需要在进程启动时建立一个epoll对象，然后在需要的时候向这个epoll对象中添加或者删除连接。同时，epoll_wait的效率也非常高，因为调用epoll_wait时，并没有一股脑的向操作系统复制这100万个连接的句柄数据，内核也不需要去遍历全部的连接。

应用：

Nginx的高并发得益于其采用了epoll模型，Apache就没采用，传统Apache都是多进程或者多线程来工作

epoll底层实现原理

一句话描述就是：三步曲。

第一步：epoll_create()系统调用。此调用返回一个句柄，之后所有的使用都依靠这个句柄来标识。

第二步：epoll_ctl()系统调用。通过此调用向epoll对象中添加、删除、修改感兴趣的事件，返回0标识成功，返回-1表示失败。

第三部：epoll_wait()系统调用。通过此调用收集在epoll监控中已经发生的事件。

流程概述：

epoll_create()创建一个eventpoll结构体，里面包含红黑树和双链表。epoll_ctl()将所有连接句柄数据放入红黑树，如果连接有动作，就把连接放入双链表。epoll_wait()检查双链表，有连接就返回给用户。

流程详解：

1. 当某一进程调用epoll_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关。eventpoll结构体如下所示

```
struct eventpoll{
    ...
    /*红黑树的根节点，这棵树中存储着所有添加到epoll中的需要监控的事件*/
    struct rb_root rbr;
    /*双链表中则存放着将要通过epoll_wait返回给用户的满足条件的事件*/
    struct list_head rdlist;
    ...
};
```

2. 每一个epoll对象都有一个独立的eventpoll结构体，用于存放通过epoll_ctl方法向epoll对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来（红黑树的插入时间效率是lg n，其中n为树的高度）。
3. 而所有添加到epoll中的事件都会与设备（网卡）驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个回调方法。这个回调方法在内核中叫ep_poll_callback，它会将发生的事件添加到rdlist双链表中。
4. 在epoll中，对于每一个事件，都会建立一个epitem结构体，如下所示：

```
struct epitem{
    struct rb_node rbn; //红黑树节点
    struct list_head rdllink; //双向链表节点
    struct epoll_filefd ffd; //事件句柄信息
    struct eventpoll *ep; //指向其所属的eventpoll对象
    struct epoll_event event; //期待发生的事件类型
}
```

5. 当调用epoll_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可。如果rdlist不为空，则把发生

的事件复制到用户态，同时将事件数量返回给用户。

从上面的讲解可知：通过红黑树和双链表数据结构，并结合回调机制，造就了epoll的高效。

epoll的水平触发和边缘触发

一句话解释：

- 水平触发会一直通知，边缘触发只会通知一次。
- select(),poll()模型都是水平触发模式，信号驱动IO是边缘触发模式，epoll()模型即支持水平触发，也支持边缘触发，默认是水平触发。

Level_triggered(水平触发)：当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据一次性全部读写完(如读写缓冲区太小)，那么下次调用epoll_wait()时，它还会通知你在上没读写完的文件描述符上继续读写，当然如果你一直不去读写，**它会一直通知你！！！**如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率！！！

Edge_triggered(边缘触发)：当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll_wait()时，它不会通知你，也就是**它只会通知你一次**，直到该文件描述符上出现第二次可读写事件才会通知你！！！这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符！！！

阻塞IO：当你去读一个阻塞的文件描述符时，如果在该文件描述符上没有数据可读，那么它会一直阻塞(通俗一点就是一直卡在调用函数那里)，直到有数据可读。当你去写一个阻塞的文件描述符时，如果在该文件描述符上没有空间(通常是缓冲区)可写，那么它会一直阻塞，直到有空间可写。以上的读和写我们统一指在某个文件描述符进行的操作，不单单指真正的读数据，写数据，还包括接收连接accept()，发起连接connect()等操作...

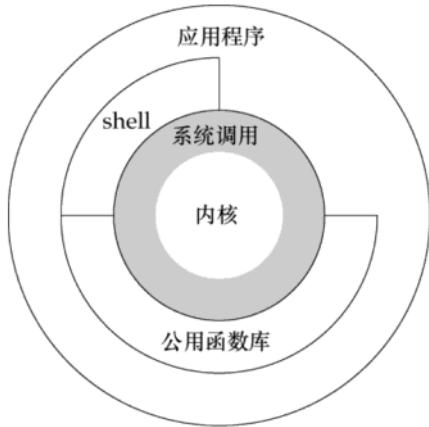
非阻塞IO：当你去读写一个非阻塞的文件描述符时，不管可不可以读写，它都会立即返回，返回成功说明读写操作完成了，返回失败会设置相应errno状态码，根据这个errno可以进一步执行其他处理。它不会像阻塞IO那样，卡在那里不动！！！

应用场景：

监听的socket文件描述符我们用sockfd，对于即要读写的文件描述符用connfd代替

1. 对于监听的sockfd，最好使用水平触发模式，边缘触发模式会导致高并发情况下，有的客户端会连接不上。如果非要使用边缘触发，网上有的方案是用while来循环accept()。
2. 对于读写的connfd，水平触发模式下，阻塞和非阻塞效果都一样，不过为了防止特殊情况，还是建议设置非阻塞。
3. 对于读写的connfd，边缘触发模式下，必须使用非阻塞IO，并要一次性全部读写完数据。

用户态和内核态



如上图所示，从宏观上来看，Linux操作系统的体系架构分为用户态和内核态（或者用户空间和内核）。**内核从本质上讲是一种软件——控制计算机的硬件资源，并提供上层应用程序运行的环境**。用户态即上层应用程序的活动空间，应用程序的执行必须依托于内核提供的资源，包括CPU资源、存储资源、I/O资源等。为了使上层应用能够访问到这些资源，内核必须为上层应用提供访问的接口：即系统调用。

系统调用是操作系统的最小功能单位，这些系统调用根据不同的应用场景可以进行扩展和裁剪，现在各种版本的Unix实现都提供了不同数量的系统调用，如Linux的不同版本提供了240-260个系统调用，FreeBSD大约提供了320个（reference: UNIX环境高级编程）。我们可以把系统调用看成是一种不能再化简的操作（类似于原子操作，但是不同概念），有人把它比作一个汉字的一个“笔画”，而一个“汉字”就代表一个上层应用，我觉得这个比喻非常贴切。因此，有时候如果要实现一个完整的汉字（给某个变量分配内存空间），就必须调用很多的系统调用。如果从实现者（程序员）的角度来看，这势必会加重程序员的负担，**良好的程序设计方法是：重视上层的业务逻辑操作，而尽可能避免底层复杂的实现细节**。库函数正是为了将程序员从复杂的细节中解脱出来而提出的一种有效方法。它实现对系统调用的封装，将简单的业务逻辑接口呈现给用户，方便用户调用，从这个角

度上看，库函数就像是组成汉字的“**偏旁**”。这样的一种组成方式极大增强了程序设计的灵活性，对于简单的操作，我们可以直接调用系统调用来自问资源，如“人”，对于复杂操作，我们借助于库函数来实现，如“仁”。显然，这样的库函数依据不同的标准也可以有不同的实现版本，如ISO C 标准库，POSIX标准库等。

Shell是一个特殊的应用程序，俗称命令行，本质上是一个命令解释器，它下通系统调用，上通各种应用，通常充当着一种“**胶水**”的角色，来连接各个小功能程序，让不同程序能够以一个清晰的接口协同工作，从而增强各个程序的功能。同时，Shell是可编程的，它可以执行符合Shell语法的文本，这样的文本称为Shell脚本，通常短短的几行Shell脚本就可以实现一个非常大的功能，原因就是这些Shell语句通常都对系统调用做了一层封装。为了方便用户和系统交互，一般，一个Shell对应一个终端，终端是一个硬件设备，呈现给用户的是一个图形化窗口。我们可以通过这个窗口输入或者输出文本。这个文本直接传递给shell进行分析解释，然后执行。

Java 语法、HashMap

2018年3月1日 16:17

目录

[面向对象和面向过程 封装、继承、多态 反射](#)

[Java中有哪些集合](#)

[final关键字](#)

[static关键字](#)

[接口与抽象类](#)

[Java重载和重写](#)

[Java数组和链表的区别](#)

[各类equals比较总结](#)

[==的用法](#)

[HashMap实现原理](#)

[HashMap扩容](#)

[HashMap均匀的存到数组里](#)

[HashMap写入](#)

[HashMap读取](#)

[JDK1.8之前和之后HashMap区别](#)

[HashMap多线程下会发生什么问题](#)

[HashTable与HashMap区别](#)

[HashTable、synchronizedMap和ConcurrentHashMap](#)

[ConcurrentHashMap底层实现 JDK1.8](#)

面向对象和面向过程

1. **面向过程：**是一种是事件为中心的编程思想。就是分析出解决问题所需的步骤，然后用函数把这写步骤实现，并按顺序调用。
2. **面向对象：**是以“对象”为中心的编程思想。
3. **举个例子**，人把冰箱打开，把大象放进去，再把冰箱关闭，这一连串的事件顺序进行，就是面向过程。而用面向对象的思想，我们并不关注事件，而是关注“人”这个对象，“人”这个对象可以做什么？可以打开冰箱，可以关闭冰箱，可以放大象。
4. **也因此有面向对象三大特性：****封装、继承、多态**。因为一切皆对象，所以一切都需要“封

装”成类。“继承”让我们设计相似的东西的时候更方便，而“多态”让我们使用类似的东西的时候可以不用去思考它们微弱的不同。我们关心的不是过程，而是接口，而接口来自对象，故名为面向对象。

封装、继承、多态

封装

- 隐藏了类的内部实现机制，可以在不影响使用的情况下改变类的内部结构，同时也保护了数据。对外界而已它的内部细节是隐藏的，暴露给外界的只是它的访问方法。

继承

- 是为了重用父类代码。两个类若存在IS-A的关系就可以使用继承。让我们设计相似的东西的时候更方便。同时继承也为实现多态做了铺垫。

多态

- 指程序中定义的**引用变量所指向的具体类型在程序运行期间才确定**，这样不用修改源代码，就可以让引用变量绑定到各种不同的类实现上，让程序可以选择多个运行状态，这就是多态性。
- 代码层面上就是指“**父类引用指向子类对象，子类对象会向上转型为父类**”，调用方法时会调用子类的实现，而不是父类的实现；但要注意，父类类型的引用可以调用父类中定义的所有属性和方法，对于只存在与子类中的方法和属性它就望尘莫及了”

子父类同名函数、同名变量调用：

```
fu t = new Zi();
System.out.println(t.num); //左边t的值
t.method(); //非静态方法，会调用右边Zi.method
```

- 成员函数（非静态）：运行看右边
- 成员函数（静态）：运行看左边
- 成员变量：运行看左边

多态如何实现的？用的反射

什么是反射？

- Java的反射机制允许我们动态的调用某个对象的方法、构造函数、获取某个对象的属性等；
- 无需在编码的时候确定调用的对象。

反射如何实现？

1、先获取这个类的class实例，比如：

```
Class<?> myClass = Class.forName("myClassName");
```

2、然后通过这个类实例获得一个类对象，比如：

```
Object myClassObject = myClass.newInstance();
```

3、然后调用Class类的对象的**getMethod**获取method对象；

4、获取method对象后调用method.**invoke**方法获取这个类的

field、method、construct等，在这一步中，JVM默认如果调用次数小于15次，会调用native方法实现反射，累积调用大于15次之后，会由java代码创建出字节码来实现反射。

Java中有哪些集合

实现了Collection接口的集合类：

1. Collection<--List<--Vector
2. Collection<--List<--ArrayList
3. Collection<--List<--LinkedList
4. Collection<--Set<--HashSet
 - a. HashSet的存储方式是把HashMap中的Key作为Set的对应存储项。
5. Collection<--Set<--HashSet<--LinkedHashSet
6. Collection<--Set<--SortedSet<--TreeSet
 - a. 自己实现Comparable接口后定义排序规则，就能自动排序，元素具有唯一性
<https://www.cnblogs.com/yzsoft/p/7127894.html>

实现了Map接口，和Collection接口没关系，但都属于集合类的一部分：

1. HashMap
2. HashTable
3. LinkedHashMap
4. TreeMap
5. SynchronizedMap
6. ConcurrentHashMap

final关键字

被final声明的对象即表示“我不想这个对象再被改变”，因此：

1. 被final声明的方法：这个方法不可以被子类重写
2. 被final声明的类：这个类不能被继承
3. 被final声明的变量：引用不能改变，常和static关键字一起使用作为常量

final关键字的好处：

1. final关键字提高了性能。JVM和Java应用都会缓存final变量。
2. final变量可以安全的在多线程环境下进行共享，而不需要额外的同步开销。
3. 使用final关键字，JVM会对方法、变量及类进行优化。

static关键字

1. static用来修饰成员变量和成员方法，也可以形成静态static代码块。
2. static对象可以在它的任何对象创建之前访问，无需引用任何对象。
3. 因此主要作用是**构造全局变量和全局方法**

Java数组和链表的区别

基于空间的考虑：

- 数组的存储空间是静态，连续分布的，初始化的过大造成空间浪费，过小又将使空间溢出机会增多。而链表的存储空间是动态分布的，只要内存空间尚有空闲，就不会产生溢出；链表中每个节点除了数据域外，还有链域（指向下一个节点），这样空间利用率就会变高。
- 数组从栈中分配空间，对于程序员方便快速，但是自由度小。链表从堆中分配空间，自由度大但是申请管理比较麻烦。
- 数组中的数据在内存中按顺序存储的，而链表是随机存储的。

基于时间的考虑：

数组查询快，插入与删除慢，单链表查询慢，插入与删除快。细说的话：数组中任意节点都可以在O(1)内直接存储访问，而链表中的节点，需从头指针顺着链表扫描才能获取到；而链表任意位置进行插入和删除，都只需要修改指针，而数组中插入删除节点，平均要移动一半的节点。

Java数据类型

数据类型(Java)

- ◆ boolean, byte, char
- ◆ short, int, long, float, double
- ◆ String, Enum, Array
- ◆ Object ...

- 1、Boolean\byte\char都是一个字节
- 2、int是4个字节，负2的31次方到正2的31次方减1
- 3、负数的表示，利用补码，比如-7。7的二进制代码取反再加1

浮点数 (+/-)1.xxx * 2^y

- ◆ 符号位 | 指数部分 | 基数部分
- ◆ 64 位 double 范围 : +/- 10^308
- ◆ 64 位 double 精度 : 10^15

浮点数比较

- ◆ `a == b` ?
- ◆ `Math.abs(a - b) < eps` ?

@4758724

- ◆ 使用 BigDecimal 算钱

浮点数是不精确的，如果算钱，要用Java提供的BigDecimal方法

Primitive type和Object

primitive type: int, long, float ...

Object: Integer, Long, Float, String...

primitive type

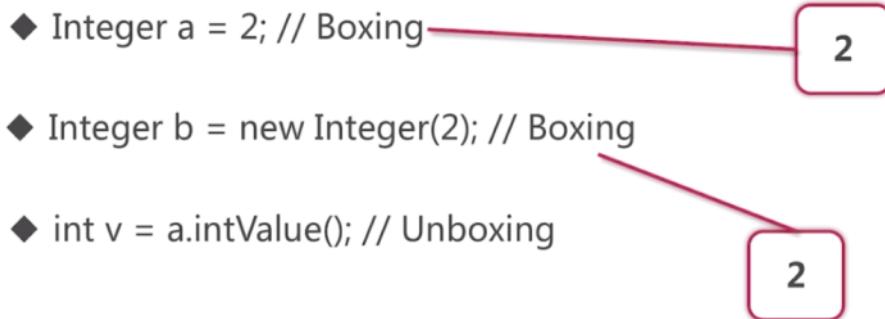
- ◆ 值类型
- ◆ 用 `a==b` 判断相等

Object

- ◆ 引用类型
 - ◆ 用 `a==b` 判断是否为同一个Object
 - ◆ 用 `a.equals(b)` , 或 `Objects.equals(a, b)` 判断是否相等
- 两个Object即使值相等，用`=`判断也是false，因为不是同一个Object
Object的值判断，用`Objects.equals(a,b)`方法

装箱和拆箱

Boxing and Unboxing



```
new Integer(2) == 2 ? true
new Integer(2) == new Integer(2) ? false
Integer.valueOf(2) == Integer.valueOf(2) ? true
Integer.valueOf(2).intValue() == 2 ? true
new Integer(2).equals(new Integer(2)) ? true
```

- 1、第一行，JVM自动帮我们把左边进行了Unboxing，取出了里面的值与2进行比较
- 2、第二行，左边新建了一个箱子装了2，右边又新建了一个箱子装了2，显然这两个箱子不是同一个箱子。
- 3、第三行，Integer.valueOf(2)，左边和右边，系统都自动给了我们一个箱子。跟进valueOf()的代码里会发现。如果是-128到127之间，系统会给我们同一个箱子，超出了这个范围，系统会给新箱子，新箱子是不同的
- 4、第四行，.intValue()是拆箱，拿出了箱子里的值
- 5、第五行，Object.equals()方法是判断值是否相等（类型也必须相等啊）

整数在内存中是怎么表示的？浮点数是怎么表示的？

整数是用补码表示，浮点数是类似科学计数法的方法表示

什么是Big-endian/Little-endian，什么是对齐？

比如整数存入内存，到底是高位放高地址还是低位放高地址，这两种顺序就是Big-endian/Little-endian。

数据在内存中是需要对齐的。比如一个char和一个int，放入一个结构体，那么这个结构体是5个字节吗？不是的，char后面补上3个字节，对齐int。

介绍一下Java的数据类型？

Java有primitive Type和Object，然后primitive type 有啥，Object有啥

什么是值传递，什么是引用传递？

在Java中，primitive type都是值传递，Object都是引用传递。Java中的Object全是引用

`String s= new String("test");`创建了多少个对象？

创建了2个对象，引号test就会创建一个String，然后 new String又创建了一个String。左边s引用指向右边new出来的对象

`equals`和`hashCode`的关系？

`hashCode`相等是`equals`的必要条件。即`hashCode`相等不一定`equals`为true，`equals`为true则`hashCode`一定相等

什么是序列化和反序列化？

对象是内存中的，当我们把对象存入硬盘或变成字节流，这就是序列化。读到或者收到字节流还原成内存中的对象就是反序列化。Java中实现`Serializable`接口后会自动的帮我们做序列化和反序列化。通常我们序列化也不是变成字节流，而是存数据库中。所以序列化都是数据库帮我们做了。网络传输也是转成JSON对象，也不是序列化成字节流传输。

二叉树如何序列化？

这类问题我们要理解为，把二叉树的结构和里面的值变成一个String，节点末尾一定要加标记

先序遍历对二叉树进行序列化

- 1、假设序列化结果为str，初始时str为空字符串。
- 2、先序遍历二叉树时如果遇到空节点，在str末尾加上“#!”。
- 3、如果遇到不为空的节点，假设节点值为3，就在str的末尾加上“3!”。



访问控制符

| | public | protected | default | private |
|------|--------|-----------|---------|---------|
| 同一个类 | ✓ | ✓ | ✓ | ✓ |
| 同一个包 | ✓ | ✓ | ✓ | ✗ |
| 子父类 | ✓ | ✓ | ✗ | ✗ |
| 不同包 | ✓ | ✗ | ✗ | ✗ |

hashCode方法

Java中的hashCode方法就是根据一定的规则将与对象相关的信息（比如对象的存储地址，对象的字段等）映射成一个数值，这个数值称作为散列值

类的特殊函数

◆ a.hashCode() == b.hashCode()



◆ a.equals(b)

1、下面如果为true，就可以推出上面为true

hashCode的作用

- 在Java集合中有两类，一类是List，一类是Set。他们之间的区别就在于List集合中的元素是有序的，且可以重复，而Set集合中元素是无序不可重复的。对于List好处理，但是对于Set而言我们要如何来保证元素不重复呢？通过迭代来equals()是否相等。数据量小还可以接受，当我们的数据量大的时候效率可想而知
- 当集合要添加新的对象时，先调用这个对象的 hashCode方法，得到对应的hashcode值，实际上在HashMap的具体实现中会用一个table保存已经存进去的对象的hashcode 值，如果table 中没有该hashcode值，它就可以直接存进去，不用再进行任何比较了；如果存在该hashcode 值，就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址
- 所以hashCode在上面扮演的角色为**快速寻域**（寻找某个对象在集合中区域位置）

在重写equals方法的同时，必须重写hashCode方法。为什么这么说呢？

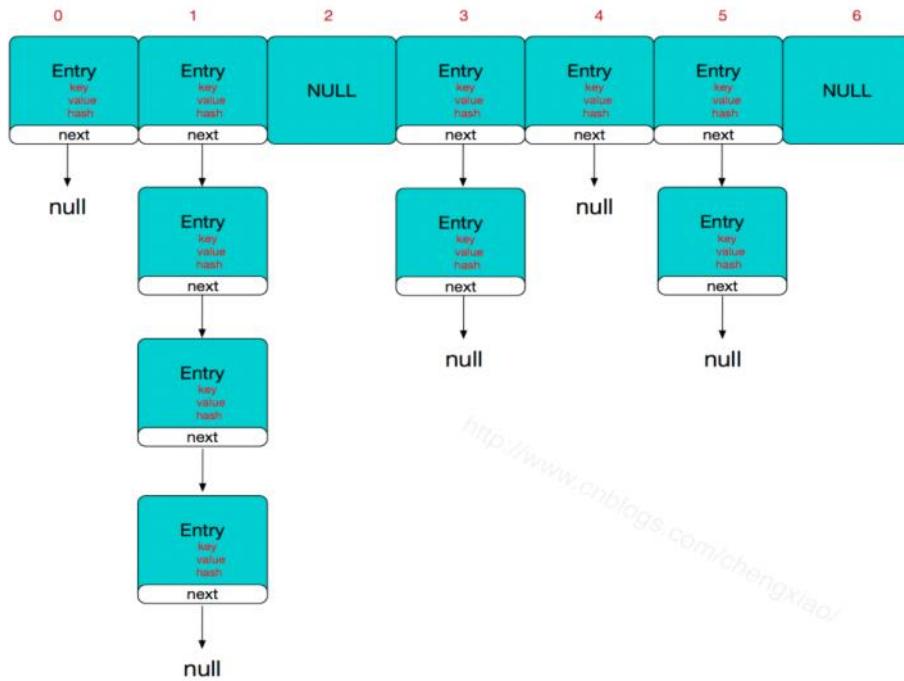
- 让equals方法和hashCode方法始终在逻辑上保持一致性
- 即让equals认为相等的两个对象，这两个对象同时调用hashCode方法，返回的值也是一样

的

HashMap实现原理

来自 <https://www.cnblogs.com/chengxiao/p/6059914.html>

所以，HashMap的整体结构如下



Entry的内部结构！！！有4个变量！！！，不止key和value

```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;
    int hash;

    /**
     * Creates new entry.
     */
    Entry(int h, K k, V v, Entry<K,V> n) {
        value = v;
        next = n;
        key = k;
        hash = h;
    }
}
```

概述：

- 1、HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，即链地址法。HashMap的主干是一个Entry数组。Entry是HashMap的基本组成单元，每一个Entry包含一个key-value键值对和一个hash值和一个指向下一个Entry的next指针。
- 2、如果定位到的数组位置不含链表（当前entry的next指向null），那么对于查找，添加等操作很快，仅需一次寻址即可
- 3、如果定位到的数组包含链表，对于添加操作，其时间复杂度依然为O(1)，操作是创建新节点，把该新节点插入到链表中的头部，该新节点的next指针指向原来的头结点，即需要简单改变

引用链即可，而对于查找操作来讲，此时就需要遍历链表，然后通过key对象的equals方法逐一比对查找。

4、所以，性能考虑，HashMap中的链表出现越少，性能才会越好。

6、当发生哈希冲突并且size大于阈值的时候，需要进行数组扩容，扩容时，需要新建一个长度为之前数组2倍的新的数组，然后将当前的Entry数组中的元素全部传输过去，扩容后的新数组长度为之前的2倍，所以扩容相对来说是个耗资源的操作

7、如果key为null，就会插入到table[0]的位置也就是数组头。如果key=null，则hash值直接赋0

8、存key时，如果链中存在该key，则用传入的value覆盖掉旧的value，同时把旧的value返回：这就是为什么HashMap不能有两个相同的key的原因。

9、计算hash值之后，如何通过hash值均匀的存到数组里！！！！！！！！！？当然是取模，但取模消耗大，因此HashMap用的&运算符（按位与操作）来实现的：`hashCode & (length-1)`。

10、这里就隐含了为什么数组长度length一定要是2的n次方。当length不是2的n次方的时候，length-1的二进制最后一位肯定是0，在&操作时，一个为0，无论另一个为1还是0，最终&操作结果都是0，这就造成了结果的二进制的最后一位都是0，这就导致了所有数据都存储在2的倍数位上，所以说，所以说当length = 2^n 时，不同的hash值发生碰撞的概率比较小，这样就会使得数据在table数组中分布较均匀，查询速度也较快。

我们重新来理一下存储的步骤：

1. 传入key和value，判断key是否为null，如果为null，则调用`putForNullKey`，以null作为key存储到哈希表中；

2. 然后计算key的hash值，根据hash值搜索在哈希表table中的索引位置，若当前索引位置不为null，则对该位置的Entry链表进行遍历，如果链中存在该key，则用传入的value覆盖掉旧的value，同时把旧的value返回，结束；

3. 否则调用`addEntry`，用key-value创建一个新的节点，并把该节点插入到该索引对应的链表的头部

读的步骤：

读取的步骤比较简单，调用`hash (key)`求得key的hash值，然后调用`indexFor (hash)`求得hash值对应的table的索引位置，然后遍历索引位置的链表，如果存在key，则把key对应的Entry返回，否则返回null。

JDK1.8之前和之后HashMap区别

- 在JDK1.8以前版本中，HashMap的实现是数组+链表，它的缺点是即使哈希函数选择的再好，也很难达到元素百分百均匀分布，而且当HashMap中有大量元素都存到同一个桶中时，这个桶会有一个很长的链表，此时遍历的时间复杂度就是O(n)，当然这是最糟糕的情况。
- 在JDK1.8及以后的版本中引入了红黑树结构，HashMap的实现就变成了数组+链表或数组+红黑树。添加元素时，若桶中链表个数超过8，链表会转换成红黑树；删除元素、扩容

时，若桶中结构为红黑树并且树中元素个数较少时会进行修剪或直接还原成链表结构，以提高后续操作性能；遍历、查找时，由于使用红黑树结构，**红黑树遍历的时间复杂度为 $O(\log n)$** ，所以性能得到提升。

HashMap多线程下会发生什么问题

多线程并发下，在HashMap扩容的时候可能会形成环形链表。至于具体过程？不用深究了吧，连Sun公司都不觉得这是个问题，要并发就用ConcurrentHashMap呀！

HashTable与HashMap区别

- 1、Hashtable 中的方法是同步的，而HashMap中的方法在缺省情况下是非同步的。
- 2、Hashtable中，key和value都不允许出现null值。在HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。
- 3、并发性不如ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁。**Hashtable不建议在新代码中使用，不需要线程安全的场合可以用HashMap替换，需要线程安全的场合可以用ConcurrentHashMap替换。**

HashSet 和 HashMap

- 1、它们底层的 Hash 存储机制完全一样，甚至 HashSet 本身就采用 HashMap 来实现的

接口与抽象类

一句话解释：

抽象类就是比普通类多了一些抽象方法而已，其他部分和普通类完全一样；而接口是特殊的抽象类。

作用上看：

- 1、接口与抽象类结构有点像，但功能完全不同
- 2、接口是强调合约、约束关系，即你要与我合作，必须实现我的功能；抽象类没这个功能

语法上看：

1. 都不能被实例化
2. 接口是特殊的抽象类
3. 接口不能有实现，Java8中可以有添加default关键字的默认实现和静态方法实现。
4. 接口中的成员变量必须是public static final修饰（编译器默认会添加上），因此是常量
5. 一个类可以实现多个接口但只能继承一个抽象类

什么是接口？

- 从表现来说：定义了很多函数，但是这些函数都没有实现，这就是接口。从作用来说：起到一个合约规范的作用。我要告诉你和我打交道的东西有什么约束
- 接口中的方法只能用public和abstract修饰或者不修饰
- 接口中的属性默认都是public static final，因此是常量

final的作用？

表示我这个东西不希望被修改了

ArrayList<String>是ArrayList<Object>吗？
ArrayList<String>是List<String>吗？

- 第一个“否”
- 第二个“是”

实现Iterable接口

- 实现Iterable接口，可以让我们一个一个拿出元素
- 如下，因为LinkedList<>实现了Iterable接口，所以我们才能用for each语法遍历出里面一个一个的对象

```
LinkedList<Employee> employees = new LinkedList<>();
employees.add(employee1);
employees.add(employee2);
employees.add(employee3);

System.out.println("Print using for each");
for (Employee employee : employees) {
    System.out.println(employee);
}
```

```
Print using for each
Employee [name=John, salary=10000]
Employee [name=Mary, salary=20000]
Employee [name=John, salary=10000]
```

派生类中修改封装可见性

◆ private → public? public → private?

◆ 不建议使用

- 子类（派生类）只能增加或修改基类里的内容，不能减少

- 因此，子类的private（父类对应的方法是private）改为public可以，但子类的public（父类对应的方法是public）改为private不行
- 如果有@Override注释，则子类权限必须与基类一致

final 关键字

- ◆ 类申明 → 类不可以被继承
- ◆ 函数申明 → 函数不可以在派生类中重写
- ◆ 变量申明 → 变量不可以指向其它对象

final 关键字

- ◆ static final 变量 → 用于定义常量，名称一般大写

HashTable、synchronizedMap和ConcurrentHashMap

HashTable、synchronizedMap效率低下

- 现在基本不用HashTable。HashTable容器使用synchronized来保证线程安全，但是锁的是整个hash表，当一个线程使用put方法时，另一个线程不但不可以使用put方法，连get方法都不可以。
- synchronizedMap比HashTable强一分钱，synchronizedMap提供一个不同步的基类和一个同步的包装。允许需要同步的用户可以拥有同步，而不需要同步的用户则不必为同步付出代价，get方法与HashTable一样锁住整个hash表，区别是get()和put()之类的简单操作可以在不需要额外同步的情况下安全地完成。但多个操作组成的操作序列却可能导致数据争用，总之就是不好用。

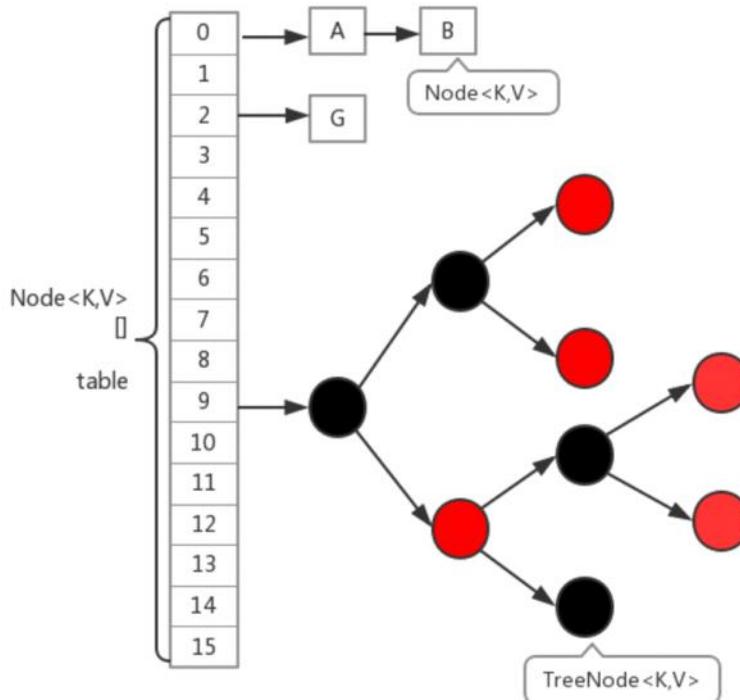
ConcurrentHashMap效率高，因为用了分段锁（JDK8之前），16个

- HashTable容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问HashTable的线程都必须竞争同一把锁
- 那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率
- 这就是 ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，默认分成16个段，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。
- 上面说到的16个线程指的是写线程，而读操作大部分时候都不需要用到锁。只有在size等操作时才需要锁住整个hash表。

ConcurrentHashMap JDK1.8

基本结构：Node<K,V>数组+链表（红黑树）的结构。

- 而对于锁的粒度，调整为对每个数组元素加锁（Node），即没有分段锁了，而是Node锁，粒度更小。
- 使用CAS操作来确保Node的一些操作的原子性，这种方式代替了锁。
- ConcurrentHashMap在线程安全的基础上提供了更好的写并发能力，但同时降低了读一致性。ConcurrentHashMap的get操作上面并没有加锁。所以在多线程操作的过程中，并不能完全的保证一致性。这里和1.7当中类似，是弱一致性的体现。
- 代码中使用synchronized而不是ReentrantLock，说明JDK8中synchronized有了足够的优化。
- 然后是定位节点的hash算法被简化了，这样带来的弊端是Hash冲突会加剧。
- 因此在链表节点数量大于8时，会将链表转化为红黑树进行存储。这样一来，查询的时间复杂度就会由原先的O(n)变为O(logN)。



- ConcurrentHashMap的设计与实现非常精巧，大量的利用了volatile, final, CAS等lock-free技术来减少锁竞争对于性能的影响。
- HashEntry中的value以及next都被volatile修饰，这样在多线程读写过程中能够保持它们的可见性。

```
static final class HashEntry<K,V> {
    final int hash;
    final K key;
    volatile V value;
    volatile HashEntry<K,V> next;
```

网络

2018年2月28日 23:19

目录

[DHCP 五层协议 OSI七层模型](#)
[UDP/TCP区别 TCP如何保证可靠](#)
[三次握手 四次挥手 服务器大量CLOSE_WAIT 服务器大量TIME_WAIT](#)
[遇到的一个TIME_WAIT\CLOSE_WAIT实例](#)

[TCP计时器管理](#)
[TCP拥塞控制](#)
[TCP流量控制](#)
[滑动窗口协议](#)

[TCP\UDP常用端口](#)

[IP协议 IPv4 IPv6](#)

[一次完整的HTTP请求](#)
[HTTPS](#)

[RPC](#)

1到1000，其中一个数出现了2次，乱序

1001个自然数

利用异或的特性，与另外1到1000的数组依次异或，最后剩下的就是这个出现了2次的数

DHCP动态主机配置协议

所有的Internet协议都做了这样一个假设，即主机配置了一些基本信息，比如IP地址。主机如何获得此信息？手动设置是可以的，但太麻烦，因此有了DHCP，即自动给主机配置IP地址、网络掩码等信息。

DHCP如何实现分配IP？

1. 首先每个网络必须有一个DHCP服务器
2. 计算机启动时，在自己的网络上广播一个报文，请求IP地址
3. 这个请求就是DHCP DISCOVER包，这个包必须到达DHCP服务器

4. DHCP收到请求了就给主机分配一个IP地址，并通过DHCP OFFER包返回给主机
5. 为了在主机没有IP地址的情况下完成此项工作，服务器用主机的以太网地址来标识这台主机

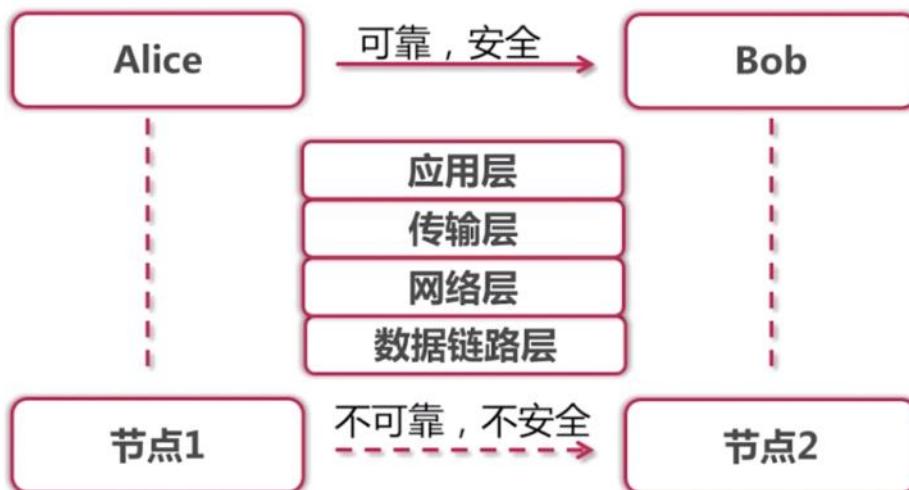
防止IP地址过期

为了避免主机离开网络，并没有把IP地址返回给DHCP服务器，因此DHCP服务器在分配IP地址的时候都会指定一个有效期。在有效期满之前，主机必须请求续订。

DHCP数据包格式及其原理

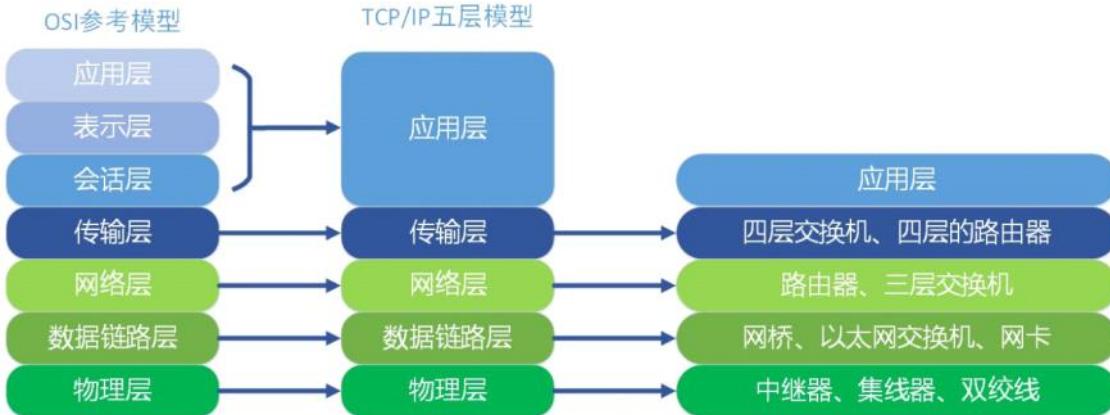
DHCP可为主机配置除了IP地址以外的其他各种参数，比如网络掩码、默认网关的IP地址，DNS服务器和时间服务器的IP地址。

五层因特网协议栈



- 0、两个节点物理线路、传输介质是**物理层**。
- 1、在**数据链路层**中传递数据包，并进行校验，**DHCP协议**是数据链路层的。
- 2、节点和节点之间可以用数据链路层，但传递给其他学校或者国家，这就不是两个节点之间的传输了，就需要**网络层**。网络层会有路由，包发给路由器，路由器再发给路由器，转转转，通过**IP协议**，每个节点都有IP地址，最终传到目标节点。网络层还包括ICMP和IGMP协议。**ICMP**是IP协议的附属协议。IP层用它来与其他主机或路由器交换错误报文和其他重要信息。**IGMP**是Internet组管理协议。它用来把一个UDP数据报多播到多个主机。网络层还有**ARP协议**（地址解析协议），作用是将IP地址映射到数据链路层的以太网地址（mac地址），因为网卡只认以太网地址（mac地址）。
- 3、虽然在数据链路层进行了校验，但并不可靠，需要出错重传，就有了**传输层**。在传输层有**TCP/UDP协议**。TCP协议是基于连接的，UDP无连接。
- 4、数据是为哪个应用服务呢？是**HTTP**还是**FTP**还是**SMTP**、**POP3**、**DNS**、**AMQP** **AMQP协议** 因此需要有**应用层**。
- 5、**FTP**、**Telnet**、**SMTP**、**HTTP**、**POP3**是基于TCP的；**DNS**、**SNMP**、**RIP**是基于UDP的。**socket**只是对TCP/IP协议栈操作的抽象，即socket是TCP/IP协议的API

OSI七层模型



UDP/TCP区别

1. **TCP面向连接** (如打电话要先拨号建立连接); UDP是无连接的, 即发送数据之前不需要建立连接。
2. **TCP提供可靠的服务**。也就是说, 通过TCP连接传送的数据, 无差错, 不丢失, 不重复, 且按序到达; UDP尽最大努力交付, 即不保证可靠交付。
3. **TCP实时性比UDP差**, UDP更适用于对高速传输和实时性有较高的通信 (比如LOL这种实时对战网络游戏) 或广播通信。
4. 每一条**TCP连接只能是点到点的**, 即TCP不支持组播或者广播传输模式。UDP支持一对
一, 一对多, 多对一和多对多的交互通信。
5. **TCP对系统资源要求较多**, UDP对系统资源要求较少。

为什么UDP在越来越多的场景下取代了TCP?

