Data Structures

Array

Sort

Time complexity

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Heap sort

```java
// Java program for implementation of Heap Sort
public class HeapSort
{
    public void sort(int arr[])
    {
        int n = arr.length;

        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i=n-1; i>0; i--)
        {
```

```
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    // To heapify a subtree rooted with node i which is
    // an index in arr[]. n is size of heap
    void heapify(int arr[], int n, int i)
    {
        int largest = i; // Initialize largest as root
        int l = 2*i + 1; // left = 2*i + 1
        int r = 2*i + 2; // right = 2*i + 2

        // If left child is larger than root
        if (l < n && arr[l] > arr[largest])
            largest = l;

        // If right child is larger than largest so far
        if (r < n && arr[r] > arr[largest])
            largest = r;

        // If largest is not root
        if (largest != i)
        {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }
```

```
    // Driver program
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};
        int n = arr.length;

        HeapSort ob = new HeapSort();
        ob.sort(arr);

        System.out.println("Sorted array is");
        printArray(arr);
    }
}
```

Merge sort

时间复杂性稳定，但是占用空间大，空间换效率

```
void merge(int arr[], int l, int m, int r)
{
    // Find sizes of two subarrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[] = new int[n1];
    int R[] = new int[n2];

    /*Copy data to temp arrays*/
    for (int i = 0; i < n1; ++i)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays */

    // Initial indexes of first and second subarrays
    int i = 0, j = 0;

    // Initial index of merged subarry array
    int k = l;
    // 轮流放入原array
    while (i < n1 && j < n2) {
```

```
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy remaining elements of L[] if any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy remaining elements of R[] if any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r) {
        // Find the middle point
        int m = (l + r) / 2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

Quick sort

每次寻找一个pivot，将比pivot小的划至左侧，大的划至右侧。对左右侧依次分割排序。

```java
// Java program for implementation of QuickSort
class QuickSort
{
    /* This function takes last element as pivot,
       places the pivot element at its correct
       position in sorted array, and places all
       smaller (smaller than pivot) to left of
       pivot and all greater elements to right
       of pivot */
    int partition(int arr[], int low, int high)
    {
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<high; j++)
        {
            // If current element is smaller than the pivot
            if (arr[j] < pivot)
            {
                i++;

                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1;
    }


    /* The main function that implements QuickSort()
      arr[] --> Array to be sorted,
      low  --> Starting index,
      high  --> Ending index */
    void sort(int arr[], int low, int high)
    {
        if (low < high)
        {
            /* pi is partitioning index, arr[pi] is
              now at right place */
```

```
            int pi = partition(arr, low, high);

            // Recursively sort elements before
            // partition and after partition
            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = arr.length;

        QuickSort ob = new QuickSort();
        ob.sort(arr, 0, n-1);

        System.out.println("sorted array");
        printArray(arr);
    }
}
/*This code is contributed by Rajat Mishra */
```

Tim sort

java 内置的排序算法，调用方法为Arrays.sort(arr)。

如需传入自定义比较方式可创建一个继承自Comparator 的类