- UDP以其简单、传输快的优势, 在越来越多场景下取代了TCP, **不能容忍延迟**比如**LOL、Dota用UDP, 可以容忍延迟的游戏如RPG还是用TCP**。
- **网速的提升**给UDP的稳定性提供可靠网络保障, 丢包率很低, 如果使用应用层重传, 能够确保传输的可靠性。
- 网络游戏如果采用**TCP**, **一旦发生丢包**, TCP会将后续的包缓存起来, 等前面的包重传并接收到后再继续发送, **延时会越来越大**, 基于UDP对实时性要求较为严格的情况下, 采用自定义重传机制, 能够把丢包产生的延迟降到最低, 尽量减少网络问题对游戏性造成影响。

设计UDP丢包检查来保证可靠性

1. 可以采用一个近似TCP的ack机制, 可以给每个数据包都添加一个**sequence ID**, 然后服务端就依次发送数据包, 客户端收到数据包后就可以根据**sequence ID**来判断是否有丢包了。
2. **接下来是重点**, 客户端需要发该**sequenceID**的ack给服务端, 服务端才会知道这个包是否已经送达。但这是一笔不小的开销, 而且, ack本身也有可能丢包。
3. 可以这样, **客户端发送一个sequence ID的ack时, 附加一个32bit的位序列**, 表示当前**sequence ID**之前的32个连续顺位的数据包是否已经送达, 其实就是冗余的发送连续32

个包的送达状态，如果bit为0说明这个包还没到，如果为1，说明已经收到了。这样一来，除非连续丢包30多次，ack是一定会送到的，这种几率已经非常小了。

4. 相应的，在服务端设置一个超时机制，这个时间差不多比连续发30个ack的时间长一点，如果发送一个包后开始计时，达到超时还没有收到ack，这个包就丢失了。
5. **但即使丢了也不一定需要重发！** 是否需要重发，如何重发可以和游戏的逻辑结合起来，没有必要实现类似TCP那样的完全可靠的机制，毕竟战斗中的同步速率很高，丢一个一般也没啥事情。

TCP如何保证可靠

一句话解释：

Tcp通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。

详细解释：

1. 数据被分割成TCP认为最合适的数据块
2. 包是按序收到的，即发送顺序和接受顺序一致。
3. TCP发送一个段之后会启动一个定时器，等待目标端的确认信息，如果没有收到确认信息，会重发
4. 防止序号回绕机制。即在连接期间，时间戳被用来辅助扩展32位序号，而且采用了伪随机的初始序号，防止重复数据包。
5. TCP保证收到的包不出错，即保持首部和数据的校验和，如果收到的校验和出错，TCP将丢弃这个数据报也不会确认这个数据报
6. TCP会对收到的数据进行重新排序，保证数据以正确的顺序交给应用层
7. TCP还有流量控制和拥塞控制
8. 滑动窗口协议也是来保证可靠的

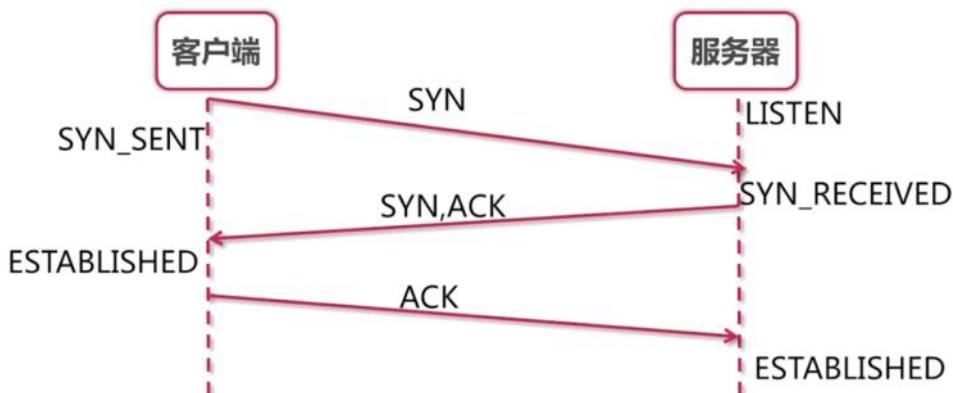
三次握手

TCP使用了三次握手法来建立连接。

为什么是三次握手、四次挥手？而不是五次六次

首先讲这个思想1、网络是不可靠的，任何包都有可能丢。2、遇见问题，解决问题，不断迭代。3、因此网络协议能用就好。然后回答具体流程

建立连接 - 三次握手



- 1、某一端，比如服务器必须先依次执行LISTEN和ACCEPT原语，然后被动的等待入境连接请求。
- 2、另一端，比如说客户，执行CONNECT原语，同时说明它希望连接的IP地址、端口等参数。CONNECT原语发送一个SYN标志位置为ON和ACK标志位置为OFF的TCP段，进入SYN_SENT状态，然后等待服务器响应。
- 3、当这个TCP段到达接收方时，接收方的TCP实体检查是否有一个进程已经在目标端口上执行了LISTEN。如果没有，则它发送一个设置了RST的应答报文，拒绝客户的连接请求。
- 4、如果是LISTEN，那么TCP实体将入境的TCP段交给该进程处理，进程可以接受或者拒绝这个连接请求。如果接受，则进入SYN RECEIVED状态，并发送回一个确认段。
- 5、正常情况下，发送的TCP段顺序如下

- (1)第一次握手，客户发送SYN=x到服务器
- (2)第二次握手，服务器确认后，进入SYN RECEIVED状态并发送SYN=y,ACK=x+1到客户
- (3)第三次握手，客户确认后，进入ESTABLISHED状态，发送SYN=x+1,ACK=y+1到服务器，此包发送完毕，服务器收到后进入ESTABLISHED状态。客户端和服务器TCP连接成功，完成三次握手，开始传送数据。如果这里服务器没收到，客户进入ESTABLISHED状态后也会自认为连接完成，开始发送数据，不过发数据一直没有响应，最后会超时断开连接。
- (4)客户不知道服务器是否收到了它的SYN=x+1，如此下去是个死循环，因此够用就好。

这里可以看视频7分钟

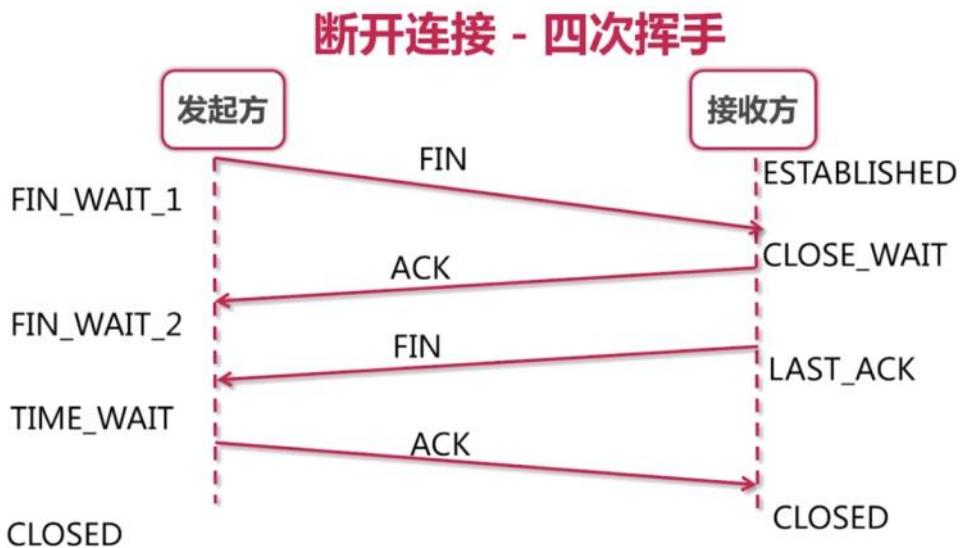
<https://coding.imooc.com/lesson/132.html#mid=8907>

TCP连接如何释放？

0、简称四次挥手

- 1、虽然TCP连接是全双工的，这里我们把TCP看成单工的。
- 2、为了释放一个连接，任何一方（比如A）都可以发送一个设置了FIN标志位的TCP段，这表示它已经没有数据要发送了。当FIN被另一方（比如B）确认后，这个方向（A到B）的连接就被关闭。
- 3、另一方向上或许还继续传着数据流。当两个方向都关闭后，连接才算彻底关闭。
- 4、为了避免“两军对垒”问题，需要使用计时器，如果两倍最大数据包生存期内，针对FIN的响应没有出现（A发了FIN，A没有收到B的FIN响应），那么FIN发送端（A）会直接释放连接。另一方最终会发现没人监听，超时后也会释放连接。

5、四次握手具体如下：



6、注意这个TIME_WAIT，因为最后接收方不会发送ACK回来了，因此发起方会进入TIME_WAIT状态等待一段时间，如果没啥变化，再进入CLOSED关闭状态。
这个好形象

结束聊天 - 男生发起



断开为什么是四次握手？

因为A到B断开了，B也许还有数据没有发送完，B到A这方向不能立刻断，因此设计上和三次握手有区别。再讲四次握手流程

服务器出现异常？

如果服务器出现异常，百分之八九十都是下面两种情况：

1. 服务器保持了大量TIME_WAIT状态
2. 服务器保持了大量CLOSE_WAIT状态

因为linux分配给一个用户的文件句柄是有限的，而TIME_WAIT和CLOSE_WAIT两种状态如果一直被保持，那么意味着对应数目的通道就一直被占着，而且是“占着茅坑不使劲”，一旦达

到句柄数上限，新的请求就无法被处理了，接着就是大量Too Many Open Files异常，tomcat崩溃。。。

服务器大量CLOSE_WAIT状态的原因？？

- CLOSE_WAIT产生的原因在于：TCP Server 已经ACK了过来的FIN数据包，但是上层应用程序迟迟没有发命令关闭Server到client 端的连接。所以TCP一直在那等啊等.....
- 所以说如果发现自己的服务器保持了大量的CLOSE_WAIT，问题的根源十有八九是自己的server端程序代码的问题。

服务器大量TIME_WAIT状态原因？？

- 服务器处理大量连接并主动关闭连接时，将导致服务器端存在大量的处于TIME_WAIT 状态的socket。
- 因为主动关闭方会进入TIME_WAIT的状态，然后在保持这个状态2MSL (max segment lifetime) 时间（1到4分钟）之后，彻底关闭回收资源（被占用的是一个五元组：（协议，本地IP，本地端口，远程IP，远程端口）。对于 Web 服务器，协议是 TCP，本地 IP 通常也只有一个，本地端口默认的 80 或者 443。只剩下远程 IP 和远程端口可以变了。如果远程 IP 是相同的话，就只有远程端口可以变了。这个只有几万个）。
- 所以如果大量关闭，资源还没来得及回收，会导致大量TIME_WAIT。
- 解决方案是修改linux内核，允许将TIME-WAIT sockets重新用于新的TCP连接，并开启TCP连接中TIME-WAIT sockets的快速回收，这些默认都是关闭的。

遇到的一个TIME_WAIT\CLOSE_WAIT实例

事件经过：

7月份我们上线了一个改动，这个改动会产生慢查询，增加数据库压力。大量的慢查询导致后台处理系统不能及时处理峰值请求，请求数据严重重发，如此恶性循环最终导致系统崩溃，服务中有大量TIME_WAIT状态和大量CLOSE_WAIT状态，当时线程池的数量为20，使用线程数为20，等待线程数为1000。

原因分析：

1. 服务端阻塞在数据库访问上，因此服务端无法处理请求，客户端超时发起关闭TCP，服务端这边因为处理请求的线程还阻塞在数据库，没有发起服务端到客户端的关闭，因此服务端这边积累了大量CLOSE_WAIT状态。
2. 服务发起方（客户端服务器），因为数据库阻塞了，所以服务发起方的请求大量超时，大量关闭，请求资源还没来得及回收（1到4分钟才能回收完），因此服务发起方这边就会积累大量TIME_WAIT。

为什么会慢查询呢？

比如数据库配置的连接数只有10，实际要连接的数目>配置的连接数时，就会导致实例不能连接数据库，导致数据库慢查询。在评估数据库连接数的时候，需要把程序并发数考虑进去。

改进措施

1. 请求优化,如果存在请求积压时主动放弃请求
2. 检查SQL, 对潜在问题进行优化
3. 服务增加只访问缓存的紧急预案, 保证数据库出现问题时能够灵活切换, 减少服务损失
同时减少数据库请求防止雪崩发生

TCP协议要达成什么目的?

要实现可靠的传输, 什么叫可靠的传输呢? 1、包是按序收到的, 即发送顺序和接受顺序一致。2、保证收到的包不出错。3、流量控制和拥塞控制

TCP计时器管理

- 1、TCP有三个计时器, 分别是持续计时器、保活计时器和重传计时器, 其中最重要的是重传计时器。
- 2、**重传计时器**: 当TCP实体发出一个段时, 它同时启动一个重传计时器, 如果在该计时器超时前被确认, 则计时器被停止。如果在确认到来之前计时器超时, 则段被重传。
- 3、那么问题来了, **超时间隔应该设置为多长?** 解决方案是使用一个动态算法, 它根据网络性能的连续测量情况, 不断地调整超时间隔。

TCP如何进行拥塞控制?

- 1、当路由器上的队列增长到很大时, 网络层检测到拥塞, 并试图通过丢弃数据包来管理拥塞。
- 2、传输层 (TCP所在) 收到从网络层反馈来的拥塞信息, 就会减慢它发送到网络的流量速率。
- 3、TCP会并发的维持一个拥塞窗口和一个流量控制窗口。拥塞窗口大小限制发送端可以发送的字节数, 流量控制窗口指出了接收端可以缓冲的字节数。有效发送是这两个窗口的较小者。

TCP如何进行流量控制?

滑动窗口协议。滑动窗口协议不仅用于流量控制, 还参与了一部分拥塞控制

网络传输

为什么网络传输不可靠?

丢包、重复包、包出错、包乱序、安全

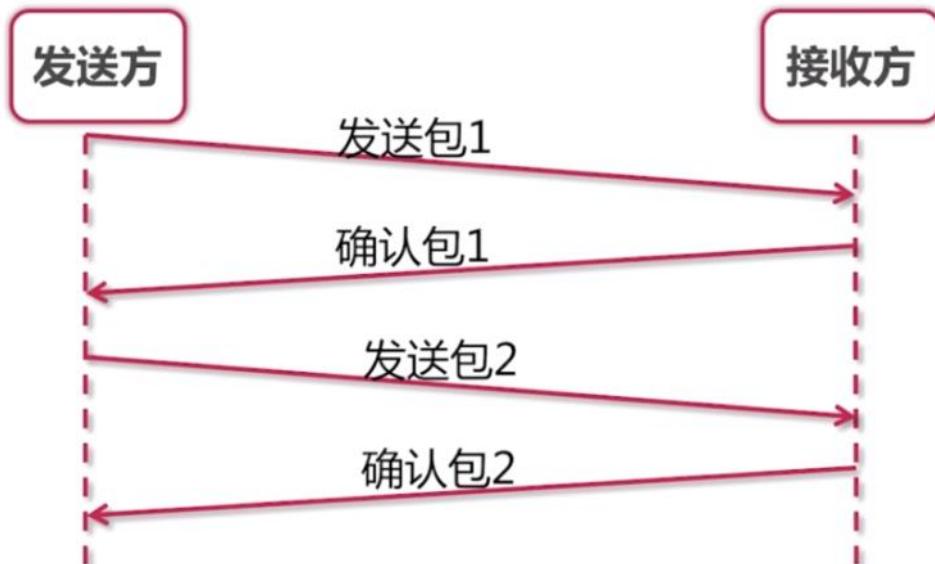
TCP协议如何解决网络传输不可靠问题?

滑动窗口协议。是在TCP协议中使用的非常重要的部分, 用来维持发送方和接收方缓冲区。这个缓冲区就是为了解决网络传输不可靠的问题。发送方和接收方各自维护自己的缓冲区, 商定如何accept传递包和重传机制。

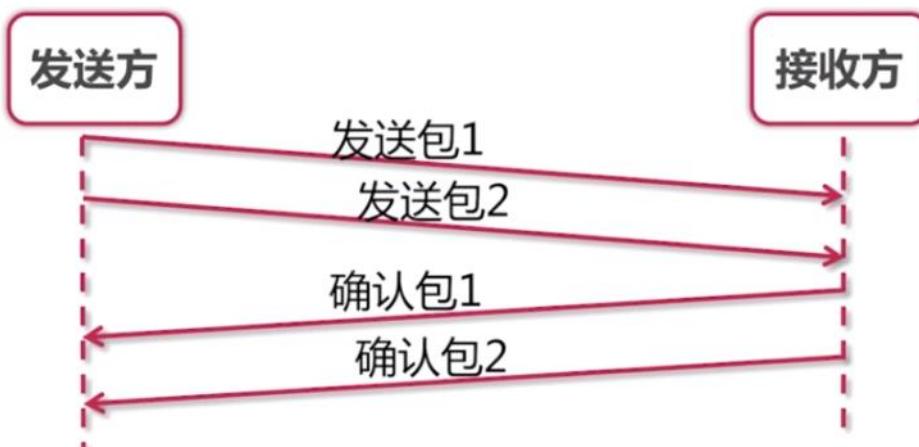
0、窗口大小最大65535, 不仅用于流量控制, 还参与了一部分拥塞控制, 在传输控制过程中

窗口大小可以调整，大小为0是合法的。

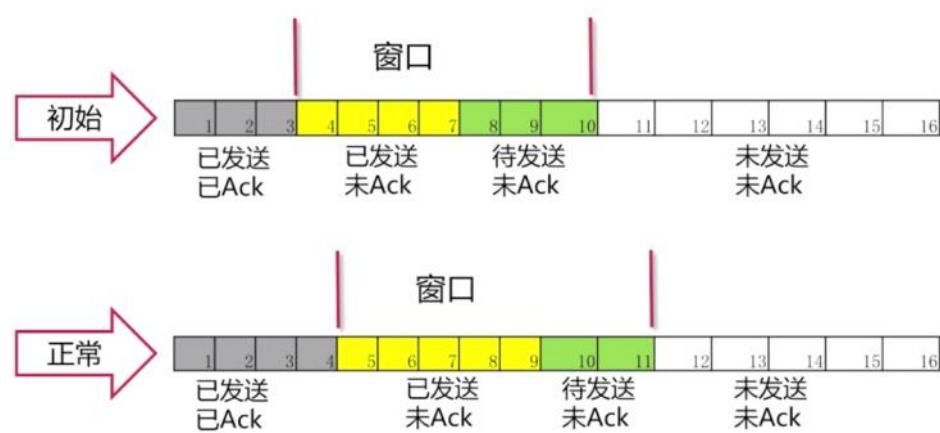
1、最原始的无滑动窗口协议时，如下。发送一个包，确认一个包。缺点也很明显，很慢



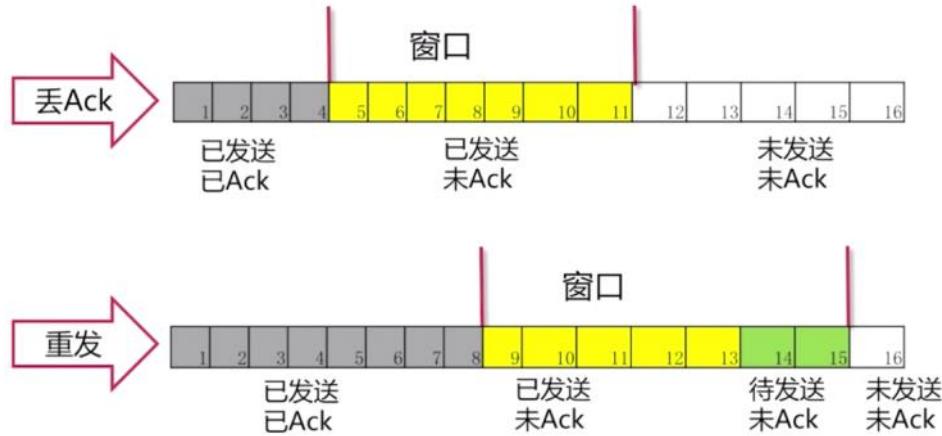
2、改进方案，如下。两个包一起发，一起确认。缺点也很明显，收到了包1确认，没收到包2确认，那我一直等下去？发不发后面的包？



3、滑动窗口协议，每确认一个包，就往后移动



如果出现丢包，也就是未收到确认信息，那么就有一个超时重传机制。注意接收方发送 Ack (确认字符) 必须按照顺序来，即使678号包收到了，但5号包一直没收到，就不会发送678号包的Ack信息。只有收到了5号包，才会把5678的Ack一起发送出去。



TCP\UDP常用端口

端口号的范围是从1~65535。其中1~1024是被RFC 3232规定好了的，被称作“众所周知的端口”(Well Known Ports)；从1025~65535的端口被称为动态端口(Dynamic Ports)，可用来建立与其它主机的会话，也可由用户自定义用途。

TCP 21端口: FTP 文件传输服务

TCP 23端口: TELNET 终端仿真服务，远程登录

TCP 25端口: SMTP 简单邮件传输服务，发送邮件

UDP 53端口: DNS 域名解析服务

TCP 80端口: HTTP 超文本传输服务

TCP 110端口: POP3 “邮局协议版本3”使用的端口，接收邮件

TCP 443端口: HTTPS 加密的超文本传输服务

网络抓包演示

<https://coding.imooc.com/lesson/132.html#mid=6558>

网络例题

阿里巴巴有两个相距 1500km 的机房 A 和 B。现有 100GB 数据需要通过一条 FTP 连接在 100s 内从 A 传输到 B。已知 FTP 连接建立在 TCP 协议之上，而 TCP 协议通过 ACK 来确认每个数据包是否正确传送。网络信号传输速率为 $2 \times 100000 \text{ km/s}$ ，假设机房间带宽足够高，那么 A 节点的发送缓冲区最小可以设置为_____。

- A、6MB B、12MB C、18MB D、24MB

解答：

- ◆ 一个来回的时间： $1500 / (2 \times 10^5) * 2 = 0.015\text{s}$
- ◆ 来回的次数至多： $100 / 0.015 = 6666.667$ 次
- ◆ 每次来回传输至少： $100\text{G} / 6666.667 = 100\text{ 000M} / 6666.67 = 15\text{M}$

因此选C:18MB

IP协议

IPv4协议是32位地址，IPv6是128位地址。

IPv4地址不够用怎么办？

1. NAT~~网络地址转换~~，比如一个IP地址作为一个大学的地址，大学内各个计算机的地址用内网私有地址。
2. 动态分配，为一台连在网上并使用的计算机动态分配一个IP地址，当该主机不活跃的时候收回分配给他的IP地址
3. 最终解决方案是IPv6

IPv6相比IPv4的提升

1. 有更长的地址，一个从32位提升到了128位
2. 对头进行了简化，从13个字段下降到7个字段，因此路由器可以更快的处理数据包
3. 更好的支持选项，即以前必须的字段现在变成了可选
4. 安全性的改进，后来这些特征也被引入到了IPv4中，因此现在安全性差异没那么大

HTTP、HTTPS、Cookie、Session

2018年3月11日 17:16

[一次完整的HTTP请求](#)

[Cookie和Session](#)

[RPC](#)

HTTP是基于TCP的，在TCP建立连接之后，才用HTTP来传送数据。

HTTP状态码

- 2开头（请求成功）表示成功处理了请求的状态代码。
- 3开头（请求被重定向）表示要完成请求，需要进一步操作。通常，这些状态代码用来重定向。
- 4开头（请求错误）这些状态代码表示请求可能出错，妨碍了服务器的处理。
- 5开头（服务器错误）这些状态代码表示服务器在尝试处理请求时发生内部错误。这些错误可能是服务器本身的问题，而不是请求出错。

[301 moved permanently](#)、[302 found](#)、[303 see other](#)

301/302/303都表示重定向，所以放在一起讲解。

- 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。
- 新url指的是，第一次请求返回的location。

举例说明

- 1、浏览器访问<http://write.blog.csdn.net>, csdn中“我的博客”
- 2、服务器，返回状态码302（url临时改变）和location
- 3、浏览器，请求location指定的地址，完成请求。也就是说，浏览器一共请求了2次！

[304 not modified](#)

客户端发送附带条件的请求时（if-matched, if-modified-since, if-none-match, if-range, if-unmodified-since任一个）服务器端允许请求访问资源，但因发生请求未满足条件的情况下，直接返回304Modified（[服务器端资源未改变，可直接使用客户端未过期的缓存](#)）。304状态码返回时，不包含任何响应的主体部分。304虽然被划分在3xx类别中，但是和重定向没有关系。

举例说明

请求[hao123主页](http://www.hao123.com)，js、css、图片状态码很多是304

| | | |
|-----|-----------|--|
| 200 | (成功) | 服务器已成功处理了请求。通常，这表示服务器提供了请求的网页。 |
| 301 | (永久移动) | 请求的网页已永久移动到新位置。服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。 |
| 302 | (临时移动) | 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。 |
| 404 | (未找到) | 服务器找不到请求的网页。 |
| 415 | (请求错误) | 后台程序不支持提交的content-type |
| 405 | (请求错误) | 请求的方式 (get、post、delete) 方法与后台规定的方式不符合 |
| 400 | (请求错误) | 请求的报文中存在语法错误，比如url含有非法字符。 提交json时，如果json格式有问题，接收端接收json，也会出现400 bad request |
| 401 | | 未授权，比如访问SpringSecurity限制了权限的资源 |
| 500 | (服务器内部错误) | 服务器遇到错误，无法完成请求。 |

一次完整的HTTP请求

1. **域名解析**：在系统的hosts文件里查找域名对应的IP地址，没有的话向本地DNS发送一个请求报文，本地服务器必须返回完整结果，称为**递归查询**。如果本地DNS服务器没有，则启动一次远程查询，请求根域名服务器，每次返回一部分域名，直到找到域名对应的IP地址并返回一个响应报文，这个机制称为**迭代查询**。
 - a. 查询报文和响应报文都作为UDP数据包发送，即**DNS是基于UDP的**。
 - b. DNS所有的查询答案，包括所有的部分答案都会被缓存，缓存也是有过期时间的。

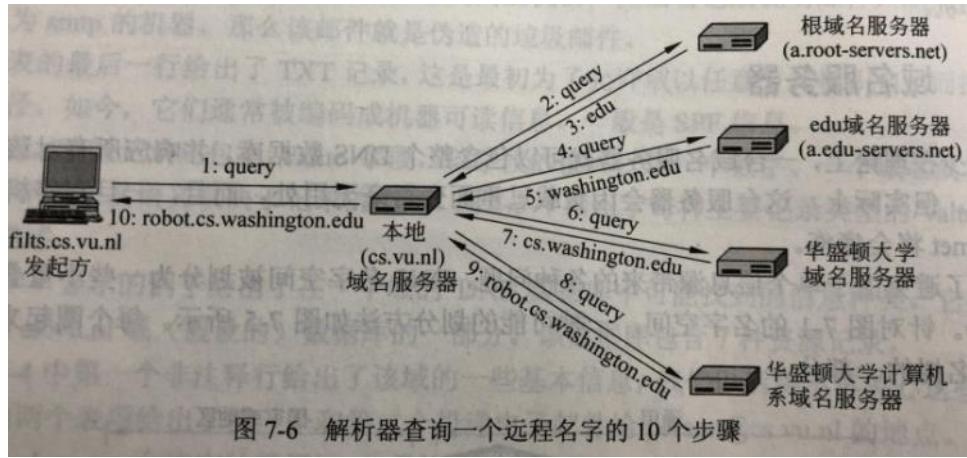


图 7-6 解析器查询一个远程名字的 10 个步骤

2. 发起TCP的3次握手建立连接 [三次握手](#)
3. 建立TCP连接后发起HTTP请求
4. 服务器响应HTTP请求，业务逻辑处理，返回HTTP响应给浏览器
5. 浏览器得到html代码，会解析html代码，并请求html代码中的资源（如js、css、图片等，并对页面进行渲染最终呈现给用户

HTTP请求格式

主要有四部分组成，分别是：**请求行、请求头、空行、消息体**，每部分内容占一行

| | |
|--|------------------------|
| GET /index.html HTTP/1.1 | Request Line |
| Date: Thu, 20 May 2004 21:12:55 GMT | General Headers |
| Connection: close | |
| Host: www.myfavoriteamazingsite.com | Request Headers |
| From: joebloe@somewebsitewhere.com | |
| Accept: text/html, text/plain | |
| User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1) | Entity Headers |
| | HTTP Request |
| | Message Body |

- **请求行**：由三部分组成，GET/POST请求方法、请求资源URL、HTTP版本号
- **请求头**：和缓存相关的头（Cache-Control, If-Modified-Since），**客户端身份信息**（User-Agent）等等
- **消息体**：客户端发给服务端的请求数据，这部分数据并不是每个请求必须的

HTTP响应格式

包括：**状态行、响应头、空行、消息体**。每部分内容占一行。

| | |
|--|------------------|
| HTTP/1.1 200 OK | Status Line |
| Date: Thu, 20 May 2004 21:12:58 GMT | General Headers |
| Connection: close | |
| Server: Apache/1.3.27 | Response Headers |
| Accept-Ranges: bytes | |
| Content-Type: text/html | Entity Headers |
| Content-Length: 170 | |
| Last-Modified: Tue, 18 May 2004 10:14:49 GMT | |
| <html> | HTTP Response |
| <head> | |
| <title>Welcome to the Amazing Site!</title> | |
| </head> | |
| <body> | Message Body |
| <p>This site is under construction. Please come back later. Sorry!</p> | |
| </body> | |
| </html> | |

- **状态行**: HTTP协议版本号, 状态码和状态说明三部分构成
- **响应头**: 响应头是服务器传递给客户端用于说明服务器的一些信息 (Content-Type, charset等), 以及将来继续访问该资源时的策略。
- **响应体**: 返回给客户端的HTML文本内容, 或者其他格式的数据, 比如: 视频流、图片或者音频数据。

Socket

- **Socket**只是对TCP/IP协议栈操作的抽象, 即Socket是TCP/IP协议的API
- WEB Server都是基于Socket编程, 又称之为网络编程, 网络协议通过一个叫做Socket的对象抽象出来, Socket可以建立网络连接, 读数据, 写数据。

HTTPS

由于 http 请求是明文的, 链路上的任意一个路由, 都可以任意修改数据, 没有任何安全可言。因此 HTTPS 就诞生了。

1. https协议需要到ca申请证书, 一般免费证书很少, 需要交费。
2. http是超文本传输协议, 信息是明文传输, https则是具有安全性的ssl加密传输协议。
3. http和https使用的是完全不同的连接方式, 用的端口也不一样, 前者是80, 后者是443.
4. http的连接很简单, 是无状态的, https协议是由ssl (安全套接层) +http协议构建的可进行加密传输、身份认证的网络协议, 比http协议安全。
5. **HTTPS比HTTP页面加载慢一些, 不如HTTP高效, 但更安全。**
 - **对称加密**: 数据使用 A 密钥加密, 变成加密后的数据, 加密后的数据使用 A 密钥解密, 变成数据。
 - **非对称加密**: 数据使用 A 密钥加密, 变成加密后的数据, 然后加密后的数据经过 B 密钥解密, 变成明文。然后相反的, 数据经过 B 加密以后, 也能通过 A 解密。

Cookie和Session

概述:

因为HTTP协议是无状态的协议, 因此需要引入一种机制跟踪会话。因此有了Cookie和Session。Cookie通过在客户端记录信息确定用户身份, Session通过在服务器端记录信息确定用户身份。

Cookie机制:

1. Cookie以文本文件的方式存放在客户端，每次客户端向服务器发送请求都会带上这些信息。
2. Cookie的内容主要包括：名字、值、过期时间、路径和域。路径与域一起构成cookie的作用范围。域限定网站，路径限定这个网站下的路径。
3. 若不设置过期时间，则表示这个cookie的生命期为浏览器会话期间，关闭浏览器窗口，cookie就消失。这种生命期为浏览器会话期的cookie被称为会话cookie。
4. 会话cookie一般不会存储在硬盘上而是保存在内存里。若设置了过期时间，浏览器就会把cookie保存到硬盘上
5. Cookie信息都是放在请求头和响应头里
6. Cookie具有不可跨域名性，即一个网站只能取得它放在你的电脑中的信息，它无法从其它的cookie文件中取得信息，也无法得到你的电脑上的其它任何东西。
7. 客户端保存的Cookie中的内容大多数经过了加密处理，因此一般用户看来只是一些毫无意义的字母数字组合，只有服务器的CGI处理程序才知道它们真正的含义。

Session机制:

Session思想：分布式和中心化 Session共享

1. Session是由Tomcat创建的，Cookie应该也是。
2. session机制是一种服务器端的机制，当用户请求来自应用程序的 Web 页时，如果该用户还没有会话，则 Web 服务器将自动创建一个 Session 对象。当会话过期或被放弃后，服务器将终止该会话。
3. Session一般存在内存里，可以设置有效期，只要用户访问，服务器就会更新session的有效期。超期会删除。
4. 当程序需要为某个客户端的请求创建一个session时，服务器首先检查这个客户端的请求里是否已包含了一个session ID。
5. 如果已包含则说明以前已经为此客户端创建过session，服务器就按照session id把这个session检索出来使用（检索不到，会新建一个）；
6. 如果客户端请求不包含session id，则为此客户端创建一个session并且生成一个与此session相关联的session id。
7. session id的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个session id 将被在本次响应中通过Cookie机制返回给客户端保存。
8. URL重写：当客户端不支持Cookie时，可以采用URL重写来解决session id客户端保存的问题。即把服务器返回页面的a标签的URL后面加上sessionid=XXXXXXXX.

Cookie和Session的区别：

- 1、Cookie保存在客户端，Session保存在服务端。
- 2、Cookie不是很安全，因为存放在本地。
- 3、单个Cookie的数据不能超过4K，很多浏览器都限制一个站点最多保存50个Cookie。

数据库

2018年3月1日 11:08

目录

[关系型、非关系型数据库](#)

[Innodb、MyIASM引擎](#)

[事务、ACID、隔离级别、脏读、幻读](#)

[Innodb多版本并发控制（MVCC）](#)

[乐观锁 悲观锁 CAS](#)

[索引的实现](#)

[什么时候不建议使用索引](#)

[mysql索引，最左匹配原则](#)

[MySQL查询优化](#)

[数据库事务断电怎么办](#)

[三大范式](#)

[JDBC、ODBC](#)

[MySQL主从复制原理](#)

[Redis主从同步原理](#)

[Redis如何保证和MySQL数据一致？实时同步](#)

关系型和非关系型数据库

非关系型数据库的优势：

1. 性能NOSQL是基于键值对的，可以想象成表中的主键和值的对应关系，而且不需要经过SQL层的解析，所以性能非常高。
2. 可扩展性同样也是因为基于键值对，数据之间没有耦合性，所以非常容易水平扩展。

关系型数据库的优势：

1. 复杂查询可以用SQL语句方便的在一个表以及多个表之间做非常复杂的数据查询。
2. 事务支持使得对于安全性能很高的数据访问要求得以实现。

Innodb引擎和MyIASM引擎

区别：

1. MyIASM是非事务安全的，而InnoDB是事务安全的
2. MyIASM锁的粒度是表级的，而InnoDB支持行级锁
3. MyIASM不支持外键，InnoDB支持外键
4. MyIASM支持全文类型（FullText）索引，而InnoDB不支持全文类型索引
5. MyIASM保存了表的行数，InnoDB没有保存表的行数
6. MyIASM相对简单，效率上要优于InnoDB，小型应用可以考虑使用MyIASM

应用场景：

- 1、InnoDB用于事务处理，具有ACID事务支持等特性，如果在应用中执行大量insert和update操作，应该选择InnoDB
- 2、MyIASM管理非事务表，提供高速存储和检索以及全文搜索能力，如果再应用中执行大量select操作，应该选择MyIASM
- 3、对于一般的Web应用来说，应该选择MyIASM，效率更高，特定场景再用InnoDB

关系型数据库

- ◆ 基于关系代数理论
- ◆ 缺点：表结构不直观，实现复杂，速度慢
- ◆ 优点：健壮性高，社区庞大

而这些年分布式系统的兴起，对健壮性要求也没有这么高，关系型数据库的健壮性是针对一个节点来说的，当我们使用多个节点之后，这个健壮性也就不存在了，所以我们宁可用健壮性低的产品，获得性能提升或实现简单，比如KEY-VALUE型的。

JOIN GROUP_BY COUNT MIN ON 子查询

<https://coding.imooc.com/lesson/132.html#mid=6561>

事务

需要符合ACID特性

- 1、Atomicity, 原子性
- 2、Consistency, 关系型数据库的约束在事务整个期间都要保持一致
- 3、Isolation, 各个事务之间是相互独立的，隔离的
- 4、Durability, 事务做完之后，事务的结果是持久性的，即使断电了，结果也能存下来。

隔离级别、由低到高

<https://coding.imooc.com/lesson/132.html#mid=6562> 视频5分钟！！直观例子更好理解

1、Read Uncommitted

- Read uncommitted就是别人事务做到一半还没有提交的值，可以被我读出来。每一个事务Begin之后，事务操作的数据都是uncommitted的，事务结束后数据全部committed到数据库中。这中间uncommitted的数据都可以回滚。
- 脏读：事务可以读取未提交的数据，就叫脏读

2、Read Committed

是只能读到别人Committed之后的值，比如B事务修改了数据但没提交，A事务看到的是修改之前的数据值。

3、Repeatable Read

- 重复读，就是在开始读取数据（A事务开启）时，即使其他B事务修改了数据，但A事务读到的数据，再多select几次，也不会改变。
- 解决了脏读的问题，保证了同一事务中多次读取同一记录结果是一致的。
- 幻读：可重复读可能产生幻读，A事务开始准备插入一条记录id=6，B事务同时开始并成功插入一条记录id=6，此时A执行插入id=6操作，结果插入失败，因为id=6记录已经存在，这就是幻读。[Innodb通过多版本并发控制\(MVCC\)解决了幻读问题。](#)

[Innodb多版本并发控制](#)

- 注意：视频11分钟，用Repeatable Read权限，事务开启后，其他事务仍可以修改数据，只是Repeatable Read读到的数据不变（实际上变了）

4、Serializable

- 读加共享锁，写加排它锁。
- 视频中A事务读数据后，B事务开始读数据是可以的，B事务写数据会卡住。等A事务写完数据COMMIT后，B事务写数据。这样还是买了两双鞋，数据库只减了1，因为AB事务读的时候，读的是一样的库存，各自减1都是在这个一样的库存上减1写入。

5、值得一提的是：

大多数数据库默认的事务隔离级别是Read committed，比如Sql Server , Oracle。

Mysql的默认隔离级别是Repeatable read。

Innodb多版本并发控制 (MVCC)

概述：

- 可以认为MVCC是行级锁的一个变种，但是它在很多情况下避免了加锁操作，因此开销更低。虽然实现机制所有不同，但大都实现了非阻塞的读操作，写操作也只锁定必要的行。
- InnoDB的MVCC，是通过在每行纪录后面保存两个隐藏的列来实现的。这两个列，一个保存了行的创建时间，一个保存了行的过期时间，（存储的并不是实际的时间值，而是系统版本号）。每开始一个新的事务，系统版本号都会自动递增。事务开始时刻的系统版本号会作为事务的版本号，用来和查询到的每行纪录的版本号进行比较。

MVCC多版本并发控制是用来解决 “读-写冲突” 的无锁并发控制

- 同一个数据有多个版本，事务开启时看到是哪个版本就看到这个版本，最大的好处是读写不冲突，只有写于写是冲突的，这个特性可以很大程度上提升性能，避免了脏读

乐观锁是用来解决 “写-写冲突” 的无锁并发控制

- 认为事务间争用没有那么多，所以先进行修改，在提交事务前，检查一下事务开始后，有没有新提交改变，如果没有就提交，如果有就放弃并重试。乐观并发控制类似自选锁。乐观并发控制适用于低数据争用，写冲突比较少的环境。

乐观锁 (加入版本保护)

乐观锁的机制就是CAS，版本保护就是CAS中的期望值 CAS

顾名思义，就是很乐观，每次去拿（取）数据的时候都认为别人不会修改，所以上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型

并发量不是很高的时候可以用，并发量高时，比如抢票，数据就存在Redis这类内存中了，就不存mysql了，mysql太慢了。

<https://coding.imooc.com/lesson/132.html#mid=6562> 15分钟

乐观锁演示

- ◆ 读取数据，记录Timestamp
- ◆ 修改数据
- ◆ 检查和提交数据

```
SELECT count FROM `product` WHERE `productId` = 2;
UPDATE `product` SET `count` = 46 WHERE `productId` = 2 AND `count` = 47;
```

后面AND 'count'=47就是**版本保护**，就是我在读取数据为47后，写数据时要保证这个数据（实际的）仍然为47。如果有其他事务修改了这个值，不等于47了，那么我这个UPDATE操作就会显示如下：

执行成功，影响了[0]行，耗时：[3ms.]

执行成功，但影响了0行，这个会返回给程序，我们就知道UPDATE并没有更新数据

悲观锁

顾名思义，就是很悲观，每次去拿（取）数据的时候都认为别人会修改，所以每次在拿（取）数据的时候都会上锁，这样别人想拿这个数据就会block（阻塞）直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

FOR UPDATE就是悲观锁

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;
SET AUTOCOMMIT = 0;
SELECT @@tx_isolation;
SELECT count FROM `product` WHERE `productId` = 2 FOR UPDATE;
```

这里的**FOR UPDATE**会把select这行锁住，那么另一个线程再读同样的数据（也要加FOR UPDATE）就读不了，就能防止两个线程同时读同一个数据，同时减1，卖了两双鞋，数据库却只减了1这种情况。

这个**FOR UPDATE**就是加锁，实际中这种加锁是很耗费资源的，因此我们有乐观锁

FOR UPADTE会产生行锁、表锁或不加锁

MySQL 行锁、表锁

◆ SELECT ... FOR UPDATE

◆ 使用InnoDB引擎

- 1、有明确主键，并且有结果集的时候使用行锁
- 2、有明确主键，但无结果集的时候不加锁
- 3、无明确主键的时候使用表锁

行锁例子

```
SELECT * FROM mmall_product WHERE id=' 66'  
FOR UPDATE;
```

不加锁例子

```
· SELECT * FROM mmall_product  
    WHERE id=' -100' FOR UPDATE;
```

表锁例子

```
SELECT * FROM mmall_product  
    WHERE name= 'iphone' FOR UPDATE;
```

```
SELECT * FROM mmall_product  
    WHERE id <> ' 66' FOR UPDATE;
```

注意：这里id<>也是无明确主键

介绍一下关系型数据库的理论基础？

一二三范式 [MySQL基础概念](#)

给定表结构，请按要求写出SQL语句？

熟练掌握JOIN\外联结\GROUP BY\子查询等

什么是事务的ACID属性？事务有哪些隔离级别？

事务的性能太慢怎么办？

索引

- 1、建表的时候主键和外键都会自动创建索引
- 2、辅助索引，就是我们常规所指的索引，原文是SECONDARY KEY。辅助索引里还可以再分为唯一索引，非唯一索引。
- 3、唯一索引其实应该叫做唯一性约束，它的作用是避免一列或多列值存在重复，是一种约束性索引。
- 4、对100万行数据InnoDB表（Mysql默认引擎是InnoDB）的话，唯一索引比主键索引效率约慢9%，普通索引比主键索引约慢了50%以上。

5、

```
EXPLAIN SELECT * FROM repair WHERE hospital_name='同济医院'
```

| 信息 | 结果1 | 概况 | 状态 | | |
|----|-------------|--------|------------|------|-------------------|
| id | select_type | table | partitions | type | possible_keys |
| 1 | SIMPLE | repair | (Null) | ref | idx_hospital_name |

EXPLAIN 是解释查询的过程。这里显示可能用到的索引是idx_hospital_name，即我们查询条件 where/on中有索引，就会利用索引。

6、

```
EXPLAIN SELECT * FROM repair WHERE id >12
```

| select_type | table | partitions | type | possible_keys | key |
|-------------|--------|------------|-------|---------------|---------|
| SIMPLE | repair | (Null) | range | PRIMARY,pk_id | PRIMARY |

这就是用的主键，PRIMARY指主键，pk_id使我们起的主键别名。

7、

```
EXPLAIN SELECT * FROM `category` c JOIN `product` p  
ON p.`categoryName` = c.`categoryName`;
```

| 结果集1 | | | | |
|------|-------------|-------|------------|------|
| id | select_type | table | partitions | type |
| 1 | SIMPLE | p | null | ALL |
| 1 | SIMPLE | c | null | ALL |

这个例子中，**type都是ALL**（ALL是全表遍历查找），都是ALL的意思是这两张表是笛卡尔积，比如各10个数据，这就遍历了100个数据，这是非常慢的。遇见这种情况，我们就要给ON条件下的两个categoryName建索引，这样就会变快了。实际工作中漏建索引是很正常的。

```
执行(F8) SQL诊断 格式化 执行计划 数据库：hwj1  
1 EXPLAIN SELECT * FROM `category` c JOIN `product` p  
2   ON p.`categoryName` = c.`categoryName`;  
3  
4
```

| 结果集1 | | | | | | |
|------|-------------|-------|------------|------|-------------------|-------------------|
| | select_type | table | partitions | type | possible_keys | key |
| 1 | SIMPLE | p | null | ALL | null | |
| 1 | SIMPLE | c | ref | ref | idx_category_name | idx_category_name |

给数据多的那个表的categoryName建立索引后，看Type，变成了ref，根据索引查找的。

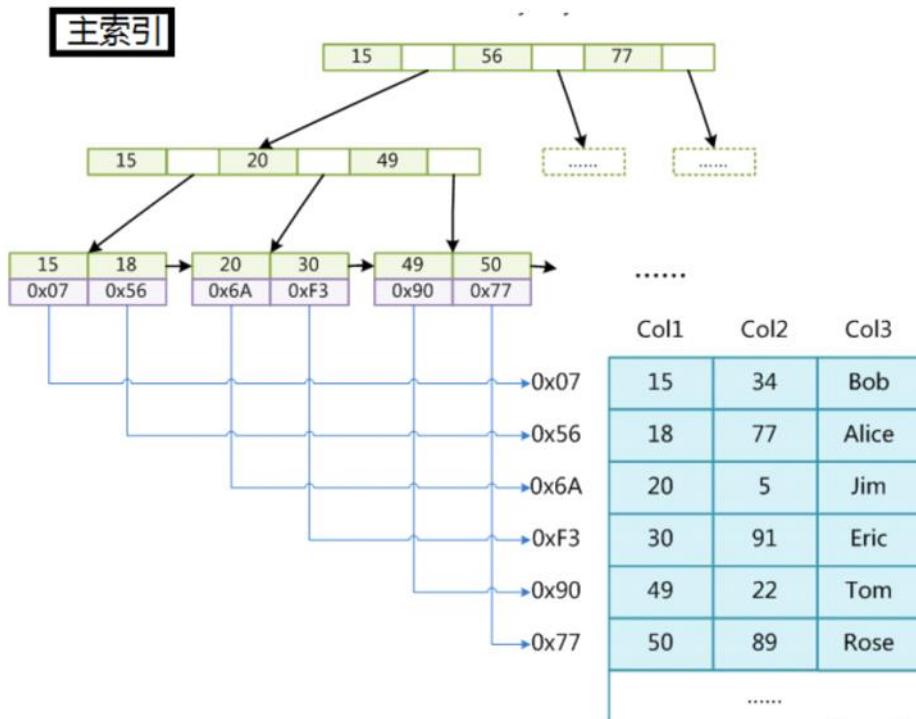
索引的实现

基础知识：

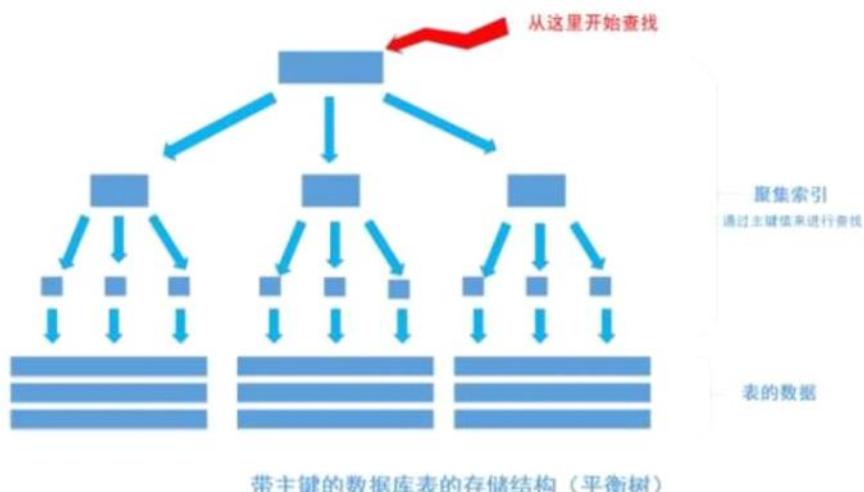
- **B树和B+树**这两种实现方式是最常见的，B树、B+树都是平衡树。
- MySQL的Innodb用B+树做索引。

- MySQL的MyISAM的索引有两种，主索引和辅助索引，主索引使用具有唯一性的键值，辅助索引键值可以重复。和Innodb不同的是，最后的叶子节点存的是地址，而Innodb最后的叶子节点存的是完整的数据。MyISAM索引如下图：

1.MyISAM



- 以下都是针对Innodb，先说主键（聚集索引），一个没加主键的表，它的数据无序的放置在磁盘存储器上，一行一行的排列的很整齐，如果给表上了主键，那么表在磁盘上的存储结构就由整齐排列的结构转变成了树状结构，也就是上面说的「平衡树」结构，换句话说，就是整个表就变成了一个索引，也就是所谓的聚集索引（Clustered Index），这就是为什么一个表只能有一个主键，一个表只能有一个「聚集索引」，因为主键的作用就是把「表」的数据格式转换成「索引」（平衡树）」的格式放置。



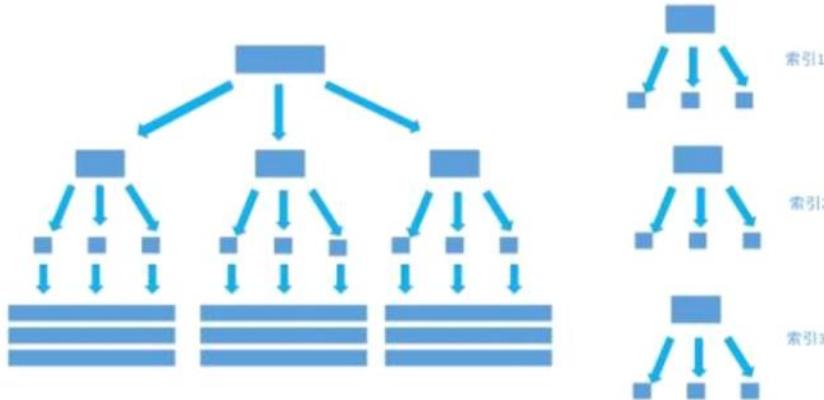
- 如上图：其中树的所有结点（底部除外）的数据都是由主键字段中的数据（比如主键 id）构成。最下面部分是真正表中的数据。假如我们执行`select * from table where id = 1256;`这个数有几层，就只需要几次查找。假如一张表有一亿条数据，需要查找其中某一条数据，按照常规逻辑，一条一条的去匹配的话，最坏的情况下需要匹配一亿次

才能得到结果，如果把这张表转换成平衡树结构（一棵非常茂盛和节点非常多的树），假设这棵树有10层，那么只需要10次IO开销就能查找到所需要的数据，速度以指数级别提升。

- 索引能让数据库查询数据的速度上升，而使写入数据的速度下降，原因很简单的，因为平衡树这个结构必须一直维持在一个正确的状态，**增删改数据**都会改变平衡树各节点中的索引数据内容，**破坏树结构**，因此，在每次数据改变时，DBMS必须去重新梳理树（索引）的结构以确保它的正确，这会带来不小的性能开销，也就是为什么索引会给查询以外的操作带来副作用的原因。
- 查找的时间复杂度**

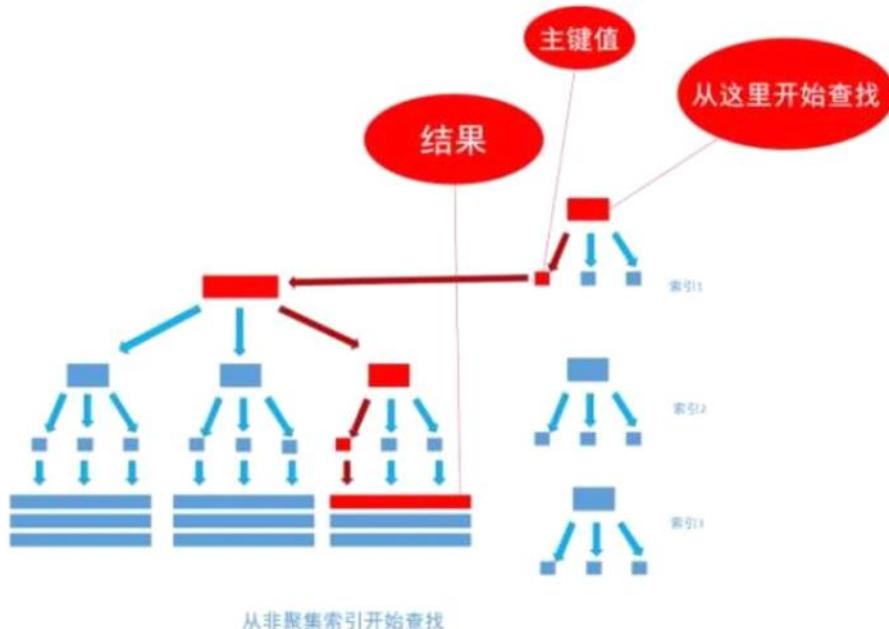
$$\log_{\text{树的分叉数}} \text{记录总数} = \text{查找次数}$$

- 非聚集索引**，每次给字段建一个新索引，索引字段中的数据就会被复制一份出来（**只复制索引字段，不复制其他的**），用于生成索引。因此，给表添加索引，会增加表的体积，占用磁盘存储空间。非聚集索引互相之间不存在关联。



带有主键和三个非聚集索引的表的存储结构

- 非聚集索引和聚集索引的区别**在于，通过聚集索引可以查到需要查找的数据，而通过非聚集索引可以查到记录对应的主键值，再使用主键的值通过聚集索引查找到需要的数据，如下图：

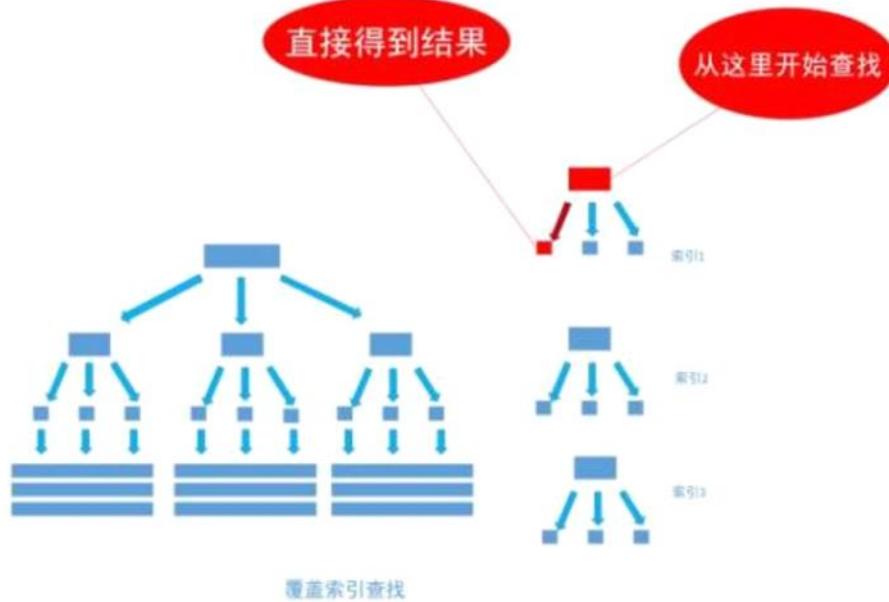


- 复合索引（覆盖索引、多字段索引）查询，比如建一个复合索引：index_b_u，这个索引包含birthday和user_name两个字段。执行查询语句

`select user_name from user_info where birthday = '1991-11-1';`

通过复合索引index_b_u查找birthday等于1991-11-1的叶节点的内容，然而，叶节点中除了有user_name表主键ID的值以外，user_name字段的值也在里面，因此不需要通过主键ID值的查找数据行的真实所在，直接取得叶节点中user_name的值返回即可。

通过这种覆盖索引直接查找的方式，可以省略不使用覆盖索引查找的后面两个步骤，大大的提高了查询性能，如下图：



什么时候不建议使用索引

1. **数据唯一性差的字段不要使用索引**: 比如性别，只有两种可能数据。意味着索引的二叉树级别少，多是平级。这样的二叉树查找无异于全表扫描。
2. **频繁更新的字段不要使用索引**: 比如logincount登录次数，频繁变化导致索引也频繁变化，增大数据库工作量，降低效率。
3. **字段不在where语句出现时不要添加索引**: 只有在where语句出现，mysql才会去使用索引
4. **数据量少的表不要使用索引**: 使用了改善也不大
5. 另外，如果mysql估计使用全表扫描要比使用索引快，则不会使用索引。

为什么不用二叉树？

二叉树优化**比较**次数，B/B+树优化**磁盘**读写次数

B树/B+树的每一个节点可以放更多的元素，这样做的原因是优化硬盘的读写次数。放的元素越多，读写次数越少。

B树/B+树定义

m阶B/B+树

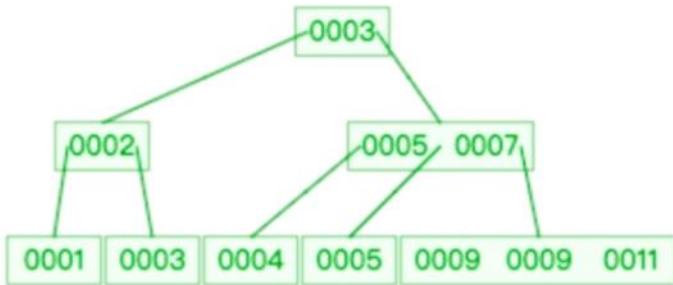
每个非叶子节点（除根外）至多有m个儿子，至少有 $\lceil m/2 \rceil$ 个儿子

根节点（如果不是叶子）至少有两个儿子

所有叶子节点在同一层

B树

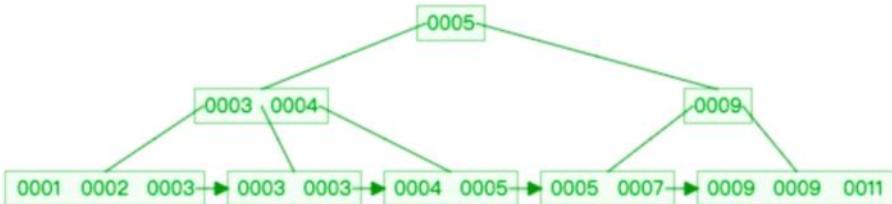
[1,2,3,3,3,4,5,5,7,9,9,11]



- 1、注意B树这是放在硬盘里的
- 2、**B树的缺点**。比如我select * from table，顺序返回整个表。顺着这个B树就很难返回所有的值，比如第一个1，第二个2往上找，然后找完了又要往下找3，接着再往上找，有时上有时下，就比较麻烦。

B+树

◆ [1,2,3,3,3,4,5,5,7,9,9,11]



优化后就是B+树。

- 1、跟B树的区别，所有原始的数值最终都会出现在叶子结点上，并且**串起来就是原始数据的顺序**。
- 2、根节点和中间的节点都是用来帮助索引的节点

数据库索引的作用？

加快查找速度，约束数据的值，如唯一索引

数据库索引的分类？

Clustered Index, 每个表至多一个，通常作为主键存在

Non-clustered Index, 可以建很多个，比Clustered Index稍微慢一点

为什么要给表加上主键？

为什么加索引后会使查询变快？

为什么加索引后会使写入、修改、删除变慢？

什么情况下要同时在两个字段上建索引？

B树和B+树的区别？

数据的值可以出现在B树的非叶子结点。而B+树，值全在叶子节点，非叶子节点是作为帮助找到值得目的出现的。

B树和二叉搜索树（如红黑树）的区别？

B树的每一个节点可以放更多的元素，这样做的原因是优化硬盘的读写次数。放的元素越多，读写次数越少。二叉树每一个节点只有一个值。

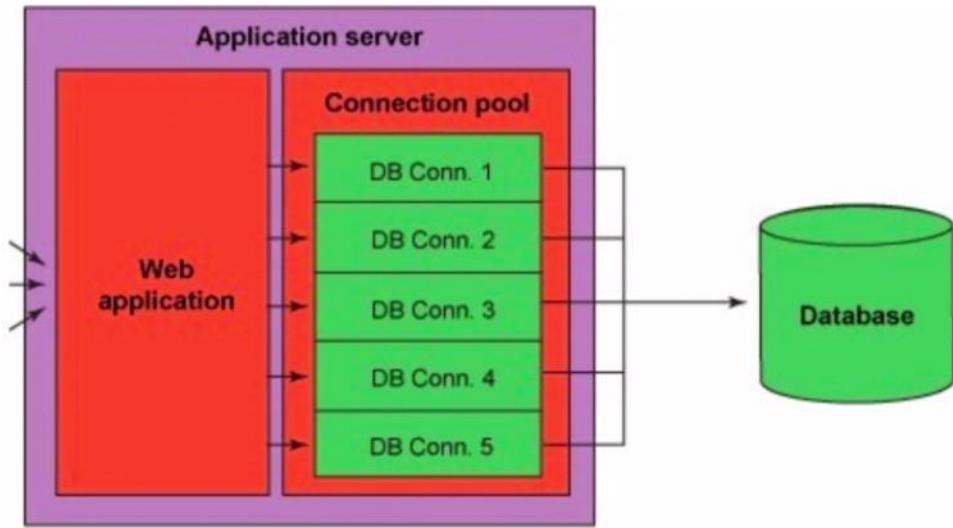
怎样选择给哪些字段加上索引？

经常需要查找的字段需要加上索引，另外通过EXPLAIN关键字解释查询语句，分析哪些字段可以加索引。

数据库的连接池

不用连接池的话，就要根据每个请求或者每个用户来建立连接。这样的缺点是显而易见的。1、这样需要建立很多连接，建立连接是要花很多时间的。2、有的用户建立了连接，却没有使用，造成了资源浪费。

因此需要用连接池，如下先建好5个连接（Tomcat默认的连接是10到100个，可修改），每次请求来了直接用，用完了还回去，如果请求太多，来不及处理，超时会报错（线程池请求太多会排队，不会超时报错）



综合例题

下列方法中，_____不可以用来程序调优。

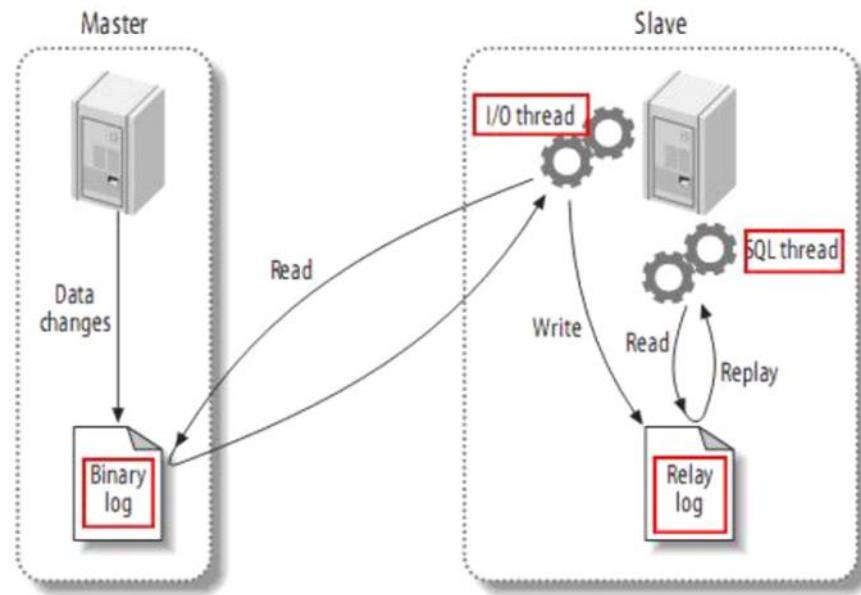
- A、改善数据访问方式以提升缓存命中率
 - B、使用多线程的方式提高 I/O 密集型操作的效率
 - C、利用数据库连接池替代直接的数据库访问
 - D、使用迭代替代递归
 - E、合并多个远程调用批量发送
 - F、共享冗余数据提高访问效率
-
- A、可以
 - B、I/O密集型指现有线程已经在等待I/O端口响应了，你再增加线程，排队等待的人更多，因此不可以用来程序调优
 - D、迭代消耗小于递归
 - E、类似滑动窗口，与其发送一个等待一个，不如发送多个一起等待
 - F、这个要注意共享冗余数据是可以提高访问效率的

JDBC和ODBC

JDBC使用起来更方便，ODBC因为是C编写，性能更快一些。

- **JDBC**: (Java Data Base Connectivity,java数据库连接) 是一种用于执行SQL语句的Java API，它是Java十三个规范之一。可以为多种关系数据库提供统一访问，它由一组用Java语言编写的类和接口组成。 JDBC的最大特点是它独立于详细的关系数据库。
- **ODBC**: 是微软公司开放服务结构(WOSA, Windows Open Services Architecture)中有关数据库的一个组成部分。一个基于ODBC的应用程序对数据库的操作不依赖数据库类型，能以统一的方式处理全部的数据库。

MySQL主从同步原理



一句话解释：

Slaver读取Master的binlog并顺序执行

概述：

- MySQL的主从复制是一个**异步的复制过程**（虽然一般情况下感觉是实时的），在Master与Slave之间实现整个主从复制的过程是由三个线程参与完成的。其中有两个线程（SQL线程和IO线程）在Slave端，另一个线程（IO线程）在Master端。
- 要实现MySQL的主从复制，首先必须**打开Master端的binlog记录功能**，否则就无法实现。因为整个复制过程实际上就是Slave从Master端获取binlog日志，然后再在Slave上以**相同顺序执行获取的binlog日志中的记录的各种SQL操作**

详细过程

- 在Slave服务器上执行`start slave`命令开启主从复制开关，开始进行主从复制。
- 此时，Slave服务器的IO线程会通过在master上已经授权的复制用户权限请求连接master服务器，并请求从执行binlog日志文件的指定位置（日志文件名和位置就是在配置主从复制服务时执行`change master`命令指定的）之后开始发送binlog日志内容
- Master服务器接收到来自Slave服务器的IO线程的请求后，其上负责复制的IO线程会根据Slave服务器的IO线程请求的信息分批读取指定binlog日志文件指定位置之后的binlog日志信息，然后返回给Slave端的IO线程。返回的信息中除了binlog日志内容外，还有在Master服务器端记录的IO线程。返回的信息中除了binlog中的下一个指定更新位置。
- 当Slave服务器的IO线程获取到Master服务器上IO线程发送的日志内容、日志文件及位置点后，**会将binlog日志内容依次写到Slave端自身的Relay Log**（即中继日志）文件（`Mysql-relay-bin.xxx`）的最末端，并将新的binlog文件名和**位置**记录到`master-info`文件中，以便下一次读取master端新binlog日志时**能告诉Master服务器从新binlog日**

志的指定文件及位置开始读取新的binlog日志内容

5. Slave服务器端的SQL线程会实时检测本地Relay Log 中新增的日志内容，然后及时把 Relay LOG 文件中的内容解析成sql语句，并在自身Slave服务器上按解析SQL语句的位置顺序执行应用这样sql语句，并在relay-log.info中记录当前应用中继日志的文件名和位置点

知识点

1. 3个线程，主库IO，从库IO和SQL及作用
2. master.info (从库) 作用
3. relay-log 作用
4. 异步复制
5. binlog作用 (如果需要级联需要开启Binlog)

小结

1. 主从复制是异步的逻辑的SQL语句级的复制
2. 复制时，主库有一个I/O线程，从库有两个线程，I/O和SQL线程
3. 实现主从复制的必要条件是主库要开启记录binlog功能
4. 作为复制的所有Mysql节点的server-id都不能相同
5. binlog文件只记录对数据库有更改的SQL语句 (来自主库内容的变更)，不记录任何查询 (select, show) 语句

工作中常用主从模式

1、单向主从复制逻辑图



2、双向主主同步逻辑图，此架构可以在Master1端或Master2端进行数据写入



Redis主从同步原理

直接看这篇博文，写的太好了

来自 <https://blog.csdn.net/qq_23923485/article/details/73456784>

一句话解释：

主Redis所有数据生成RDB文件发送给从Redis，并把存在缓冲区中的最近的写、删命令发送给从Redis让其执行命令。同步以后主Redis的写、删命令都会发给从Redis让其执行来保持数据一致状态。

如何部署主从Redis

- 假设现在有两个Redis服务器，地址分别为127.0.0.1:6379和127.0.0.1:12345，如果我们向服务器127.0.0.1:12345发送以下命令：
 - 127.0.0.1:12345> **SLAVEOF** 127.0.0.1 6379
 - OK
- 那么服务器127.0.0.1:12345将成为127.0.0.1:6379的从服务器，而服务器127.0.0.1:6379则会成为127.0.0.1:12345的主服务器。

旧版复制功能

Redis的复制功能分为**同步** (sync) 和**命令传播** (command propagate) 两个操作：

- 同步**操作用于将从服务器的数据库状态更新至主服务器当前所处的数据库状态；
- 命令传播**操作则用于在主服务器的数据库状态被修改，导致主从服务器的数据库状态出现不一致时，让主从服务器的数据库重新回到一致状态。

同步

- 当客户端向从服务器发送SLAVEOF命令，要求从服务器复制主服务器时，从服务器首先需要执行同步操作，也即是，将从服务器的数据库状态更新至主服务器当前所处的数据库状态。
- 从服务器对主服务器的同步操作需要通过向主服务器发送SYNC命令来完成，以下是**SYNC命令的执行步骤**：
 - 从服务器向主服务器发送SYNC命令；
 - 收到SYNC命令的主服务器执行BGSAVE命令，在后台生成一个RDB文件，并使用一个**缓冲区记录从现在开始执行的所有写命令**；
 - 当主服务器的BGSAVE命令执行完毕时，主服务器会将BGSAVE命令生成的RDB文件发送给从服务器，从服务器接收并载入这个RDB文件，将自己的数据库状态更新至主服务器执行BGSAVE命令时的数据库状态。
 - 主服务器将记录在**缓冲区里面的所有写命令发送给从服务器**，从服务器执行这些写命令，将自己的数据库状态更新至主服务器数据库当前所处的状态。

命令传播

- 在执行完同步操作之后，主从服务器之间数据库状态已经相同了。但这个状态并非一成不变，如果主服务器执行了写操作，那么主服务器的数据库状态就会修改，并导致主从服务器状态不再一致。
- 所以为了让主从服务器再次回到一致状态，主服务器需要对从服务器执行命令传播操作：**主服务器会将自己执行的写命令，也即是造成主从服务器不一致的那条写命令，发送给从服务器执行**，当从服务器执行了相同的写命令之后，主从服务器将再次回到一致状态。

旧版复制功能的缺陷

SYNC命令是一个非常耗费资源的操作，若断线后为了弥补一小部分缺失数据而重新复制，这样的效率非常低。

1. 在Redis中，从服务器对主服务器的复制可以分为以下两种情况：
 - **初次复制**：从服务器以前没有复制过任何主服务器，或者从服务器当前要复制的主服务器和上一次复制的主服务器不同；
 - **断线后重复制**：处于命令传播阶段的主从服务器因为网络原因而中断了复制，但从服务器通过自动重连接重新连上了主服务器，并继续复制主服务器。
2. 对于初次复制来说，旧版复制功能能够很好地完成任务，但对于断线后重复制来说，旧版复制功能虽然也能让主从服务器重新回到一致状态，但效率却非常低。
3. 主从服务器断开的时间越短，主服务器在断线期间执行的写命令就越少，而执行少量写命令所产生的数据量通常比整个数据库的数据量要少得多，在这种情况下，为了让从服务器补足一小部分缺失的数据，却要让主从服务器重新执行一次SYNC命令，这种做法无疑是低效的。**SYNC命令是一个非常耗费资源的操作，所以Redis最好在真需要的时候才需要执行SYNC命令，因此有了新版复制功能。**

新版复制功能的实现

一句话解释：**如果偏移量的数据主服务器缓冲区内还有，那么就用部分同步，反之就用完整同步。**

- 为了解决旧版复制功能在处理断线重复制情况时的低效问题，Redis从2.8版本开始，使用PSYNC命令代替SYNC命令来执行复制时的同步操作。
- **PSYNC命令具有完整重同步** (full resynchronization) 和**部分重同步** (partial resynchronization) 两种模式：
 1. **完整重同步**用于处理初次复制情况：完整重同步的执行步骤和SYNC命令的执行步骤基本一样，它们都是通过让主服务器创建并发送RDB文件，以及向从服务器发送保存在缓冲区里面的写命令来进行同步；
 2. **部分重同步**则用于处理断线后重复制情况：当从服务器在断线后重新连接主服务器时，如果条件允许，主服务器可以将主从服务器连接断开期间执行的写命令发送给从服务器，从服务器只要接收并执行这些写命令，就可以将数据库更新至主服务器当前所处的状态。
- 其实看到这里的时候心里还是有一个疑问的：如果上面的例子是T3时候从服务器掉线，然后在T10093的时候才连接上或者更长的时间呢！！！你这样一条指令一条指令地传输过去还不如直接来一个SYNC命令快一些。所以在我看来**使用PSYNC进行操作时，什么时候部分重同步，什么时候全部重同步是一个策略问题**。当然Redis会解决这个问题。

部分重同步的实现

部分重同步功能由以下三个部分构成：

1. 主服务器的**主复制偏移量** (replication offset) 和从服务器的**从复制偏移量**；
2. 主服务器的**复制积压缓冲区** (replication backlog)；

3. 服务器的运行ID (run ID)。

复制偏移量

1. 执行复制的双方——主服务器和从服务器会分别维护一个复制偏移量：
 - a. 主服务器每次向从服务器传播N个字节的数据时，就将自己的复制偏移量的值加上N；
 - b. 从服务器每次收到主服务器传播来的N个字节的数据时，就将自己的复制偏移量的值加上N；

(我靠！！难道从服务器没有反馈吗？丢了怎么办？难道是用TCP？大家继续看，我只是想穿插一些我的思路)
2. 通过对比主从服务器的复制偏移量，程序可以很容易地知道主从服务器是否处于一致状态：
 - a. 如果主从服务器处于一致状态，那么主从服务器两者的偏移量总是相同的；
 - b. 相反，如果主从服务器两者的偏移量并不相同，那么说明主从服务器并未处于一致状态。

假设从服务器A在断线之后就立即重新连接主服务器，并且成功，那么接下来，**从服务器将向主服务器发送PSYNC命令，报告从服务器A当前的复制偏移量为10086和主服务器不一样，那么这时，主服务器应该对从服务器执行完整重同步还是部分重同步呢？**如果执行部分重同步的话，主服务器又如何补偿从服务器A在断线期间丢失的那部分数据呢？以上问题的答案都和**复制积压缓冲区有关。**

复制积压缓冲区

1. 复制积压缓冲区是由主服务器维护的一个固定长度 (fixed-size) 先进先出 (FIFO) 队列，默认大小为1MB。
2. 和普通先进先出队列随着元素的增加和减少而动态调整长度不同，固定长度先进先出队列的长度是固定的，当入队元素的数量大于队列长度时，最先入队的元素会被弹出，而新元素会被放入队列。
3. 当主服务器进行命令传播时，它不仅会将写命令发送给所有从服务器，还会将写命令入队到复制积压缓冲区里面。
4. 因此，主服务器的复制积压缓冲区里面会保存着一部分最近传播的写命令。
5. 当**从服务器重新连上主服务器时，从服务器会通过PSYNC命令将自己的复制偏移量offset发送给主服务器，主服务器会根据这个复制偏移量来决定对从服务器执行何种同步操作：**
 - a. 如果offset偏移量之后的数据（也即是偏移量offset+1开始的数据）仍然存在于复制积压缓冲区里面，**即偏移量的数据仍在缓冲区内，那么主服务器将对从服务器执行部分重同步操作；**
 - b. 相反，**如果offset偏移量之后的数据已经不存在于复制积压缓冲区，那么主服务器将对从服务器执行完整重同步操作。**

根据需要调整复制积压缓冲区的大小

- Redis为复制积压缓冲区设置的默认大小为1MB，如果主服务器需要执行大量写命令，又或者主从服务器断线后重连接所需的时间比较长，那么这个大小也许并不合适。

- 如果主服务器平均每秒产生1 MB的写数据，而从服务器断线之后平均要5秒才能重新连接上主服务器，那么复制积压缓冲区的大小就不能低于5MB，一般设双倍也就是10MB了。
- 为了安全起见，可以将复制积压缓冲区的大小设 $2 * \text{second} * \text{write_size_per_second}$ ，这样可以保证绝大部分断线情况都能用部分重同步来处理。
- 至于复制积压缓冲区大小的修改方法，可以参考配置文件中关于repl-backlog-size选项的说明。

服务器运行ID

1. 除了复制偏移量和复制积压缓冲区之外，实现部分重同步还需要用到服务器运行ID。
2. 每个Redis服务器，不论主服务器还是从服务，都会有自己的运行ID。
3. 运行ID在服务器启动时自动生成，由40个随机的十六进制字符组成，例如53b9b28df8042fdc9ab5e3fcbbbabbff1d5dce2b3。
4. 当从服务器对主服务器进行初次复制时，主服务器会将自己的运行ID传送给从服务器，而从服务器则会将这个运行ID保存起来（注意哦，是从服务器保存了主服务器的ID）。
5. **当从服务器断线并重新连上一个主服务器时，从服务器将向当前连接的主服务器发送之前保存的运行ID。**
 - a. 如果从服务器保存的运行ID和当前连接的主服务器的运行ID相同，那么说明从服务器断线之前复制的就是当前连接的这个主服务器，主服务器可以继续尝试**执行部分重同步操作**；
 - b. 相反地，如果从服务器保存的运行ID和当前连接的主服务器的运行ID并不相同，那么说明从服务器断线之前复制的主服务器并不是当前连接的这个主服务器，主服务器将对从服务器**执行完整重同步操作**。

心跳检测

1. 在命令传播阶段，从服务器默认会以每秒一次的频率，向主服务器发送命令。
2. 如果主服务器超过一秒钟没有收到从服务器发来的心跳检测命令，那么主服务器就知道主从服务器之间的连接出现问题了。
3. 我们可以配置连接丢失时间过长时让主服务器拒接执行写命令，虽然我觉得这样做并没有什么鸟用。。。

MySQL查询优化

包含count(*)的语句会扫描大量数据，但很可能我们想要的只是一个总数也就是一个数据。

比如 SELECT actor_id,COUNT(*) FROM article GROUP BY actor_id;

针对这个，可以改变库表结构，增加单独的汇总表，记录每个作者的文章总数。

把一个十分复杂的查询拆成小的查询，比如Delete大量数据的操作

```
mysql> DELETE FROM messages WHERE created < DATE_SUB(NOW(), INTERVAL 3 MONTH);
```

一个查询要删除所有大于3个月的数据，假设数据量十分庞大，这样一个查询可能一次锁

住很多数据，占满整个事务日志，耗尽系统资源，阻塞其他小的重要的查询，因此可以修改如下：

```
rows_affected = 0
do {
    rows_affected = do_query(
        "DELETE FROM messages WHERE created < DATE_SUB(NOW(), INTERVAL 3 MONTH)
        LIMIT 10000")
} while rows_affected > 0
```

一次删除10000条数据一般是比较高效的，分多次进行避免了服务器压力太大。

分解关联查询

很多高性能的应用都会对关联查询进行分解。简单地，可以对每一个表进行一次单表查询，然后将结果在应用程序中进行关联。例如，下面这个查询：

```
mysql> SELECT * FROM tag
      ->     JOIN tag_post ON tag_post.tag_id=tag.id
      ->     JOIN post ON tag_post.post_id=post.id
      -> WHERE tag.tag='mysql';
```

可以分解成下面这些查询来代替：

```
mysql> SELECT * FROM tag WHERE tag='mysql';
mysql> SELECT * FROM tag_post WHERE tag_id=1234;
mysql> SELECT * FROM post WHERE post.id in (123,456,567,9098,8904);
```

1. 好处是，让缓存更加高效，某些数据已经缓存了，就不用执行这部分的缓存了。如果用联表，不管有没有缓存，都得查询。
2. 将查询分解后，可以减少锁的竞争。
3. 查询本身效率也有提升，使用IN () 代替关联查询，可以让MySQL按照ID顺序进行查询，比随机的关联效率更高。

尽量不要用外键约束，如果是要限制数值约束，用触发器显示地限制取值会更好一些。

因为外键约束使得查询需要额外访问一些表，如果是写入操作，还会额外的锁其他表

数据库事务断电怎么办

本地事务数据库断电的这种情况，它是怎么保证数据一致性的呢？

我们使用SQL Server来举例，我们知道我们在使用 SQL Server 数据库是由两个文件组成的，一个数据库文件和一个日志文件，通常情况下，日志文件都要比数据库文件大很多。**数据库进行任何写入操作的时候都是要先写日志的**，同样的道理，我们在执行事务的时候数据库首先会记录下这个事务的redo操作日志，然后才开始真正操作数据库，在操作之前首先会把日志文件写入磁盘，那么当突然断电的时候，即使操作没有完成，在重新启动数据库时候，数据库会根据当前数据的情况进行undo回滚或者是redo前滚，这样就保证了数据的强一致性。

设计模式

2018年3月1日 19:14

通过反射可以破坏单例模式！！！

如果要抵御这种攻击，要防止构造函数被成功调用两次。需要在构造函数中对实例化次数进行统计，大于一次就抛出异常。

<https://www.cnblogs.com/wyb628/p/6371827.html>

JDK中哪些类实现了单例模式

- java.lang.reflect.Proxy类（动态代理类）
- java.lang.Runtime类

Singleton优缺点

确保全局至多只有一个对象

用于：构造缓慢的对象，需要统一管理的资源

缺点：很多全局状态，线程安全性

需要统一管理的资源，比如线程池，连接池。比如不可能创建多个连接池。

Singleton的创建

◆ 双重锁模式 Double checked locking

◆ 作为Java类的静态变量

◆ 使用框架提供的能力

- 1、双重锁模式用来保证线程安全，加锁前检查对象是否为null，加锁后再检查一次对象是否为null
- 2、作为Java类的静态变量，让这个变量指向Singleton对象，显然更简单，缺点是程序初始化的时候就要创建这个变量。否则你想控制它的创建，别人也想控制，就很麻烦了。
- 3、最简单的，使用框架，比如Spring的@Autowired

变继承关系为组合关系

继承关系

- ◆ 描述is-a关系
- ◆ 不要用继承关系来实现复用
- ◆ 使用设计模式来实现复用

Manager是继承Employee的，如果Employee升级成了Manager怎么办？

```
public class Manager extends Employee {
```

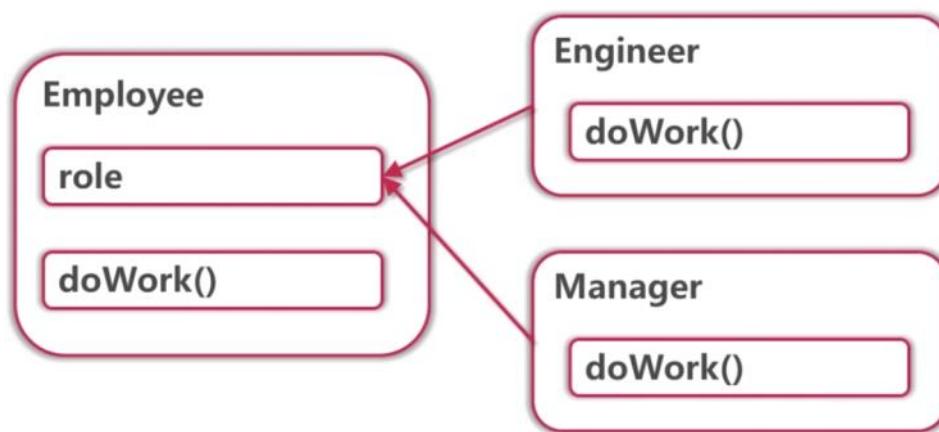
如果是这种继承关系，那么Employee无法升级成Manager，我们new一个Employee，那么这个Employee就永远是Employee。所以继承关系面对这种Employee升级成了Manager的应用场景，必须new一个，并把原来的引用指向这个新new的Manager

```
Employee employee2 = new Employee("Mary", 20000);  
Employee employee3 = new Employee("John", 10000);
```