```
// 创建一个comparator类
class NumComparator implements Comparator<NameTag> {
    public int compare (NameTag left,NameTag right) {
        return(left.getNumber() - right.getNumber());
    }
}
// 调用如下
Arrays.sort(tags, new NumComparator());
```

LinkedList

快慢指针能解决很多问题

Tree

通常tree类的题目会使用很多的递归，做题时要想好返回条件，遍历顺序。

Binary search tree

Algorithms

Graph

BFS

把遍历的node加入一个**queue**，first in first out。

可以用一个while-loop来实现

DFS

访问的node加入一个stack，first in last out。

可以用while-loop加上stack实现或者递归。

Network

时延

总时延 = 排队时延 + 处理时延 + 传输时延 + 传播时延

排队时延

分组在路由器的输入队列和输出队列中排队等待的时间，取决于网络当前的通信量。

处理时延

主机或路由器收到分组时进行处理所需要的时间，例如分析首部、从分组中提取数据、进行差错检验或查找适当的路由等。

传输时延

主机或路由器传输数据帧所需要的时间。

$delay = \frac{l}{v}$

其中 l 表示数据帧的长度，v 表示传输速率。

传播时延

电磁波在信道中传播所需要花费的时间，电磁波传播的速度接近光速。

其中 l 表示信道长度，v 表示电磁波在信道上的传播速度。

网络结构

1. 五层协议

- **应用层**： 为**特定应用程序**提供数据传输服务，例如 HTTP、DNS 等协议。数据单位为**报文**。
- **传输层**： 为**进程**提供通用数据传输服务。由于应用层协议很多，定义通用的传输层协议就可以支持不断增多的应用层协议。运输层包括两种协议：传输控制协议 TCP，提供面向连接、可靠的数据传输服务，数据单位为**报文段**；用户数据报协议 UDP，提供无连接、尽最大努力的数据传输服务，数据单位为用户数据报。TCP 主要提供完整性服务，UDP 主要提供及时性服务。
- **网络层**： 为**主机**提供数据传输服务。而传输层协议是为主机中的进程提供数据传输服务。网络层把传输层传递下来的报文段或者用户数据报**封装成分组**。
- **数据链路层**： 网络层针对的还是主机之间的数据传输服务，而主机之间可以有很多链路，链路层协议就是为同一链路的主机提供数据传输服务。数据链路层把网络层传下来的分组**封装成帧**。
- **物理层**： 考虑的是怎样在传输媒体上传输数据比特流，而不是指具体的传输媒体。物理层的作用是尽可能屏蔽传输媒体和通信手段的差异，使数据链路层感觉不到这些差异。

数据链接层

信道复用技术

- 频分复用： 相同时间占用不同频率的带宽资源

- 时分复用： 不同时间占用相同的频率带宽资源
- 统计时分复用： 不固定时分复用帧的位置，有数据就集中一组发送

MAC

固定的机器编码，用与交换机的机器对端口映射

交换机会自学习，更新翻译表

网络层

VPN

VPN 使用公用的互联网作为本机构各专用网之间的通信载体。专用指机构内的主机只与本机构内的其它主机通信；虚拟指好像是，而实际上并不是，它有经过公用的互联网。

（场所A）-- （路由A）-- 【互联网/隧道】-- （路由B）-- （场所B）

NAT

将内网ip+端口转化为外网ip+端口，内网主机公用一个外网ip，使用不同端口来转发不同内网ip

路由协议

1. RIP
    1. 两个路由之间的跳跃数计做开销，最多跳跃15次
    2. 周期性与相邻路由器更新路由表，最终知道到达每一个网络内路由的距离以及下一跳的路由
    3. 实现简单但是限制网络规模
2. OSPF
    1. 向所有路由广播自己的链接状态（仅在链接变化时广播）。最终所有路由都具有全网的拓扑结构
    2. 路径计算使用Dijkstra的最短路径算法
    3. OSPF收敛速度比RIP快
3. BGP
    1. 每个AS之间选用比较好的路由尽心通信
    2. 可以限制其他AS通过自己转发
    3. 必须配置BGP发言人来进行路由信息交换

传输层

网络层只把分组发送到目的主机，但是真正通信的并不是主机而是主机中的进程。传输层提供了进程间的逻辑通信，传输层向高层用户屏蔽了下面网络层的核心细节，使应用程序看**起来像是在两个传输层实体之间有一条端到端的逻辑通信信道。**

UDP

- 用户数据报协议 UDP（User Datagram Protocol）是无连接的，尽最大可能交付，没有拥塞控制，面向报文（对于应用程序传下来的报文不合并也不拆分，只是添加 UDP 首部），支持一对一、一对多、多对一和多对多的交互通信。

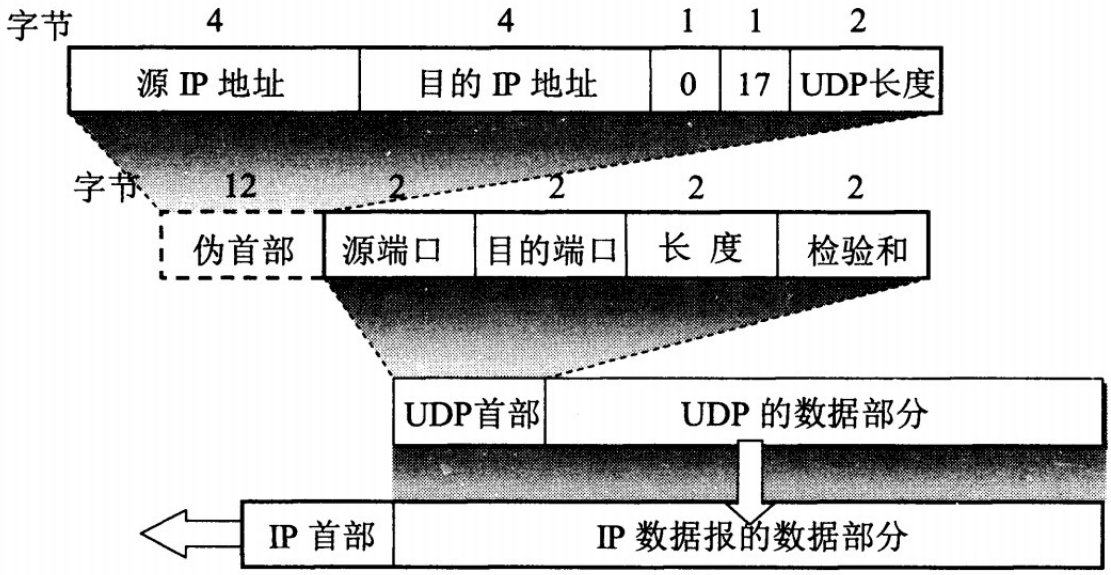


图 5-5　UDP 用户数据报的首部和伪首部

首部字段只有 8 个字节，包括源端口、目的端口、长度、检验和。12 字节的伪首部是为了计算检验和临时添加的。

TCP

- 传输控制协议 TCP（Transmission Control Protocol）是面向连接的，提供可靠交付，有流量控制，拥塞控制，提供全双工通信，面向字节流（把应用层传下来的报文看成字节流，把字节流组织成大小不等的数据块），每一条 TCP 连接只能是点对点的（一对一）。

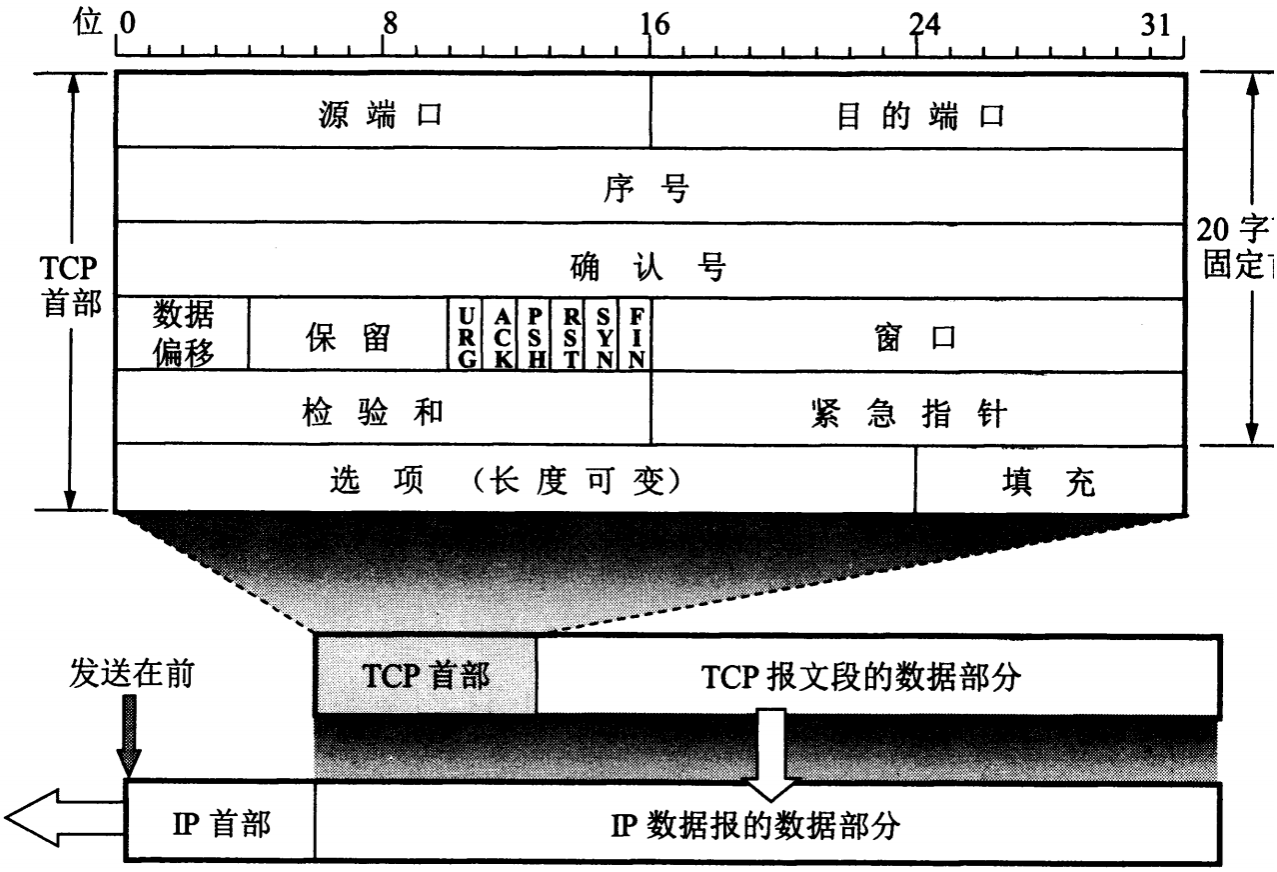


图 5-14　TCP 报文段的首部格式

- **序号**： 用于对字节流进行编号，例如序号为 301，表示第一个字节的编号为 301，如果携带的数据长度为 100 字节，那么下一个报文段的序号应为 401。
- **确认号**： 即ack，期望收到的下一个报文段的序号。例如 B 正确收到 A 发送来的一个报文段，序号为 501，携带的数据长度为 200 字节，因此 B 期望下一个报文段的序号为 701，**B 发送给 A 的确认报文段**中确认号就为 701。
- **数据偏移**： 指的是数据部分距离报文段起始处的偏移量，实际上指的是首部的长度。
- **确认 ACK**： 当 ACK=1 时确认号字段有效，否则无效。TCP 规定，在连接建立后所有传送的报文段都必须把 ACK 置 1。
- **同步 SYN**： 在连接建立时用来同步序号。当 SYN=1，ACK=0 时表示这是一个连接请求报文段。若对方同意建立连接，则响应报文中 SYN=1，ACK=1。
- **终止 FIN**： 用来释放一个连接，当 FIN=1 时，表示此报文段的发送方的数据已发送完毕，并要求释放连接。
- **窗口**： 窗口值作为接收方让发送方设置其发送窗口的依据。之所以要有这个限制，是因为接收方的数据缓存空间是有限的

TCP三次握手

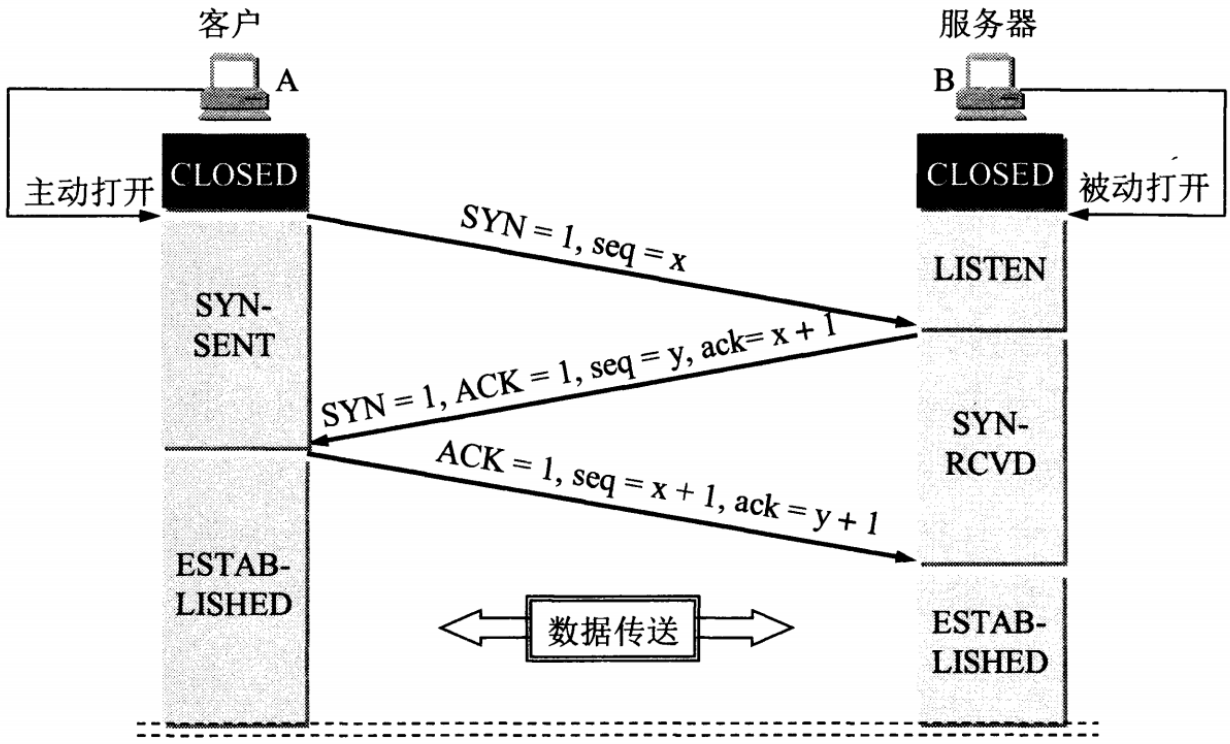


图 5-28 用三报文握手建立 TCP 连接

假设 A 为客户端，B 为服务器端。

- 首先 B 处于 LISTEN（监听）状态，等待客户的连接请求。
- A 向 B 发送连接请求报文，SYN=1，ACK=0，选择一个初始的序号 x。
- B 收到连接请求报文，如果同意建立连接，则向 A 发送连接确认报文，SYN=1，ACK=1，确认号为 x+1，同时也选择一个初始的序号 y。
- A 收到 B 的连接确认报文后，还要向 B 发出确认，确认号为 y+1，序号为 x+1。
- B 收到 A 的确认后，连接建立。

三次握手的原因（建立连接）

第三次握手是为了防止失效的连接请求到达服务器，让服务器错误打开连接。

客户端发送的连接请求如果在网络中滞留，那么就会隔很长一段时间才能收到服务器端发回的连接确认。客户端等待一个超时重传时间之后，就会重新请求连接。但是这个滞留的连接请求最后还是会到达服务器，如果不进行三次握手，那么服务器就会打开两个连接。如果有第三次握手，客户端会忽略服务器之后发送的对滞留连接请求的连接确认，不进行第三次握手，因此就不会再次打开连接。

TCP四次挥手

以下描述不讨论序号和确认号，因为序号和确认号的规则比较简单。并且不讨论 ACK，因为 ACK 在连接建立之后都为 1。

- A 发送连接释放报文，FIN=1。
- B 收到之后发出确认，此时 TCP 属于半关闭状态，B 能向 A 发送数据但是 A 不能向 B 发送数据。
- 当 B 不再需要连接时，发送连接释放报文，FIN=1。
- A 收到后发出确认，进入 TIME-WAIT 状态，等待 2 MSL（最大报文存活时间）后释放连接。

- B 收到 A 的确认后释放连接。

四次挥手的原因

客户端发送了 FIN 连接释放报文之后，服务器收到了这个报文，就进入了 CLOSE-WAIT 状态。这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕之后，服务器会发送 FIN 连接释放报文。

TIME_WAIT

客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间 2MSL。这么做有两个理由：

- 确保最后一个确认报文能够到达。如果 B 没收到 A 发送来的确认报文，那么就会重新发送连接释放请求报文，A 等待一段时间就是为了处理这种情况的发生。
- 等待一段时间是为了让本连接持续时间内所产生的所有报文都从网络中消失，使得下一个新的连接不会出现旧的连接请求报文。

应用层

应用	应用层协议	端口号	传输层协议	备注
域名解析	DNS	53	UDP/TCP	长度超过 512 字节时使用 TCP
动态主机配置协议	DHCP	67/68	UDP	
简单网络管理协议	SNMP	161/162	UDP	
文件传送协议	FTP	20/21	TCP	控制连接 21，数据连接 20
远程终端协议	TELNET	23	TCP	
超文本传送协议	HTTP	80	TCP	
简单邮件传送协议	SMTP	25	TCP	
邮件读取协议	POP3	110	TCP	
网际报文存取协议	IMAP	143	TCP	

DNS域名系统

DNS 是一个分布式数据库，提供了主机名和 IP 地址之间相互转换的服务。这里的分布式数据库是指，每个站点只保留它自己的那部分数据。

域名具有层次结构，从上到下依次为：根域名、顶级域名、二级域名。

通常使用UDP传输，限制大小为512字节。

当大小超过512字节或者主域名服务器向辅助服务器发送变化的时候使用TCP协议

动态主机配置协议

DHCP (Dynamic Host Configuration Protocol) 提供了**即插即用**的连网方式，用户不再需要手动配置 IP 地址等信息。

DHCP 配置的内容不仅是 IP 地址，还包括子网掩码、网关 IP 地址。

DHCP 工作过程如下：

1. 客户端发送 Discover 报文，该报文的目的地址为 255.255.255.255:67，源地址为 0.0.0.0:68，被放入 UDP 中，该报文被广播到同一个子网的所有主机上。如果客户端和 DHCP 服务器不在同一个子网，就需要使用中继代理。
2. DHCP 服务器收到 Discover 报文之后，发送 Offer 报文给客户端，该报文包含了客户端所需要的信息。因为客户端可能收到多个 DHCP 服务器提供的信息，因此客户端需要进行选择。
3. 如果客户端选择了某个 DHCP 服务器提供的信息，那么就发送 Request 报文给该 DHCP 服务器。
4. DHCP 服务器发送 Ack 报文，表示客户端此时可以使用提供给它的信息。

电子邮件协议

一个电子邮件系统由三部分组成：用户代理、邮件服务器以及邮件协议。

邮件协议包含发送协议和读取协议，发送协议常用 SMTP，读取协议常用 POP3（读取后会删除邮件） 和 IMAP（读取后服务器上邮件不会被删除）。

Web网页请求过程

1. DHCP配置主机信息
    1. 向DHCP服务器请求IP地址，之后就配置它的 IP 地址、子网掩码和 DNS 服务器的 IP 地址，并在其 IP 转发表中安装默认网关。
2. ARP(address resolution protocol)解析MAC地址
    1. 现在只知道网关路由器的IP，需要得到MAC地址
    2. 主机生成ARP查询报文并广播，网管路由器收到后解开ARP、报文发现查询的IP与自己一致，回复MAC地址给主机
3. DNS解析域名
    1. 网关路由器将DNS解析请求转发给DNS服务器，DNS查询域名并通过网关发送回主机
4. HTTP请求页面
    1. 有了网页服务器IP后可以申请TCP连接
    2. 通过HTTP GET向主机请求web内容，服务器将web页面放入报文并发送回诸暨
    3. 渲染web页面内容并显示

2. OSI

其中表示层和会话层用途如下：

- **表示层**： 数据压缩、加密以及数据描述，这使得应用程序不必关心在各台主机中数据内部格式不同的问题。
- **会话层**： 建立及管理会话。

五层协议没有表示层和会话层，而是将这些功能留给应用程序开发者处理。

3. TCP/IP

它只有四层，相当于五层协议中数据链路层和物理层合并为网络接口层。

TCP/IP 体系结构不严格遵循 OSI 分层概念，应用层可能会直接使用 IP 层或者网络接口层。

4. 数据在各层之间的传递过程

在向下的过程中，需要添加下层协议所需要的首部或者尾部，而在向上的过程中不断拆开首部和尾部。

路由器只有下面三层协议，因为路由器位于网络核心中，不需要为进程或者应用程序提供服务，因此也就不需要传输层和应用层。

编译器

1.词法分析（lexical analysis）

词法分析是编译过程的第一个阶段。这个阶段的任务是从左到右的读取每个字符，然后根据构词规则识

别单词。词法分析可以用 lex 等工具自动生成。

2.语法分析（syntax analysis）

语法分析是编译过程的一个逻辑阶段。语法分析在词法分析的基础上，将单词序列组合成各类语法短

语，如“程序”，“语句”，“表达式”等等。语法分析程序判断程序在结构上是否正确。

3.语义分析（semantic analysis）

属于逻辑阶段。对源程序进行上下文有关性质的审查，类型检查。如赋值语句左右端类型匹配问题。

Machine Learning

Precision: 挑出的瓜中好瓜的比例

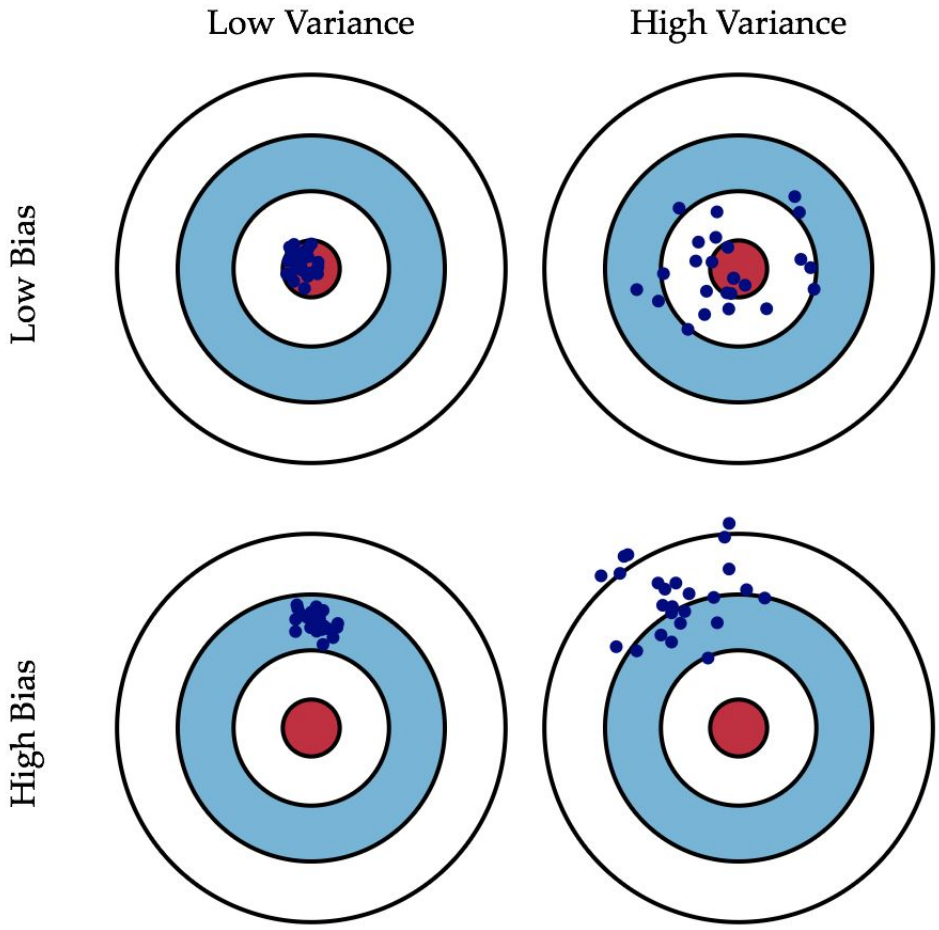
Recall：好瓜中挑出的比例

mAP：mean average percision，pr曲线下面积对处以p

ssd：用vgg16作为骨干网络，在不同的卷积层引出一个全联接层做classification，好处是对于不同大小的物体可以更好的提取特征

数学基础

- 方差（variance）以及偏差（bias）



减少过拟合 Prevent overfitting

To recap: here are the most common ways to prevent overfitting in neural networks:

- Get more training data.
  - 最直接的方式，获取更多的训练数据
- Reduce the capacity of the network.
  - 使用更小的网络，即减少网络可训练变量的的容积。构建较大的网络时应当从小向大逐渐增加
- Add weight regularization.
  - 正则化，本质上是对weight权重添加限制，分为L1正则化以及L2正则化
  - L1正则化 where the cost added is proportional to the absolute value of the weights coefficients (i.e. to what is called the "L1 norm" of the weights).
  - L2正则化 where the cost added is proportional to the square of the value of the weights coefficients (i.e. to what is called the squared "L2 norm" of the weights). L2 regularization is also called weight decay in the context of neural networks. Don't let the different name confuse you: weight decay is mathematically the exact same as L2 regularization.