```
employee2 = new Manager("Mary", 400000,  
    Arrays.asList(employee1));
```

上面问题更好的方案，利用State设计模式

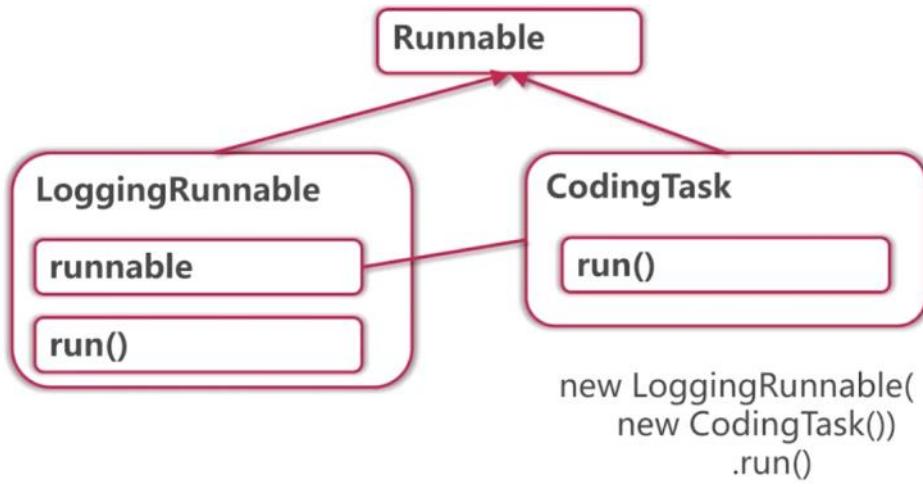
State Pattern (状态模式)



- 1、这个Employee有doWork()和role，role是可变的。
- 2、当role= "Engineer"，就调用Engineer的doWork(),当role= "Manager",就调用Manager的doWork()
- 3、以上就叫做state模式，变继承为组合的思想

Decorator Pattern (装饰模式)

装饰模式指的是在不必改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。



实例在下面

```

package com.js.designpattern.decorator;
public class Test {
    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("-----普通Worker的功能-----");
        new Worker().run();
        System.out.println("-----增加Worker的记日志功能-----");
        new LoggingRunnable(new Worker()).run();
        System.out.println("-----增加Worker的记日志和检查功能-----");
        new CheckingRunnable(new LoggingRunnable(new Worker())).run();
    }
}

```

状态模式和装饰模式的核心思想都是变继承为组合！！！！

工厂模式

<http://www.runoob.com/design-pattern/factory-pattern.html>

1、复杂对象适合使用工厂模式，而简单对象，特别是只需要通过 new 就可以完成创建的对象，无需使用工厂模式

2、优点：

一个调用者想创建一个对象，只要知道其名称就可以了。

扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。

屏蔽产品的具体实现，调用者只关心产品的接口。

3、缺点：

每次增加一个产品时，都需要增加一个具体类，在一定程度上增加了系统的复杂度，这并不是什么好事

Abstract Factory Pattern(抽象工厂模式)

抽象工厂模式就是根据输入需要创建不同的工厂，是在工厂模式的基础上又加了一层。看实现代码就懂了

<http://www.runoob.com/design-pattern/factory-pattern.html>

<http://www.runoob.com/design-pattern/abstract-factory-pattern.html>

我们使用new来创建对象时有两个缺点，1、编译时必须决定创建哪个类的对象，即new类名，类名一定要写出来。2、参数意义不明确，new类名（参数）

抽象工厂模式和建造者模式就是解决上面两个问题

比较

◆ task = new LoggingTask(new CodingTask());

◆ task = taskFactory.createCodingTask();

- 1、第一行是传统代码
- 2、第二行是抽象工厂模式代码，taskFactory是一个抽象的东西，是个接口。这个接口有一个createCodingTask方法。这一行就把编译时必须知道的类型全部扔掉了。
- 3、我们通过实现taskFactory，就能很方便的配置createCodingTask到底实现什么具体功能。比如写好几套taskFactory的实现，根据具体场景应用具体的实现。

Builder Pattern (建造者模式)

比较

◆ employee = new Employee(
 oldEmployee.getName(), 15000);

◆ employee = Employee.fromExisting(oldEmployee)
 .withSalary(15000)
 .build();

◆ 不可变对象往往配合Builder使用

- 1、第一行是传统代码，只看这行代码并不知道15000是个什么意思
- 2、第二行是建造者模式代码，显然15000是薪水显而易见，解决了new对象参数不明确的问题
- 3、SpringSecurity的配置类就是明显的建造者模式啊！！！

Top K 排序

2018年3月10日 19:18

无重复数组找Top K

例子：有一个长度为1000万的int数组，各元素互不重复。如何以最快的速度找出其中最大的100个元素？

- 1、**快速排序**，时间复杂度 $N \log N$
- 2、**堆排序**，时间复杂度 $N \log N$ ，优点是节省内存
- 3、**如果没有对内存和系统资源的要求**，可以采用**多线程**，将1000万大小的数组分割为1000个元素组成的若干小数组，利用JDK自带的高效排序算法void `java.util.Arrays.sort(int[] a)`来进行排序，多线程处理，主线程汇总结果后取出各个小数组的top 100，归并后再进行一次排序得出结果。速度比1、2、快。
- 4、**位图数组**，前提是**无重复数组**，时间复杂度 $O(N)$ ，内存占用为数组最大长度除以32+1

有重复数组找Top K

- 1、**快速排序或者堆排序**，时间复杂度 $N \log N$
- 2、**当 $N >> K$** 。可以使用**快速排序中的partition函数**，时间复杂度为 $O(N \log K)$ 。
 - a. 将数组分为两个组，A和B。
 - b. 若A组的个数大于K，则继续在A分组中找取最大的K个数字。
 - c. 若A组中的数字小于K，其个数为T，则继续在B中找取 $K-T$ 个数字。
- 3、**当N个数都是正整数，且取值范围不大的时候**。可以使用**空间换时间的方法**，使用一个数组记录每个元素出现的次数，数组长度为maxN，即N个数中的最大元素的值，然后找出最大的K个数。时间复杂度为 $O(N) + O(\max N)$ ，近似为 $O(N)$ 。

并行计算、外部排序、归并排序

2018年3月3日 10:55

并行计算

并行计算的方法

◆ 将数据拆分到每个节点上 如何拆分

◆ 每个节点并行的计算出结果 什么结果

◆ 将结果汇总 如何汇总

外部排序

内存不足的情况下，如何排序10G个元素？？？

<https://coding.imooc.com/lesson/132.html#mid=6605>

1、如果内存足够，用一般的快排啥的，几分钟也能搞定10G个元素，**100M对于NlogN算法是个秒级运算。**

2、数据不全在内存里，就要用到外部排序

3、用扩展的归并排序来解决这个问题

4、普通的归并排序如下

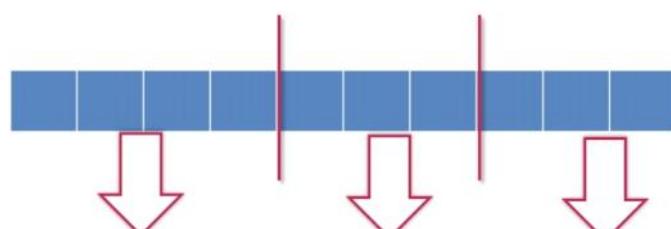
如何归并：

- [1,3,6,7],[1,2,3,5] → 1
- [3,6,7],[1,2,3,5] → 1
- [3,6,7],[2,3,5] → 2
- [3,6,7],[3,5] → 3
- ...

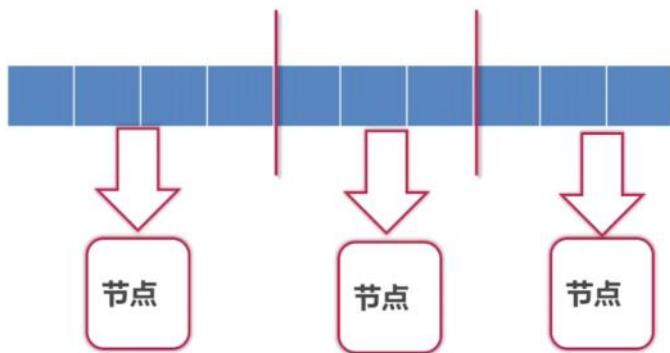
数据分为左右两半有序序列后，各自最小的拿出来比较，其中较小的选出来，如此循环

5、如何外部排序？

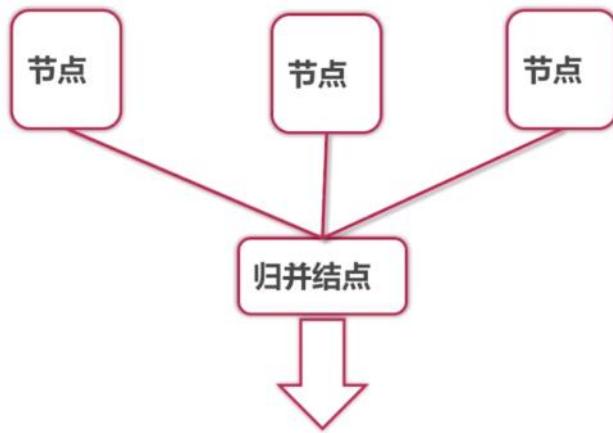
例：外部排序



例：外部排序



(1)把数据分为切分成很多段，每一段能分别送进一个节点进行排序（这个节点的内存足够放下这段数据），这个节点自己用任何排序方法都可以。

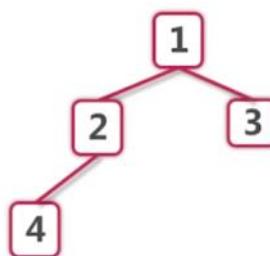
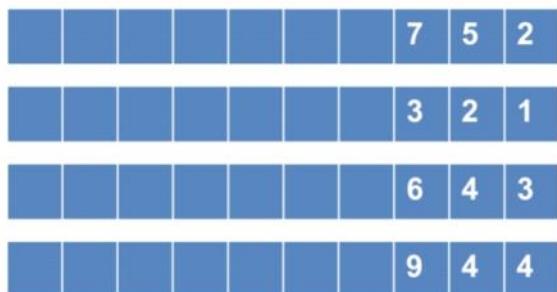


(2)各个节点排出来的结果都是有序的，然后把这些节点的结果送入归并节点，进行归并排序。因为只用把各个节点最小的数送入归并节点，因此内存是足够的，但是这里要注意，比如下面的图，1拿掉之后，要把1拿掉的那个队列后面的2送进来！！！！

6、归并节点的算法实现

归并结点实现

k路归并



每个节点最小值拿出来后，组成右边的完全二叉树（从上到下，从左到右依次排满，只可能右下角缺掉），很多库都有这个结构，叫PriorityQueue优先队列，直接拿来用就行

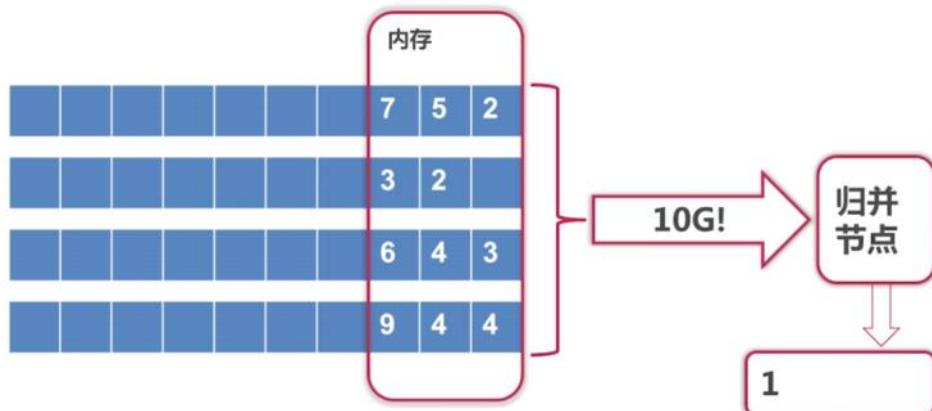
7、PriorityQueue怎么用？

```
Q.push(2); Q.push(1);
Q.push(3); Q.push(4);
```

`Q.pop() == 1`

数据一个一个push进去，然后依次拿出来，第一次拿出来的就是最小的

8、解决从硬盘读取节点数据很慢的问题，如下增加一个内存缓冲区，每次送一批数据进来，而不止各个节点的最小数。



9、使用`Iterable<T>`来作为缓冲区

归并：使用`Iterable<T>`接口

- ◆ 可以不断获取下一个元素的能力
- ◆ 元素存储/获取方式被抽象，与归并结点无关
- ◆ `Iterable<T> merge(List<Iterable<T>> sortedData);`

归并数据源来自 `Iterable<T>.next()`

- ◆ 如果缓冲区空，读取下一批元素放入缓冲区
- ◆ 给出缓冲区第一个元素
- ◆ 可配置项：缓冲区大小，如何读取下一批元素

即每个节点的数据是`Iterable<T>`，`merge()`就是归并算法，输出也是`Iterable<T>`，`Iterable<T>.next()`方法获得下一个元素。

多线程、锁、CAS、AQS

2018年3月3日 11:27

目录

[Java中锁的分类](#)

[如何实现可重入锁\(ReentrantLock\)](#)

[Lock和synchronized的区别](#)

[什么时候用synchronized，什么时候用Lock](#)

[Java中concurrent包的实现](#)

[原子类AtomicInteger的实现原理](#)

[synchronized底层实现原理](#)

[ReentrantLock底层实现原理 AQS CAS CAS存在的问题](#)

[volatile和synchronized的区别](#)

[典型的volatile使用场景](#)

[死锁条件](#)

[死锁防止](#)

[Java实现多线程的四种方法](#)

[线程的5种状态, sleep、wait、notify](#)

[线程池种类](#)

什么是线程安全？

当多个线程访问某个类时，这个类始终能表现出正确的行为，那么就称这个类是线程安全的。

什么是竞态条件？

在多线程环境下，由于不恰当的执行顺序而出现不正确的结果。换句话说，就是正确的结果取决于运气。这种情况就叫竞态条件，出现了竞态条件，就代表了线程不安全。

Java中锁的分类

[公平锁/非公平锁](#)

- 公平锁是指多个线程按照申请锁的顺序来获取锁。如果休眠队列中有线程了，则新进入竞争的线程一定要在休眠队列上排队。
- 非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿

现象。新进入的线程是无视休眠队列直接抢占锁的。因此占有锁的线程放弃锁后，唤醒线程需要时间，此时被唤醒的线程就会与新进入的线程争锁。

- 对于Java ReentrantLock而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。
- 对于Synchronized [底层实现原理](#)而言，也是一种非公平锁，不能改变。、

可重入锁

- 可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。
- Synchronized和Java ReentrantLock都是可重入锁。可重入锁可一定程度避免死锁。如下：

```
synchronized void setA() throws Exception{
    Thread.sleep(1000);
    setB();
}

synchronized void setB() throws Exception{
    Thread.sleep(1000);
}
```

上面的代码就是一个可重入锁的一个特点，如果不是可重入锁的话，setB可能不会被当前线程执行，可能造成死锁。

独享锁/共享锁

- 独享锁是指该锁一次只能被一个线程所持有。
- 共享锁是指该锁可被多个线程所持有。
- 对于Lock的另一个实现类ReadWriteLock，其读锁是共享锁，其写锁是独享锁。
- Synchronized和Java ReentrantLock都是独享锁

自旋锁

锁的等待者会原地忙等，不停的询问，直到获得锁。采用让当前线程不停地在循环体内执行实现，当循环的条件被其他线程改变时才能进入临界区。

如何实现可重入锁(ReentrantLock)？

为每个锁关联一个获取计数值和一个所有者线程。当计数值为0时，这个锁就被认为是没有被任何线程持有。当线程请求一个未被持有的锁时，JVM将记下锁的所有者，并且将获取计数值置为1。如果同一个线程再次获取这个锁，计数值将递增，而当线程退出同步代码块时，计数器会相应地递减。当计数值为0时，这个锁将被释放。

可重入锁(ReentrantLock)的实现原理

- 和上面是一样的，细节是ReentrantLock有公平锁模型和非公平锁模型。
- 对于公平锁，如果休眠队列中有线程了，则新进入竞争的线程一定要在休眠队列上排队。
- 对于非公平锁，新进入的线程是无视休眠队列直接抢占锁的。因此占有锁的线程放弃锁后，唤醒线程需要时间，此时被唤醒的线程就会与新进入的线程

争锁。

Lock接口类

<https://www.cnblogs.com/baizhanshi/p/6419268.html>

ReentrantLock是Lock的实现类，也是Lock唯一的实现类

Lock和synchronized的区别

Lock提供了比synchronized更多的功能：

1. Lock可以让等待线程只等待一定的时间或者响应中断，Synchronized则是无限等下去。
2. Lock可以让多个线程只是进行读操作的时候共享锁，Synchronized则是一个线程读操作时，其他线程只能等待。
3. Lock可以知道线程有没有成功获取到锁，Synchronized则不行。
4. 但是Lock必须用户手动写代码释放锁，如果没有主动释放锁，就有可能导致出现死锁现象，因此使用Lock时需要在finally块中释放锁。而synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生

什么时候用synchronized，什么时候用Lock？

在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。

为什么？看上面Lock比synchronized更多的功能。

Lock中声明的方法

1. Lock(): 是平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。由于在前面讲到如果采用Lock，必须主动去释放锁，并且在发生异常时，不会自动释放锁。因此一般来说，使用Lock必须在try{} catch{}块中进行，并且将释放锁的操作放在finally块中进行，以保证锁一定被释放，防止死锁的发生。通常使用Lock来进行同步的话，是以下面这种形式去使用的：

```
Lock lock = ...;
lock.lock();
try{
    //处理任务
} catch(Exception ex){

} finally{
    lock.unlock(); //释放锁
}
```

2. tryLock(): 方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就是说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。
3. tryLock(long time, TimeUnit unit): 方法和tryLock()方法是类似的，只不过

区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

所以，一般情况下通过tryLock来获取锁时是这样使用的：

```
Lock lock = ...;
if(lock.tryLock()) {
    try{
        //处理任务
    }catch(Exception ex){
        //如果不能获取锁，则直接做其他事情
    }
}
```

4. `lockInterruptibly()`: 方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就是说，当两个线程同时通过`lock.lockInterruptibly()`想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用`threadB.interrupt()`方法能够中断线程B的等待过程。因此`lockInterruptibly()`一般的使用形式如下：

```
public void method() throws InterruptedException {
    lock.lockInterruptibly();
    try {
        //.....
    }
    finally {
        lock.unlock();
    }
}
```

Synchronized底层实现原理

对象的同步Synchronized的底层是通过monitor来完成

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行`monitorenter`指令时尝试获取monitor的所有权，过程如下：

1. 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
2. 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1。
3. 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的

进入数为0，再重新尝试获取monitor的所有权。

释放锁则是通过**monitorexit指令**，执行monitorexit的线程必须是objectref所对应的monitor的所有者，指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

方法的synchronized同步

相对于普通方法，其常量池中多了**ACC_SYNCHRONIZED标示符**。JVM就是根据该标示符来实现方法的同步的：当方法调用时，调用指令将会检查方法的ACC_SYNCHRONIZED访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。

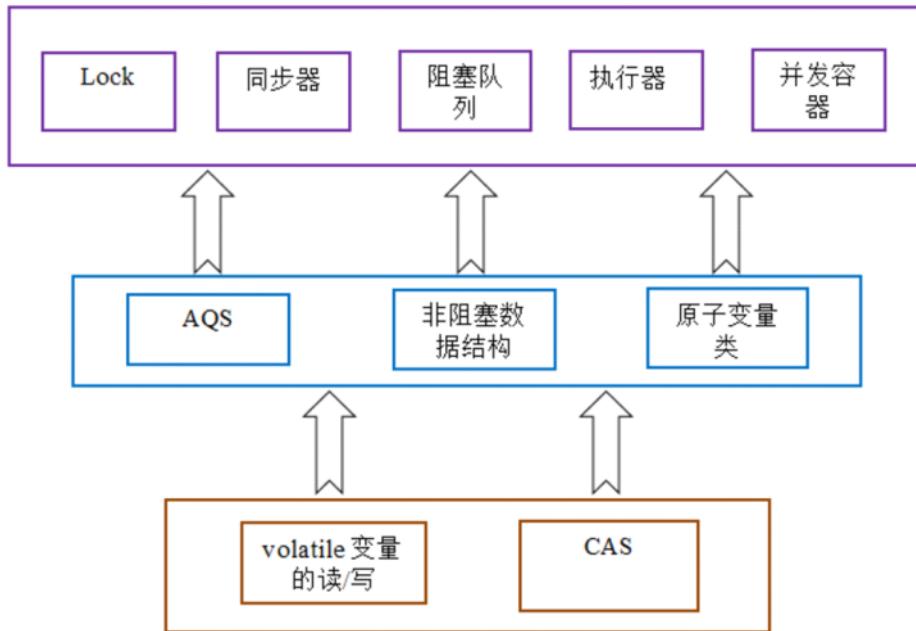
ReentrantLock (Lock接口类) 底层实现原理

整体来看Lock主要是通过两个东西来实现的分别是CAS和ASQ，如下：

- AQS维护了一个volatile int state（代表共享资源）和一个FIFO线程等待队列（多线程争用资源被阻塞时会进入此队列）。
- 对于刚来竞争的线程首先会通过CAS设置状态，如果设置成功那么直接获取锁，执行临界区的代码；
- 反之如果已经存在Running线程，那么CAS肯定会失败，则新的竞争线程会通过CAS的方式被追加到队尾。

以**ReentrantLock**为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多少次，这样才能保证state是能回到零态的。

Java中concurrent包的实现



1. volatile变量和CAS机制是concurrent包实现的基石
2. AQS、非阻塞数据结构和原子变量类都是通过volatile和CAS实现的
3. 实现的模式如下：
 - a. 首先，声明共享变量为volatile；
 - b. 然后，使用CAS的原子条件更新来实现线程之间的同步；
 - c. 同时，配合以volatile的读/写和CAS所具有的volatile读和写的内存语义来实现线程之间的通信。

原子类AtomicInteger的实现原理

volatile和CAS机制，类似的AtomicBoolean和AtomicLong是一样的

AtomicInteger如何实现原子操作的？源代码如下：

变量声明为volatile类型

```
private volatile int value;
```

加1方法，JDK1.7

```
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

加1方法，JDK1.8(调用了本地系统的fetch-and-add方法，性能更好)

```
public final int incrementAndGet() {  
    return unsafe.getAndAddInt(this, valueOffset, 1) + 1;  
}
```

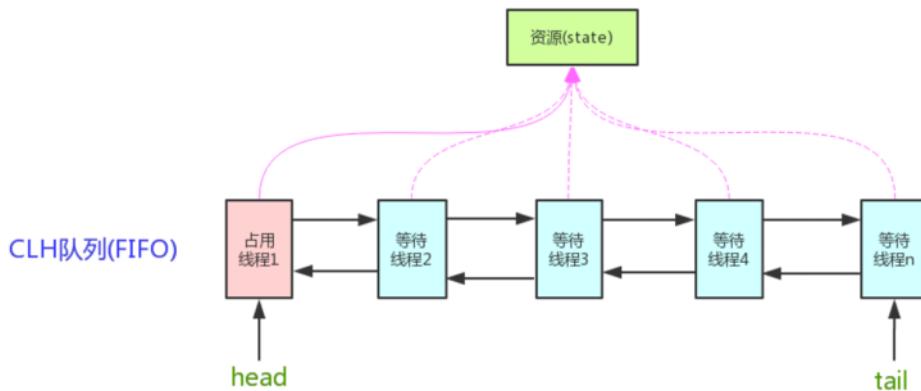
我们还是看JDK1.7的源码，这个是+1操作，逻辑是：

1. 先获取当前的value值
2. 对value加一
3. 第三步是关键步骤，调用**compareAndSet**方法来进行原子更新操作，这个方法的语义是：

先检查当前value是否等于current，如果相等，则意味着value没被其他线程修改过，更新并返回true。如果不相等，compareAndSet则会返回false，然后循环继续尝试更新。

AQS (AbstractQueuedSynchronizer)

AQS (AbstractQueuedSynchronizer)，抽象的队列式的同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch....。



AQS有以下几种方法：

- isHeldExclusively(): 该线程是否正在独占资源。只有用到condition才需要去实现它。
- tryAcquire(int): 独占方式。尝试获取资源，成功则返回true，失败则返回false。
- tryRelease(int): 独占方式。尝试释放资源，成功则返回true，失败则返回false。

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该

锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多少次，这样才能保证state是能回到零态的。

上面是AQS定义的资源独占方式，其实还有资源共享方式，采用以下两种方法：

- tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
- tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

CAS (Compare and swap)

- CAS (Compare and swap) 比较和替换是设计并发算法时用到的一种技术。简单来说，**比较和替换是使用一个期望值和一个变量的当前值进行比较，如果当前变量的值与我们期望的值相等，就使用一个新值替换当前变量的值。**
- 现在CPU内部已经执行原子的CAS操作，Java5+中内置的CAS特性可以让你利用底层的你的程序所运行机器的CPU的CAS特性，这会使代码运行更快。
- Java5以来，你可以使用java.util.concurrent.atomic包中的一些原子类来使用CPU中的这些功能

CAS示例：

```
public static class MyLock {  
    private AtomicBoolean locked = new AtomicBoolean(false);  
  
    public boolean lock() {  
        return locked.compareAndSet(false, true);  
    }  
}
```

上面是一个使用AtomicBoolean类实现lock()方法的例子。

locked变量不再是boolean类型而是AtomicBoolean。这个类中有一个compareAndSet()方法，它使用一个期望值和AtomicBoolean实例的值比较，若两者相等，则使用一个新值替换原来的值。在这个例子中，它比较locked的值和false，如果locked的值为false，则把修改为true。即compareAndSet()返回true，如果值被替换了，返回false。

CAS用于同步（乐观锁的机制就是CAS）

- 通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行多步计算来获得新值 B，然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改，则 CAS 操作成功。
- 类似于 CAS 的指令允许算法执行读-修改-写操作，而无需害怕其他线程同时修改变量，因为如果其他线程修改变量，那么 CAS 会检测它（并失败），算

法可以对该操作重新计算。

CAS存在的问题

有三个，ABA问题，循环时间长开销大和只能保证一个共享变量的原子操作。

1. ABA问题。

- a. 因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A - B - A 就会变成1A-2B - 3A。
- b. 从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。
- c. 关于ABA问题参考文档: <http://blog.hesey.net/2011/09/resolve-aba-by-atomicstampedreference.html>

2. 循环时间长开销大。

- a. 自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。
- b. 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline），使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

3. 只能保证一个共享变量的原子操作。

- a. 当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性。
- b. 这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量 $i = 2, j = a$ ，合并一下 $ij = 2a$ ，然后用CAS来操作 ij 。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

死锁分析

死锁分析

```
void transfer(Account from, Account to, int amount) {  
  
    from.setAmount(from.getAmount() - amount);  
    to.setAmount(to.getAmount() + amount);  
  
}
```

1、比如这是一个银行转账代码，单线程下没问题，多线程下明显有安全隐患

2、加synchronized如下

```
void transfer(Account from, Account to, int amount) {  
    synchronized (from) {  
        synchronized (to) {  
            from.setAmount(from.getAmount() - amount);  
            to.setAmount(to.getAmount() + amount);  
        }  
    }  
}
```

针对对象加锁，好处是转出者A进来后，另一个转出者B也能进来。

3、但这样可能会造成死锁！！！要记住：在任何地方都可以线程切换，甚至在一
句语句中间。要考虑对自己最不利的情况

4、对我们不利的情况如下

synchronized(from) → 别的线程在等待from

synchronized(to) → 别的线程已经锁住了to

可能死锁：transfer(a, b, 100) 和 transfer(b, a, 100) 同时进行

a向b转账，a拿到了a锁，还需要b锁才能进行。同时b向a转账，b拿了b锁，还需要
a锁才能进行。即a->b缺b锁，b->a缺a锁。这种情况下就死锁了。

死锁必须同时满足的条件

1、互斥等待，即必须有锁

2、hold and wait，即拿着一个锁还在等另一个锁

3、循环等待，即A对象拿了A锁等B锁，而B对象拿了B锁等A锁

4、无法剥夺的等待，即没有超时自动放弃锁这一说（synchronized关键字就是会
无限等待，没有超时自动放弃锁）

死锁防止

针对上面4个条件，破除任何一个都可以，一般针对条件2，3来做

死锁防止

- ◆ 破除互斥等待 → 一般无法破除
- ◆ 破除hold and wait → 一次性获取所有资源
- ◆ 破除循环等待 → 按顺序获取资源
- ◆ 破除无法剥夺的等待 → 加入超时

- 1、比如针对最开始的例题，按顺序获取资源，我们每次锁Account数值较小的那个。比如a向b转账和b向a转账同时发生，我们先锁a和b中数值小的。坏处是实际业务中未必能比较大小
- 2、同时拿两个锁，但大部分系统不支持同时拿2个锁。因此我们只能先拿from锁，再去拿to锁，如果很短时间内拿不到to锁，则from锁放掉。过一会再尝试拿from锁和to锁。坏处是多久尝试呢？尝试几次呢？

Spring的线程安全

- 1、Spring MVC (Springboot) 开发的web项目，**默认的Controller, Service, Dao组件的作用域都是单例模式，无状态的，因此是线程安全的**
- 2、无状态的Bean适合用不变模式，技术就是单例模式，这样可以共享实例，提高性能。有状态的Bean，多线程环境下不安全，那么适合用Prototype原型模式。Prototype: 每次对bean的请求都会创建一个新的bean实例。
- 3、默认情况下，从Spring bean工厂所取得的实例为singleton (scope属性为singleton)，容器只存在一个共享的bean实例。
- 4、理解了两者的关系，那么scope选择的原则就很容易了：有状态的bean都使用prototype作用域，而对无状态的bean则应该使用singleton作用域。

Servlet的线程安全

- 1、ServletContext、HttpSession是线程安全的；ServletRequest是非线程安全的
- 2、Servlet是否线程安全是由它的实现来决定的，如果它内部的属性或方法会被多个线程改变，它就是线程不安全的，反之，就是线程安全的。
- 3、Spring是一种线程安全的Servlet实现

Thread中的join()

主线程创建并启动子线程，如果子线程中要进行大量的耗时运算，主线程往往将在子线程运行结束前结束。如果主线程想等待子线程执行完成后再结束（如，子线程处理一个数据，主线程需要取到这个值），则需要用到join()。即**子线程名称.join()**
作用是：等待线程对象销毁

```

Thread thread1 = tester.createThread1();
Thread thread2 = tester.createThread2();

thread1.start();
thread2.start();

thread1.join();
thread2.join();

System.out.println(String.format(
    "(%d, %d)", tester.x_read, tester.y_read));

```

如上，这是写在main函数里的。主线程会在thread1和thread2运行完毕后才调用后面的代码

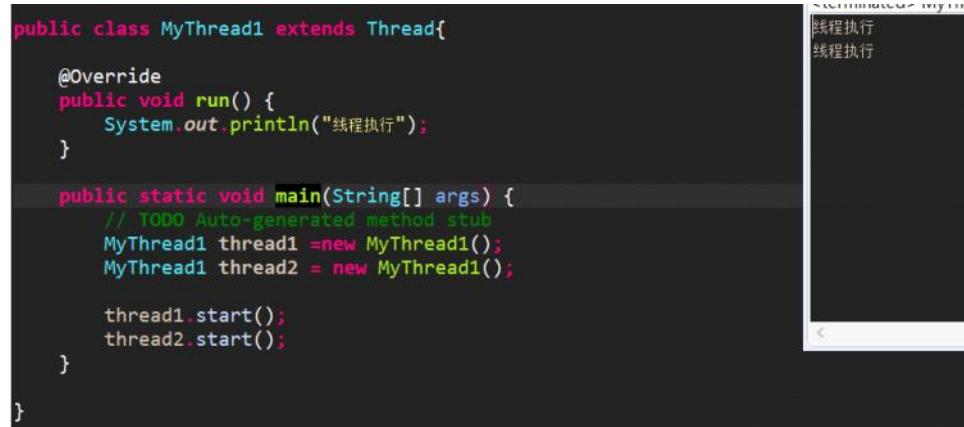
为什么new Thread(new Runnable)的时候要传入Runnable?

- 1、Thread是单继承，Runnable可以多实现，Callable多实现可以返回值Future。
- 2、Thread调用start()方法，则会通过JVM找到run()方法。但是在使用Runnable定义的子类中没有start()方法，只有Thread类中才有。
- 3、因此通过Thread类来启动Runnable实现的多线程

Java实现多线程的四种方法

1、继承Thread类创建线程

- (1) Thread类内部implements了Runnable接口，因此可以复写run()方法，执行start()后会自动调用run方法



The screenshot shows a Java code editor with the following content:

```

public class MyThread1 extends Thread{

    @Override
    public void run() {
        System.out.println("线程执行");
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MyThread1 thread1 = new MyThread1();
        MyThread1 thread2 = new MyThread1();

        thread1.start();
        thread2.start();
    }
}

```

To the right of the code editor is a terminal window showing the output of the program:

```

线程执行
线程执行

```

2、实现Runnable接口创建线程

- (1) 如果自己的类已经extends另一个类，就无法直接继承Thread类，此时可以实现一个Runnable接口
- (2) 为了启动MyThread2，需要首先实例化一个Thread，并传入自己的MyThread2实例
- (3) Thread类start后会调用传入的Runnable的run方法

```
public class MyThread2 extends OtherClass implements Runnable{  
  
    @Override  
    public void run() {  
        System.out.println("线程启动");  
    }  
  
    public static void main(String[] args) {  
        // 必须创建Thread类并传入我们自己实现了Runnable接口的类  
        Thread thread1 = new Thread(new MyThread2());  
        Thread thread2 = new Thread(new MyThread2());  
  
        thread1.start();  
        thread2.start();  
    }  

```

线程启动
线程启动

3、实现Callable接口通过FutureTask包装器来创建线程

- (1) Callable的call()方法类似于Runnable接口中run()方法，都定义任务要完成的工作
- (2) 使用Callable的好处是，call()方法是有返回值的运行Callable任务可以拿到一个Future对象，表示异步计算的结果，通过Future对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果
- (3) 通过futureTask.get()可以得到call()的运行结果

```
* 实现Callable接口通过FutureTask包装器来创建线程  
* @author Jinsong  
* @email 188949420@qq.com  
* @date 2018年3月4日  
  
*/  
//注意类名后面有<V>泛型，这里我们自己设置泛型内容为Integer，当然也可以设置其他的  
public class MyThread3 implements Callable<Integer>{  
  
    @Override  
    public Integer call() throws Exception {  
  
        int sum = 0;  
        for(int i=0;i<100;i++)  
            sum += i;  
        System.out.println("线程启动，sum结果是：" + sum);  
        return sum;  
    }  
  
    public static void main(String[] args) throws InterruptedException, ExecutionException  
        // TODO Auto-generated method stub  
        MyThread3 callable = new MyThread3();  
        FutureTask<Integer> futureTask1 = new FutureTask<>(callable);  
        FutureTask<Integer> futureTask2 = new FutureTask<>(callable);  
  
        //注意！！！！通过这种方式创建的线程，即使我们创建2个，也只有一个会运行  
        //因此Callable, FutureTask基本都是与线程池ExecutorService来结合使用  
        Thread thread1 = new Thread(futureTask1);  
        Thread thread2 = new Thread(futureTask2);  
  
        thread1.start();  
        thread2.start();  
  
        //通过futureTask.get()可以得到call()的运行结果  
        System.out.println(futureTask1.get());  
        System.out.println(futureTask2.get());  

```

<terminated> MyThread3 [Java]
线程启动，sum结果是：4950
线程启动，sum结果是：4950
4950
4950

```
//通过futureTask.get()可以得到call()的运行结果  
System.out.println(futureTask.get());
```

4、使用ExecutorService、Callable、Future实现在有返回结果的线程

(1)有返回值的任务必须实现Callable接口。类似的，无返回值的任务必须实现Runnable接口。

(2) 执行 Callable 任务后，可以获取一个 Future 的对象，在该对象上调用 get 就可以获取到 Callable 任务返回的 Object 了。

(3) get方法是阻塞的，即：线程无返回结果，get方法会一直等待

```
12 * 使用ExecutorService< Callable< Future< ? extends Object > >>实现有返回结果的线程
13 *
14 * @author Jinsong
15 * @email 188949420@qq.com
16 * @date 2018年3月4日
17 *
18 */
19 public class MyThread4 implements Callable< Integer > {
20
21     private int name;
22
23
24
25     public MyThread4(int name) {
26         super();
27         this.name = name;
28     }
29
30     @Override
31     public Integer call() throws Exception {
32         int sum = 0;
33         for (int i = 0; i < 100; i++)
34             sum += i;
35         System.out.println("Callable任务" + name + "启动, sum结果是: " + sum);
36         System.out.println("当前处理线程名称是: " + Thread.currentThread().getName());
37         return sum;
38     }
39
40     public static void main(String[] args) throws InterruptedException, ExecutionException {
41         // 创建线程池
42         ExecutorService pool = Executors.newFixedThreadPool(3);
43
44         // 创建多个有返回值的任务
45         List< Future> list = new ArrayList< Future>();
63
64         for (int i = 0; i < 3; i++) {
65             Future< Integer > future = pool.submit(new MyThread4(i));
66             list.add(future);
67         }
68
69         // 打印所有线程的返回值
70         for (Future< Integer > future : list) {
71             System.out.println("线程" + future.get().toString());
72         }
73     }
74 }
```

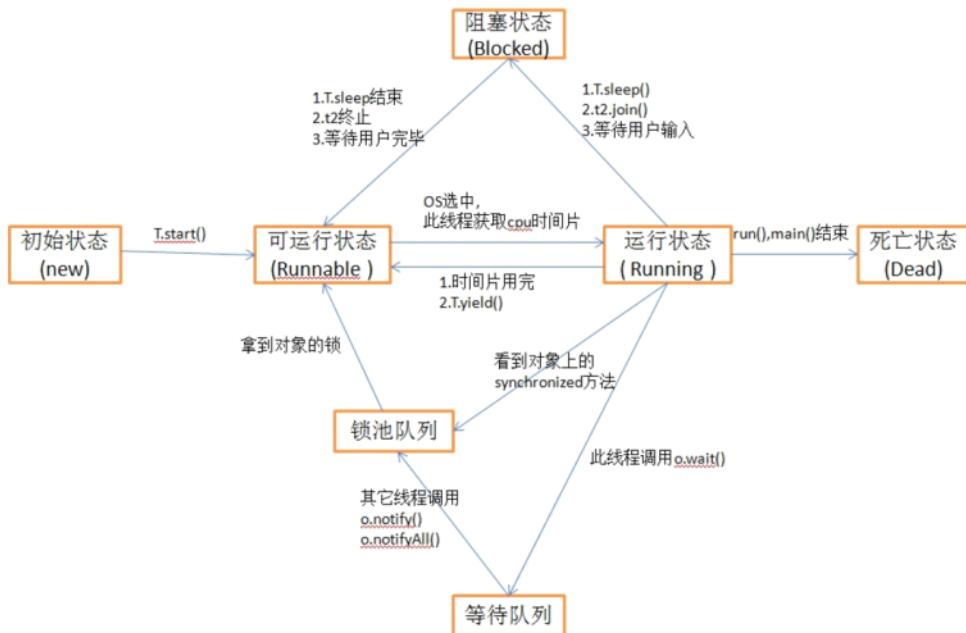
```
for (int i = 0; i < 10; i++) {
    MyThread4 callable = new MyThread4(i);
    Future result = pool.submit(callable);
    list.add(result);
}

// 关闭线程池
pool.shutdown();

// 获取所有并发任务的运行结果
for (Future f : list) {
    // 从Future对象上获取任务的返回值，并输出到控制台
    System.out.println(">>>" + f.get().toString());
}
```

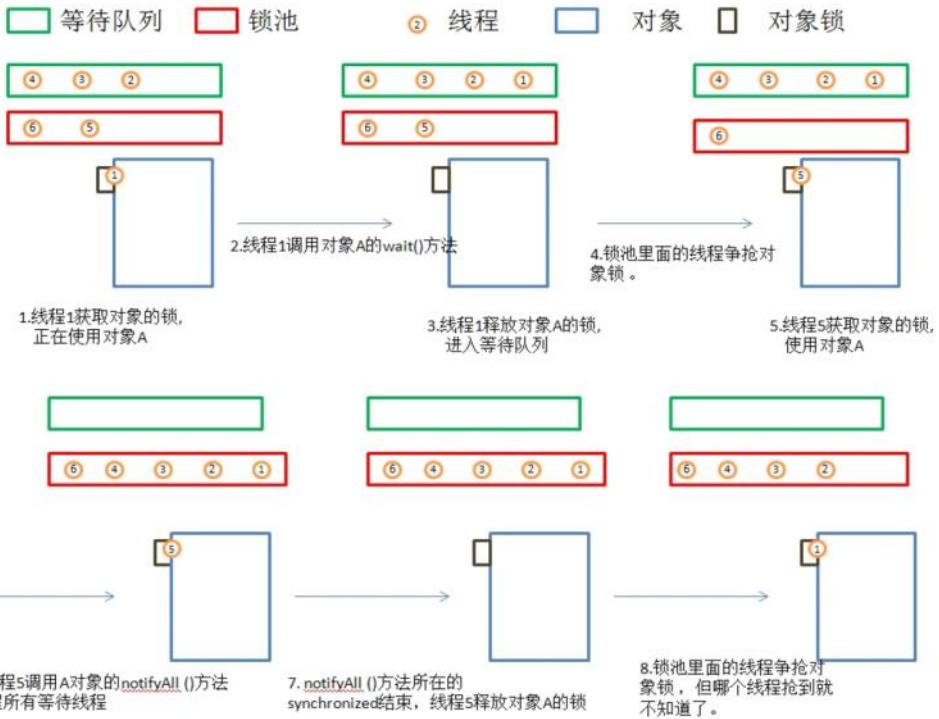
Java线程的5种状态

Java中的线程的生命周期大体可分为5种状态。



1. **新建(NEW)**: 新创建了一个线程对象。
2. **可运行(RUNNABLE)**: 线程对象创建后，其他线程(比如main线程) 调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权。
3. **运行(RUNNING)**: 可运行状态(runnable)的线程获得了cpu 时间片(timeslice) ，执行程序代码。
4. **阻塞(BLOCKED)**: 阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：
 - a. **等待阻塞**: 运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。
 - b. **同步阻塞**: 运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。
 - c. **其他阻塞**: 运行(running)的线程执行Thread.sleep(long ms)或运行在当前线程里的其它线程调用了join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。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，线程重新转入可运行(runnable)状态。
5. **死亡(DEAD)**: 线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