```
kernel_regularizer=regularizers.l2(0.0001),
相当于 cost = 0.0001 * weight_coefficient_value^2
```

    - L1 regularization pushes weights towards exactly zero encouraging a sparse model. L2 regularization will penalize the weights parameters without making them sparse since the penalty goes to zero for small weights. one reason why L2 is more common. L2比L1更常见，L2对于小权重的惩罚很低而L1会将很多权重清零

- Add dropout.
    - 对于一部分weight随机清零，layers.Dropout(0.5) 这个layer代表50%的weight会被随机放弃清零

Normalization

1. 为什么需要 Normalization

1.1 独立同分布与白化

机器学习界的炼丹师们最喜欢的数据有什么特点？窃以为，莫过于“**独立同分布**”了，即*independent and identically distributed*，简称为 *i.i.d.* 独立同分布并非所有机器学习模型的必然要求（比如 Naive Bayes 模型就建立在特征彼此独立的基础之上，而Logistic Regression 和 神经网络 则在非独立的特征数据上依然可以训练出很好的模型），但独立同分布的数据可以简化常规机器学习模型的训练、提升机器学习模型的预测能力，已经是一个共识。

因此，在把数据喂给机器学习模型之前，“**白化（whitening）**”是一个重要的数据预处理步骤。白化一般包含两个目的：

（1）*去除特征之间的相关性* —> 独立；

（2）*使得所有特征具有相同的均值和方差* —> 同分布。

白化最典型的方法就是PCA，可以参考阅读 PCAWhitening。

1.2 深度学习中的 Internal Covariate Shift

深度神经网络模型的训练为什么会很困难？其中一个重要的原因是，深度神经网络涉及到很多层的叠加，而每一层的参数更新会导致上层的输入数据分布发生变化，通过层层叠加，高层的输入分布变化会非常剧烈，这就使得高层需要不断去重新适应底层的参数更新。为了训好模型，我们需要非常谨慎地去设定学习率、初始化权重、以及尽可能细致的参数更新策略。

Google 将这一现象总结为 Internal Covariate Shift，简称 ICS. 什么是 ICS 呢?

@魏秀参

在一个回答中做出了一个很好的解释：

> 大家都知道在统计机器学习中的一个经典假设是“源空间（source domain）和目标空间（target domain）的数据分布（distribution）是一致的”。如果不一致，那么就出现了新的机器学习问题，如 transfer learning / domain adaptation 等。而 covariate shift 就是分布不一致假设之下的一个分支问题，它是指源空间和目标空间的条件概率是一致的，但是其边缘概率不同，即：对所有$\boldsymbol{x} \in \boldsymbol{\mathcal{X}}$，
>
> $$P_s(Y|X=x) = P_t(Y|X=x)$$
>
> 但是
>
> $$P_s(X) \neq P_t(X)$$
>
> 大家细想
>
> 便会发现，的确，对于神经网络的各层输出，由于它们经过了层内操作作用，其分布显然与各层对应的输入信号分布不同，而且差异会随着网络深度增大而增大，可是它们所能“指示”的样本标记（label）仍然是不变的，这便符合了covariate shift的定义。由于是对层间信号的分析，也即是“internal”的来由。

1.3 ICS 会导致什么问题?

简而言之，每个神经元的输入数据不再是“独立同分布”。

其一，上层参数需要不断适应新的输入数据分布，降低学习速度。

其二，下层输入的变化可能趋向于变大或者变小，导致上层落入饱和区，使得学习过早停止。

其三，每层的更新都会影响到其它层，因此每层的参数更新策略需要尽可能的谨慎。

2. Normalization 的通用框架与基本思想

我们以神经网络中的一个普通神经元为例。神经元接收一组输入向量

$$\mathbf{x} = (x_1, x_2, \cdots, x_d)$$

通过某种运算

后，输出一个标量值：

$$y = f(\mathbf{x})$$

由于 ICS 问题的存在， $\mathbf{x}$ 的分布可能相差很大。要解决独立同分布的问题，“理论正确”的方法就是对每一层的数据都进行白化操作。然而标准的白化操作代价高昂，特别是我们还希望白化操作是可微的，保证白化操作可以通过反向传播来更新梯度。

因此，以 BN 为代表的 Normalization 方法退而求其次，进行了简化的白化操作。基本思想是：在将 $\mathbf{x}$ 送给神经元之前，先对其做**平移和伸缩变换**， 将 $\mathbf{x}$ 的分布规范化成在固定区间范围的标准分布。

通用变换框架就如下所示：

$$h = f\left(\mathbf{g} \cdot \frac{\mathbf{x} - \mu}{\sigma} + \mathbf{b}\right)$$

我们来看看这个公式中的各个参数。

（1） μ 是**平移参数**（shift parameter）， σ 是**缩放参数**（scale parameter）。通过这两个参数进行 shift 和 scale 变换：

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sigma}$$

得到的数据符合均值为 0、方差为 1 的标准分布。

（2） $\mathbf{b}$ 是**再平移参数**（re-shift parameter）， $\mathbf{g}$ 是**再缩放参数**（re-scale parameter）。将 上一步得到的 $\hat{\mathbf{x}}$ 进一步变换为：

$$\mathbf{y} = \mathbf{g} \cdot \hat{\mathbf{x}} + \mathbf{b}$$

最终得到的数据符合均值为 $\mathbf{b}$ 、方差为 $\mathbf{g}^2$ 的分布。

奇不奇怪？奇不奇怪？

说好的处理 ICS，第一步都已经得到了标准分布，第二步怎么又给变走了？

答案是——**为了保证模型的表达能力不因为规范化而下降**。

我们可以看到，第一步的变换将输入数据限制到了一个全局统一的确定范围（均值为 0、方差为 1）。下层神经元可能很努力地在学习，但不论其如何变化，其输出的结果在交给上层神经元进行处理之前，将被粗暴地重新调整到这一固定范围。

沮不沮丧？沮不沮丧？

难道我们底层神经元人民就在做无用功吗？

所以，为了尊重底层神经网络的学习结果，我们将规范化后的数据进行再平移和再缩放，使得每个神经元对应的输入范围是针对该神经元量身定制的一个确定范围（均值为 $\mathbf{b}$ 、方差为 $\mathbf{g}^2$ ）。rescale 和 reshift 的参数都是可学习的，这就使得 Normalization 层可以学习如何去尊重底层的学习结果。

除了充分利用底层学习的能力，另一方面的重要意义在于保证获得非线性的表达能力。Sigmoid 等激活函数在神经网络中有着重要作用，通过区分饱和区和非饱和区，使得神经网络的数据变换具有了非线性计算能力。而第一步的规范化会将几乎所有数据映射到激活函数的非饱和区（线性区），仅利用到了线性变化能力，从而降低了神经网络的表达能力。而进行再变换，则可以将数据从线性区变换到非线性区，恢复模型的表达能力。

那么问题又来了——

经过这么的变回来再变过去，会不会跟没变一样?

不会。因为，再变换引入的两个新参数 g 和 b，可以表示旧参数作为输入的同一族函数，但是新参数有不同的学习动态。在旧参数中， $\mathbf{x}$ 的均值取决于下层神经网络的复杂关联；但在新参数中， $\mathbf{y} = \mathbf{g} \cdot \hat{\mathbf{x}} + \mathbf{b}$ 仅由 $\mathbf{b}$ 来确定，去除了与下层计算的密切耦合。新参数很容易通过梯度下降来学习，简化了神经网络的训练。

那么还有一个问题——

这样的 Normalization 离标准的白化还有多远?

标准白化操作的目的是"独立同分布"。独立就不说了，暂不考虑。变换为均值为 $\mathbf{b}$ 、方差为 $\mathbf{g}^2$ 的分布，也并不是严格的同分布，只是映射到了一个确定的区间范围而已。（所以，这个坑还有得研究呢！）

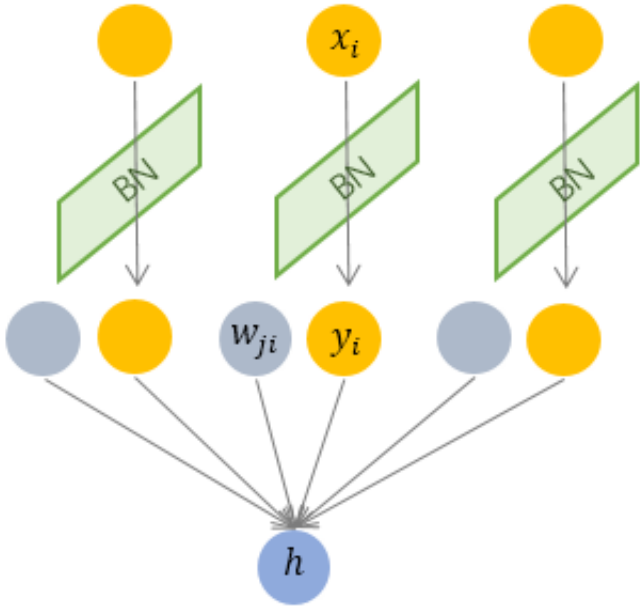
3. 主流 Normalization 方法梳理

在上一节中，我们提炼了 Normalization 的通用公式：

$$h = f\left(\mathbf{g} \cdot \frac{\mathbf{x} - \mu}{\sigma} + \mathbf{b}\right)$$

对照于这一公式，我们来梳理主流的四种规范化方法。

3.1 Batch Normalization —— 纵向规范化



Batch Normalization 于2015年由 Google 提出，开 Normalization 之先河。其规范化针对单个神经元进行，利用网络训练时一个 mini-batch 的数据来计算该神经元 x_i 的均值和方差,因而称为 Batch Normalization。

$$\mu_i = \frac{1}{M}\sum x_i, \quad \sigma_i = \sqrt{\frac{1}{M}\sum (x_i - \mu_i)^2 + \epsilon}$$

其中 M 是 mini-batch 的大小。

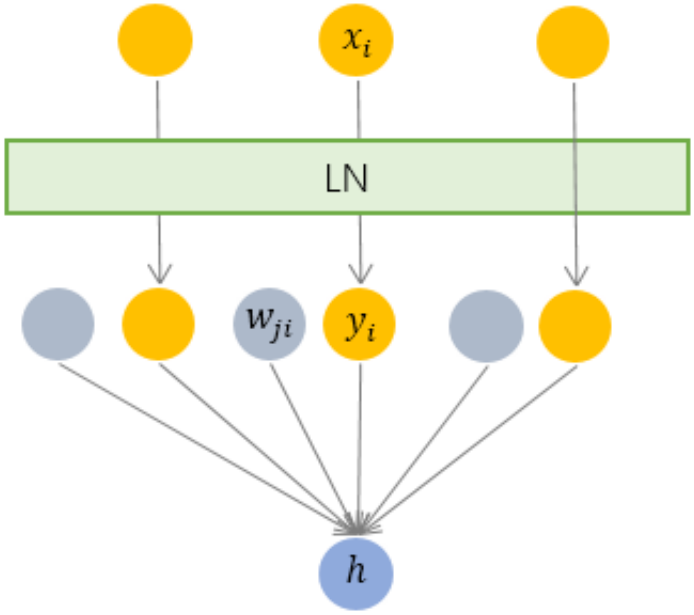
按上图所示，相对于一层神经元的水平排列，BN 可以看做一种纵向的规范化。由于 BN 是针对单个维度定义的，因此标准公式中的计算均为 element-wise 的。

BN 独立地规范化每一个输入维度 x_i ，但规范化的参数是一个 mini-batch 的一阶统计量和二阶统计量。这就要求 每一个 mini-batch 的统计量是整体统计量的近似估计，或者说每一个 mini-batch 彼此之间，以及和整体数据，都应该是近似同分布的。分布差距较小的 mini-batch 可以看做是为规范化操作和模型训练引入了噪声，可以增加模型的鲁棒性；但如果每个 mini-batch的原始分布差别很大，那么不同 mini-batch 的数据将会进行不一样的数据变换，这就增加了模型训练的难度。

因此，BN 比较适用的场景是：每个 mini-batch 比较大，数据分布比较接近。在进行训练之前，要做好充分的 shuffle. 否则效果会差很多。

另外，由于 BN 需要在运行过程中统计每个 mini-batch 的一阶统计量和二阶统计量，因此不适用于 动态的网络结构 和 RNN 网络。不过，也有研究者专门提出了适用于 RNN 的 BN 使用方法，这里先不展开了。

3.2 Layer Normalization —— 横向规范化



层规范化就是针对 BN 的上述不足而提出的。与 BN 不同，LN 是一种横向的规范化，如图所示。它综合考虑一层所有维度的输入，计算该层的平均输入值和输入方差，然后用同一个规范化操作来转换各个维度的输入。

$$\mu = \sum_i x_i, \quad \sigma = \sqrt{\sum_i (x_i - \mu)^2 + \epsilon}$$

其中 i 枚举了该层所有的输入神经元。对应到标准公式中，四大参数 μ, σ, $\mathbf{g}$, $\mathbf{b}$ 均为标量（BN中是向量），所有输入共享一个规范化变换。

LN 针对单个训练样本进行，不依赖于其他数据，因此可以避免 BN 中受 mini-batch 数据分布影响的问题，可以用于 小mini-batch场景、动态网络场景和 RNN，特别是自然语言处理领域。此外，LN 不需要保存 mini-batch 的均值和方差，节省了额外的存储空间。

但是，BN 的转换是针对单个神经元可训练的——不同神经元的输入经过再平移和再缩放后分布在不同的区间，而 LN 对于一整层的神经元训练得到同一个转换——所有的输入都在同一个区间范围内。如果不同输入特征不属于相似的类别（比如颜色和大小），那么 LN 的处理可能会降低模型的表达能力。

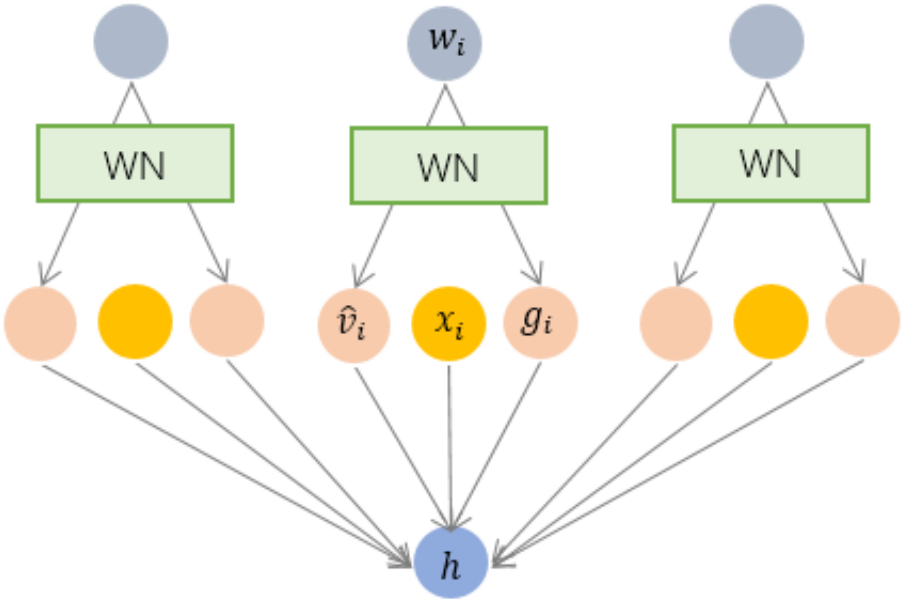
3.3 Weight Normalization —— 参数规范化

前面我们讲的模型框架

$$h = f\left(\mathbf{g} \cdot \frac{\mathbf{x} - \mu}{\sigma} + \mathbf{b}\right)$$

中，经过规范化之后的$\mathbf{y}$ 作为输入送到下一个神经元，应用以 $\mathbf{w}$ 为参数的$f_{\mathbf{w}}(\cdot)$ 函数定义的变换。最普遍的变换是线性变换，即 $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$.

BN 和 LN 均将规范化应用于输入的特征数据 $\mathbf{x}$，而 WN 则另辟蹊径，将规范化应用于线性变换函数的权重 $\mathbf{w}$，这就是 WN 名称的来源。



具体而言，WN 提出的方案是，**将权重向量 $\mathbf{w}$ 分解为向量方向 $\hat{\mathbf{v}}$ 和向量模 g 两部分**：

$$\mathbf{w} = g \cdot \hat{\mathbf{v}} = g \cdot \frac{\mathbf{v}}{||\mathbf{v}||}$$

其中 $\mathbf{v}$ 是与 $\mathbf{w}$ 同维度的向量，$||\mathbf{v}||$是欧氏范数，因此$\hat{\mathbf{v}}$ 是单位向量，决定了 $\mathbf{w}$ 的方向；g 是标量，决定了 $\mathbf{w}$ 的长度。由于 $||\mathbf{w}|| \equiv |g|$，因此这一权重分解的方式将权重向量的欧氏范数进行了固定，从而实现了正则化的效果。

乍一看，这一方法似乎脱离了我们前文所讲的通用框架?

并没有。其实从最终实现的效果来看，异曲同工。我们来推导一下看。

$$f_{\mathbf{w}}(WN(\mathbf{x})) = \mathbf{w} \cdot WN(\mathbf{x}) = g \cdot \frac{\mathbf{v}}{||\mathbf{v}||} \cdot \mathbf{x}$$
$$= \mathbf{v} \cdot g \cdot \frac{\mathbf{x}}{||\mathbf{v}||} = f_{\mathbf{v}}(g \cdot \frac{\mathbf{x}}{||\mathbf{v}||})$$

对照一下前述框架：

$$h = f\left(\mathbf{g} \cdot \frac{\mathbf{x} - \mu}{\sigma} + \mathbf{b}\right)$$

我们只需令：

$$\sigma = ||\mathbf{v}||, \quad \mu = 0, \quad \mathbf{b} = 0$$

就完美地对号入座了！

回忆一下，BN 和 LN 是用输入的特征数据的方差对输入数据进行 scale，而 WN 则是用 神经元的权重的欧氏范式对输入数据进行 scale。**虽然在原始方法中分别进行的是特征数据规范化和参数的规范化，但本质上都实现了对数据的规范化，只是用于 scale 的参数来源不同。**

另外，我们看到这里的规范化只是对数据进行了 scale，而没有进行 shift，因为我们简单地令 $\mu = 0$. 但事实上，这里留下了与 BN 或者 LN 相结合的余地——那就是利用 BN 或者 LN 的方法来计算输入数据的均值 μ 。

WN 的规范化不直接使用输入数据的统计量，因此避免了 BN 过于依赖 mini-batch 的不足，以及 LN 每层唯一转换器的限制，同时也可以用于动态网络结构。

3.4 Cosine Normalization —— 余弦规范化

Normalization 还能怎么做?

我们再来看看神经元的经典变换 $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$.

对输入数据 $\mathbf{x}$ 的变换已经做过了，横着来是 LN，纵着来是 BN。

对模型参数 $\mathbf{w}$ 的变换也已经做过了，就是 WN。

好像没啥可做的了。

然而天才的研究员们盯上了中间的那个点，对，就是·

他们说，我们要对数据进行规范化的原因，是数据经过神经网络的计算之后可能会变得很大，导致数据分布的方差爆炸，而这一问题的根源就是我们的计算方式——点积，权重向量 $\mathbf{w}$ 和 特征数据向量 $\mathbf{x}$ 的点积。向量点积是无界（unbounded）的啊！

那怎么办呢？我们知道向量点积是衡量两个向量相似度的方法之一。哪还有没有其他的相似度衡量方法呢？有啊，很多啊！夹角余弦就是其中之一啊！而且关键的是，夹角余弦是有确定界的啊，[-1, 1] 的取值范围，多么的美好！仿佛看到了新的世界！

于是，Cosine Normalization 就出世了。他们不处理权重向量 $\mathbf{w}$，也不处理特征数据向量 $\mathbf{x}$，就改了一下线性变换的函数：

$$f_{\mathbf{w}}(\mathbf{x}) = cos\theta = \frac{\mathbf{w} \cdot \mathbf{x}}{||\mathbf{w}|| \cdot ||\mathbf{x}||}$$

其中 θ 是 $\mathbf{w}$ 和 $\mathbf{x}$ 的夹角。然后就没有然后了，所有的数据就都是 [-1, 1] 区间范围之内的了！

不过，回过头来看，CN 与 WN 还是很相似的。我们看到上式中，分子还是 $\mathbf{w}$ 和 $\mathbf{x}$ 的内积，而分母则可以看做用 $\mathbf{w}$ 和 $\mathbf{x}$ 二者的模之积进行规范化。对比一下 WN 的公式：

$$f_{\mathbf{w}}(WN(\mathbf{x})) = f_{\mathbf{v}}(g \cdot \frac{\mathbf{x}}{||\mathbf{v}||})$$

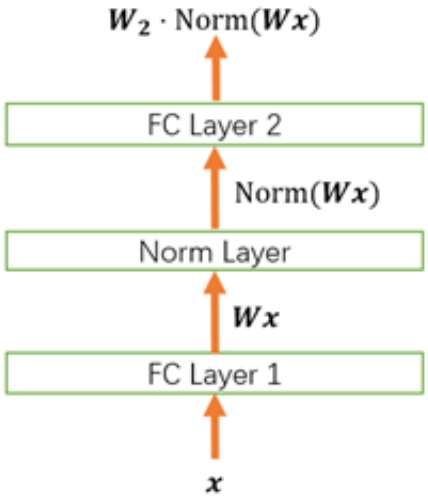
一定程度上可以理解为，WN 用 权重的模 $||\mathbf{v}||$ 对输入向量进行 scale，而 CN 在此基础上用输入向量的模 $||\mathbf{x}||$ 对输入向量进行了进一步的 scale.

CN 通过用余弦计算代替内积计算实现了规范化，但成也萧何败萧何。原始的内积计算，其几何意义是 输入向量在权重向量上的投影，既包含 二者的夹角信息，也包含 两个向量的scale信息。去掉scale信息，可能导致表达能力的下降，因此也引起了一些争议和讨论。具体效果如何，可能需要在特定的场景下深入实验。

现在，BN, LN, WN 和 CN 之间的来龙去脉是不是清楚多了?

4. Normalization 为什么会有效?

我们以下面这个简化的神经网络为例来分析。



4.1 Normalization 的权重伸缩不变性

权重伸缩不变性（weight scale invariance）指的是，当权重 $\mathbf{W}$ 按照常量 $\boldsymbol{\lambda}$ 进行伸缩时，得到的规范化后的值保持不变，即：

$$Norm(\mathbf{W}'\mathbf{x}) = Norm(\mathbf{W}\mathbf{x})$$

其中 $\mathbf{W}' = \lambda\mathbf{W}$ 。

上述规范化方法均有这一性质，这是因为，当权重 $\mathbf{W}$ 伸缩时，对应的均值和标准差均等比例伸缩，分子分母相抵。

$$\begin{aligned} Norm(\mathbf{W}'\mathbf{x}) &= Norm\left(\mathbf{g}\cdot\frac{\mathbf{W}'\mathbf{x}-\mu'}{\sigma'}+\mathbf{b}\right) \\ &= Norm\left(\mathbf{g}\cdot\frac{\lambda\mathbf{W}\mathbf{x}-\lambda\mu}{\lambda\sigma}+\mathbf{b}\right) \\ &= Norm\left(\mathbf{g}\cdot\frac{\mathbf{W}\mathbf{x}-\mu}{\sigma}+\mathbf{b}\right) = Norm(\mathbf{W}\mathbf{x}) \end{aligned}$$

权重伸缩不变性可以有效地提高反向传播的效率。

由于

$$\frac{\partial Norm(\mathbf{W}'\mathbf{x})}{\partial\mathbf{x}} = \frac{\partial Norm(\mathbf{W}\mathbf{x})}{\partial\mathbf{x}}$$

因此，权重的伸缩变化不会影响反向梯度的 Jacobian 矩阵，因此也就对反向传播没有影响，避免了反向传播时因为权重过大或过小导致的梯度消失或梯度爆炸问题，从而加速了神经网络的训练。

权重伸缩不变性还具有参数正则化的效果，可以使用更高的学习率。

由于 $$\frac{\partial Norm(\mathbf{W}'\mathbf{x})}{\partial \mathbf{W}'} = \frac{1}{\lambda} \cdot \frac{\partial Norm(\mathbf{W}\mathbf{x})}{\partial \mathbf{W}}$$

因此，下层的权重值越大，其梯度就越小。这样，参数的变化就越稳定，相当于实现了参数正则化的效果，避免参数的大幅震荡，提高网络的泛化性能。

4.2 Normalization 的数据伸缩不变性

数据伸缩不变性（data scale invariance）指的是，当数据 $\mathbf{x}$ 按照常量 $\boldsymbol{\lambda}$ 进行伸缩时，得到的规范化后的值保持不变，即：

$$Norm(\mathbf{W}\mathbf{x}') = Norm(\mathbf{W}\mathbf{x})$$

其中 $\mathbf{x}' = \boldsymbol{\lambda}\mathbf{x}$ 。

数据伸缩不变性仅对 BN、LN 和 CN 成立。因为这三者对输入数据进行规范化，因此当数据进行常量伸缩时，其均值和方差都会相应变化，分子分母互相抵消。而 WN 不具有这一性质。

数据伸缩不变性可以有效地减少梯度弥散，简化对学习率的选择。

对于某一层神经元 $\boldsymbol{h_l = f_{\mathbf{W}_l}(\mathbf{x}_l)}$ 而言，展开可得

$$h_l = f_{\mathbf{W}_l}(\mathbf{x}_l) = f_{\mathbf{W}_l}(f_{\mathbf{W}_{l-1}}(\mathbf{x}_{l-1})) = \cdots = \mathbf{x}_0 \prod_{k=0}^{l} \mathbf{W}_k$$

每一层神经元的输出依赖于底下各层的计算结果。如果没有正则化，当下层输入发生伸缩变化时，经过层层传递，可能会导致数据发生剧烈的膨胀或者弥散，从而也导致了反向计算时的梯度爆炸或梯度弥散。

加入 Normalization 之后，不论底层的数据如何变化，对于某一层神经元 $\boldsymbol{h_l = f_{\mathbf{W}_l}(\mathbf{x}_l)}$ 而言，其输入 $\mathbf{x}_l$ 永远保持标准的分布，这就使得高层的训练更加简单。从梯度的计算公式来看：

$$\frac{\partial Norm(\mathbf{W}\mathbf{x}')}{\partial \mathbf{W}} = \frac{\partial Norm(\mathbf{W}\mathbf{x})}{\partial \mathbf{W}}$$

数据的伸缩变化也不会影响到对该层的权重参数更新，使得训练过程更加鲁棒，简化了对学习率的选择。

Java语言特性

可能的面试问题

1. 垃圾回收机制。。。(主要从下面几方面解答 GC原理、最好画图解释一下年轻代（Eden区和Survival区）、年老代、比例分配及为啥要这样分代回收)
2. 对象分配问题，堆栈里的问题，详细的会问道方法区、堆、程序计数器、本地方法栈、虚拟机栈，问题入口从String a,new String("")开始
3. 关键字，private protected public static final 组合着问
4. Object类里面有哪几种方法，作用
5. equals 和 hashCode方法，重写equals的原则()
6. 向上转型
7. Java引用类型(强引用，软引用，弱引用，虚引用)
8. 线程相关的，主要是volitate，synchorized，wait()，notify()，notifyAll()，join()
9. Exception和Error
10. 反射的用途
11. HashMap实现原理(数组+链表)，查找数据的时间复杂度
12. List有哪些子类，各有什么区别
13. NIO相关，缓冲区、通道、selector。。。(不熟，面了这么多，挂在这里。其实主要是表现在同步阻塞和异步，传输方式不同。标准IO无法实现非阻塞模式、文件锁、读选择、分散聚集等)
14. 内存泄露，举个例子
15. OOM是怎么出现的，有哪几块JVM区域会产生OOM，如何解决(对于该问题，建议去《Java特种兵》的3.6章)
16. Java里面的观察者模式实现
17. 单例实现(我一般用enum写，不容易被挑毛病)
18. 用Java模拟一个栈，并能够做到扩容，并且能有同步锁。（用数组实现）
19. Java泛型机制，泛型机制的优点，以及类型变量

Java内部类

https://www.cnblogs.com/dolphin0520/p/3811445.html

1.根据注释填写(1)，(2)，(3)处的代码