一个线程获取锁、用完锁的例子



几个方法的比较

1. **Thread.sleep(long millis)**, 一定是当前线程调用此方法，当前线程进入阻塞，但不释放对象锁，millis后线程自动苏醒进入可运行状态。作用：给其它线程执行机会的最佳方式。
2. **Thread.yield()**, 一定是当前线程调用此方法，当前线程放弃获取的cpu时间片，由运行状态变会可运行状态，让OS再次选择线程。作用：让相同优先级的线程轮流执行，但并不保证一定会轮流执行。实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。Thread.yield()不会导致阻塞。
3. **t.join()/t.join(long millis)**, 当前线程里调用其它线程1的join方法，当前线程阻塞，但不释放对象锁，直到线程1执行完毕或者millis时间到，当前线程进入可运行状态。
4. **obj.wait()**, 当前线程调用对象的wait()方法，当前线程释放对象锁，进入等待队列。依靠notify()/notifyAll()唤醒或者wait(long timeout)timeout时间到自动唤醒。
5. **obj.notify()**唤醒在此对象监视器上等待的单个线程，选择是任意性的。**notifyAll()**唤醒在此对象监视器上等待的所有线程。

线程池

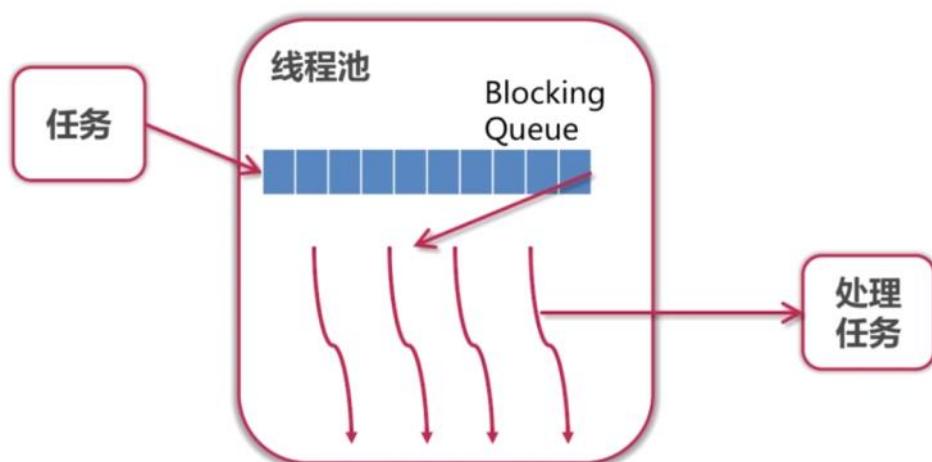
2018年3月3日 12:56

目录

[线程池 实例](#)
[线程池 种类](#)
[线程池 参数](#)
[线程池 缓冲队列种类](#)
[线程池 线程数量](#)
[BlockingQueue规则](#)
[Java实现多线程的四种方法](#)

线程池

- ◆ 创建线程开销大
- ◆ 线程池：预先建立好线程，等待任务派发



- 1、下面的红线是线程
- 2、先进先出的队列里有任务进来后，线程就去处理。

线程池的实例

<https://coding.imooc.com/lesson/132.html#mid=6610>

```

public class ExecutorTester {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        for (int i = 0; i < 10; i++) {
            executor.submit(new CodingTask(i));
        }
        System.out.println("10 tasks dispatched successfully.");
    }
}

```

1. 用`Executors`可以new很多种不同参数的线程池
2. `submit`方法是把任务放进线程池的队列
3. 线程池的任务都做完后，即使main函数运行完了，线程池并不会自己关闭，线程也并不会自己销毁，需要我们手动销毁
4. 手动关闭线程池，直接在最后面加调用`shutdown()`，这个函数会在线程池处理完所有队列里的任务后自动关闭线程池

```

public static void main(String[] args)
    throws InterruptedException, ExecutionException {
    ExecutorService executor = Executors.newFixedThreadPool(3);
    List<Future<?>> taskResults = new LinkedList<>();
    for (int i = 0; i < 10; i++) {
        taskResults.add(executor.submit(new CodingTask(i)));
    }
    System.out.println("10 tasks dispatched successfully.");

    for (Future<?> taskResult : taskResults) {
        taskResult.get();
    }
    System.out.println("All tasks finished.");
    executor.shutdown();
}

```

5. `submit`会返回`Future<?>`类型的结果，根据这个结果可以知道线程是否处理完任务等等细节的东西
6. 泛型里是问号`<?>`指任意类型都可以，不会报错

线程池种类

可以看到，每种线程池都已经设定好了使用哪种队列

- `newSingleThreadExecutor`

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
            0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>()));
}
```

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

- **newFixedThreadPool**

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());
}
```

创建固定大小的线程池。可以看到，corePoolSize和maximumPoolSize的大小是一样的，采用了LinkedBlockingQueue。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

- **newCachedThreadPool**

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
        60L, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>());
}
```

创建一个可缓存的无界线程池，可以自动进行线程回收，可以看到corePoolSize=0，maximumPoolSize=最大。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

- **newScheduledThreadPool**

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
        new DelayedWorkQueue());
}
```

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。这里DelayedWorkQueue()实现了BlockingQueue，是一个无界阻塞队列，只有在延迟期满时才能从中提取元素。该队列的头部是延迟期满后保存时间最长的Delayed元素。如果延迟都还没有期满，则队列没有头部，并且poll将返回null。

线程池参数

1. **corePoolSize** - 池中所保存的线程数，包括空闲线程。
2. **maximumPoolSize**-池中允许的最大线程数。
3. **keepAliveTime** - 当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长

时间，设置的越长，空闲线程没任务时等待的时间越长。

4. **unit** - `keepAliveTime` 参数的时间单位。
5. **workQueue** - 执行前用于保持任务的队列。此队列仅保持由 `execute` 方法提交的 **Runnable** 任务。[缓冲队列的种类](#)
6. **threadFactory** - 执行程序创建新线程时使用的工厂。
7. **handler** - 由于超出线程范围和队列容量而使执行被阻塞时所使用的处理程序。[有四种策略：](#)
 - a. 减缓新任务的提交速度
 - b. 抛异常
 - c. 直接删除不能执行的任务
 - d. 如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序（如果再次失败，则重复此过程）

BlockingQueue规则

1. 除了 `SynchronizedQueue`，其他都实现了 `BlockingQueue`。
2. 如果运行的线程少于 `corePoolSize`，则 `Executor` 始终首选添加新的线程，而不进行排队。（如果当前运行的线程小于 `corePoolSize`，则任务根本不会添加到 `queue` 中，而是直接找家伙（`thread`）开始运行）
3. 如果运行的线程等于或多于 `corePoolSize`，则 `Executor` 始终首选将请求加入队列，而不添加新的线程。
4. 如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 `maximumPoolSize`，在这种情况下，任务将被拒绝。

缓冲队列的种类

1. **SynchronizedQueue**: 工作队列的[默认选项](#)是 `SynchronousQueue`，队列不保存任务而是将任务直接提交给线程，如果线程不够，则新开线程，因此常要求无界 `maximumPoolSizes` 以避免拒绝新提交的任务。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。
2. **LinkedBlockingQueue**: [无界队列](#)，所有的线程就不会超过 `corePoolSize`，任务存队列里，换句说，永远也不会触发产生新的线程，池子里有多少线程就永远是多少线程。
3. **PriorityBlockingQueue**: [具有优先级的无界队列](#)，它可以让优先级高的任务先得到执行，需要注意的是如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。
4. **ArrayBlockingQueue**: [有界队列](#)，有助于防止资源耗尽，通常配置大队列小池子或者

小队列大池子。使用大队列小池子可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。使用小队列大池子，CPU 使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。

合理配置线程池中线程数量

从任务性质分析：

1. **CPU密集型任务** 配置尽可能少的线程数量，如配置 $N+1$ 个线程的线程池。（ N 是CPU数量）
2. **IO密集型任务** 则由于需要等待IO操作，线程并不是一直在执行任务，则配置尽可能多的线程，如 $2*N$ 。
3. **混合型的任务**，如果可以拆分，则将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。
4. 我们可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。

从优先级分析：

1. 优先级不同的任务可以使用**优先级队列PriorityBlockingQueue**来处理。它可以让优先级高的任务先得到执行。
2. 需要注意的是如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

是否依赖数据库：

1. **依赖数据库的任务，线程数量要设置大。**
2. 因为线程提交SQL后需要等待数据库返回结果，如果等待的时间越长CPU空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用CPU。

建议使用有界队列 **ArrayBlockingQueue**：

有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点，比如几千。防止队列撑满内存，导致系统不可用。

服务器Socket编程

2018年3月3日 13:32

虽然工作中不用自己编写，但一定要编写过或者熟悉Socket编程

BIO和NIO

BIO: 同步阻塞式IO，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端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

NIO: 同步非阻塞式IO，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

普通Socket编程

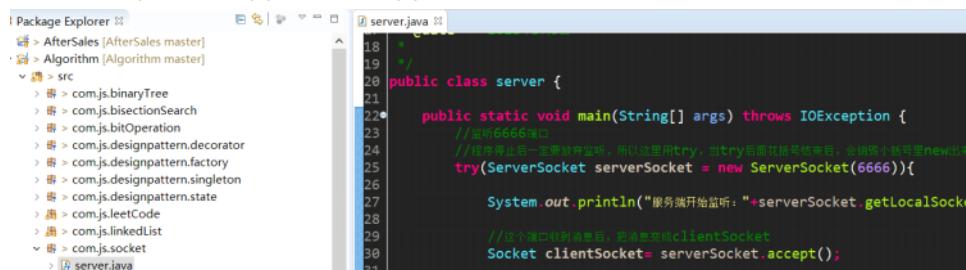
使用线程池并行处理客户请求

使用Java NIO异步处理客户请求

从上到下依次可以提高服务器的处理能力

单客户服务器实例

同时只能响应一个客户请求，客户退出了，服务器就退出了



```
server.java
1.8
1.9
1.10 */
1.11 public class server {
1.12
1.13     /**
1.14      * @param args
1.15     */
1.16    public static void main(String[] args) throws IOException {
1.17        //监听6666端口
1.18        //线程看守后一定要放弃监听，所以这里用try，当try后面代码执行后，会自动小括号里new出的
1.19        try(ServerSocket serverSocket = new ServerSocket(6666)){
1.20
1.21            System.out.println("服务器开始监听：" + serverSocket.getLocalSocketAddress());
1.22
1.23            //这个语句执行完成后，线程将调用clientSocket
1.24            Socket clientSocket= serverSocket.accept();
1.25
1.26            //向客户端写入数据
1.27            OutputStream os = clientSocket.getOutputStream();
1.28            os.write("hello XXX".getBytes());
1.29
1.30            //向客户端读取数据
1.31            InputStream is = clientSocket.getInputStream();
1.32            byte[] buffer = new byte[1024];
1.33            int len = is.read(buffer);
1.34            String str = new String(buffer, 0, len);
1.35            System.out.println(str);
1.36
1.37            //关闭连接
1.38            clientSocket.close();
1.39        }
1.40    }
1.41 }
```

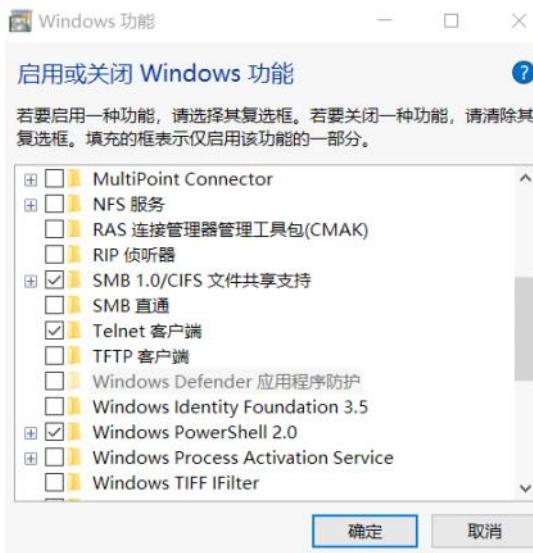
- 1、实现客户端发送xxx，服务端回应hello XXX
- 2、运行这个server.java(Run as JavaApplication就可以，不用任何配置)
- 3、打开命令行第一次输入telnet localhost 6666 会碰见如下



```
Microsoft Windows [版版本本 10.0.16299.
(c) 2017 Microsoft Corporation。保保留留所

C:\Users\18894>telnet localhost 6666
'telnet' 不不是是内内部部或或外外部部命命令令,
或或批批处处理
```

在“启用或关闭Windows功能”窗口勾选Telnet客户端



4、再次输入telnet localhost 6666，进入连接，实现功能

客户端输入hahahaha,服务端返回了Hello hahahaha

客户端输入quit，就退出了

```
cmd
Hello hello
      nihao
Hello nihao
      hahahaha
Hello hahahaha
      quit

遗失对主机的连接。
```

A screenshot of a terminal window titled "cmd". It shows a Telnet session. The client sends "Hello hello", "Hello nihao", "Hello hahahaha", and "quit". The server responds with "nihao", "hahahaha", and nothing for "quit". A message at the bottom says "遗失对主机的连接。" (Lost connection to host).

线程池服务器实例

可以同时响应多个客户请求

```
Package Explorer ThreadPoolServer.java RequestHandler.java
AfterSales [AfterSales master]
Algorithm [Algorithm master]
src
  com.js.binaryTree
  com.js.bisectionSearch
  com.js.bitOperation
  com.js.designpattern.decorator
  com.js.designpattern.factory
  com.js.designpattern.singleton
  com.js.designpattern.state
  com.js.leetCode
  com.js.linkedList
  com.js.socket
    RequestHandler.java
    Server.java
    ThreadPoolServer.java
  com.js.sort
  com.js.stackAndQueue
  com.js.string
JRE System Library [JavaSE-1.8]
blog [blog master]
GuanGongWei [GuanGongWei master]

/*
 * 基于线程池的服务器
 * 实现客户端发送XXX
 * 服务端回应Hello XXX
 * 可以同时响应多个客户请求
 */
public class ThreadPoolServer {
    public static void main(String[] args) throws IOException {
        //建立固定有3个线程的线程池
        ExecutorService executor = Executors.newFixedThreadPool(3);
        try(ServerSocket serverSocket = new ServerSocket(7777)) {
            while(true) {
                Socket socket = serverSocket.accept();
                executor.execute(new RequestHandler(socket));
            }
        }
    }
}
```

A screenshot of the Eclipse IDE interface. On the left is the Package Explorer showing Java projects and files. On the right is the code editor with the file "ThreadPoolServer.java" open. The code implements a thread pool server using the ExecutorService API to handle multiple client requests simultaneously.

```

Telnet localhost
Hello nihao

```



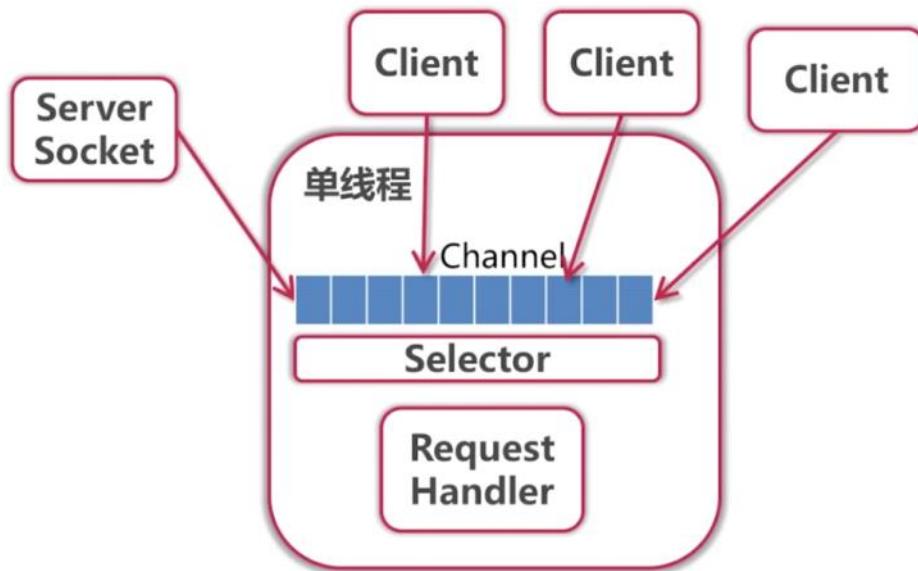
```

Telnet localhost
Hello hhahhh

```

如上能同时响应多个客户同时请求

Java NIO异步服务器实例



这是一个单线程程序，却可以同时服务多个用户

```

10 import java.nio.channels.SocketChannel;
11 import java.util.Iterator;
12 import java.util.Set;
13
14 /**
15 *
16 * Java NIO异步服务器
17 *
18 * NIO就是new io的简称
19 *
20 * 这是一个单线程程序，却可以同时服务多个用户
21 */
22 @author Jinsong
23 @email 188949420@qq.com
24 @date 2018年3月3日
25
26 */
27 public class NioServer {

```

```
C:\> Telnet localhost  
鍊嶅姁鎸ㄥ淇塞箇細Hello j  
k鍊嶅姁鎸ㄥ淇塞箇細Hello k  
C:\> Telnet localhost  
  
奔欸ello l  
k鍊嶅姁鎸ㄥ淇塞箇細H鉢簰锛欵ello h  
鍊嶅姁鎸ㄥ淇塞箇細Hello l  
鍊嶅姁鎸ㄥ淇塞箇細Hello  
s鍊嶅姁  
  
欵ello f  
鍊嶅姁鎸ㄥ淇塞箇細Hello  
r鍊嶅姁  
llo f  
f鍊嶅姁鎸ㄥ淇塞箇細Hello f  
s鍊嶅姁  
鍊嶅姁鎸ㄥ淇塞箇細Hello
```

NIO服务器时基于select的模型，有什么缺点？

- 1、因为是单线程，所以不适用于运算密集型。select模型适合处理IO，如果应用很复杂，应该再开线程，由select模型处理完IO后扔给再开的线程处理具体业务逻辑。
- 2、select模型需要轮询所有fd (channel的代号) 看哪个channel好了，因此比较慢，而且最大只有1024个fd，因此不能处理数量太多例如同时上万个请求。
- 3、不同的系统 (Linux、 Mac、 Windows) 有各自的技术去解决上面select需要轮询的问题，Java Nio根据部署系统的不同自动选择底层实现。

JIT=动态编译+优化

JIT之后，Java并不比C++慢

JIT技术是JVM中最重要的核心模块之一。我的课程里本来没有计划这一篇，但因为不断有朋友问起，Java到底是怎么运行的？既然Hotspot是C++写的，那Java是不是可以说运行在C++之上呢？为了澄清这些概念，我才想起来了加了这样一篇文章，算做番外篇吧。

Just In Time

- Just in time编译，也叫做运行时编译，不同于 C / C++ 语言直接被翻译成机器指令，javac把java的源文件翻译成了class文件，而class文件中全都是Java字节码。那么，JVM在加载了这些class文件以后，针对这些字节码，逐条取出，逐条执行，这种方法就是解释执行。
- 还有一种，就是**把这些Java字节码重新编译优化，生成机器码，让CPU直接执行。这样编出来的代码效率会更高**。通常，我们不必把所有的Java方法都编译成机器码，只需要把调用最频繁，占据CPU时间最长的方法找出来将其编译成机器码。这种调用最频繁的Java方法就是我们常说的热点方法（Hotspot，说不定这个虚拟机的名字就是从这里来的）。
- 这种在运行时按需编译的方式就是Just In Time。

主要技术点

其实JIT的主要技术点，从大的框架上来说，非常简单，就是申请一块既有写权限又有执行权限的内存，然后把你想要编译的Java方法，翻译成机器码，写入到这块内存里。当再需要调用原来的Java方法时，就转向调用这块内存。

JIT方法内联

比如A方法里调用了B方法，方法内联就是把B方法的代码直接写到A方法里，使A\B方法合成一个方法。这就是getter/setter无需优化的原因。

JIT逃逸分析

1、同步消除

如果实际只有一条线程访问对象，则消除对象方法的Synchronized，例外是StringBuffer，因内联父类失败

2、标量替换

如果对象只在方法内使用，则不会在栈上分配对象，而是只创建对象的属性。

比如：

A a = new A(); a.b=1; 变成 int b=1;

解释器，C1和C2

在Hotspot中，解释器是为每一个字节码生成一小段机器码，在执行Java方法的过程中，每次取一条指令，然后就去执行这一个指令所对应的那一段机器码。256条指令，就组成了一个表，在这个表里，每一条指令都对应一段机器码，当执行到某一条指令时，就从这个表里去查这段机器码，并且通过jmp指令去执行这段机器码就行了。

这种方式被称为模板解释器。

模板解释器生成的代码有很多冗余，就像我们上面的第一个例子那样。为了生成更精简的机器码，我们可以引入编译器优化手段，例如全局值编码，死代码消除，标量展开，公共子表达式消除，常量传播等等。这样生成出来的机器码会更加优化。

但是，生成机器码的质量越高，所需要的时间也就越长。JIT线程也是要挤占Java应用线程的资源的。**所以C1是一个折衷，编译时间既不会太长，生成的机器码的指令也不是最优化的，但肯定比解释器的效率要高很多。**

如果一个Java方法调用得足够频繁，那就更值得花大力气去为它生成更优质的机器码，这时就会触发C2编译，**C2是一个运行得更慢，但却能生成更高效代码的编译器。**

由此，我们看到，其实Java的运行，几乎全部都依赖运行时生成的机器码上。所以，对于文章开头的那个问题“Java是运行在C++上的吗？” ，大家应该都有自己的答案了。这个问题无法简单地回答是或者不是，正确答案就是Java的运行依赖模板解释器和JIT编译器。

来自 <<https://zhuanlan.zhihu.com/p/28476709>>

JVM GC

2018年3月3日 15:37

目录

[JVM内存区域](#) [JVM类加载机制](#)

[回收区域](#)

[确定垃圾](#)

[垃圾回收的根](#)

[朴素垃圾回收算法](#)

[分代垃圾回收算法](#)

[JVM如何发起GC、在哪里进行GC](#)

[垃圾收集器](#)

[内存分配与回收策略](#)

[元空间](#)

[垃圾回收调试](#)

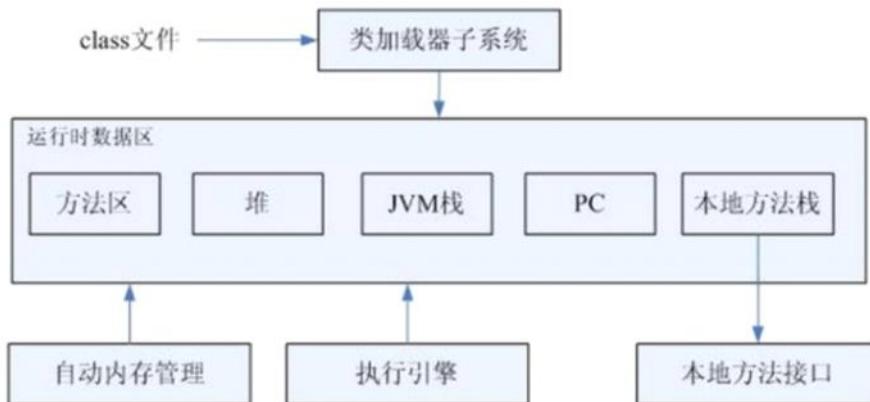
[JVM GC 调优](#)

[类加载的详细过程 双亲委派模型](#)

[Class类文件的结构](#)

JVM组成

JVM由类加载器子系统、运行时数据区、执行引擎以及本地方法接口组成。

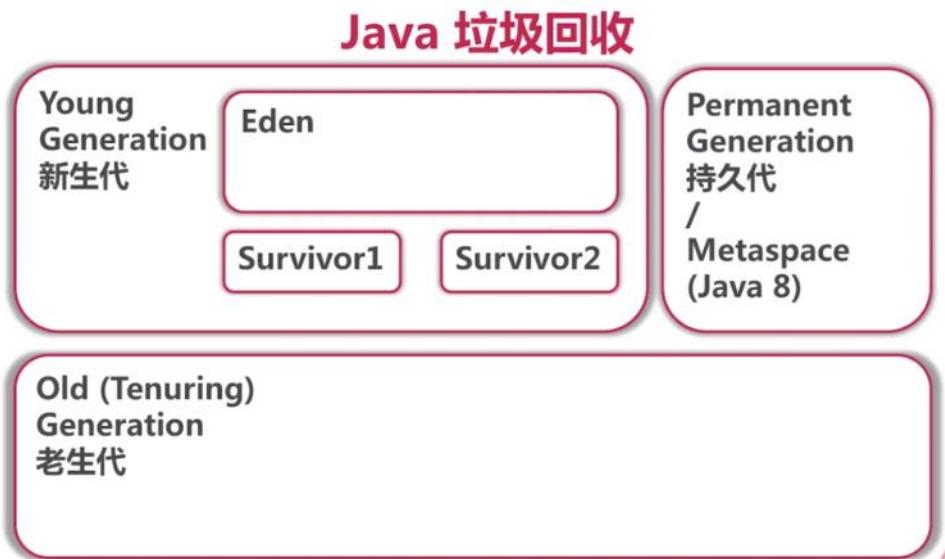


JVM运行原理

Java源文件经编译器，编译成字节码程序，通过JVM将每一条指令翻译成不同平台机器码，通过特定平台运行

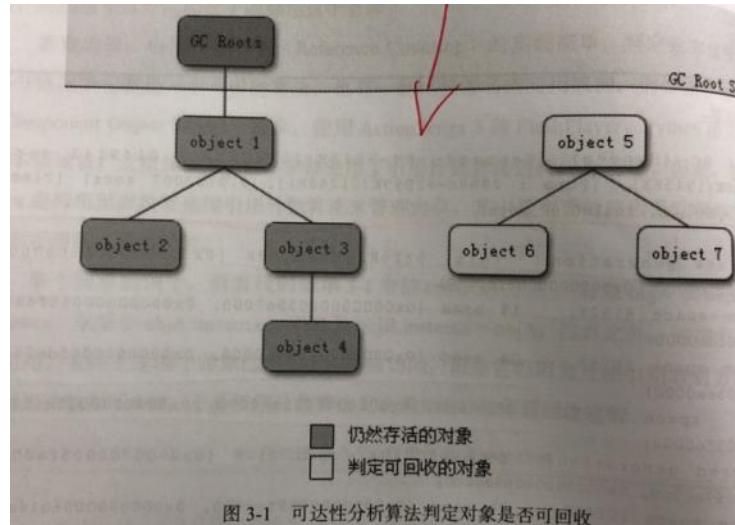
Java垃圾回收区域

- 1、Java垃圾回收只针对堆和方法区的内存。
- 2、程序计数器、虚拟机栈、本地方法栈随线程而生，随线程而灭，因此不用管。



如何确定垃圾？

- 1、采取引用计数是否可行？即统计有多少人引用了，如果0人引用，则判断为垃圾。缺点是：东西很多时，消耗太大；A引用了B，B引用了A，其实A\B都是垃圾，但他们引用永远都是1，不会被回收掉，陷入了死循环。
- 2、从垃圾回收的根出发判断是否可见，这也是JVM采用的可达性分析算法



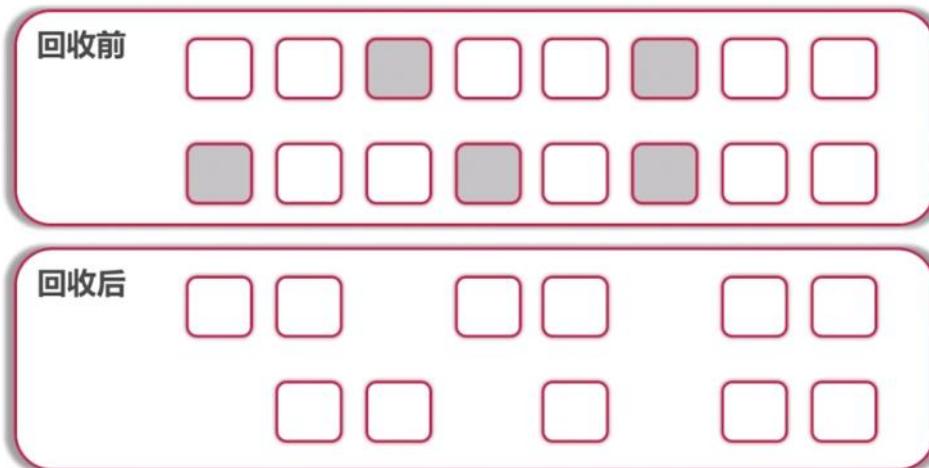
- 3、Object5、6、7就是垃圾

什么是垃圾回收的根？

- 1、就是虚拟机认为一定有用的
- 2、有局部变量（即虚拟机栈中引用的对象）。JVM认为垃圾回收时，暂停的那一刻，所有的局部变量都是有用的
- 3、还有方法区中的静态变量、常量引用的对象。
- 4、本地方法栈中Native方法引用的对象，活动线程，等待中的Monitor（比如synchronized等待的代码段）都是JVM认为的根

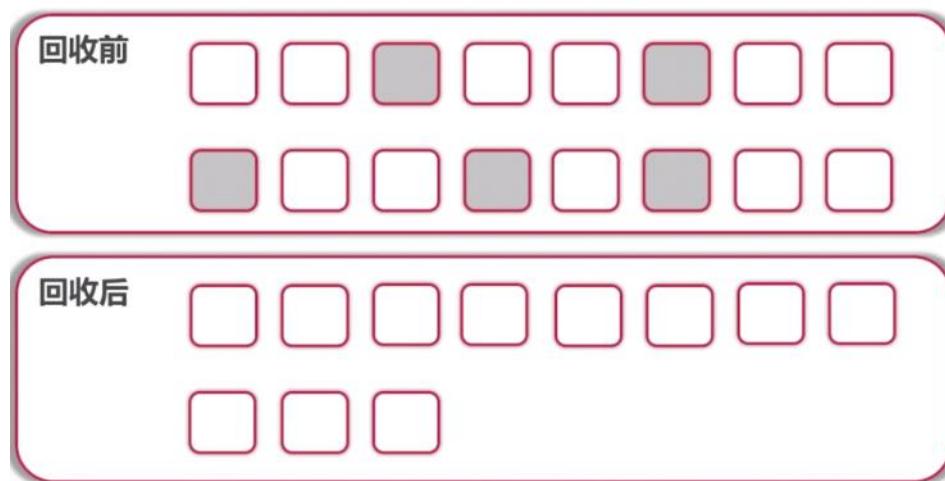
垃圾回收算法

朴素的算法 – Mark and Sweep



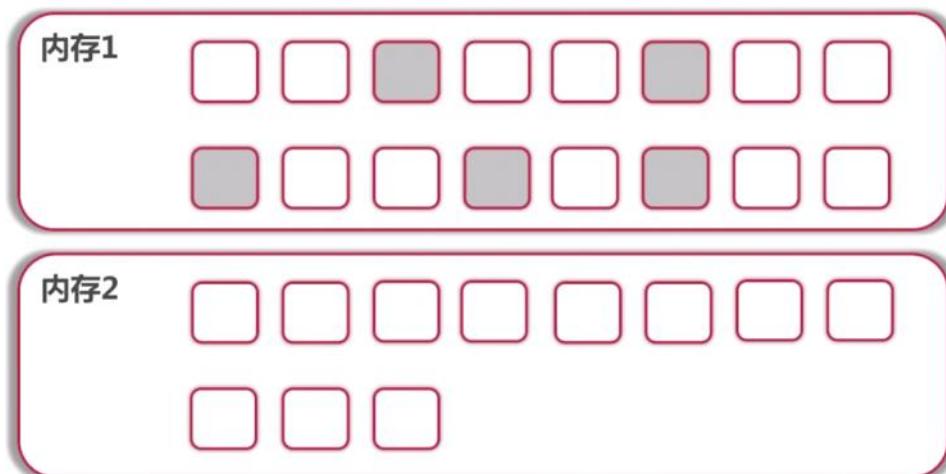
1、**“标记-清除”** 算法，缺点是碎片化很严重，会导致程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前进行另一次GC。

朴素的算法 – Compact



1、**“标记-整理”** 算法。缺点是实际的实现消耗很大，垃圾是一块一块的，要把回收掉空出来的区域往前移动，整块移是移不动的，要切成很多小块移动，想想就消耗很大。

朴素的算法 – Copy

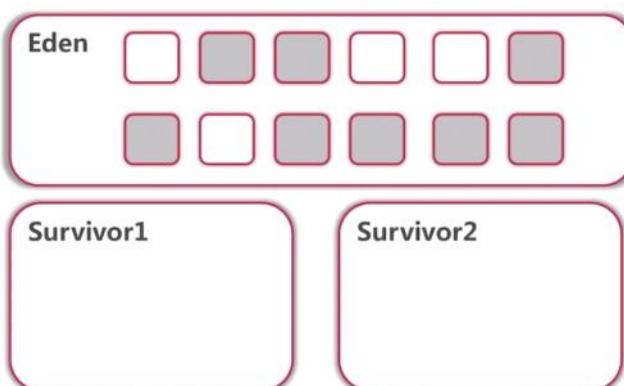


1、 “**拷贝**” 算法，先把内存切成两块。内存1往内存2拷贝，内存1全部扔掉。比 Compact有改进，但缺点是内存每次只能用一半。

JVM实际使用的分代垃圾回收算法，就是把上面的的朴素算法综合一下

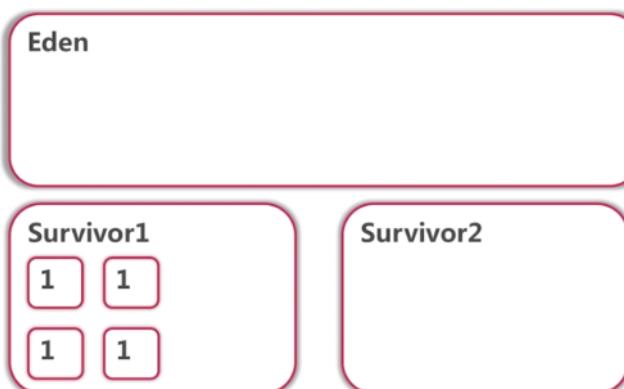
- 1、基础假设：大部分对象只存在很短的时间，这个假设确实也是对的
- 2、基于上面的假设，JVM将内存分为新生代和老生代。大部分只存在很短的时间，就放在新生代。对于少部分存在很长时间的对象，放在老生代去。
- 3、对新生代和老生代采取不同的做法
- 4、新生代经常性进行GC，因此要优化性能，采取类似Copy算法
- 5、**将新生代分为Eden, Survivor1和Survivor2。**

Minor GC



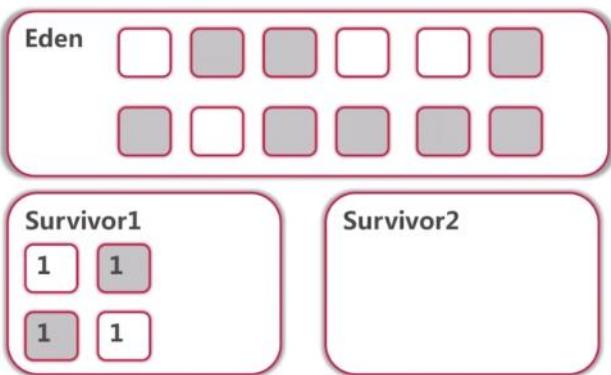
Eden是要回收的区域。1表示存活了1次垃圾回收，2表示存活了2次垃圾回收

Minor GC



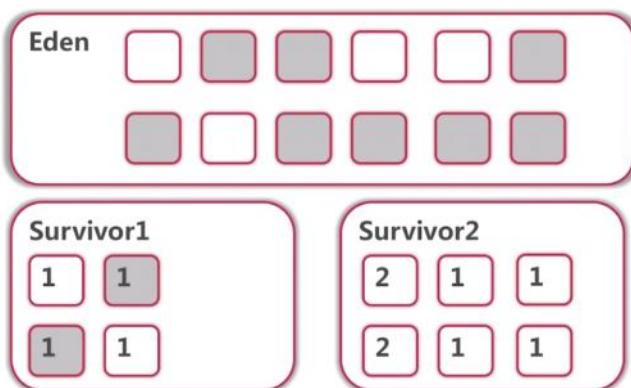
第一次回收，从Eden回收存活下来的对象放入Survivor1，然后清掉Eden。

Minor GC



又产生了垃圾

Minor GC



第二次回收，从Eden和Survivor1存活的对象放入Survivor2

Minor GC



清掉Eden和Survivor1

就这样如此反复！！！

当Survivor不够了，就会把Survivor里面年纪大的，即12345这种数字高的放入老生代

并且JVM会认为你的年纪足够大了，就是长期对象，会直接转入老生代

6、Full GC会对老生代做GC，老生代采取[标记-整理](#)算法 [内存分配与回收策略](#)

JVM如何发起GC、在哪里进行GC

枚举根节点

1、要GC就得枚举根节点

- 2、如果逐一去检查引用，效率很低。因此JVM使用一组称为OopMap的数据结构，直接知道哪些地方存放着对象引用。

安全点

- 可能导致引用关系、或者说OopMap内容变化的指令非常多
- 不可能为每一条指令都生成对应的OopMap
- 因此有了安全点，在安全点才记录OopMap，在安全点才能进行GC
- 例如方法调用、循环跳转、异常跳转等，具有这些功能的指令才会产生安全点

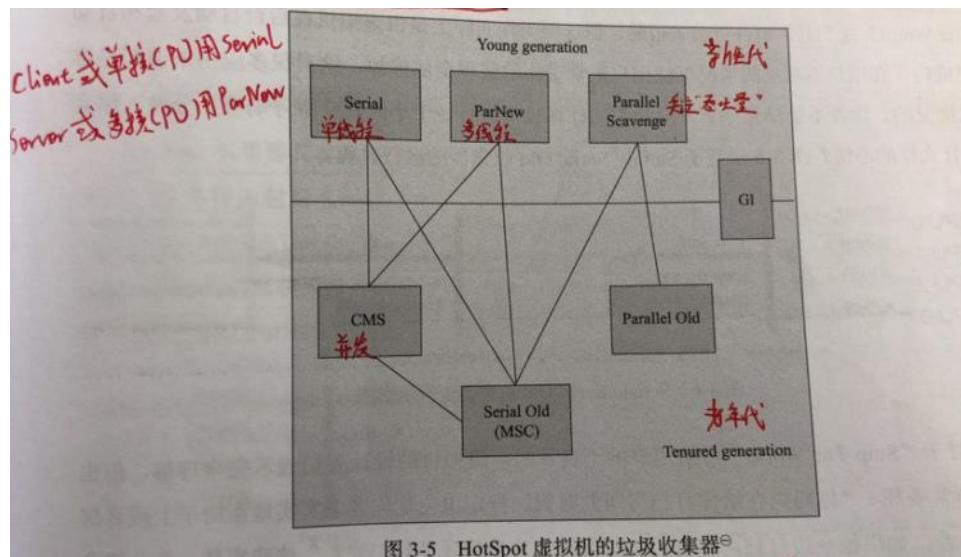
如何让GC发生时线程都跑到安全点

- 采用主动式中断思想
- GC时，不直接对线程操作，而是设置一个中断标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起
- 轮询标志的地方和安全点是重合的

安全区域

- 如果程序不执行时，比如sleep了，岂不是就进不了安全点？
- 因此有了安全区域
- 安全区域指在一段代码中，引用关系不会发生变化，在这个区域内GC都是安全的。
- 线程进入安全区域后，会标志自己进入了。JVM要GC时就不会管这些线程。
- 线程要离开安全区域时，必须检查GC是否完成，如果GC完成了线程就继续执行，否则一直等待直到GC完成。

垃圾收集器



- 图 3-5 HotSpot 虚拟机的垃圾收集器^②
- Serial**，单线程，GC的时候必须暂停其他所有工作线程。JVM Client模式下的默认新生代收集器。
 - ParNew (小米用的这个)**，Serial的多线程版本。JVM Server模式下的默认

新生代收集器。实现了GC线程与工作线程同时工作。**举例就是，你妈打扫卫生的时候，你还可以一边往地上扔纸屑。**

- 3、单CPU环境中，ParNew绝对不会有比Serial更好的效果。
- 4、Serial和ParNew都是与CMS配合工作。
- 5、Parallel Scavenge，关注“吞吐量”。比如JVM总共运行了100分钟，其中GC花了1分钟，那吞吐量就是99%。
- 6、新生代的垃圾收集器都是“复制”算法。
- 7、Serial Old，单线程，使用“标记-整理”算法负责老年代。
- 8、Parallel Old，多线程，“标记-整理”，和Parallel Scavenge同时使用提高“吞吐量”。
- 9、**CMS（小米用的这个）**，并发，唯一的“标记-清除”因此可能导致老年代碎片化严重，无法容纳新生代提升上来的大对象，从而CMS失败，退回到Serial Old算法，导致GC时间过长，可以尝试调大Survivor的空间和调整CMS垃圾收集在老年代占比达到多少时启动来减少问题发生频率，越早启动问题发生频率越低，但是会降低吞吐量，具体得多调整几次找到平衡点；另外，如果**GC频率太快**，说明空间不足，首先可以尝试调大新生代空间和晋升阈值，专注最短GC停顿时间，使网站响应更快，服务不会出现长时间停滞。
- 10、**G1**，多线程，分代收集，“标记-整理”，可预测的停顿。如果追求低停顿，可以尝试G1。

内存分配与回收策略

- 1、对象优先在Eden分配
- 2、老年代GC (Full GC/Major GC) 一般比新生代GC (Minor GC) 慢10倍以上
- 3、**大对象直接进入老年代**，大对象指需要大量连续内存空间的Java对象，比如很长的字符串和数组。可通过参数设置。
- 4、**长期存活的对象将进入老年代**。默认15岁
- 5、**动态对象年龄判定**。如果Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代。
- 6、**空间分配担保**。当出现大量对象Minor GC后仍然存活的情况，需要老年代进行分配担保，让Survivor无法容纳的对象直接进入老年代。

JVM垃圾回收参数，一般不改这个参数

参数配置

- ◆ -XX:NewRatio 老生代/新生代比例，默认2
- ◆ -XX:SurvivorRatio Eden/Survivor比例，默认8
- ◆ -XX:MaxTenuringThreshold 新生代转至老生代阈值，默认15

持久代 (JDK8改进成元空间)

Permanent Generation 持久代

- ◆ 放置ClassLoader读进来的Class，除系统Class外
- ◆ 放置String.intern后的结果
- ◆ 易出现OutOfMemoryError: PermGen Space

如何解决OutOfMemoryError:PermGen Space的问题？

- 1、使用-XX:MaxPermSize调整，调大一点。。。
- 2、Java 1.8帮我们做了改进，取消了持久代，改进成元空间

元空间

PermGen Space vs Metaspace

Java 1.8使用Metaspace，取消PermGen Space

String.intern的结果被放入堆

Metaspace默认不设限制，使用系统内存

- 1、调用String.intern()方法的时候，会将共享池中的字符串与外部的字符串(s)进行比较，如果共享池中有与之相等的字符串，则不会将外部的字符串放到共享池中的，返回的只是共享池中的字符串，如果不同则将外部字符串放入共享池中，并返回其字符串的句柄（引用）--这样做的好处就是能够节约空间
- 2、一个初始时为空的字符串池，它由类 String 私有地维护。当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（该对象由 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添

加到池中，并且返回此 String 对象的引用。它遵循对于任何两个字符串 s 和 t，当且仅当 s.equals(t) 为 true 时，s.intern() == t.intern() 才为 true

谈谈Java垃圾回收机制

- 1、回答垃圾回收在什么时候运行？JVM分配内存失败的时候会运行。还可以手动调用System.gc()，JVM会知道你想垃圾回收了，至于到底是否进行GC由JVM自行判断，一般都是进行的。
- 2、垃圾回收对什么对象进行回收？从垃圾回收的根节点出发，看得见的不回收，看不见的都回收。
- 3、垃圾回收算法对内存划分成了哪些区域？新生代、老生代、Metaspace，新生代又分为了Eden、Survivor1、Survivor2。并回答具体的算法。

垃圾回收调试

Java垃圾回收的调试

获取信息

- ◆ -verbose:gc
- ◆ -XX:+HeapDumpOnOutOfMemoryError
- ◆ -XX:+PrintGCDetails -Xloggc:<GC-log-file-path>
- ◆ Spring Actuator

用Spring的话可以dependency一个Spring Acuator来帮助打印垃圾回收信息

查看信息

- ◆ 官方：visualvm , jmap
- ◆ Eclipse Memory Analyzer (MAT)
- ◆ 在线：gceeasy.io fastthread.io

- 1、垃圾回收信息可视化的可用这些工具
- 2、这里列举的在线工具是最好用的

JVM GC 调优

优化背景

信息流视频推荐服务，从物理机迁移到Ocean上后，观察JVM的YoungGC的

Stop The World(以下简称STW)时间从正常的50ms左右最高时飙升至2s之多。

JVM Options(主要配置):

- Xmx10240M
- Xms10240M
- XX:+UseParNewGC
- XX:+UseConcMarkSweepGC
- Xmn768M
- XX:+CMSParallelRemarkEnabled
- XX:CMSInitiatingOccupancyFraction=50
- XX:+UseCMSInitiatingOccupancyOnly
- XX:+CMSParallelInitialMarkEnabled
- XX:+ExplicitGCIInvokesConcurrent

优化过程概览

1、我们首先分析影响JVM STW时间的因素和Docker资源隔离机制,寻求解决方法

从JVM配置选项可知, YoungGen 使用ParNew GC算法, OldGen使用CMS GC算法.