```
public class Test{
    public static void main(String[] args){
            // 初始化Bean1
            // 创建外部类，然后通过外部类调用内部类的构造函数
            Test test = new Test();
            Test.Bean1 bean1 = test.new Bean1();
            bean1.I++;
            // 初始化Bean2
            // 直接调用内部静态类创建
            Test.Bean2 bean2 = new Test.Bean2();
            bean2.J++;
            //初始化Bean3
            // 同1，创建外部类
```

```
            Bean bean = new Bean();
            Bean.Bean3 bean3 = bean.new Bean3();
            bean3.k++;
    }
    class Bean1{
            public int I = 0;
    }

    static class Bean2{
            public int J = 0;
    }
}

class Bean{
    class Bean3{
            public int k = 0;
    }
}
```

从前面可知，对于成员内部类，必须先产生外部类的实例化对象，才能产生内部类的实例化对象。而静态内部类不用产生外部类的实例化对象即可产生内部类的实例化对象。

创建静态内部类对象的一般形式为： 外部类类名.内部类类名 xxx = new 外部类类名.内部类类名()

创建成员内部类对象的一般形式为： 外部类类名.内部类类名 xxx = 外部类对象名.new 内部类类名()

因此，（1），（2），（3）处的代码分别为：

```
Test test = new Test();
Test.Bean1 bean1 = test.new Bean1();
```

```
Test.Bean2 b2 = new Test.Bean2();
```

```
Bean bean = new Bean();
Bean.Bean3 bean3 =  bean.new Bean3();
```

```
public class Test {
    public static void main(String[] args)  {
        Outter outter = new Outter();
        outter.new Inner().print();
    }
}
```

```
class Outter
{
    private int a = 1;
    class Inner {
        private int a = 2;
        public void print() {
            int a = 3;
            System.out.println("局部变量: " + a);
            System.out.println("内部类变量: " + this.a);
            System.out.println("外部类变量: " + Outter.this.a);
        }
    }
}
```

```
输出:
3
2
1
```

Java泛型

泛型有三种使用方式，分别为：泛型类、泛型接口和泛型方法。

https://blog.csdn.net/ShuSheng0007/article/details/80720406

泛型类

在类的申明时指定参数，即构成了泛型类，例如下面代码中就指定`T`为类型参数，那么在这个类里面就可以使用这个类型了。例如申明`T`类型的变量`name`，申明`T`类型的形参`param`等操作。

```
public class Generic<T> {
    public T name;
    public Generic(T param){
        name=param;
    }
    public T m(){
        return name;
    }
}
```

那么在使用类时就可以传入相应的类型，构建不同类型的实例，如下面代码分别传入了`String`,`Integer`,`Boolean`3个类型:

```
 private static void genericClass()
 {
     Generic<String> str=new Generic<>("总有刁民想害朕");
     Generic<Integer> integer=new Generic<>(110);
     Generic<Boolean> b=new Generic<>(true);

     System.out.println("传入类型："+str.name+"  "+integer.name+"  "+b.name);
}
```

输出结果为：`传入类型：总有刁民想害朕 110 true`

如果没有泛型，我们想要达到上面的效果需要定义三个类，或者一个包含三个构造函数，三个取值方法的类。

泛型接口

泛型接口与泛型类的定义基本一致

```
public interface Generator<T> {
    public T produce();
}
```

泛型方法

这个相对来说就比较复杂，当我首次接触时也是一脸懵逼，抓住特点后也就没有那么难了。

```
public class Generic<T> {
    public T name;
    public  Generic(){}
    public Generic(T param){
        name=param;
    }
    public T m(){
        return name;
    }
    public <E> void m1(E e){ }
    public <T> T m2(T e){ }
}
```

重点看`public <E> void m1(E e){ }`这就是一个泛型方法，判断一个方法是否是泛型方法关键看方法返回值前面有没有使用`<>`标记的类型，有就是，没有就不是。这个`<>`里面的类型参数就相当于为这个方法声明了一个类型，这个类型可以在此方法的作用块内自由使用。 上面代码中，`m()`方法不是泛型方法，`m1()`与`m2()`都是。值得注意的是`m2()`方法中声明的类型`T`与类申明里面的那个参数`T`不是一个，也可以说方法中的`T`**隐藏**了类型中的`T`。下面代码中类里面的`T`传入的是`String`类型，而方法中的`T`传入的

是`Integer`类型。

```
Generic<String> str=new Generic<>("总有刁民想害朕");
str.m2(123);
```

泛型的使用方法

如何继承一个泛型类

如果不传入具体的类型，则子类也需要指定类型参数，代码如下：

```
class Son<T> extends Generic<T>{}
```

如果传入具体参数，则子类不需要指定类型参数

```
class Son extends Generic<String>{}
```

如何实现一个泛型接口

```
class ImageGenerator<T> implements Generator<T>{
    @Override
    public T produce() {
        return null;
    }
}
```

如何调用一个泛型方法

和调用普通方法一致，不论是实例方法还是静态方法。

通配符?

? 代表任意类型，例如有如下函数：

```
public void m3(List<?>list){
    for (Object o : list) {
        System.out.println(o);
    }
}
```

其参数类型是`?`，那么我们调用的时候就可以传入任意类型的`List`,如下

```
str.m3(Arrays.asList(1,2,3));
str.m3(Arrays.asList("总有刁民","想害","朕"));
```

但是说实话，单独一个`?`意义不大，因为大家可以看到，从集合中获取到的对象的类型是`Object`类型的，也就只有那几个默认方法可调用，几乎没什么用。如果你想要使用传入的类型那就需要**强制类型转换**，这是我们接受不了的，不然使用泛型干毛。其真正强大之处是可以通过设置其上下限达到类型的灵活使用，且看下面分解**重点内容**。

通配符上界

通配符上界使用`<? extends T>`的格式，意思是需要一个**T类型**或者**T类型的子类**，一般T类型都是一个具体的类型，例如下面的代码。

```
public void printIntValue(List<? extends Number> list) {
    for (Number number : list) {
        System.out.print(number.intValue()+" ");
    }
}
```

这个意义就非凡了，无论传入的是何种类型的集合，我们都可以使用其父类的方法统一处理。

通配符下界

通配符下界使用`<? super T>`的格式，意思是需要一个**T类型**或者**T类型的父类**，一般T类型都是一个具体的类型，例如下面的代码。

```
public void fillNumberList(List<? super Number> list) {
    list.add(new Integer(0));
    list.add(new Float(1.0));
}
```

至于什么时候使用通配符上界，什么时候使用下界，在**《Effective Java》**中有很好的指导意见：遵循**PECS**原则，即**producer-extends,consumer-super.** 换句话说，如果参数化类型表示一个生产者，就使用`<? extends T>`；如果参数化类型表示一个消费者，就使用`<? super T>`。

Java泛型擦除

流程

- Replace all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
    - 将泛型替换成他们的边界，没有边界就替换成Object类型
- Insert type casts if necessary to preserve type safety.

    - 将类型转换放到合适的地方来保证类型安全
- Generate bridge methods to preserve polymorphism in extended generic types.
    - 生成桥接方法来保持多态，例如泛型的字类型

泛型擦除

Consider the following generic class that represents a node in a singly linked list:

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

Because the type parameter `T` is unbounded, the Java compiler replaces it with `Object`:

```
public class Node {

    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() { return data; }
    // ...
}
```

In the following example, the generic `Node` class uses a bounded type parameter:

```
public class Node<T extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

The Java compiler replaces the bounded type parameter `T` with the first bound class, `Comparable`:

```
public class Node {

    private Comparable data;
    private Node next;

    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData() { return data; }
    // ...
}
```

Bridge Methods

When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, called a *bridge method*, as part of the type erasure process. You normally don't need to worry about bridge methods, but you might be puzzled if one appears in a stack trace.