JVM在启动伊始如果没有相关参数设置,会根据读取到宿主机的CPU逻辑核心数目(即24)设置执行GC的线程数目,

实际观察得知在Docker中YoungGC时线程并发数为18, CMS GC并发数为5

实际上Docker容器只被分配了4个CPU, 过多的GC线程会造成不必要的线程之间的资源竞争和上下文切换损耗。

我们可以通过**设置相应的JVM启动参数**-XX:ParallelGCThreads=4 -
XX:ConcGCThreads=4 解决两者之间的矛盾

设置该参数后, 线上观察YoungGC的STW的时间**从最高飙升到2s下降到250ms**, 并且稳定在250ms左右.

说明根据Docker的资源限制设置相应的JVM启动参数是有效的

2、STW时间虽然下降到250ms, 但是和物理机STW时间在50ms左右差距较大.

我们继续从影响YoungGC的因素着手, 查阅相关资料得知YoungGC中有一个阶段是扫描OldGen的CardTable来判断YoungGen中的对象是否被OldGen的对象引用. JVM中参数ParGCCardsPerStrideChunk是该阶段的并发粒度, 默认是256。

调整该值尝试优化, 线上观察发现该值在设置为8192(XX+UnlockDiagnosticVMOptions -XX:ParGCCardsPerStrideChunk=8192)时,能够减少STW时间5~10ms, 虽然有效果, 但是效果有限. 而且该参数和具体业务应用是有关联的。

3、在有限的时间内解决该业务的GC性能问题, 我们把方向调整为针对该业务进行JVM调优

为了能达到物理机kernel2.6.32 的YoungGC的STW时间表现, 我们**调小了**

YoungGen的空间.并且打开并发处理引用的标记.

(-Xmn512M -XX:+ParallelRefProcEnabled)

最后我们达到了YoungGC的STW 时间稳定在60~70ms 之间, 平均每12s进行一次YoungGC

4、优化JVM的STW时间达到60~70ms后, 我们观察线上打点延迟情况, 通过降低STW时间, 使得请求平均响应时间下降了10ms

最后使用的JVM主要参数汇总:

```
#内存
-Xmx10240M -Xms10240M -Xmn512M -XX:MaxPermSize=512M -
XX:PermSize=512M
# GC 针对Docker CPU 4
-XX:+ParallelRefProcEnabled -XX:ConcGCThreads=4 -XX:
+ParallelRefProcEnabled
-XX:+UseParNewGC -XX:MaxTenuringThreshold=2 -XX:SurvivorRatio=
4
-XX:+CMSParallelRemarkEnabled -
XX:CMSInitiatingOccupancyFraction=50
-XX:+UseCMSInitiatingOccupancyOnly -XX:
+CMSParallelInitialMarkEnabled
-XX:+ExplicitGCIInvokesConcurrent
# CardTable
-XX:+UnlockDiagnosticVMOptions -XX:ParGCCardsPerStrideChunk=
8192
```

经验小结

通过该优化过程, 我们能够得出的结论主要有以下三点:

1、在根据Docker资源限制调整JVM启动参数后, Docker对JVM的GC基本上没有影响。

JVM读取的宿主机CPU信息和Docker的CPU限制的之间矛盾是JVM在Docker中运行普遍遇到的, 针对这一点,

Ocean平台近期会推出Docker JVM的优化方案. 详情参见Docker JVM 优化说明

2、使用JVM(version1.7.0_80)的业务在不同内核下运行, GC表现可能会有差距

这个差距涉及到JVM和业务本身两大因素. 目前我们没有收到Ocean上其他Java类业务的GC问题的反馈.

更具体的原因需要花时间进一步定位问题.

3、通过针对业务特定JVM调优, GC表现可以达到原物理机同等水平

最后的结果也表明, 通过我们的针对业务的特定调优, 是可以达到原物理机的同等GC水平的. 从YoungGC的执行间隔时间单方面来看
新kernel表现更优.

JVM 内存区域、字节码执行引擎

2018年3月5日 11:50

内存区域

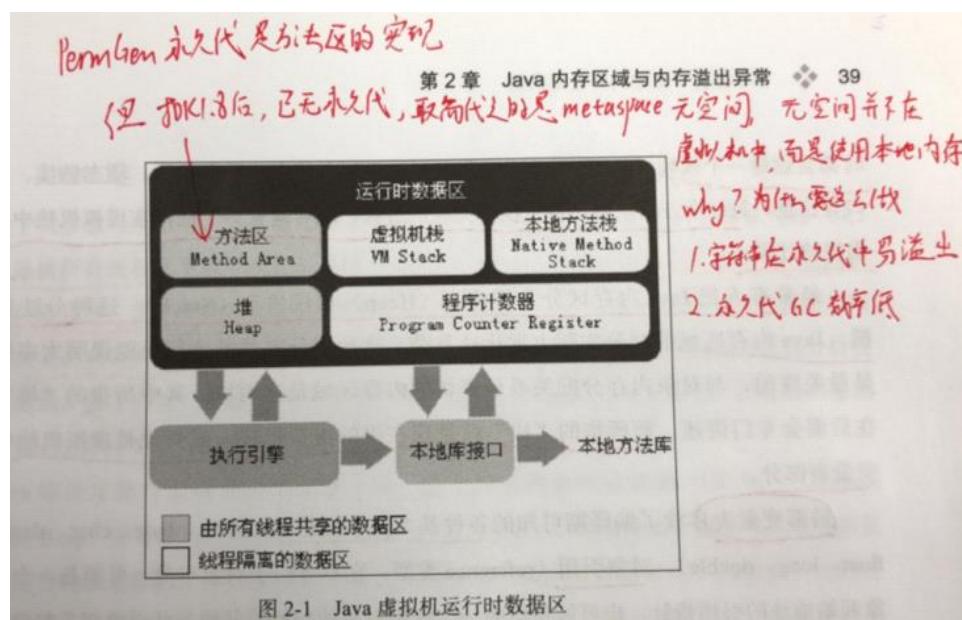
对象的创建、new一个对象

对象的内存布局

对象的访问定位

执行引擎

内存区域



- 1、**程序计数器**：知道线程执行位置，保证线程切换后能恢复到正确的执行位置。
- 2、**虚拟机栈**：存栈帧。栈帧里存局部变量表、操作栈、动态连接、方法返回地址。局部变量表又存了各种基本数据类型和**对象引用（句柄）**。
- 3、**本地方法栈**：为Native方法服务
- 4、**堆**：存放**对象实例**和**数组**，可以处于物理上不连续的内存空间
- 5、**方法区**：存类信息、**常量**、**静态变量**。有运行时常量池，存放类的符号引用

堆主要用来存放对象，栈主要用来执行程序。

对象的创建

- 1、虚拟机遇到一条new指令时，会先去常量池检测能否找到new对应的类的符号引用，并检测

这个类是否加载、初始化。

- 2、如果加载检查通过，则分配内存。分配内存有两种方式：(1)指针碰撞，针对连续内存区域；(2)空闲列表，针对不连续内存区域
- 3、内存分配完之后，会对内存初始化零值，保证实例字段能在java代码不赋初值也能使用。
- 4、接下来对对象信息进行设置，把类的元数据信息、对象的哈希码、对象的GC分代年龄等信息存放在对象头之中
- 5、最后执行用户的Init方法

对象的内存布局

- 1、分为三部分，对象头、实例数据、对齐填充
- 2、对象头：(1)对象自身运行时数据，如哈希码、GC分代年龄、锁状态标志、线程持有的锁等。(2)类型指针，虚拟机通过这个来确定这个对象是哪个类的实例。(3)如果对象是一个Java数组，那么对象头中还必须有一块用于记录数组长度的数据。
- 3、实例数据：对象真正存储的有效信息，也是在程序代码中定义的各种类型的字段内容。
- 4、对齐填充：JVM要求对象的起始地址必须是8字节的整数倍，因此当对象实例数据没有对齐时，这部分来补全。

对象的访问定位

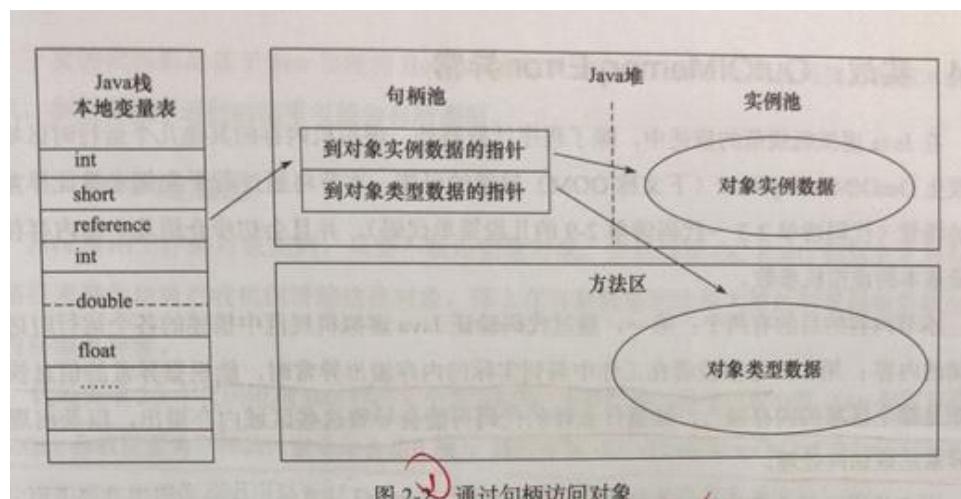


图 2-2 通过句柄访问对象

- 如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象地址，如图 2-3 所示。

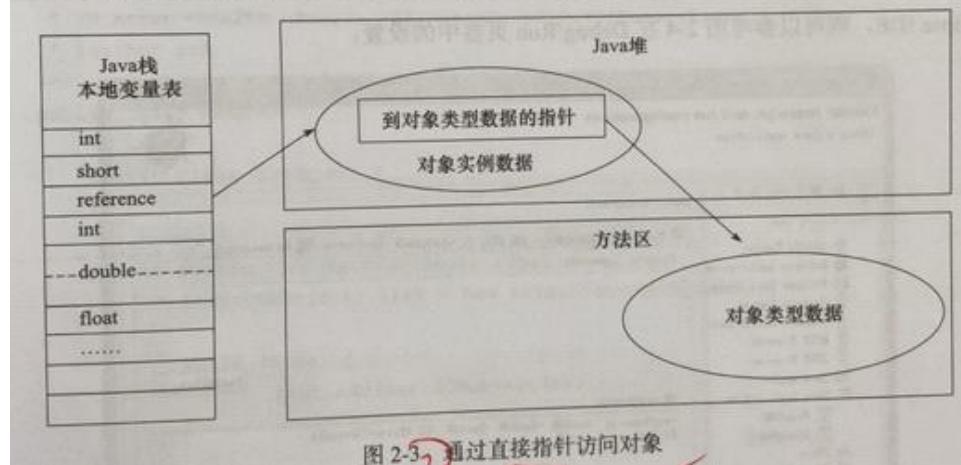


图 2-3 通过直接指针访问对象

- 取决于虚拟机的实现而定，有“**句柄**”和“**直接指针**”两种方式
- “句柄”的好处是，在对象被移动（垃圾回收时很普遍），只用修改句柄中的实例数据指针，而reference本身不用修改。
- “直接指针”的好处是，速度更快，毕竟节省了一次指针定位的时间开销。由于对象的访问在Java中非常频繁，因此这部分开销节省下来也很可观。

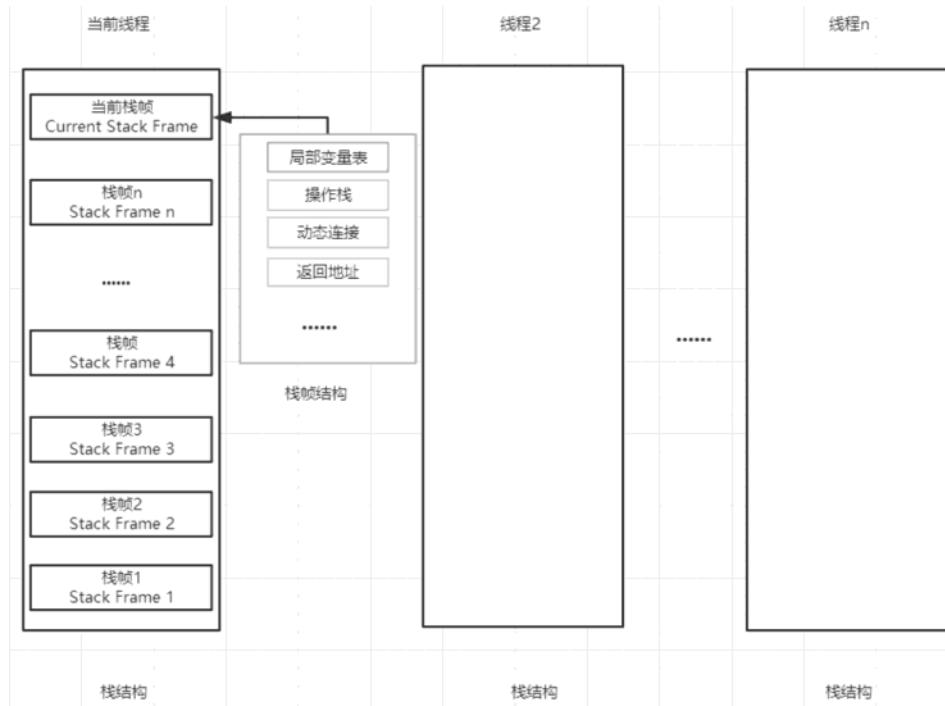
JVM字节码执行引擎

字节码文件即类文件被加载后，就能送入执行引擎了：

- 输入**：字节码文件
- 处理**：字节码解析
- 输出**：执行结果。

物理机的执行引擎是由硬件实现的，虚拟机的执行引擎由自己实现的。

运行时栈帧结构



- 栈帧 (Stack Frame)** 是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈 (Virtual Machine Stack) 的栈元素。
- 每个栈帧都包括了一下几部分：局部变量表、操作数栈、动态连接、方法的返回地址和一些额外的附加信息。
- 每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。
- 一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

在活动线程中，只有位于栈顶的栈帧才是有效的，称为**当前栈帧**，与这个栈帧相关联的方法称为**当前方法**，执行引擎运行的所有字节码指令都只针对当前栈帧进行操作。

局部变量表：

一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。以变量槽slot为单位，一个slot可以放32位数据类型，对于long\double占用2个slot。

操作数栈：

即用来存放操作数的栈结构，当一个方法刚开始执行的时候，这个方法的操作数栈是空的，在方法的执行过程中，会有各种字节码指令向操作数栈中写入和提取内容，也就是入栈和出栈的操作。

java虚拟机的解释执行引擎称为基于栈的执行引擎，其中所指的栈就是操作数栈。

动态连接：

运行期将相关的符号引用转换为直接引用

方法返回地址：

方法执行完成的结果值

方法调用：

解析方法的符号引用和确定方法的版本

(1) 虚方法和非虚方法

只有在允许方法重载的情况下才有虚方法和非虚方法之分，因为在允许重载的情况下方法的版本不止一个，在方法的执行前需要特定的机制确定将要执行的方法版本。

非虚方法：能够在解析阶段（将符号引用转换为直接引用）确定方法执行版本的方法。在解析阶段能够确定方法版本的有：静态方法，私有方法，实例构造器，父类方法，final修饰的方法。

虚方法：只有在运行期才能够最终确定方法执行版本的方法。

(2) 解析

方法调用时，方法执行前将方法内的符号引用转换为直接引用的过程

解析调用时一个静态的过程，在编译期内就能够完全确定。类加载过程中解析阶段就会把涉及的符号引用直接转换为直接引用。

(3) 分派

确定方法最终调用的版本的步骤。

1) 静态分派和动态分派：

静态分派（编译期分派）：在编译期根据静态类型来定位方法执行的版本的分配过程，因为静态类型在编译期可知。Java虚拟机方法重载的规则就是根据参数的静态类型确定最终执行的方法版本的。

动态分派（运行期分派）：在运行期根据实际类型来确定最终执行的方法版本的分派过程

2) 方法的重载和重写

方法的执行

1. 解释执行（通过解释器执行）
2. 编译执行（通过JIT编译器产生本地代码执行）

基于栈的代码执行示例

下面我们用简单的案例来解释一下JVM代码执行的过程，代码实例如下：

```

public class MainTest {
    public static int add(){
        int result=0;
        int i=2;
        int j=3;
        int c=5;
        return result =(i+j)*c;
    }

    public static void main(String[] args) {
        MainTest.add();
    }
}

```

使用javap指令查看字节码：

```

public static int add();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=4, args_size=0      //栈深度2, 局部变量4个, 参数0个
0:  iconst_0  //对应result=0,0入栈
1:  istore_0  //取出栈顶元素0, 将其存放在第0个局部变量slot中
2:  iconst_2  //对应i=2,2入栈
3:  istore_1  //取出栈顶元素2, 将其存放在第1个局部变量slot中
4:  iconst_3  //对应 j=3, 3入栈
5:  istore_2  //取出栈顶元素3, 将其存放在第2个局部变量slot中
6:  iconst_5  //对应c=5, 5入栈
7:  istore_3  //取出栈顶元素, 将其存放在第3个局部变量slot中
8:  iload_1   //将局部变量表的第一个slot中的数值2复制到栈顶
9:  iload_2   //将局部变量表中的第二个slot中的数值3复制到栈顶
10: iadd     //两个栈顶元素2,3出栈, 执行相加, 将结果5重新入栈
11: iload_3   //将局部变量表中的第三个slot中的数字5复制到栈顶
12: imul     //两个栈顶元素出栈5,5出栈, 执行相乘, 然后入栈
13: dup      //复制栈顶元素25, 并将复制值压入栈顶.
14: istore_0  //取出栈顶元素25, 将其存放在第0个局部变量slot中
15: ireturn   //将栈顶元素25返回给它的调用者

```

JVM Class文件，类加载机制、编译过程

2018年3月11日 19:48

[Java代码编译过程](#)

[Class文件](#)

[类加载的详细过程](#)

[类加载器](#)

[Java类的加载过程](#)

[双亲委派模型](#)

Java编译器先把Java代码编译为存储字节码的Class文件，再通过Class文件进行类加载。

Class类文件的结构

Java编译器可以把Java代码编译为存储字节码的Class文件。[Java代码编译过程](#)

Class文件格式采用一种类似C语言结构体的伪结构来存储数据。这种伪结构中只有两种数据类型：**无符号数和表**。整个Class文件本质上就是一张表。

无符号数：属于基本数据类型，以u1、u2、u4分别代表1个字节、2个字节、4个字节。

表：由多个无符号数或其他表作为数据项构成的复合数据类型。

| 表 6-1 Class 文件格式 | | |
|------------------|---------------------|-----------------------|
| 类 型 | 名 称 | 数 量 |
| u4 | magic | 1 |
| u2 | minor_version | 1 |
| u2 | major_version | 1 |
| u2 | constant_pool_count | 1 |
| cp_info | constant_pool | constant_pool_count-1 |
| u2 | access_flags | 1 |
| u2 | this_class | 1 |
| u2 | super_class | 1 |
| u2 | interfaces_count | 1 |
| u2 | interfaces | interfaces_count |
| u2 | fields_count | 1 |
| field_info | fields | fields_count |
| u2 | methods_count | 1 |
| method_info | methods | methods_count |
| u2 | attributes_count | 1 |
| attribute_info | attributes | attributes_count |

Class类文件结构详解：

- 魔数**：每个Class文件的头4个字节称为魔数，唯一的作用是确定这个文件是否能被虚拟机接受。如GIF\JPEG等在文件头都存有魔数。
- 版本号**：紧接着魔数的4个字节是Class文件的版本号。

3. 常量池：接着版本号的是常量池入口
4. 访问标志：接着常量池的是访问标志，标志着这个Class是类还是接口、是否为public等
5. 类索引、父类索引与接口索引集合：之后接着这三个
6. 字段表集合：接着是字段集合，用于描述接口或者类中声明的变量
7. 方法表集合：
8. 属性表集合：

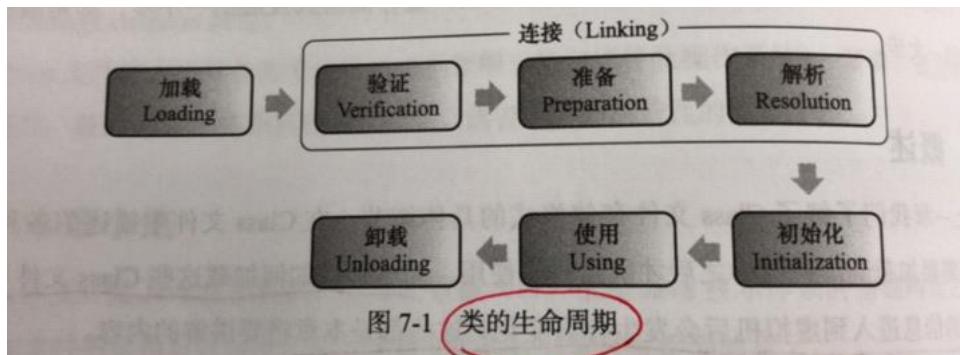
一句话解释

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型，这就是JVM的类加载机制。

类加载时期

- 在Java语言里，类的加载、连接和初始化过程都是在程序运行期间完成的。
- Java语言运行期加载类的特性，为Java应用程序提供了高度的灵活性，比如一个本地程序可以在运行时从网络或其他地方加载一个二进制流作为程序代码的一部分。

类的生命周期



- 类从被加载到虚拟机内存中开始，到卸载出内存为止，整个生命周期如上图。
- 其中加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的
- 解析阶段则不一定，可能在初始化之后才开始。

类与接口加载时的区别

只有一点，当一个类在初始化的时候，要求其父类全部都已经初始化过了，但一个接口在初始化的时候，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量）才会初始化。

类加载的详细过程

一、加载

1. 通过一个类的全限定名来获取定义此类的二进制字节流，JVM把这个阶段的动作放在了虚拟机外部的“类加载器”中实现。未指明从哪里获取，因此有各种花样，比如从JAR

包、WAR包，或者网络，或者运行时计算生成等等。

2. 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。
3. 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。即对象类型数据（非对象实例数据）存在方法区。

二、验证

验证的目的是确保Class文件的字节流中包含的信息不会危害虚拟机自身的安全，直接决定了Java虚拟机是否能承受恶意代码的攻击。

- 验证阶段分为4个
 - a. 文件格式验证
 - b. 元数据验证
 - c. 字节码验证
 - d. 符合引用验证

三、准备

准备阶段为类变量在方法区中分配内存并设置类变量的零值

1. 这里只包含类变量（即被static修饰的变量），而不是实例变量
2. 实例变量会在对象实例化时随着对象一起分配在Java堆中
3. 比如 public static int value =123;在准备阶段过后value=0，只有在初始化阶段之后，value才等于123.

四、解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程

1. 符号引用也是描述所引用目标的，但引用的目标并不一定以及加载在内存中
2. 直接引用是指针、句柄这种，直接引用的目标必定以及在内存中存在。

五、初始化

初始化时类加载的最后一步，根据程序员的主观去初始化类变量和其他资源

类加载器ClassLoader

- 虚拟机把类加载阶段中的通过一个类的全限定名来获取定义此类的二进制字节流这个动作放在了虚拟机外部的“类加载器”中实现。
- 对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性
- 比较两个类是否“相等”，只有在这个两个类是由同一个类加载器加载的前提下才有意义。

Java类的加载过程

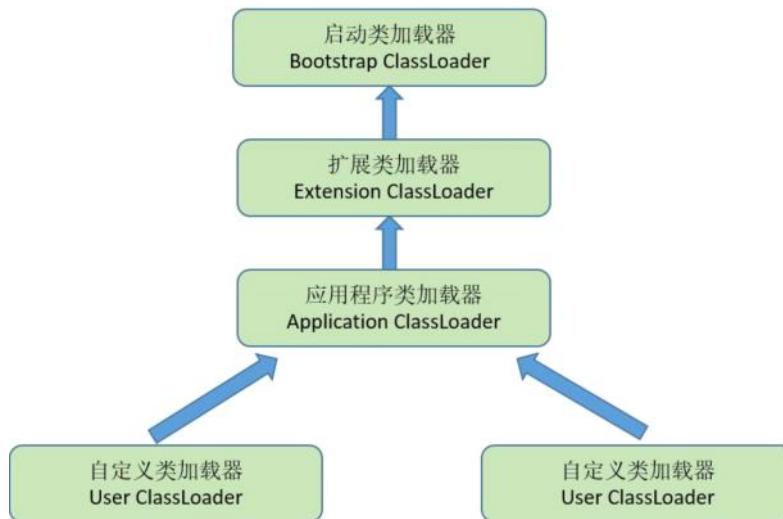
Java类的加载过程应用了双亲委派模型



双亲委派模型

绝大部分Java程序都会用到以下3种系统提供的类加载器

1. 启动类加载器
2. 扩展类加载器
3. 应用程序类加载器



上图就是类加载器双亲委派模型

工作流程:

某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

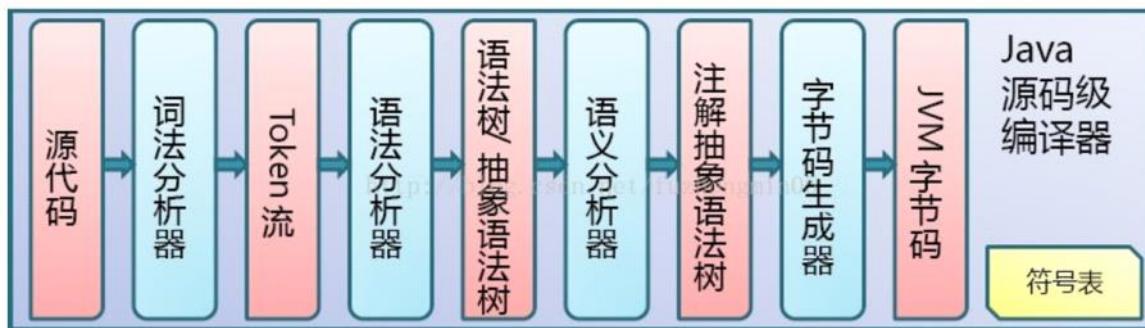
好处:

Java类随着它的类加载器一起具备了一种带有优先级的层次关系。

为什么需要双亲委派模型:

1. 例如类`java.lang.Object`，它存在在`rt.jar`中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的`Bootstrap ClassLoader`进行加载，因此`Object`类在程序的各种类加载器环境中都是同一个类。相反，如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个`java.lang.Object`的同名类并放在`ClassPath`中，那系统中将会出现多个不同的`Object`类，程序将混乱。
2. 如果不采用双亲委派模型，那么由各个类加载器自己去加载的话，那么系统中会存在多种不同的`Object`类。

Java代码编译过程



代码编译是由Javac编译器来完成，流程如上图所示。

Javac的任务就是将Java源代码编译成Java字节码，也就是JVM能够识别的二进制代码，从表面看是将.java文件转化为.class文件。而实际上是将Java源代码转化成一连串二进制数字，这些二进制数字是有格式的，只有JVM能够真确的识别他们到底代表什么意思。

具体流程：

1. **词法分析**：读取源代码，一个字节一个字节的读进来，找出这些词法中我们定义的语言关键词如：if、else、while等，识别哪些if是合法的哪些是不合法的。这个步骤就是词法分析过程。
2. **语法分析**：就是对词法分析中得到的token流进行语法分析，这一步就是检查这些关键词组合在一起是不是符合Java语言规范。如if的后面是不是紧跟着一个布尔型判断表达式。
3. **语义分析**：语法分析完成之后也就不存在语法问题了，语义分析的主要工作就是把一些难懂的，复杂的语法转化成更简单的语法。比如将foreach转化为for循环。
4. **字节码生成**：将会根据经过注释的抽象语法树生成字节码，也就是将一个数据结构转化为另外一个数据结构，结果就是生成符合java虚拟机规范的字节码。

Java内存模型

2018年3月3日 16:24

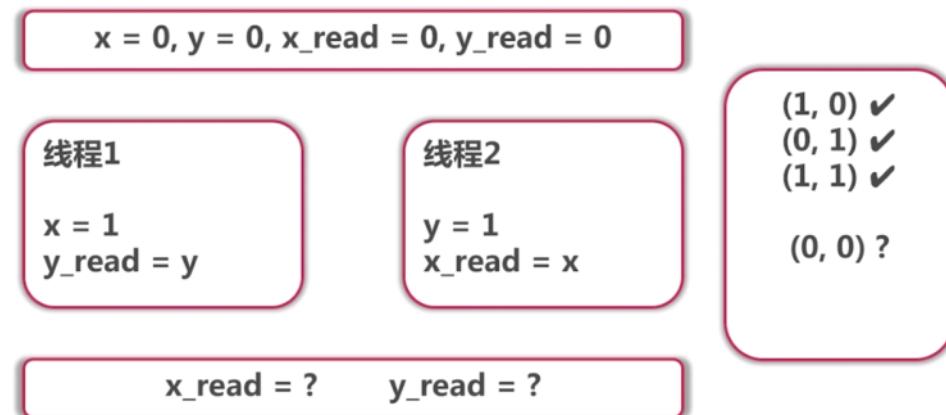
目录

[Java内存模型定义](#)
[happens-after关系](#)
[主内存与工作内存\(本地内存\)](#)
[竞争现象](#)
[volatile和synchronized的区别](#)
[典型的volatile使用场景](#)

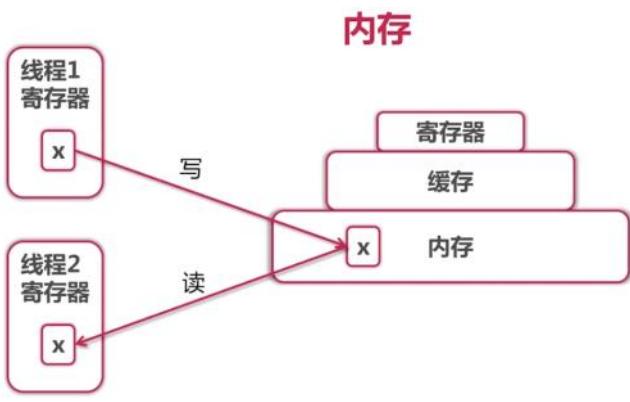
Java内存模型定义

1. 描述多线程环境中线程与内存的关系
2. Java内存模型定义了程序中各个变量的访问规则，即虚拟机将变量存储到内存和从内存取出变量的底层细节。
3. 这里的变量可以理解为堆和方法区的，不包括线程私有的栈。
4. 解决了多线程之间共享变量的可见性以及如何在需要的时候对共享变量进行同步。
5. 用来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果。

一个例子



- 1、先定义x,y,x_read,y_read都等于0
- 2、然后启动线程1和线程2
- 3、 (x_read, y_read) 可能是右边4种结果， $(0,0)$ 也可能
- 4、一个线程内的代码概率是按照顺序执行的，即这里线程1要执行的代码顺序肯定是先 $x=1$ ，再 $y_read=y$ ，而概率不会反过来。但真反过来了（线程要执行的代码乱序执行）也是会发生的，就出现了 $(0,0)$ ，只不过这个反过来的概率很小。
- 5、由于内存可见性也会出现 $(0,0)$ ，即线程1执行了 $x=1$ ，把结果写入了自己的寄存器，但还没有写入内存，（什么时候写入内存呢？这个不确定）此时线程2读取内存看到 x 自然就是0。



6、如何解决内存可见性的问题？因此有了Java内存模型

Java内存模型是一个规范，思想

1、首先定义了一个关系：happens-after关系

即：如果操作执行顺序具有先后性，那么后执行的操作能够看到先执行的操作（在内存中）的结果。

2、JVM自动保证以下操作遵守happens-after关系，(1)(2)最重要

(1) Unlock发生在Lock之前，即第一个人拿了锁做了一些事情，释放了锁之后，第二个人再拿这个锁，必须知道第一个人做的结果

(2)写volatile发生在读volatile之前，即加了volatile的变量的变化都是所有线程可见的

(3)线程start()发生在线程所有动作之前

(4)线程中所有操作发生在线程join之前

(5)构造函数完成发生在finalizer开始之前

3、上面我们随便定义的x,y,x_read,y_read并不遵守这个happens-after关系

4、与程序员密切相关的happens-before规则如下：

(1)程序顺序规则：一个线程中的每个操作，happens-before于该线程中任意的后续操作。

(2)监视器锁规则：对一个锁的解锁操作，happens-before于随后对这个锁的加锁操作。

(3)volatile域规则：对一个volatile域的写操作，happens-before于任意线程后续对这个volatile域的读。

(4)传递性规则：如果 A happens-before B，且 B happens-before C，那么A happens-before C。

线程之间的通信

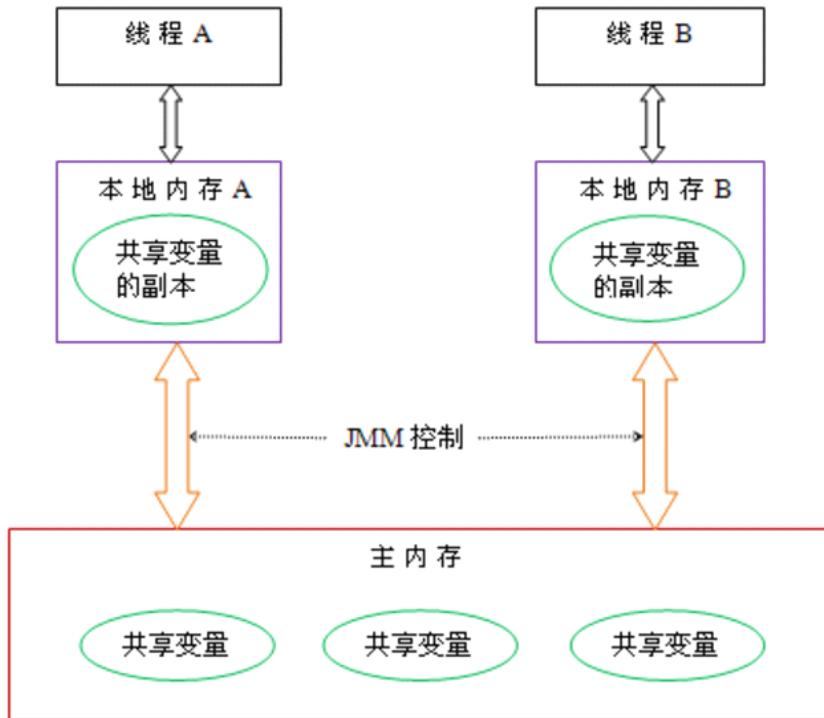
1、线程之间的通信机制有两种共享内存和消息传递

2、典型的共享内存通信方式就是通过共享对象进行通信

3、在java中典型的消息传递方式就是wait()和notify()。

主内存与工作内存（本地内存）

- 1、Java线程之间的通信采用的是共享内存模型，这里提到的共享内存模型指的就是Java内存模型(简称JMM)，JMM决定一个线程对共享变量的写入何时对另一个线程可见。
- 2、JMM定义了线程和主内存之间的抽象关系：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。



内存间交互操作

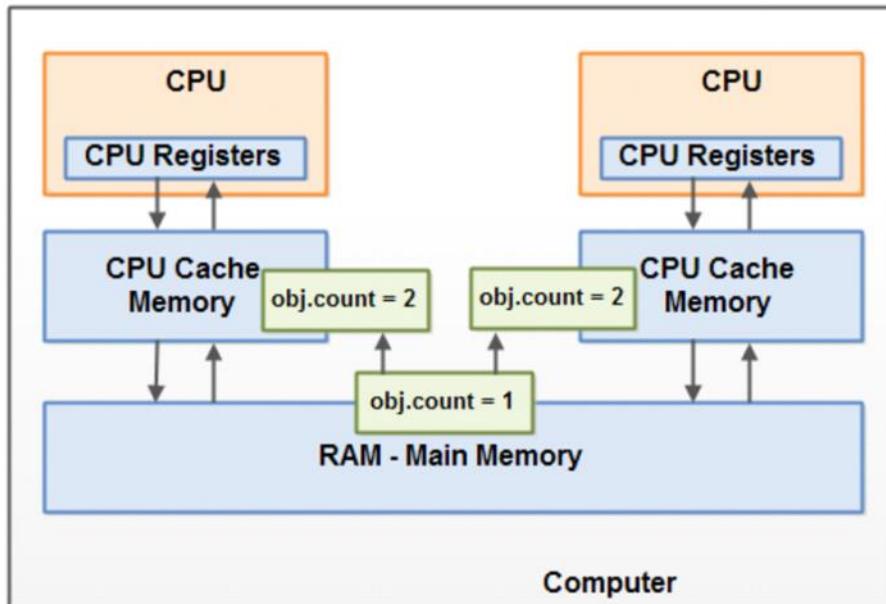
- 1、JMM定义了8种操作，保证这8种操作都是原子的。包括lock,unlock,read,load,use,assign,store,write
- 2、同时定义了8种操作必须满足的规则，总结下来就是先行发生原则。

共享对象的可见性

- 1、当多个线程同时操作同一个共享对象时，如果没有合理的使用volatile和synchronization关键字，一个线程对共享对象的更新有可能导致其它线程不可见。
- 2、**volatile** 关键字可以保证变量会直接从主存读取，而对变量的更新也会直接写到主存

竞争现象

- 1、如果多个线程共享一个对象，如果它们同时修改这个共享对象，这就产生了竞争现象。
- 2、如下图，CPUa和CPUb同时，真正意义上的同时，对obj.count进行了+1操作，即使加了volatile，最终主内存中的obj.count也只会等于2



- 3、解决这个问题只能用synchronized
- 4、为什么volatile无效？ volatile不是内存修改可见吗？但面对真正的同时， volatile也没办法
- 5、因为**volatile关键字解决的是内存可见性的问题；synchronized关键字解决的是执行控制的问题**

volatile和synchronized的区别

- 1、volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住
- 2、volatile不会造成线程的阻塞； synchronized可能会造成线程的阻塞。
- 3、volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性
- 4、volatile仅能使用在变量级别； synchronized则可以使用在变量、方法、和类级别的

典型的volatile使用场景

要使用volatile，必须同时满足两个条件：

1. 对变量的写操作不依赖当前值！！！！！！！！！！！！！
2. 该变量没有包含在具有其他变量的不变式中。

例子1：

```
volatile boolean shutdownFlag;

while(!shudownFlag){
    do something;
}
```

这是一个典型的volatile使用场景，如果不加volatile，当shutdownFlag被另一个线程修改时，执行判断的线程却发现不了，就无法及时退出循环。

例子2：

单例模式双重锁检查中使用volatile

```
private volatile static TestInstance instance;

public static TestInstance getInstance() { //1
    if (instance == null) { //2
        synchronized (TestInstance.class) { //3
            if (instance == null) { //4
                instance = new TestInstance(); //5
            }
        }
    }
    return instance; //6
}
```

在并发情况下，如果没有volatile关键字，在第5行会出现问题

对于第5行可以分解为3行伪代码

1. memory=allocate();// 分配内存 相当于c的malloc
2. ctorInstanc(memory) //初始化对象
3. instance=memory //设置instance指向刚分配的地址

上面的代码在编译器运行时，可能会出现重排序 从1-2-3 排序为1-3-2

如此在多线程下就会出现问题

例如现在有2个线程A,B。线程A在执行第5行代码时，B线程进来，而此时A执行了1和3，没有执行2，此时B线程判断instance不为null 直接返回一个未初始化的对象，就会出现问题。

而用了volatile，就会禁止重排序，就不会出现上述问题。

例子

Singleton中的双重锁模式

```
private Connection conn = null;  
Connection getSingletonConnection() {  
    if (conn == null) {  
        synchronized (this) {  
            if (conn == null)  
                conn = new Connection();  
        }  
    }  
}
```

- 1、注意private Connetcion conn =null;这一句没有加volatile
- 2、因此会导致synchronized外面的if(conn==null)这句判断出错。即Connection的构造函数乱序，线程看到conn不等于null，但其实Connection的构造函数还没运行完，因此我们会获得一个构建到一半的Connection，这个Connection并不能用。
- 3、为了解决这个问题，要改写成private volatile Connetcion conn =null;

Java异常处理

2018年3月3日 19:37

异常分为Checked (检查) 和Unchecked (非检查) 异常

Java异常处理

Checked Exception 与 Unchecked Exception

- ◆ RuntimeException及其子类 : Unchecked
- ◆ IllegalArgumentException, ClassCastException, IllegalStateException, NullPointerException, ...
- ◆ 其他Exception : Checked
 - ◆ IOException, InterruptedException, ParseException, ...

1、Unchecked异常在编译器不会报错，只有在运行时才会报错

如何处理异常

- 1、写日志
- 2、执行相关处理逻辑，比如换种方法、调整代码避免异常发生

如何不处理Checked Exception

- 1、如果能，添加throws定义。比如调用Thread.sleep，编译器会让你处理异常，我们不想处理的方法就是在调这个Thread.sleep的函数名后面添加throws定义。
- 2、不处理的精髓就是把Exception扔出去
- 3、如果加不了throws定义，就把Checked Exception转化为Unchecked Exception。
- 4、如何转化？Throw new RuntimeException(**checkedException**)；一定要把Checked Exception放进去，如下

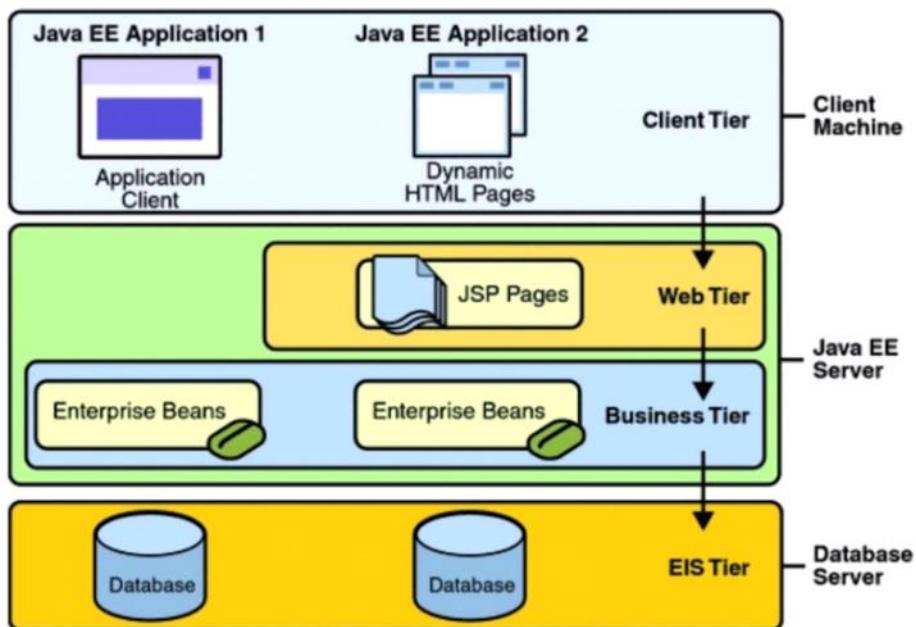
```
        }
    }catch(IOException e) {
        System.out.println("IO异常"+e);
        throw new RuntimeException(e);
    }
```

服务器架构的演进

2018年3月3日 19:47

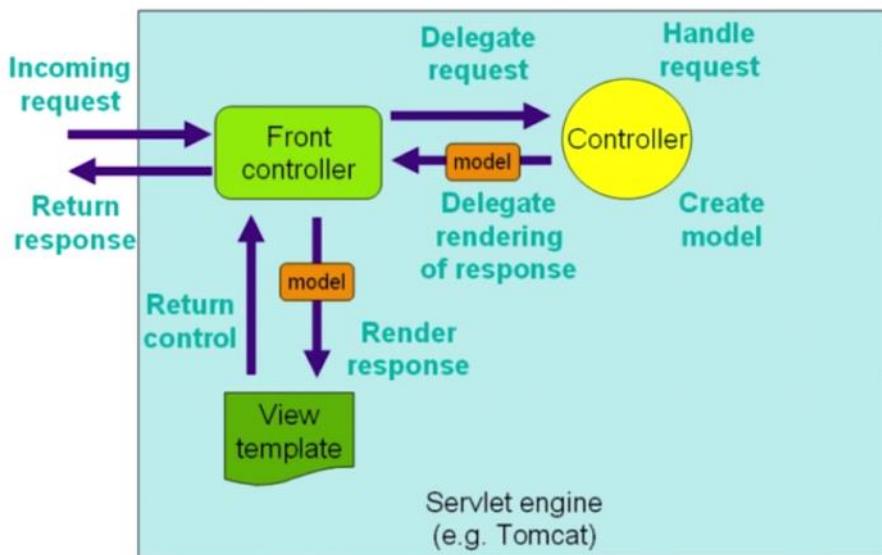
服务器架构演进

三层架构



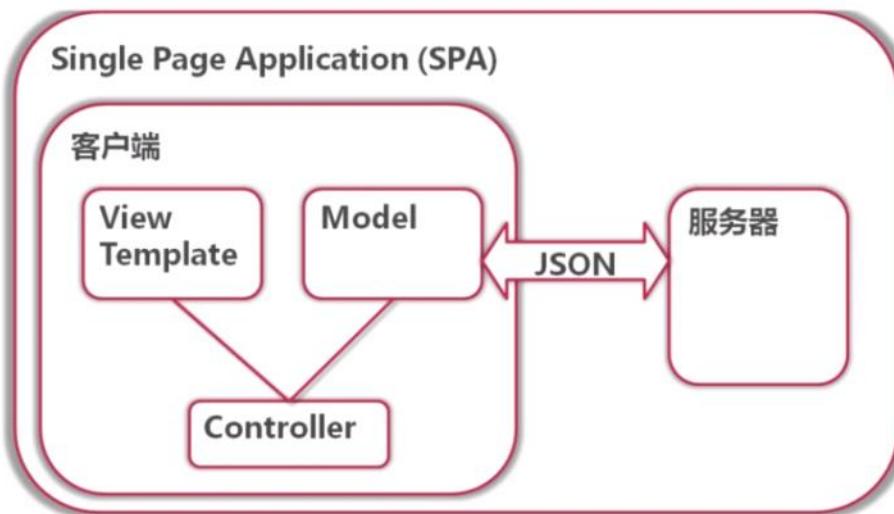
- 1、最早是三层架构，客户、服务器、数据库
- 2、还没来得及考虑客户量上涨，就发现了其他问题
- 3、这个问题是：客户层依赖于Web层，Web层依赖于业务逻辑层。**Web层依赖于业务逻辑层这里不对**，因为Web层应该只是展示数据，为什么还要依赖业务逻辑？应该依赖的是数据，所以不对
- 4、因此有了MVC结构

MVC



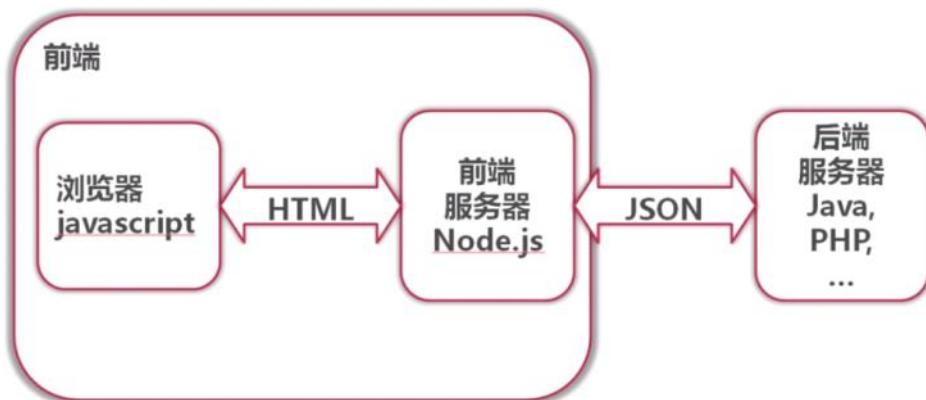
- 1、前端View Template只依赖于Model，黄色的Controller是业务逻辑，通过Model与前端连接
- 2、缺点是：黄的的Controller和绿色的View离的越来越远，越来越独立，我们却还把他们两放在同一个JVM里面，每次要一起release出去，这个在灵活性上面就有很大的约束。
- 3、因此考虑前后端分离，因此有了Ajax

基于Ajax的前后端分离



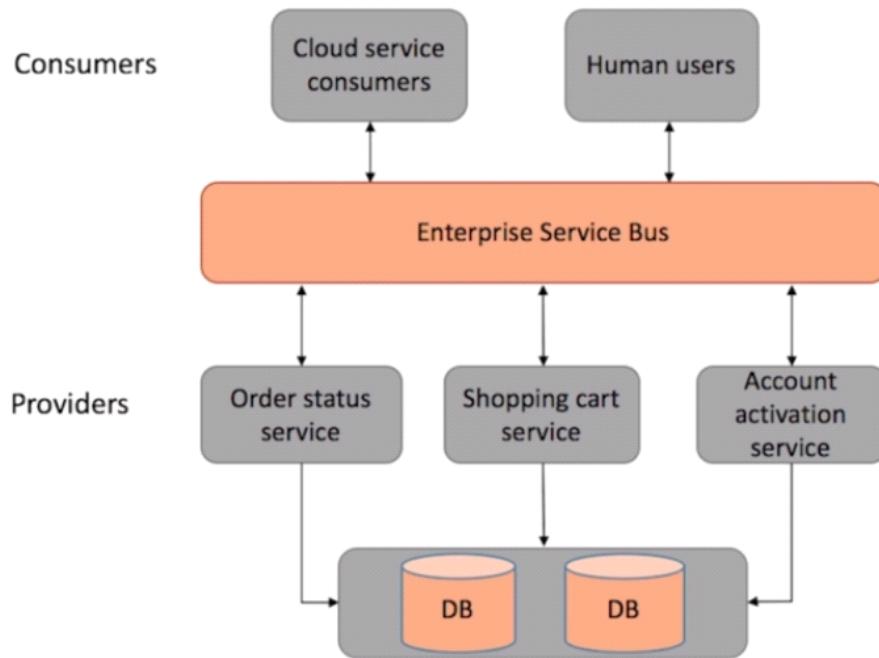
- 1、缺点是前端太复杂了
- 2、另一个缺点是不利于SEO (Search Engine Optimization, 搜索引擎优化)，即搜索引擎会认为我们只有一个页面，却不知道这个页面上的按钮按下去之后会有很多功能，这就导致我们的产品无法在搜索引擎这里有一个好的位置，排位不靠前，互联网搜索第一页以后的东西基本都没人看了。
- 3、前后端隔的太远也不好，因此有了Node.js

基于Node.js的前后端分离

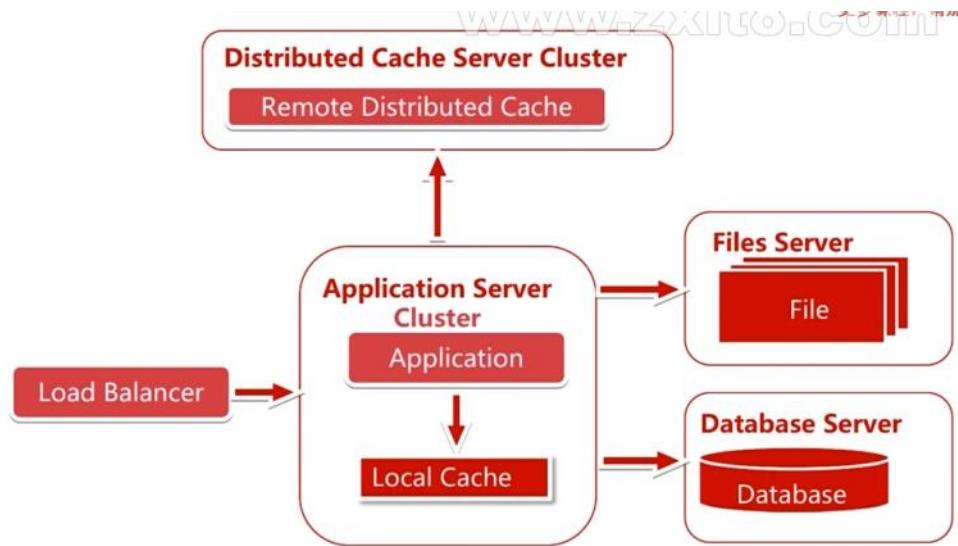


- 1、在前端加了一个服务器，前后端仍然是JSON交互
- 2、用户浏览器和前端服务器交互是HTML文本
- 3、**前端这方面没什么缺点了，再来看后端**
- 4、后端全扔一个JVM里面，显然性能有限
- 5、因此有了SOA

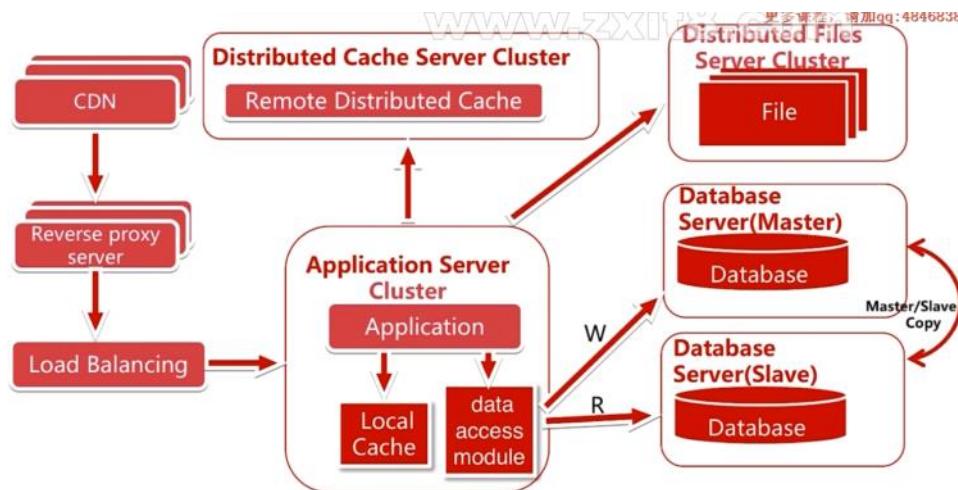
SOA



- 1、后端服务分成很多个服务器，每个服务器负责自己的service。即一个功能一个服务器
- 2、服务和服务之间通过一个Service接口进行交互
- 3、SOA最大的缺点是服务会down掉，比如这张图中间的Enterprise Service Bus如果down掉就全挂了

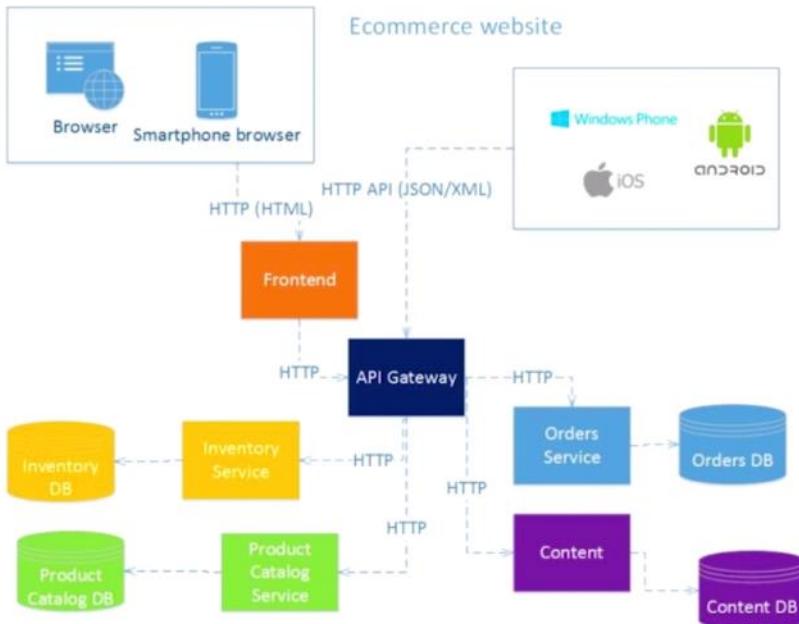


- 增加多个服务器的时候，需要增加一个Load Balancer即负载均衡调度器，与之相关有很多种负载均衡策略，比如轮询、最小连接等。



- CDN解决不同地区访问速度的问题
- Reverse proxy server反向代理服务器，可以缓冲用户请求
- 数据库读写分离

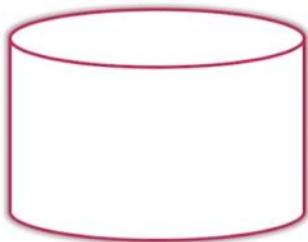
微服务



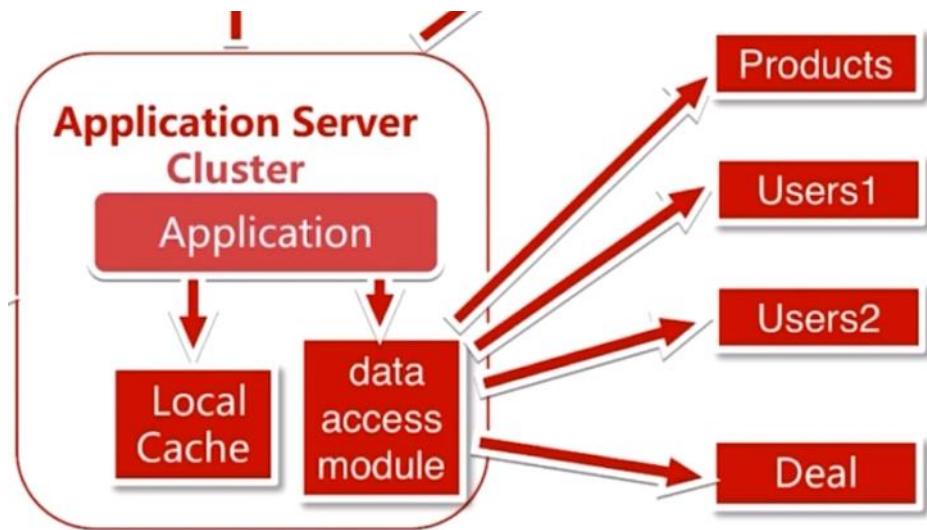
- 1、首先服务按照业务的领域来分，order订单就是订单，Inventory库存就是库存，product catalog产品目录就是产品目录
- 2、划分的非常彻底，数据库都是分开的，SOA里数据库不是分开的
- 3、各个服务相当于分开了，可以分开测试，分开上线
- 4、**微服务是现在最火的服务器架构**

数据库架构的演进

单个数据库



- 1、最早单个数据库
- 2、单个数据库太大了、太乱了，要拆一下，**分库分表**
- 3、分库，有按照领域拆，也有按照地域拆分的



- 1、**分表**，把User表拆分成两张表，存入两个数据库
- 2、解决单数据库瓶颈问题

根据领域拆分数据库



- 1、缺点是：交易里面有一个userId联系用户信息，还要有一个productId联系产品信息，在单个数据库里可以通过外键很方便的连接，但这里就不好做了。
- 2、这个只适合微服务

根据分区拆分数据库



- 1、缺点是：华北的同学访问其他的数据库就比较慢甚至不能访问
- 2、因此有读写分离

数据库读写分离



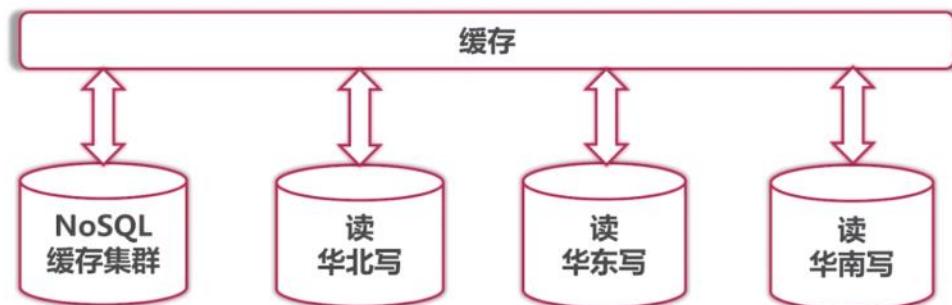
- 1、每个数据库都放相同的数据，包含所有领域和所有地区
- 2、但是我们总共有三个拷贝，不能全都支持读写，这样太复杂了
- 3、两个读，一个读写，这样“读”的性能就解决了
- 4、但“写”的问题还没解决
- 5、因此与区域相结合去解决“写”的问题

数据库读写分离+区域



- 1、所有的数据库都能读，但是写只能就近写，写完之后再同步到其他数据库
- 2、数据库结构调整好了，但是访问还是慢怎么办？
- 3、因此加上缓存

加入缓存

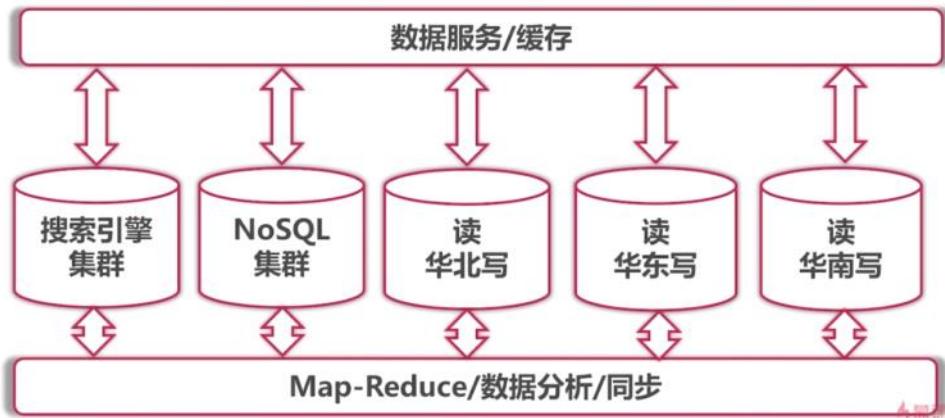


- 0、因为80%的访问都在20%的数据上，著名的28原则，因此可将这些高频数据放入缓存。
- 1、缓存放在NoSQL（泛指非关系型数据库，比如Redis, Memcached）的集群里面
- 2、关系型数据库啥都好，就是慢，NoSQL就是解决这个慢的问题

3、Memcached性能比Redis略强，但不支持持久化，Redis支持持久化，Redis还支持更多的数据结构

4、加入缓存变快了，但我们希望更快，因此添加搜索引擎

添加搜索引擎



- 1、把业务数据做成搜索引擎的Document，让搜索引擎给索引起来，这样“读”起来更快
- 2、同样数据存在这么多地方，又是数据库、又是NoSQL集群，又是搜索引擎集群的。因此后台会运行Map-Reduce来进行数据分析和同步

Map-Reduce

1. MapReduce是一个概念。
2. 假设我们手上有很多复杂数据，那么怎样来处理呢？显然需要分类。
3. map的工作就是切分数据，然后给他们分类，分类的方式就是输出key,value键值对，key就是对应“类别”了。
4. 分类之后，把数据送给Reduce，一个reduce处理一个key，reducer拿到的都是同类数据，这样处理就很容易了。

谈谈你做过的项目

在谈项目的时候，要谈到项目的架构，要讲采用了什么做法，解决了什么问题。如果实在没的说，也要讲这种做法本身想解决什么问题。我们可以给项目润润色，让其有这个问题，然后正好被我们所解决了。

如果流量/数据量/需求变化，你要如何扩展你的项目？

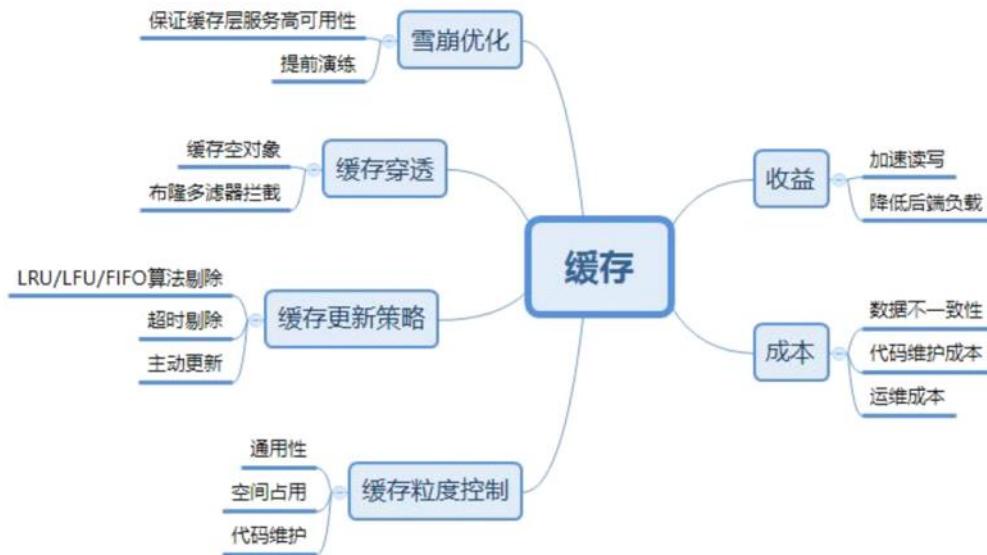
就要往架构演进这条路上靠

为何采用xxx框架，xxx存储引擎？

简略提到了这些框架、引擎，都要讲一个为什么用。要思考自己用的这些框架和这些框架的竞争品相比，各自有什么优缺点

缓存设计

2018年3月17日 11:04



缓存优缺点

缓存的**优点**就是“快”，一个快字基本能概括了。如上文说的加速读写，分流对数据库的压力，归根结底就是对快字的应用及其本身。

缺点主要是下面三点：

1. **数据不一致性**：DB的数据与缓存中的数据不一致；
2. **开发成本**：需要同时处理缓存层跟DB层的逻辑，增加了开发成本；
3. **维护成本**：例如需要对缓存层进行一个监控，增加了运维的成本。

因为有成本，所以要考虑缓存对象

被缓存的数据一般具有以下特点：

1. **经常被访问**：不经常被访问的数据即使缓存了对系统的性能吞吐也没太大的改善，没什么必要做缓存，直接访问DB、File、其他系统即可。
2. **改动不频繁**：如果一个数据改动很频繁，缓存的数据很容易就过期或者失效，保证数据一致性成本很高，命中率也很难上去，缓存的效果也不会好。
3. **时效性不强**：业务上要能容忍缓存失效前数据的不准确性。比如产品价格在页面展示的时候可以取缓存的数据，但在结算的时候一定要取数据库的值。

针对数据不一致性，要考虑缓存的更新策略

一般来说缓存也是需要有生命周期的，需要被更新或者删除，这样才能保持缓存的可控

性。

解决方案：

对数据实时性、一致性要求高用主动更新，对实时性要求不高用超时删除。

- **主动更新**: MySQL binlog增量订阅消费+消息队列+处理并把数据更新到redis 具体实现
- **超时删除**: 在设置缓存的时候可以设置过期时间，在时间到期之后自动删除。

考虑缓存粒度

- 假设一张用户表有20个字段，那是否需要将全部字段都放到缓存中？这就涉及到一个粒度的问题！
- 数据字段放少了，就会出现了不通用的问题；数据字段放多了，空间占用也多，序列化跟反序列化消耗的性能更多了。

解决方案：

在粒度这个问题上还是需要根据通用性，代码维护，性能跟空间占用这几点上进行考虑，简单来说就是靠经验了。

考虑缓存穿透

缓存穿透指的是查询一个不存在的数据，DB跟缓存都不会命中的数据。这样的话每次查询都会到DB层中查询，DB层负载加大还有可能造成死机，这样缓存就失去了保护DB层的意义。出现这种情况有两种：1.攻击，爬虫的大量请求；2.业务自身有问题。

解决方案：

1. **缓存空对象，一般用这个**，当DB层也查不到数据的时候，缓存一个null值进缓存，这样下一次的话就直接从缓存中读取，保护了后端。不过这种带来的后果是缓存了更多的键，需要更多的空间。
2. **布隆过滤器拦截**，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。这样在查询缓存之前先去过滤器中查询缓存是否有存在该key。不过这个适合于数据量固定且较少，实时性低的应用中，因为要维护这一个过滤器，数据大的时候布隆过滤器误算率高。

布隆过滤器 来自 <http://blog.csdn.net/better_jh/article/details/77187897>

考虑雪崩优化

雪崩指的是原先的缓存层承载了大量的请求，有效的保护了DB层，但是假如缓存层炸了，那所有的请求都直接穿透到DB层，会容易造成DB层也炸了。

解决方案：

1. 保证缓存层的高可用性，比如Redis的Sentinel哨兵模式（主从模式下，额外有一个Redis当哨兵，如果master挂了，哨兵从Slaver中重新选一个master出来）和Redis的Cluster集群模式都实现了高可用性。
2. 提前演练，模拟某一层挂了，看看怎么调整配置。

面试题目合集

2016年9月12日 14:43

目录

- [基本磁盘和动态磁盘的区别](#)
- [GPT磁盘与MBR磁盘的区别](#)
- [HTTPS和HTTP的区别](#)
- [POST和GET的区别](#)
- [Web开发中的cookie和session](#)
- [Cookie和Session的区别](#)
- [常见的Web安全漏洞](#)

如何翻墙

使用VPN（虚拟专用网络）翻墙啊，推荐用LoCoVPN，下载客户端，使用客户端连接。

基本磁盘和动态磁盘的区别

基本磁盘和动态磁盘是Windows中的两种硬盘配置类型，大多数个人计算机都配置为基本磁盘，该类型最易于管理。

基本磁盘：

受26个字母限制，盘符只能从C到Z（A\B被软驱占用）。基本磁盘上在一个硬盘上只能最多建立四个主分区，分区必须相邻。基本磁盘一旦分区不能更改大小，除非借助第三方工具。

动态磁盘：

动态磁盘不再采用基本磁盘的分区方式，而是叫做卷集。分为简单卷、跨区卷、带区卷、镜像卷和RAID-5卷。动态磁盘在一个硬盘上可创建的卷集数没有限制。动态磁盘可以把几个不同的硬盘建成一个卷，并且这些分区可以非相邻。动态磁盘可以不重启机器的情况下调整分区大小，并且不会丢失数据。动态磁盘还有容错功能。

基本磁盘可以直接转换为动态磁盘，但该过程不可逆。想要转回基本磁盘，只有把数据全部拷出，然后删除硬盘所有分区后才能转回去。

可在基本磁盘上创建的分区个数取决于分区形式是MBR还是GPT。

基本磁盘、动态磁盘针对的对象时整个磁盘。GPT和MBR是针对磁盘的某个分区的分区方式。GPT\MBR和基本磁盘\动态磁盘根本就是两个不同的概念，互相没有联系！！！！

GPT磁盘与MBR磁盘的区别

在使用新磁盘之前，必须对其进行分区。MBR和GPT是在磁盘上存储分区信息的两种不同的方式。

GPT是一种新的标准，并在逐渐取代MBR，但MBR仍拥有最好的兼容性。MBR已经成为磁盘分区和启动的工业标准。

MBR的局限性：

MBR的意思是“主引导记录”，MBR支持最大2TB硬盘，无法处理大于2TB容量的磁盘。MBR还只支持最多4个主分区（如果你想要更多分区，你需要创建所谓“扩展分区”，并在其中创建逻辑分区）。

GPT的优势：

GPT意为“GUID分区表”（GUID意为全局唯一标识符）。与UEFI相辅相成（UEFI取代老旧的BIOS）。

可以保证你的驱动器上每个分区都有一个全球唯一的标识符。

磁盘容量可以超级大，大到操作系统不支持，支持无限个分区数量，限制在于操作系统——Windows支持最多128个GPT分区，而且不需要创建扩展分区。

在MBR磁盘上，分区和启动信息是保存在一起的，如果这部分数据被损坏，就无法启动计算机了。但GPT在磁盘的上保存多个这部分信息的副本，如果数据被破坏，可以进行修复。

PHP、CSS、HTML、JS之间的关系

HTML是一种标签语言，静态页面，由客户端的浏览器负责解析。

CSS是一种样式控制，也就是如何定义一个网页的布局、颜色等外观，也由浏览器负责解析。

JS是一种客户端动态脚本，使用户可以控制页面上的动态内容显示，也就是用户交互。

PHP是一种服务端动态语言，最终会动态生成html供客户端浏览器解析。

总结：

html,css,js都是客户端语言，都是由浏览器解析执行。php是服务端语言（和java,C#一样是编程语言的一种，做Web开发都可以用），运行在远程服务器上，其最终需要生成html才可以被浏览器识别。

形象比喻：

简单的说就像是装修房子一样，装修队就是浏览器

你把你的设计（html==装修图纸）告诉浏览器，它就会按照你给的装修图纸进行房屋的摆设和装饰。

所以html包含了两大块，一是屋子里是怎么样摆设，二摆设是需要怎样的装饰点缀，那么他们就分别是html基本元素和css层叠样式表。也就是说css对html进行了详细的说明，是对细节的描写。

js则可以根据客户的需求动态的改html里的摆设和装饰。

html包含了css和js形成了完成的装修方案，是属于前端的描述语言和脚本

而php是在服务器端运行的脚本，通过与数据库和其他组件进行交互的操作，可以动态的生成多套设计方案即html(装修图纸)

WAMP,LAMP,LNMP的区别

以上都是一个集成的Web应用程序平台。

WAMP是Windows下的Apache+Mysql+PHP

LAMP是Linux下的Apache+Mysql+PHP

LNMP是Linux下的Nginx+Mysql+PHP

Apache是世界上使用排名第一的Web服务器软件，Nginx是一个高性能的HTTP和反向代理服务器。

MySQL是关系型数据库管理系统。

目前有不少AMP的集成软件，可以让我们一次性安装并设置好，是绝佳的一站式环境配置，例如WampServer。

HTTPS和HTTP的区别

HTTP即超文本传输协议是一种详细规定了浏览器和万维网服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。

网站的访问都要先看协议，协议不一样，则访问不了。

HTTPS是以安全为目标的HTTP通道，简单讲就是HTTP的安全版。即HTTP下加入SSL层。HTTPS存在不同于HTTP的默认端口以及一个加密身份验证层（在HTTP于TCP之间）。

HTTPS和HTTP的区别主要为以下四点：

- 1、https协议需要到ca申请证书，一般免费证书很少，需要交费。
- 2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
- 3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
- 4、http的连接很简单，是无状态的，https协议是由ssl+http协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

POST和GET的区别

来自 <<http://www.cnblogs.com/hyddd/archive/2009/03/31/1426026.html>>

Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是GET，POST，PUT，DELETE。URL全称是资源描述符，我们可以这样认为：一个URL地址，它用于描述一个网络上的资源，而HTTP中的GET，POST，PUT，DELETE就对应着对这个资源的查，改，增，删4个操作。到这里，大家应该有个大概的了解了，GET一般用于获取/查询资源信息，而POST一般用于更新资源信息。

1. 根据HTTP规范，GET用于信息获取，而且应该是安全的和幂等的。

(1). 所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，GET 请求一般不应产生副作用。就是说，它仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。

* 注意：这里安全的含义仅仅是指是非修改信息。

(2). 幂等的意味着对同一URL的多个请求应该返回同样的结果。这里我再解释一下幂等这个概念：

```
/*
```

幂等 (idempotent、idempotence) 是一个数学或计算机学概念，常见于抽象代数中。

幂等有以下几种定义：

对于单目运算，如果一个运算对于在范围内的所有的一个数多次进行该运算所得的结果和进行一次该运算所得的结果是一样的，那么我们就称该运算是幂等的。比如绝对值运算就是一个例子，在实数集中，有 A 的绝对值= A 的绝对值的绝对值。

对于双目运算，则要求当参与运算的两个值是等值的情况下，如果满足运算结果与参与运算的两个值相等，则称该运算幂等，如求两个数的最大值的函数，有在在实数集中幂等，即 $\max(x, x) = x$ 。

```
*/
```

看完上述解释后，应该可以理解GET幂等的含义了。

但在实际应用中，以上2条规定并没有这么严格。引用别人文章的例子：比如，新闻站点的头版不断更新。虽然第二次请求会返回不同的新闻，该操作仍然被认为是安全的和幂等的，因为它总是返回当前的新闻。从根本上说，如果目标是当用户打开一个链接时，他可以确信从自身的角度来看没有改变资源即可。

2. 根据HTTP规范，POST表示可能修改改变服务器上的资源的请求。继续引用上面的例子：还是新闻以网站为例，读者对新闻发表自己的评论应该通过POST实现，因为在评论提交后站点的资源已经不同了，或者说资源被修改了。

上面大概说了一下HTTP规范中GET和POST的一些原理性的问题。但在实际的做的时候，很多人却没有按照HTTP规范去做，导致这个问题的原因有很多，比如说：

1. 很多人贪方便，更新资源时用了GET，因为用POST必须要到FORM（表单），这样会麻烦一点。
2. 对资源的增，删，改，查操作，其实都可以通过GET/POST完成，不需要用到PUT和DELETE。
3. 另外一个是，早期的Web MVC框架设计者们并**没有有意识地将URL当作抽象的资源来看待和设计**，所以导致一个比较严重的问题是传统的Web MVC框架基本上只支持GET和POST两种HTTP方法，而不支持PUT和DELETE方法。

以上3点典型地描述了老一套的风格（没有严格遵守HTTP规范），随着架构的发展，现在出现REST(Representational State Transfer)，一套支持HTTP规范的新风格，这里不多说了，可以参考《RESTful Web Services》。

简单解释一下MVC：

MVC本来是存在于Desktop程序中的，M是指数据模型，V是指用户界面，C则是控制器。使用MVC的目的是将M和V的实现代码分离，从而使同一个程序可以使用不同的表现形式。

说完原理性的问题，我们再从表面现像上面看看GET和POST的区别：

1. GET请求的数据会附在URL之后（就是把数据放置在HTTP协议头中），以?分割URL和传输数据，参数之间以&相连，如：login.action?name=hyddd&password=idontknow&verify=%E4%BD%A0%E5%A5%BD。如果数据是英文字母/数字，原样发送，如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用BASE64加密，得出如：%E4%BD%A0%E5%A5%BD，其中%XX中的XX为该符号以16进制表示的ASCII。

POST把提交的数据则放置在是HTTP包的包体中。

2. “**GET方式提交的数据最多只能是1024字节，理论上POST没有限制**，可传较大量的数据，IIS4中最大为80KB，IIS5中为100KB”？？！

以上这句是我从其他文章转过来的，其实这样说是错误的，不准确的：

(1). 首先是“**GET方式提交的数据最多只能是1024字节**”，因为GET是通过URL提交数据，那么GET可提交的数据量就跟URL的长度有直接关系了。而实际上，**URL不存在参数上限的问题，HTTP协议规范没有对URL长度进行限制**。这个限制是特定的浏览器及服务器对它的限制。IE对URL长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持。

注意这是限制是整个URL长度，而不仅仅是你的参数值数据长度。[见参考资料5]

(2). 理论上讲，**POST是没有大小限制的，HTTP协议规范也没有进行大小限制**，说“**POST数据量存在80K/100K的大小限制**”是不准确的，POST数据是没有限制的，起限制作用的是服务器的处理程序的处理能力。

对于ASP程序，Request对象处理每个表单域时存在100K的数据长度限制。但如果使用Request.BinaryRead则没有这个限制。

由这个延伸出去，对于IIS 6.0，微软出于安全考虑，加大了限制。我们还需要注意：

- 1). IIS 6.0默认ASP POST数据量最大为200KB，每个表单域限制是100KB。
- 2). IIS 6.0默认上传文件的最大大小是4MB。
- 3). IIS 6.0默认最大请求头是16KB。

IIS 6.0之前没有这些限制。[见参考资料5]

所以上面的80K，100K可能只是默认值而已(注：关于IIS4和IIS5的参数，我还没有确认)，但肯定是可以自己设置的。由于每个版本的IIS对这些参数的默认值都不一样，具体请参考相关的IIS配置文档。