After type erasure, the `Node` and `MyNode` classes become:

```
public class Node {

    public Object data;

    public Node(Object data) { this.data = data; }
```

```
    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node {

    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

After type erasure, the method signatures do not match. The `Node` method becomes `setData(Object)` and the `MyNode` method becomes `setData(Integer)`. Therefore, the `MyNode` `setData` method does not override the `Node` `setData` method.

To solve this problem and preserve the polymorphism of generic types after type erasure, a Java compiler generates a bridge method to ensure that subtyping works as expected. For the `MyNode` class, the compiler generates the following bridge method for `setData`:

```
class MyNode extends Node {

    // Bridge method generated by the compiler
    //
    public void setData(Object data) {
        setData((Integer) data);
    }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }

    // ...
}
```

As you can see, the bridge method, which has the same method signature as the `Node` class's `setData` method after type erasure, delegates to the original `setData` method.

Java 引用类型

https://juejin.im/post/6844903665241686029

通过表格来说明一下，如下：

引用类型	被垃圾回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM停止运行时终止
软引用	当内存不足时	对象缓存	内存不足时终止
弱引用	正常垃圾回收时	对象缓存	垃圾回收后终止
虚引用	正常垃圾回收时	跟踪对象的垃圾回收	垃圾回收后终止

强引用 strong reference

在一个**方法的内部**有一个**强引用**，这个引用保存在`Java`**栈**中，而真正的引用内容(`Object`)保存在`Java`**堆**中。 当这个**方法运行完成**后，就会退出**方法栈**，则引用对象的**引用数**为`0`，这个对象会被回收。

当**内存空间不足**时，`Java`虚拟机宁愿抛出`OutOfMemoryError`错误，使程序**异常终止**，也不会靠随意**回收**具有**强引用**的**对象**来解决内存不足的问题。 如果强引用对象**不使用时**，需要弱化从而使`GC`能够回收

可以显式的将引用设为null，这样对象就会被回收 `object=null;`

软引用 soft reference

如果一个对象只具有**软引用**，则**内存空间充足**时，**垃圾回收器**就**不会**回收它；如果**内存空间不足**了，就会**回收**这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。

> 软引用可用来实现内存敏感的高速缓存。

```
// 强引用
String strongReference = new String("abc");
// 软引用
String str = new String("abc");
SoftReference<String> softReference = new SoftReference<String>(str);
```

软引用可以和一个**引用队列**(`ReferenceQueue`)联合使用。如果**软引用**所引用对象被**垃圾回收**，`JAVA`虚拟机就会把这个**软引用**加入到与之关联的**引用队列**中。

```
    ReferenceQueue<String> referenceQueue = new ReferenceQueue<>();
    String str = new String("abc");
    SoftReference<String> softReference = new SoftReference<>(str,
referenceQueue);

    str = null;
    // Notify GC
    System.gc();

    System.out.println(softReference.get()); // abc

    Reference<? extends String> reference = referenceQueue.poll();
    System.out.println(reference); //null
```

> 注意：软引用对象是在JVM内存不够的时候才会被回收，我们调用System.gc()方法只是起通知作用，JVM什么时候扫描回收对象是JVM自己的状态决定的。就算扫描到软引用对象也不一定会回收它，只有内存不够的时候才会回收。

应用场景：

浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。

1. 如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建；
2. 如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出。

这时候就可以使用软引用，很好的解决了实际的问题：

```
    // 获取浏览器对象进行浏览
    Browser browser = new Browser();
    // 从后台程序加载浏览页面
    BrowserPage page = browser.getPage();
    // 将浏览完毕的页面置为软引用
    SoftReference softReference = new SoftReference(page);

    // 回退或者再次浏览此页面时
    if(softReference.get() != null) {
        // 内存充足，还没有被回收器回收，直接获取缓存
        page = softReference.get();
    } else {
        // 内存不足，软引用的对象已经回收
        page = browser.getPage();
        // 重新构建软引用
        softReference = new SoftReference(page);
    }
```

弱引用 weak reference

弱引用与**软引用**的区别在于：只具有**弱引用**的对象拥有**更短暂**的**生命周期**。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有**弱引用**的对象，不管当前**内存空间足够与否**，都会**回收**它的内存。不过，由于垃圾回收器是一个**优先级很低的线程**，因此**不一定**会**很快**发现那些只具有**弱引用**的对象。

```
String str = new String("abc");
WeakReference<String> weakReference = new WeakReference<>(str);
str = null;
```

`JVM`首先将**软引用**中的**对象**引用置为`null`，然后通知**垃圾回收器**进行回收：

```
str = null;
System.gc();
```

> 注意：如果一个对象是偶尔(很少)的使用，并且希望在使用时随时就能获取到，但又不想影响此对象的垃圾收集，那么你应该用Weak Reference来记住此对象。

下面的代码会让一个**弱引用**再次变为一个**强引用**：

```
String str = new String("abc");
WeakReference<String> weakReference = new WeakReference<>(str);
// 弱引用转强引用
String strongReference = weakReference.get();
```

同样，**弱引用**可以和一个**引用队列**(`ReferenceQueue`)联合使用，如果**弱引用**所引用的**对象**被**垃圾回收**，`Java`虚拟机就会把这个**弱引用**加入到与之关联的**引用队列**中。

虚引用 phantom reference

虚引用顾名思义，就是**形同虚设**。与其他几种引用都不同，**虚引用**并**不会**决定对象的**生命周期**。如果一个对象**仅持有虚引用**，那么它就和**没有任何引用**一样，在任何时候都可能被垃圾回收器回收。

应用场景：

虚引用主要用来**跟踪对象**被垃圾回收器**回收**的活动。 **虚引用**与**软引用**和**弱引用**的一个区别在于：

> 虚引用必须和引用队列(ReferenceQueue)联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

```
String str = new String("abc");
ReferenceQueue queue = new ReferenceQueue();
// 创建虚引用，要求必须与一个引用队列关联
PhantomReference pr = new PhantomReference(str, queue);
```

程序可以通过判断引用**队列**中是否已经加入了**虚引用**，来了解被引用的对象是否将要进行**垃圾回收**。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的**内存被回收之前**采取必要的行动。

Java锁

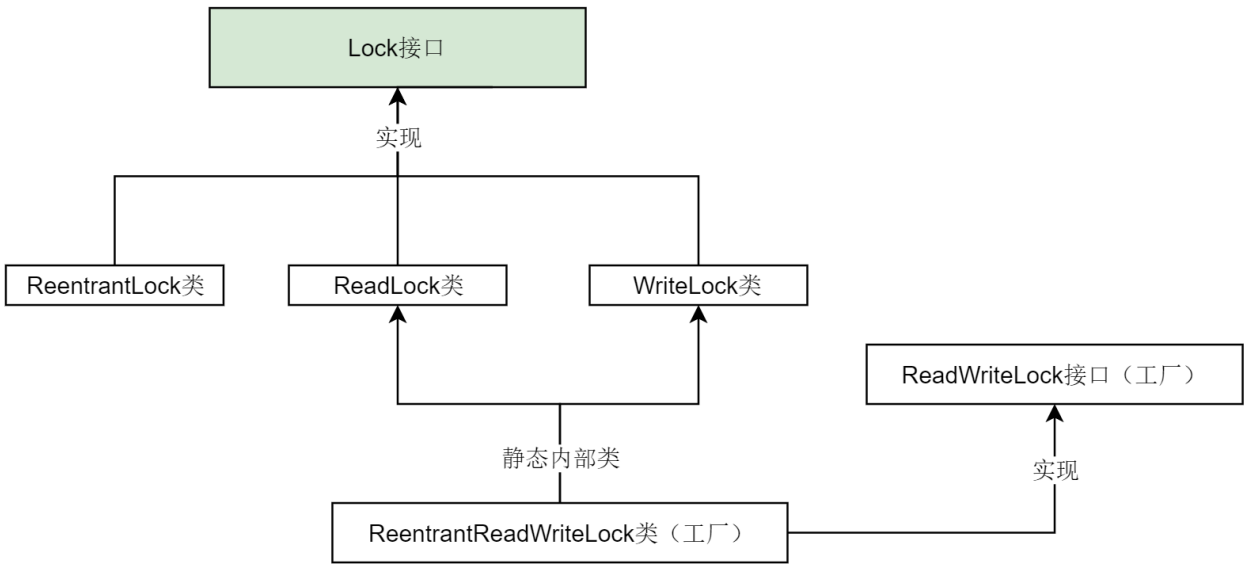
作者：Pickle Pee 链接：https://zhuanlan.zhihu.com/p/71156910 来源：知乎 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

零、synchronized与Lock

Java中有两种加锁的方式：一种是用**synchronized关键字**，另一种是用**Lock接口**的实现类。

形象地说，synchronized关键字是**自动档**，可以满足一切日常驾驶需求。但是如果你想要玩漂移或者各种骚操作，就需要**手动档**了——各种Lock的实现类。

所以如果你只是想要简单的加个锁，对性能也没什么特别的要求，用synchronized关键字就足够了。自Java 5之后，才在java.util.concurrent.locks包下有了另外一种方式来实现锁，那就是Lock。也就是说，**synchronized是Java语言内置的关键字，而Lock是一个接口**，这个接口的实现类在代码层面实现了锁的功能，具体细节不在本文展开，有兴趣可以研究下AbstractQueuedSynchronizer类，写得可以说是牛逼爆了。



其实只需要关注三个类就可以了：ReentrantLock类、ReadLock类、WriteLock类。

ReentrantLock、**ReadLock**、**WriteLock** 是Lock接口最重要的三个实现类。对应了“可重入锁”、“读锁”和“写锁”，后面会讲它们的用途。

ReadWriteLock其实是一个工厂接口，而ReentrantReadWriteLock是ReadWriteLock的实现类，它包含两个静态内部类ReadLock和WriteLock。这两个静态内部类又分别实现了Lock接口。

我们停止深究源码，仅从使用的角度看，Lock与synchronized的区别是什么？在接下来的几个小节中，我将梳理各种锁分类的概念，以及synchronized关键字、各种Lock实现类之间的区别与联系。

一、悲观锁与乐观锁

锁的一种宏观分类方式是**悲观锁**和**乐观锁**。悲观锁与乐观锁**并不是特指某个锁**（Java中没有哪个Lock实现类就叫PessimisticLock或OptimisticLock），而是在并发情况下的两种不同策略。

悲观锁（Pessimistic Lock），就是很悲观，每次去拿数据的时候都认为别人会修改。所以每次在拿数据的时候都会上锁。这样别人想拿数据就被挡住，直到悲观锁被释放。

乐观锁（Optimistic Lock），就是很乐观，每次去拿数据的时候都认为别人不会修改。所以**不会上锁，不会上锁！**但是如果想要更新数据，则会在**更新前检查在读取至更新这段时间别人有没有修改过这个数据**。如果修改过，则重新读取，再次尝试更新，循环上述步骤直到更新成功（当然也允许更新失败的线程放弃操作）。

悲观锁阻塞事务，乐观锁回滚重试，它们各有优缺点，不要认为一种一定好于另一种。像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行重试，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

二、乐观锁的基础——CAS

说到乐观锁，就必须提到一个概念：**CAS**

什么是CAS呢？Compare-and-Swap，即**比较并替换，**也有叫做Compare-and-Set的，**比较并设置**。

1、比较：读取到了一个值A，在将其更新为B之前，检查原值是否仍为A（未被其他线程改动）。

2、设置：如果是，将A更新为B，结束。[1]如果不是，则什么都不做。

上面的两步操作是原子性的，可以简单地理解为瞬间完成，在CPU看来就是一步操作。

有了CAS，就可以实现一个**乐观锁**：

```
data = 123; // 共享数据

/* 更新数据的线程会进行如下操作 */
flag = true;
while (flag) {
    oldValue = data; // 保存原始数据
    newValue = doSomething(oldValue);

    // 下面的部分为CAS操作，尝试更新data的值
    if (data == oldValue) { // 比较
        data = newValue; // 设置
        flag = false; // 结束
    } else {
  // 啥也不干，循环重试
    }
```