3. 在ASP中，服务端获取GET请求参数用Request.QueryString，获取POST请求参数用Request.Form。在JSP中，用request.getParameter(\"XXXX\")来获取，虽然jsp中也有request.getQueryString()方法，但使用起来比较麻烦，比如：传一个test.jsp?name=hyddd&password=hyddd，用request.getQueryString()得到的是：name=hyddd&password=hyddd。在PHP中，可以用\$_GET和\$_POST分别获取GET和POST中的数据，而\$_REQUEST则可以获取GET和POST两种请求中的数据。值得注意的是，JSP中使用request和PHP中使用\$_REQUEST都会有隐患，这个下次再写个文章总结。

4. **POST的安全性要比GET的安全性高**。注意：这里所说的安全性和上面GET提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而**这里安全的含义是真正的Security的含义**，比如：**通过GET提交数据，用户名和密码将明文出现在URL上**，因为(1)登录页面有可能被浏览器缓存，(2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了，除此之外，使用GET提交数据还可能会造成Cross-site request forgery攻击。

总结一下，Get是向服务器发索取数据的一种请求，而Post是向服务器提交数据的一种请求，在FORM（表单）中，Method默认为“GET”，实质上，GET和POST只是发送机制不同，并不是一个取一个发！

MVC模式

来自 <http://baike.baidu.com/link?url=hYCYH4VlmoiE8SYen_iArQJ7ZuvOYCwB4PBxX3sd7fN61NI0egZ6t18DT9qCtGU0yYgYxWPUunWSOCUFO9eXC168Wf6TeVogQodyW5TukPCwha5CgMpSwXPaGvhulGXGsKe5oA0-yjWLhGlgl7AWLnS32DHQsHJhyOxi52ljmh3>

MVC全名是model view controller，M代表业务模型，V代表用户界面，C代表控制器。是一种软件架构模式。

用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。使用MVC的目的是使M和V的代码分离，从而使一种程序有不同的表现形式。C的目的是确保M和V同步，一旦M改变，V应该同步刷新。

Model (模型) 是应用程序中用于处理应用程序数据逻辑的部分。

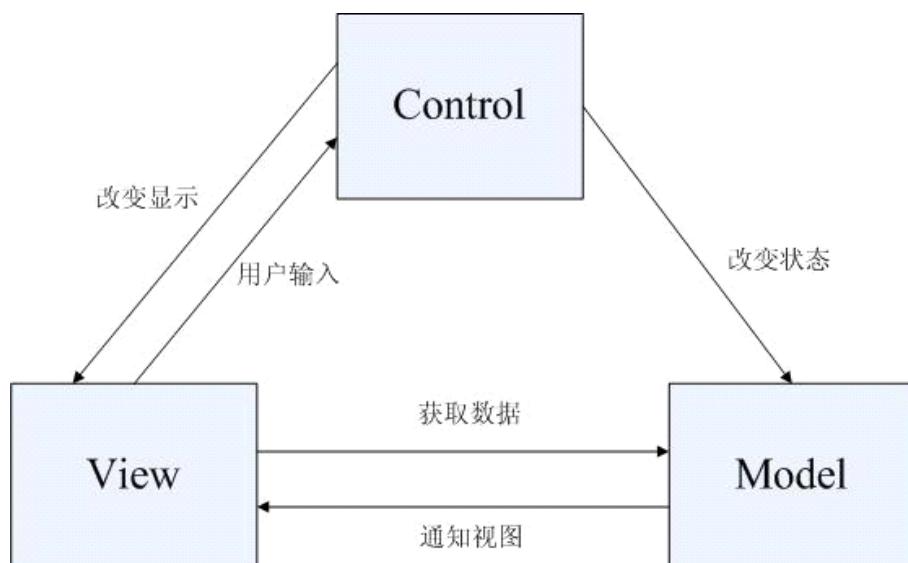
通常模型对象负责在数据库中存取数据。

View (视图) 是应用程序中处理数据显示的部分。

通常视图是依据模型数据创建的。

Controller (控制器) 是应用程序中处理用户交互的部分。

通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。



Web开发中的cookie和session

Cookie：

Cookies是一种能够让网站服务器把少量数据储存到客户端的硬盘或内存，或是从客户端的硬盘读取数据的一种技术。Cookies是当浏览某网站时，由WEB服务器置于硬盘上的一个非常小的文本文件，它可以记录用户ID、密码、浏览过的网页、停留的时间等信息。当再次来到该网站时，网站通过读取cookies,得知你的相关信息，就可以做出相应的动作，如在页面显示欢迎你的标语，或者让你不用输入ID、密码就直接登录等。

从本质上讲，它可以看作是你的身份证件。但cookie不能作为代码执行，也不会传送病毒，且为你所专有，并只能由提供它的服务器来读取。一个网站只能取得它放在你的电脑中的信息，它无法从其它的cookies文件中取得信息，也无法得到你的电脑上的其它任何东西。Cookies中的内容大多数经过了加密处理，因此一般用户看来只是一些毫无意义的字母数字组合，只有服务器的CGI处理程序才知道它们真正的含义。

服务器通过在HTTP的响应头中加上一行特殊的指示以提示浏览器按照指示生成相应的cookie。然而纯粹的客户端脚本，如JAVASCRIPT或者VBSCRIPT也可以生成cookie。而cookie的使用是由浏览器按照一定的原则在后台自动发送给服务器的。浏览器检查所有存储的cookie，如果某个cookie所声明的作用范围大于等于将要请求的资源所在的位置，则把该cookie附在请求资源的HTTP请求头上发送给服务器。

Cookie的内容主要包括：名字、值、过期时间、路径和域。路径与域一起构成cookie的作用范围。若不设置过期时间，则表示这个cookie的生命期为浏览器会话期间，关闭浏览器窗口，cookie就消失。这种生命期为浏览器会话期的cookie被称为会话cookie。会话cookie一般不会存储在硬盘上而是保存在内存里。若设置了过期时间，浏览器就会把cookie保存到硬盘上，关闭后再次打开浏览器，这些cookie仍然有效直到超过设定的过期时间。存储在硬盘上的cookie可以在不同的浏览器进程间共享，如两个IE窗口。而对于保存在内存里的cookie，不同的浏览器有不同的处理方式。

Session:

session机制是一种服务器端的机制，当用户请求来自应用程序的 Web 页时，如果该用户还没有会话，则 Web 服务器将自动创建一个 Session 对象。当会话过期或被放弃后，服务器将终止该会话。

当程序需要为某个客户端的请求创建一个session时，服务器首先检查这个客户端的请求里是否已包含了一个session标志（称为session id），如果已包含则说明以前已经为此客户端创建过session，服务器就按照session id把这个session检索出来使用（检索不到，会新建一个）；如果客户端请求不包含session id，则为此客户端创建一个session并且生成一个与此session相关联的session id，session id的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个session id 将被在本次响应中返回给客户端保存。

由于采用服务器端保持状态的方案在客户端也需要保存一个标志，所以session机制可能需要借助于cookie机制来达到保存标志的目的，但实际上它还有其他选择。

Cookie和Session的区别：

- 1、Cookie保存在客户端，Session保存在服务端。
- 2、Cookie不是很安全，因为存放在本地。
- 3、单个Cookie的数据不能超过4K，很多浏览器都限制一个站点最多保存50个Cookie。

常见的Web安全漏洞

SQL名词解释：

SQL (Structured Query Language) 即结构化查询语言，是一种特殊目的的编程语言，是一种数据库查询和程序设计语言，用于存取数据以及查询、更新和管理关系数据库系统。

SQL注入

概述：

就是通过把SQL命令插入到WEB表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意SQL命令。

可能原因：

未对用户输入执行正确的危险字符清理

技术描述：

它是利用现有的程序，将恶意的SQL命令注入到后台数据库，它可以通过WEB表单中输入恶意SQL语句得到一个存在安全漏洞的网站上的数据库。

Web 应用程序通常在后端使用数据库，以与企业数据仓库交互。查询数据库事实上的标准语言是 SQL（各大数据库供应商都有自己的不同版本）。Web 应用程序通常会获取用户输入（取自 HTTP 请求），将它并入 SQL 查询中，然后发送到后端数据库。接着应用程序便处理查询结果，有时会向用户显示结果。

如果应用程序对用户（攻击者）的输入处理不够小心，攻击者便可以利用这种操作方式。在此情况下，攻击者可以注入恶意的数据，当该数据并入 SQL 查询中时，就将查询的原始语法更改得面目全非。例如，如果应用程序使用用户的输入（如用户名和密码）来查询用户帐户的数据库表，以认证用户，而攻击者能够将恶意数据注入查询的用户名部分（和/或密码部分），查询便可能更改成完全不同的数据复制查询，可能是修改数据库的查询，或在数据库服务器上运行 Shell 命令的查询。

例子：

程序从Http请求中读取一个sql查询

```
String sql ="SELECT * FROM account WHERE name = 'Bob' AND password = '123'"  
stmt.execute(sql)
```

在执行execute(sql)之前并未对输入的字符串进行检查，因此存在SQL注入弱点，如果在password="123"后加上or'1=1'，那么这样的SQL注入会使得输入任意密码进入程序。

防范：

- 永远不要信任用户的输入，要对用户的输入输出进行校验，可以通过正则表达式，或限制长度，或者使用过滤函数，对单引号和双“”进行转换等。
- 永远不要使用动态拼装SQL，可以使用参数化的SQL或者直接使用存储过程进行数据查询存取。
- 永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的数据库连接。

4. 不要把机密信息明文存放，请加密或者hash掉密码和敏感的信息。 5. 应用的异常信息应该给出尽可能少的提示，最好使用自定义的错误信息对原始错误信息进行包装，把异常信息存放在独立的表中。

路径遍历

概述：

可能会查看WEB服务器（在WEB服务器用户的许可权限制下）上的任何文件（例如数据库、用户信息或配置文件）的内容。

可能原因：

未对用户输入执行正确的危险字符清理

未检查用户输入中是否包含“..”（两个点）字符串

技术描述：

CGI 脚本通常包含指定作为模板显示或使用的文件的参数。如果应用程序未验证为脚本提供的文件名，那么攻击者可能会操纵该参数，并请求驻留于服务器上的其他文件。

示例：

[原始的 HTML 表单]

```
<FORM METHOD=POST ACTION="/cgi-bin/vulnerable_script.cgi">  
...  
<INPUT TYPE=HIDDEN NAME="template" VALUE="/dir1/dir2/template.txt">  
...  
</FORM>
```

[受操纵的 HTML 表单]

```
<FORM METHOD=POST  
ACTION="http://target/cgi-bin/vulnerable_script.cgi">  
...  
<INPUT TYPE=HIDDEN NAME="template" VALUE="../../../../boot.ini">  
...  
</FORM>
```

如此一来，应用程序在将表单提交回服务器时，会为攻击者提供 boot.ini 文件。

跨站点脚本 (xss)

概述：

它指的是恶意攻击者往WEB页面或者客户端脚本的页面里插入恶意html代码，当用户浏览该页时，嵌入其中WEB里面的html代码会被执行，从而达到恶意用户的特殊目的。

可能原因：

WEB应用程序使用客户端创建的WEB页面

技术描述：

Stored XSS（存储式跨站脚本攻击）这是最强大的一种XSS攻击，所谓存储跨站攻击是指用户提交给Web应用程序的数据首先就被永久的保存在服务器的数据库，文件系统或其他地方，后面且未做任何编码就能显示到Web页面。

错误认证和会话管理

概述：

“遭破坏的认证和会话管理”，简而言之，就是攻击者窃听了我们访问http时的用户名和密码，或者是我们的会话，从而得到sessionID，进而冒充用户进行http访问的过程。

可能原因：

WEB应用程序将敏感的会话信息存储在永久Cookie中（磁盘上）

明文传输

技术描述：

由于HTTP本身是无状态的，也就是说HTTP的每次访问请求都是带有个人凭证的，而SessionID就是为了跟踪状态的，而sessionID本身是很容易在网络上被监听的到，所以攻击者往往通过监听sessionID来达到进一步攻击的目的。

跨站请求伪造（CSRF）

概述：

尽管听起来像跨站脚本（XSS），但它与XSS非常不同，并且攻击方式几乎相左。XSS利用站点内的信任用户，而CSRF则通过伪装来自受信任用户的请求来利用受信任的网站。与XSS攻击相比，CSRF攻击往往不大流行（因此对其进行防范的资源也相当稀少）和难以防范，所以被认为比XSS更具危险性。

技术描述：

在用户会话下对某个CGI做一些GET/POST的事情——这些事情用户未必知道和愿意做，你可以把它想做HTTP会话劫持。攻击通过在授权用户访问的页面中包含链接或者脚本的方式工作。

假设你是一台存储了丰富内容的计算机，这些内容分为秘密、公开两类，秘密信息只能让A用户查看，公开信息允许任何人查看。你和用户之间通过线缆通信，这些线缆有可能丢失、搞错信息，更糟糕的是其他人也有可能窃取线缆中的信息，现在，试着制定一套你和用户们都遵守的规章制度，让你可以为A和其他用户提供安全、准确的服务。（需要详细阐述如何处理安全和准确这两个问题）

解答：

公开信息允许任何人查看，直接进行明文传输。

秘密信息因为只能让A用户查看，为了安全性，我（计算机、服务端）首先生成一个随机的对称密钥，然后用该对称密钥对明文信息进行加密。下一步则是如何将对称密钥传

给用户A，这时候公钥密码就可以完成该任务，只需要使用接受者的公钥加密随机的对称密钥即可。这个公钥应该是用户A自己在通信开始前知道的，也相当于A的身份背书，A不应该将自己的密钥告诉别人，否则相当于把银行卡密码告诉了别人。通信开始的过程如下：

A→服务端：我是用户A，我的公开密钥是Ea，你选择一个对称密钥K，用Ea加密后传送给
我；

服务端→A：确定Ea密钥是正确的，相当于确认用户密码，使用Ea加密对称密钥K；

A→服务端：使用K加密传输信息；

服务端→A：使用K加密传输信息。

但是题目中设定有人可能窃取线缆中的信息，也就是恶意用户可能窃取到A的公开密钥Ea，所以存在安全隐患。所以这就需要能被用户信任的第三方机构来提供用户身份验证，即CA机构。因此在上述通信前双方需要获得CA机构颁发的数字证书，并验证证书的完整性、可信性，再进行通信。

为了保证信息的完整性。以上传输都可以再加上一层CRC校验，CRC校验的优点在于信息字段和校验字段的长度可以任意选定。在通信过程中，如果CRC校验错误，则重新请求，直到校验正确再返回数据。

TCP/IP

TCP/IP是因特网的通信协议，通信协议是计算机必须遵守的规则，只有遵守这些规则，计算机之间才能进行通信。

在TCP/IP中包含一系列用于处理数据通信的协议：

- TCP (传输控制协议) - 应用程序之间通信，在双方“握手”后建立连接。
- UDP (用户数据包协议) - 应用程序之间的简单通信，无连接的，可靠性稍低。
- IP (网际协议) - 计算机之间的通信，是无连接的，不占用通信线路，当一个IP包从一台计算机被发送，它会到达一个IP路由器，再由IP路由器将这个包路由至它的目的地。每个包的路径可能不同。
- ICMP (因特网消息控制协议) - 针对错误和状态
- DHCP (动态主机配置协议) - 针对动态寻址

TCP/IP意味着TCP和IP在一起协同工作，TCP负责将数据分割并装入IP包，然后在它们到达的时候重新组合。IP负责将包发送给接受者。

一个字节=8比特，因此IP地址使用了4个字节

cmd 命令行提示符

cd\ 打开根目录

cd.. 退回上一级目录

md xxx 创建xxx文件夹

rd xxx 删除xxx文件夹

dir 列出当前目录下的文件夹

cd xxx\xx 打开xxx中的xx文件夹

del xxx 删除文件

exit 退出DOS命令行

set 查看或者定义环境变量的值

Set path 先找当前目录，再找path设置好的目录

Set path=%path% 添加了原path

Set classpath=xxx 设置环境变量

Set classpath=xxx; 添加了分号，找了指定目录的环境变量后，还会在当前目录找环境变量。

但通常不建议加！

Set classpath=xxx;xxxxx 先找xxx，再找xxxxx

Set classpath=. 点代表当前路径

Set classpath= 什么也不写，就清空环境变量

Linux命令提示符

<http://blog.csdn.net/ljianhui/article/details/11100625/>

常见面试问题1

2018年3月11日 17:30

目录

[如果你的项目出现了内存泄露，怎么监控这个问题呢](#)

[走格子，从一个出发点到终点，只能向上和向右有多少种走法，连障碍点都没有](#)

[mysql索引，最左匹配原则](#)

[redis为什么要使用单线程](#)

[int范围](#)

[Java有哪些后端技术](#)

[Redis如何保证和MySQL数据一致？实时同步](#)

[在客户端抓包，看到的是加密的还是没加密的](#)

[Synchronized底层实现原理](#)

[缓冲区溢出](#)

[流式计算和批量计算](#)

[Java重载和重写](#)

[Java数组和链表的区别](#)

[CPU负载100%查找方案](#)

[MD5加密](#)

[Object类的方法有哪些](#)

[如果你的项目出现了内存泄露，怎么监控这个问题呢](#)

- 利用内存分析工具MAT，比如一个项目内存使用率每天都会增加一点。对于一个稳定运行的java项目而言，出现这种情况一般都有可能是出现了内存泄露。
- 通过jmap获取堆转储文件，然后scp到本地，然后MAT软件加载。
- 为了找到内存泄露，可以获取两个堆转储文件，两个文件获取时间间隔是一天（因为内存只是小幅度增长，短时间很难发现问题）。对比两个文件的对象，通过对比后的结果可以很方便定位内存泄露。

[怎么优化这段代码](#)

```

List<Integer> list = new ArrayList<Integer>();
for (int i = 0; i < 100; i++) {
    list.add(i);
}
StringBuilder sb = new StringBuilder();
for (Integer num: list) {
    sb = sb.append("当前数字为:").append(num).append("<br/>");
}
System.out.println(sb.toString());

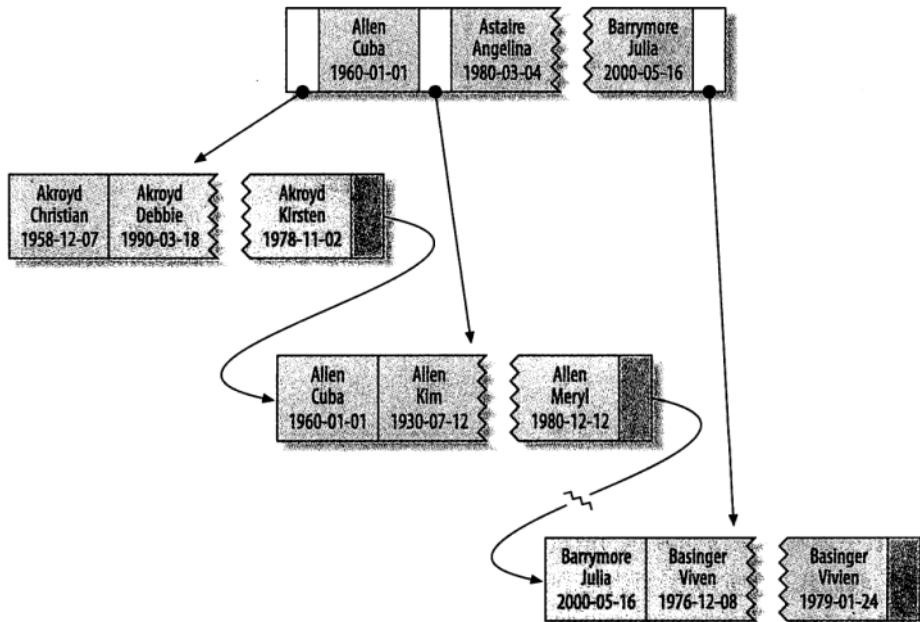
```

- 我只能想到，创建list的时候给定长度100
- 因为默认长度10，容量变化规则是
 $((旧容量 * 3) / 2) + 1$
- 一旦容量发生变化，就要带来额外的内存开销，和时间上的开销

走格子，从一个出发点到终点，只能向上和向右有多少种走法，连障碍点都没有

N行M列，从N-1中选一个，从M-1个选一个，因此是 $(N-1) \times (M-1)$

mysql索引，最左匹配原则



如上图建了复合索引 (`firstname,lastname,birthday`)

比如`select * from test where firstname=XXX and lastname=YYY and birthday=ZZZ;`

mysql也只匹配最左边的`firstname`，即找到所有`firstname=XXX`的人。

并且mysql会自动优化子句顺序，即使我写成如下

`select * from test where lastname=1 and firstname=1 and birthday=1;`即`lastname`在前面，`mysql`也会利用最左边的索引`firstname`

redis为什么要使用单线程

为了原子性操作

int范围

2的-31次方到2的31次方-1, 42到43亿

Java有哪些后端技术

主流: Nginx+tomcat+mybatis+redis

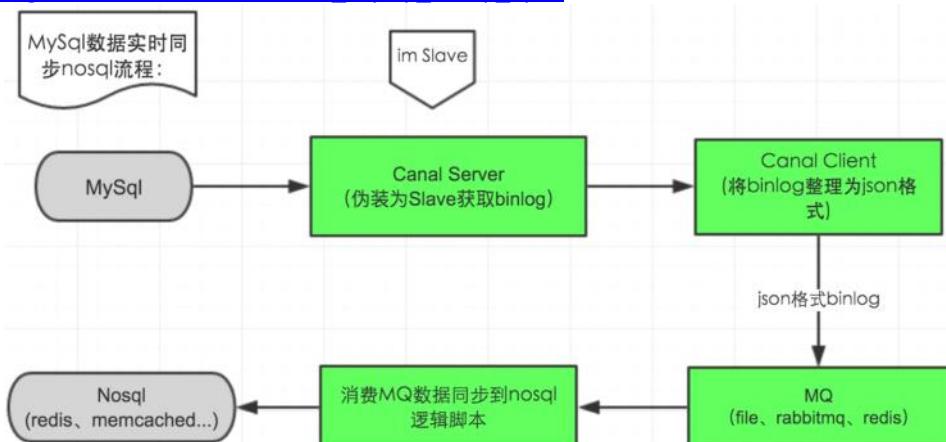
分类:

- 服务框架: Dubbo, zookeeper, Rest服务
- 缓存: redis, memecache
- 消息中间件: RabbitMQ, ActiveMQ, kafka、
- 负责均衡: Nginx
- 分布式文件: FastDFS
- 安全框架: Apache shiro
- 任务调度: quartz
- 持久层框架: mybatis
- 日志: log4j
- 项目基础搭建。spring, springmvc,
- 环境搭建: linux下,
- 开发工具: eclipse。idea等
- 服务器: tomcat, jetty等

Redis如何保证和MySQL数据一致? 实时同步

一句话解释就是: MySQL binlog增量订阅消费+消息队列+处理并把数据更新到redis

https://github.com/liukelin/canal_mysql_nosql_sync



名词解释:

- Mysql的**binlog日志**作用是用来记录mysql内部增删改查等对mysql数据库有更新的内容的记录

- 阿里开发的Canal 会将自己伪装成 MySQL 从节点 (Slave) , 并从主节点 (Master) 获取 Binlog, 解析和贮存后供下游消费端使用。

流程详解：

1. mysql主从配置
- 2.对mysql binlog(row) 解析 这一步交给canal
- 3.MQ对解析后binlog增量数据的推送
- 4.对MQ数据的消费 (接收+数据解析, 考虑消费速度, MQ队列的阻塞)
- 5.数据写入/修改到nosql (redis的主从/hash分片)
- 6.保证对应关系的简单性：一个mysql表对应一个 redis实例 (redis单线程, 多实例保证分流不阻塞) , 关联关系数据交给接口业务

数据： mysql->binlog->MQ->redis(不过期、关闭RDB、AOF保证读写性能) (nosql 数据仅用crontab脚本维护)

请求： http->webserver->redis(有数据)->返回数据 (完全避免用户直接读取 mysql) ->redis(无数据)->返回空

- 7.可将它视为一个触发器, binlog为记录触发事件, canal的作用是将事件实时通知出来, 并将binlog解析成了所有语言可读的工具。

在事件传输的各个环节 提高 可用性和 扩展性 (加入MQ等方法) 最终提高系统的稳定。

为什么要使用消息队列 (MQ) 进行binlog传输

- 1.增加缓冲, binlog生产端 (canal client) 只负责生产而不需要考虑消费端的消费能力, 不等待阻塞。
- 2.binlog 消费端: 可实时根据MQ消息的堆积情况, 动态增加/减少 消费端的数量, 达到合理的资源利用和消费

在客户端抓包, 看到的是加密的还是没加密的

没加密的

Synchronized底层实现原理

对象的同步Synchronized的底层是通过monitor来完成

每个对象有一个监视器锁 (monitor) 。当monitor被占用时就会处于锁定状态, 线程执行 **monitorenter**指令时尝试获取monitor的所有权, 过程如下:

1. 如果monitor的进入数为0, 则该线程进入monitor, 然后将进入数设置为1, 该线程即为monitor的所有者。
2. 如果线程已经占有该monitor, 只是重新进入, 则进入monitor的进入数加1.
3. 如果其他线程已经占用了monitor, 则该线程进入阻塞状态, 直到monitor的进入数为0, 再重新尝试获取monitor的所有权。

释放锁则是通过**monitorexit**指令, 执行monitorexit的线程必须是objectref所对应的 monitor的所有者, 指令执行时, monitor的进入数减1, 如果减1后进入数为0, 那线程退出 monitor, 不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

方法的synchronized同步

相对于普通方法，其常量池中多了ACC_SYNCHRONIZED标志。JVM就是根据该标志来实现方法的同步的：当方法调用时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。

缓冲区溢出

计算机对接收的输入数据没有进行有效的检测，向缓冲区（分配内存）内填充数据时超过了缓冲区本身的容量，使得溢出的数据覆盖了其他内存空间的数据。

缓冲区溢出攻击，可以导致程序运行失败、系统关机、重新启动，或者执行攻击者的指令，比如非法提升权限

流式计算和批量计算

批量计算：“收集数据 - 放到DB中 - 取出来分析”的传统的流程

流式计算：对数据流进行实时计算，它不是更快的批量计算，每次小批量计算，结果可以立刻反馈到在线系统，数据是不断无终止的，数据计算完之后就丢弃。

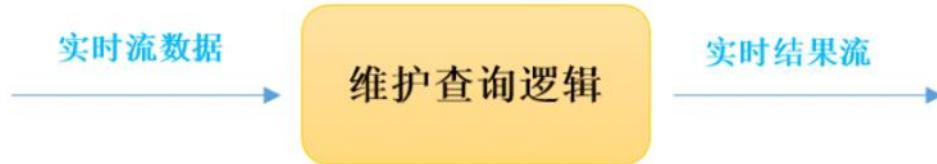
TIPS：

批量计算是维护一张表，对表进行实施各种计算逻辑。流式计算相反，是必须先定义好计算逻辑，提交到流式计算系统，这个计算作业逻辑在整个运行期间是不可更改的

批量计算



流式计算



Java重载和重写

重写：

- java中有很多的继承，继承下来的有变量、方法。在有一些子类要实现的方法中，方法名、传的参数、返回值跟父类中的方法一样，但具体实现又跟父类的不一样，这时候我们就需要重写父类的方法。
- 就比如我们有一个类叫做Animals， Animals类中有一个叫做Call,然后我们继承Animals又生成了Cat类和Dog类，各自输出不同的叫声，代码如下：

```
class Animals {  
    public void call() {  
        System.out.println("啊啊啊啊啊啊啊啊");  
    }  
}  
  
public class Cat extends Animals {  
    @Override  
    public void call() {  
  
        System.out.println("喵喵喵喵喵");  
    }  
}  
  
public class Dog extends Animals {  
  
    @Override  
    public void call() {  
        System.out.println("汪汪汪汪汪");  
    }  
}
```

重载：重载是在一个类中实现的，有多个同名方法，但参数不一样，包括参数类型、参数个数、还可以没有参数，总之每个重载的方法的参数必须不一样。比如构造器重载。

CPU负载100%查找方案

方案一：

- 1、 top命令查看哪些进程CPU占用率高，找到进程PID
- 2、 top -Hp PID命令找到这个程序的所有线程，找到占用率高的线程PID

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|------|------|----|----|--------|------|-----|---|------|------|----------|-----------|
| 6248 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 34:43.38 | AliYunDun |
| 6249 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 5:00.86 | AliYunDun |
| 6250 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 5:02.08 | AliYunDun |
| 6252 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 5:20.23 | AliYunDun |
| 6253 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 14:31.53 | AliYunDun |
| 6254 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 4:52.14 | AliYunDun |
| 6255 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 0:03.23 | AliYunDun |
| 6256 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 0:03.15 | AliYunDun |
| 6257 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 3:57.04 | AliYunDun |
| 6258 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 17:12.81 | AliYunDun |
| 6259 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 5:18.40 | AliYunDun |
| 6260 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 0:14.47 | AliYunDun |
| 6261 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 32:46.32 | AliYunDun |
| 6262 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 1:38.68 | AliYunDun |
| 6263 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 11:41.03 | AliYunDun |
| 7384 | root | 20 | 0 | 134260 | 7032 | 772 | S | 0.0 | 0.7 | 2:51.53 | AliYunDun |

- 3、 printf “0x%lx\n”线程PID命令， 将线程PID转换为 16进制，为后面查找 jstack 日志做准备
- 4、 jstack 进程PID | vim +/十六进制线程PID - // 例如：jstack 1040|vim +/0x431 - 打

印线程的堆栈信息，在此就可以查看造成CPU异常是由什么原因了。

方案二：

用top命令找到进程后，用perf工具可以查看具体哪个函数占用了CPU

MD5加密

特点：

1. 不管多长的字符串,加密后长度都是一样长
2. 一个文件,不管多大,小到几k,大到几G,你只要改变里面某个字符,那么都会导致MD5值改变
3. 你明明知道密文和加密方式,你却无法反向计算出原密码

撞库破解：

这是概率极低的破解方法，原理是建立一个大型的数据库，把日常的各个语句，通过MD5加密成为密文，不断的积累大量的句子，放在一个庞大的数据库里

除了MD5其他的加密：

- CRC32只输出32bit长度
- MD5输出128bit
- SHA1输出160bit
- SHA256输出256bit
- SHA1, SHA256这样计算相对复杂的算法，会慢很多，一般MD5够用了

Object类的方法有哪些

getClass hashCode equals clone toString notify notifyAll wait finalize

Spring相关面试问题

2018年3月19日 10:03

目录

[Spring的线程安全](#)

[Servlet的线程安全](#)

[Spring Bean的生命周期](#)

[为什么用Spring IOC? 好处?](#)

[SpringAOP实现原理](#)

Spring的线程安全

- 1、Spring MVC (Springboot) 开发的web项目，**默认的Controller, Service, Dao组件的作用域都是单例模式，无状态的，因此是线程安全的**
- 2、无状态的Bean适合用不变模式，技术就是单例模式，这样可以共享实例，提高性能。有状态的Bean，多线程环境下不安全，那么适合用Prototype原型模式。Prototype: 每次对bean的请求都会创建一个新的bean实例。
- 3、默认情况下，从Spring bean工厂所取得的实例为singleton (scope属性为singleton)，容器只存在一个共享的bean实例。
- 4、理解了两者的关系，那么scope选择的原则就很容易了：有状态的bean都使用prototype作用域，而对无状态的bean则应该使用singleton作用域。

Servlet的线程安全

- 1、ServletContext、HttpSession是线程安全的；ServletRequest是非线程安全的
- 2、Servlet是否线程安全是由它的实现来决定的，如果它内部的属性或方法会被多个线程改变，它就是线程不安全的，反之，就是线程安全的。
- 3、Spring是一种线程安全的Servlet实现

Spring Bean的生命周期

单例对象

1. 默认情况下,spring在读取xml文件的时候,就会创建对象
2. spring对bean进行依赖注入
3. 此时bean已经准备就绪,可以被应用程序使用了,他们将一直驻留在应用上下文中,直到该应用上下文被销毁

非单例对象

1. spring读取xml文件的时候,不会创建对象

2. 在每一次访问这个对象的时候，spring容器都会创建这个对象
3. spring容器一旦把这个对象交给你之后，就不再管理这个对象了

为什么用Spring IOC？好处？

为什么用IOC

对象创建统一托管

规范的生命周期管理

灵活的依赖注入

一致的获取对象

SpringAOP实现原理

AOP定义：

AOP (Aspect Orient Programming)，我们一般称为面向切面编程，用于处理系统中分布于各个模块的横切关注点，比如事务管理、日志、缓存等等。

AOP实现原理：

AOP实现的关键在于AOP框架自动创建的AOP代理，AOP代理主要分为静态代理和动态代理，静态代理的代表为AspectJ；而动态代理则以Spring AOP为代表。

AspectJ原理

AspectJ是静态代理，所谓的静态代理就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强。它会在编译阶段将Aspect织入Java字节码中，运行的时候就是经过增强之后的AOP对象。代码见 <http://www.importnew.com/24305.html>

Spring AOP原理

- Spring AOP使用的动态代理，在每次运行时生成AOP代理对象。所谓的动态代理就是说AOP框架不会去修改字节码，而是在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。
- 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。JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。
- 如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标

类。CGLIB (Code Generation Library) , 是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

编码问题

2018年3月27日 10:09

目录

[为什么要编码](#)

[Unicode和UTF-8区别](#)

[常用编码格式](#)

[为什么现在常用UTF-8?](#)

可参考

<https://www.ibm.com/developerworks/cn/java/j-lo-chinesecoding/index.html>

为什么要编码?

1. 计算机中存储信息的最小单元是一个字节即 8 个 bit，所以能表示的字符范围是 0~255 个
2. 人类要表示的符号太多，无法用一个字节来完全表示
3. 要解决这个矛盾必须需要一个新的数据结构 char，从 char 到 byte 必须编码

Unicode和UTF-8区别

Unicode 是「字符集」，UTF-8 是「编码规则」。字符集是全世界统一的，为每一个字符分配一个唯一的ID，这个ID可以根据不同的编码规则转换成不同的二进制编码然后保存到计算机里。

常用编码格式

ASCII 码

学过计算机的人都知道 ASCII 码，总共有 128 个，用一个字节的低 7 位表示，0~31 是控制字符如换行回车删除等；32~126 是打印字符，可以通过键盘输入并且能够显示出来。

ISO-8859-1

128 个字符显然是不够用的，于是 ISO 组织在 ASCII 码基础上又制定了一些列标准用来扩展 ASCII 编码，它们是 ISO-8859-1~ISO-8859-15，其中 ISO-8859-1 涵盖了大多数西欧语言字符，所有应用的最广泛。ISO-8859-1 仍然是单字节编码，它总共能表示

256 个字符。

GB2312

它的全称是《信息交换用汉字编码字符集 基本集》，它是双字节编码，总的编码范围是 A1-F7，其中从 A1-A9 是符号区，总共包含 682 个符号，从 B0-F7 是汉字区，**包含 6763 个汉字。**

GBK

全称叫《汉字内码扩展规范》，是国家技术监督局为 windows95 所制定的新的汉字内码规范，它的出现是为了扩展 GB2312，加入更多的汉字，它的编码范围是 8140 ~FEFE (去掉 XX7F) 总共有 23940 个码位，**它能表示 21003 个汉字**，它的编码是和 GB2312 兼容的，也就是说用 GB2312 编码的汉字可以用 GBK 来解码，并且不会有乱码。

UTF-16

- **定长双字节**
- 说到 UTF 必须要提到 Unicode (Universal Code 统一码)，ISO 试图想创建一个全新的超语言字典，世界上所有的语言都可以通过这本字典来相互翻译。可想而知这个字典是多么的复杂，关于 Unicode 的详细规范可以参考相应文档。Unicode 是 Java 和 XML 的基础，下面详细介绍 Unicode 在计算机中的存储形式。
- UTF-16 具体定义了 Unicode 字符在计算机中存取方法。UTF-16 用两个字节来表示 Unicode 转化格式，这个是定长的表示方法，不论什么字符都可以用两个字节表示，两个字节是 16 个 bit，所以叫 UTF-16。UTF-16 表示字符非常方便，每两个字节表示一个字符，这个在字符串操作时就大大简化了操作，这也是 Java 以 UTF-16 作为内存的字符存储格式的一个很重要的原因。

UTF-8

可变长度，不同类型的字符可以是由 1~6 个字节组成。

UTF-16 统一采用两个字节表示一个字符，虽然在表示上非常简单方便，但是也有其缺点，有很大一部分字符用一个字节就可以表示的现在要两个字节表示，存储空间放大了一倍，在现在的网络带宽还非常有限的今天，这样会增大网络传输的流量，而且也没必要。而 UTF-8 采用了一种变长技术，每个编码区域有不同的字码长度。不同类型的字符可以是由 1~6 个字节组成。

UTF-8 有以下编码规则：

1. 如果一个字节，最高位（第 8 位）为 0，表示这是一个 ASCII 字符（00 - 7F）。可见，所有 ASCII 编码已经是 UTF-8 了。
2. 如果一个字节，以 11 开头，连续的 1 的个数暗示这个字符的字节数，例如：110xxxxx 代表它是双字节 UTF-8 字符的首字节。
3. 如果一个字节，以 10 开始，表示它不是首字节，需要向前查找才能得到当前字符的首字节

为什么现在常用UTF-8？

1. 因为涉及到网络传输
2. UTF16编码效率最高，但不适合在网络之间传输，因为网络传输容易损坏字节流，一旦字节流损坏将很难恢复。
3. 相比较而言 UTF-8 更适合网络传输，对 ASCII 字符采用单字节存储，另外单个字符损坏也不会影响后面其它字符，在编码效率上介于 GBK 和 UTF-16 之间，所以 **UTF-8 在编码效率上和编码安全性上做了平衡，是理想的中文编码方式。**

Linux命令

2017年9月5日 9:44

| | |
|---|--|
| ls -l 等同于 ll | 显示当前目录下文件的属性 |
| ls -d | 仅列出目录 |
| ls -al | 显示当前目录下所有文件详细信息，包括隐藏文件 |
| cd [~] [-] | 切换目录 [~目前用户身份的主目录] [-前一个工作目录] |
| chgrp 用户组 文件名 | 修改文件的用户组 |
| chown 用户名 文件名 | 修改文件的所有者 |
| chmod 770 文件名 | 修改文件的权限 |
| cp A B | 复制文件A为B |
| rm A rm -r A | 删除A 删除非空目录或文件A |
| su root | 切换为root用户 |
| mkdir/rmdir mkdir -p test1/test2 mkdir -m 777 test2 | 新建目录、删除一个空目录 创建多层空目录 创建目录时设定权限 |
| touch | 新建空的文件 |
| man 命令 | 查询命令详细解释 |
| bzip2 | 压缩文件 |
| bunzip2 | 解压文件 |
| nano | 文本编辑器 |
| pwd | 显示当前目录 |
| echo | 打印 |
| mv 文件名 目标位置 mv A B | 移动文件 重命名A为B（目录或文件都可以） |
| PATH="\$PATH":/root | 添加/root到环境变量中 |
| vim | 进入后按i可以输入 退出按ESC然后:wq 不保存退出按ESC然后:q! 一般模式中： |

| | |
|---|--|
| | /word 向下寻找word字符串 ?word 向上寻找word字符串 n 重复前一个查找 N 与 n反向重复查找 |
| umask -S | 查看目前用户在新建文件或目录时候的默认权限 |
| find | 搜索文件，很复杂，很多参数 |
| tar -jcv -f 名字.tar.bz2 A | 压缩 “A” 变成 “名字.tar.bz2” |
| tar -jtv -f 名字.tar.bz2 | 查询 |
| tar -jxv -f 名字.tar.bz2 [-C 指定目录] | 在当前目录解压 【-C 在指定目录解压】 |
| groupadd A groupdel A | 新建用户组A 删除用户组A |
| groups [A] | 查看自己所在的用户组，【查看A所在的用户组】 |
| usermod usermod -G XXX B | 该命令有很多参数，可以修改账号各个属性 新建xxx用户组为B的支持用户组，B原来的用户组仍有 (支持用户组不是当前用户组，有效用户组才是当前用户组) |
| newgrp B | 切换当前用户的有效用户组为B |
| useradd A useradd -u 666 -g B -c "XXX" A | 新增用户A (必须要设置密码才能用) 新增用户A，用户组为B，UID为666，账号全名是XXX |
| passwd A echo "XXX" passwd --stdin A | 给用户A设置密码，若没有A，则是给自己设置密码，密码需要超过8个字符 设置用户A的密码为XXX |
| passwd -l A passwd -u A passwd -S A | 使账号A密码失效 (让其无法登陆) 使账号A密码恢复 查询账号A密码状态 |
| userdel -r A | 删除用户A，连同用户主文件夹一起删除 (慎用) |
| setfacl -m u:A:rwx B setfacl -m g:A:rx B setfacl -b A | 设置账户A针对文件B的权限为rwx (针对单独用户设置权限) 设置用户组A针对文件B的权限为rx 消除文件A的ACL权限 |
| getfacl B | 查询文件B的权限详情 |
| ctrl+c | 终止当前程序运行 |
| ctrl+alt+F1 ctrl+alt+F2-F7 | 切回图形界面 切回命令行界面 |

| | |
|--------------------------|--|
| yum install XXX | CentOS的apt-get install XXX |
| which XXX | 检测某个xxx应用是否安装 |
| reboot | 重启服务器 |
| ifconfig | 查看Linux (包括本地虚拟机的Linux) 的IP地址 |
| shift+PgUp\PgDn | 命令行界面上下滚动 |
| 启动命令 & | 在后台启动，不占用命令窗口，比如启动Redis的时候 ./redis-server & |
| kill -9 PID | 关闭服务 ，比如某程序PID=6817 kill -9 6817 就关闭了这个服务 |
| top | 查看当前系统负载情况，如果是单核CPU 那么load average低于1说明没有线程等待 |
| netstat -nap grep 5672 | 查看端口号5672是否被监听 |
| free -m | 查看内存使用情况 |
| ps -A | 显示所有运行中的进程 |
| netstat -nulp | 查看当前正在使用的端口情况 |
| God status | 查看当前部署的服务 |
| god stop 服务名 | 停止服务 |
| scp jinsong@IP地址:/路径 ./ | 复制远程主机上的文件到当前目录 |
| pwd | 显示当前路径 |