```
}
/*
    很明显，这样的代码根本不是原子性的，
    因为真正的CAS利用了CPU指令，
    这里只是为了展示执行流程，本意是一样的。
*/
```

这是一个简单直观的乐观锁实现，它允许多个线程同时读取（因为根本没有加锁操作），但是只有一个线程可以成功更新数据，并导致其他要更新数据的线程回滚重试。 CAS利用CPU指令，从硬件层面保证了操作的原子性，以达到类似于锁的效果。

```
// Method descriptor #257 (Ljava/lang/Object;JLjava/lang/Object;Ljava/lang/Object;)Z
public fina native boolean compareAndSwapObject(java.lang.Object arg0, long arg1, java.lang.Object arg2, java.lang.Object arg3);

// Method descriptor #234 (Ljava/lang/Object;JII)Z
public fina native boolean compareAndSwapInt(java.lang.Object arg0, long arg1, int arg2, int arg3);

// Method descriptor #238 (Ljava/lang/Object;JJJ)Z
public final native boolean compareAndSwapLong(java.lang.Object arg0, long arg1, long arg2, long arg3);
```

Java中真正的CAS操作调用的native方法

因为整个过程中并没有“加锁”和“解锁”操作，因此乐观锁策略也被称为**无锁编程**。换句话说，乐观锁其实不是“锁”，它仅仅是一个循环重试CAS的算法而已！

三、自旋锁

有一种锁叫**自旋锁**。所谓自旋，说白了就是一个 while(true) 无限循环。

刚刚的乐观锁就有类似的无限循环操作，那么它是自旋锁吗?

不是。尽管自旋与 while(true) 的操作是一样的，但还是应该将这两个术语分开。“自旋”这两个字，特指自旋锁的自旋。

然而在JDK中并没有自旋锁（SpinLock）这个类，那什么才是自旋锁呢？读完下个小节就知道了。

四、synchronized锁升级：偏向锁 → 轻量级锁 → 重量级锁

前面提到，synchronized关键字就像是汽车的**自动档，**现在详细讲这个过程。一脚油门踩下去，synchronized会从**无锁**升级为**偏向锁**，再升级为**轻量级锁**，最后升级为**重量级锁**，就像自动换挡一样。那么自旋锁在哪里呢？这里的轻量级锁就是一种**自旋锁**。

初次执行到synchronized代码块的时候，锁对象变成**偏向锁**（通过CAS修改对象头里的锁标志位），字面意思是“偏向于第一个获得它的线程”的锁。执行完同步代码块后，线程并**不会主动释放偏向锁**。当第二次到达同步代码块时，线程会判断此时持有锁的线程是否就是自己（持有锁的线程ID也在对象头里），如果是则正常往下执行。**由于之前没有释放锁，这里也就不需要重新加锁。**如果自始至终使用锁的线程只有一个，很明显偏向锁几乎没有额外开销，性能极高。

一旦有第二个线程加入**锁竞争**，偏向锁就升级为**轻量级锁（自旋锁）**。这里要明确一下什么是锁竞争：如果多个线程轮流获取一个锁，但是每次获取锁的时候都很顺利，没有发生阻塞，那么就不存在锁竞争。只有当某线程尝试获取锁的时候，发现该锁已经被占用，只能等待其释放，这才发生了锁竞争。

在轻量级锁状态下继续锁竞争，没有抢到锁的线程将**自旋**，即不停地循环判断锁是否能够被成功获取。获取锁的操作，其实就是通过CAS修改对象头里的锁标志位。先**比较**当前锁标志位是否为“释放”，如果是则将其**设置**为“锁定”，比较并设置是**原子性**发生的。这就算抢到锁了，然后线程将当前锁的持有者信息修改为自己。

长时间的自旋操作是非常消耗资源的，一个线程持有锁，其他线程就只能在原地空耗CPU，执行不了任何有效的任务，这种现象叫做**忙等（busy-waiting）**。如果多个线程用一个锁，但是没有发生锁竞争，或者发生了很轻微的锁竞争，那么synchronized就用轻量级锁，允许短时间的忙等现象。这是一种折衷的想法，**短时间的忙等，换取线程在用户态和内核态之间切换的开销。**

显然，此忙等是有限度的（有个计数器记录自旋次数，默认允许循环10次，可以通过虚拟机参数更改）。如果锁竞争情况严重，某个达到最大自旋次数的线程，会将轻量级锁升级为**重量级锁**（依然是CAS修改锁标志位，但不修改持有锁的线程ID）。当后续线程尝试获取锁时，发现被占用的锁是重量级锁，则直接将自己挂起（而不是忙等），等待将来被唤醒。在JDK1.6之前，synchronized直接加重量级锁，很明显现在得到了很好的优化。

一个锁只能按照 偏向锁、轻量级锁、重量级锁的顺序逐渐升级（也有叫**锁膨胀**的），不允许降级。

> 感谢评论区酷帅俊靓美的问题： 偏向锁的一个特性是，持有锁的线程在执行完同步代码块时不会释放锁。那么当第二个线程执行到这个synchronized代码块时是否一定会发生锁竞争然后升级为轻量级锁呢？ 线程A第一次执行完同步代码块后，当线程B尝试获取锁的时候，发现是偏向锁，会判断线程A是否仍然存活。**如果线程A仍然存活，**将线程A暂停，此时偏向锁升级为轻量级锁，之后线程A继续执行，线程B自旋。但是**如果判断结果是线程A不存在了**，则线程B持有此偏向锁，锁不升级。 还有人对此有疑惑，我之前确实没有描述清楚，但如果要展开讲，涉及到太多新概念，可以新开一篇了。更何况有些太底层的东西，我没读过源码，没有自信说自己一定是对的。其实在升级为轻量级锁之前，虚拟机会让线程A尽快在安全点挂起，然后在它的栈中“伪造”一些信息，让线程A在被唤醒之后，认为自己一直持有的是轻量级锁。如果线程A之前正在同步代码块中，那么线程B自旋等待即可。如果线程A之前不在同步代码块中，它会在被唤醒后检查到这一情况并立即释放锁，让线程B可以拿到。这部分内容我之前也没有深入研究过，如果有说的不对的，请多多指教啊！

五、可重入锁（递归锁）

可重入锁的字面意思是“可以重新进入的锁”，即**允许同一个线程多次获取同一把锁**。比如一个递归函数里有加锁操作，递归过程中这个锁会阻塞自己吗？如果不会，那么这个锁就是**可重入锁**（因为这个原因可重入锁也叫做**递归锁**）。

Java里只要以Reentrant开头命名的锁都是可重入锁，而且**JDK提供的所有现成的Lock实现类，包括synchronized关键字锁都是可重入的。**如果你需要不可重入锁，只能自己去实现了。网上不可重入锁的实现真的很多，就不在这里贴代码了。99%的业务场景用可重入锁就可以了，剩下的1%是什么呢？我也不知道，谁可以在评论里告诉我？

Module java.base
Package java.util.concurrent.locks

Interface Lock

All Known Implementing Classes:

ReentrantLock, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock

---

JDK提供的Lock的实现类都是可重入的

六、公平锁、非公平锁

如果多个线程申请一把**公平锁**，那么当锁释放的时候，先申请的先得到，非常公平。显然如果是**非公平锁**，后申请的线程可能先获取到锁，是随机或者按照其他优先级排序的。

对ReentrantLock类而言，通过构造函数传参**可以指定该锁是否是公平锁，默认是非公平锁**。一般情况下，非公平锁的吞吐量比公平锁大，如果没有特殊要求，优先使用非公平锁。

```
260     /**
261      * Creates an instance of {@code ReentrantLock} with the
262      * given fairness policy.
263      *
264      * @param fair {@code true} if this lock should use a fair ordering policy
265      */
266     public ReentrantLock(boolean fair) {
267         sync = fair ? new FairSync() : new NonfairSync();
268     }
269
```

ReentrantLock构造器可以指定为公平或非公平

对于synchronized而言，它也是一种**非公平锁**，但是并没有任何办法使其变成公平锁。

七、可中断锁

可中断锁，字面意思是“可以**响应中断**的锁”。

这里的关键是理解什么是**中断**。Java并没有提供任何直接中断某线程的方法，只提供了**中断机制**。何谓“中断机制”？线程A向线程B发出“请你停止运行”的请求（线程B也可以自己给自己发送此请求），但线程B并不会立刻停止运行，而是自行选择合适的时机以自己的方式响应中断，也可以直接忽略此中断。也就是说，Java的**中断不能直接终止线程**，而是需要被中断的线程自己决定怎么处理。这好比是父母叮嘱在外的子女要注意身体，但子女是否注意身体，怎么注意身体则完全取决于自己。[2]

回到锁的话题上来，如果线程A持有锁，线程B等待获取该锁。由于线程A持有锁的时间过长，线程B不想继续等待了，我们可以让线程B中断自己或者在别的线程里中断它，这种就是**可中断锁**。

在Java中，synchronized就是**不可中断锁**，而Lock的实现类都是**可中断锁**，可以简单看下Lock接口。

```
/* Lock接口 */
public interface Lock {
```

```
    void lock(); // 拿不到锁就一直等，拿到马上返回。

    void lockInterruptibly() throws InterruptedException; // 拿不到锁就一直等，如果等待时收到中断请求，则需要处理InterruptedException。

    boolean tryLock(); // 无论拿不拿得到锁，都马上返回。拿到返回true，拿不到返回false。

    boolean tryLock(long time, TimeUnit unit) throws InterruptedException; // 同上，可以自定义等待的时间。

    void unlock();

    Condition newCondition();
}
```

八、读写锁、共享锁、互斥锁

读写锁其实是一对锁，一个读锁（共享锁）和一个写锁（互斥锁、排他锁）。

看下Java里的ReadWriteLock接口，它只规定了两个方法，一个返回读锁，一个返回写锁。

```
119 public interface ReadWriteLock {
120     /**
121      * Returns the lock used for reading.
122      *
123      * @return the lock used for reading
124      */
125     Lock readLock();
126
127     /**
128      * Returns the lock used for writing.
129      *
130      * @return the lock used for writing
131      */
132     Lock writeLock();
133 }
```

记得之前的乐观锁策略吗？所有线程随时都可以读，仅在写之前判断值有没有被更改。

读写锁其实做的事情是一样的，但是策略稍有不同。很多情况下，线程知道自己读取数据后，是否是为了更新它。那么何不在加锁的时候直接明确这一点呢？如果我读取值是为了更新它（SQL的for update就是这个意思），那么加锁的时候就直接加**写锁**，我持有写锁的时候别的线程无论读还是写都需要等待；如果我读取数据仅为了前端展示，那么加锁时就明确地加一个**读锁**，其他线程如果也要加读锁，不需要等待，可以直接获取（读锁计数器+1）。

虽然读写锁感觉与乐观锁有点像，但是**读写锁是悲观锁策略**。因为读写锁并没有在**更新前**判断值有没有被修改过，而是在**加锁前**决定应该用读锁还是写锁。乐观锁特指无锁编程，如果仍有疑惑可以再回到第一、二小节，看一下什么是“乐观锁”。

JDK提供的唯一一个ReadWriteLock接口实现类是ReentrantReadWriteLock。看名字就知道，它不仅提供了读写锁，而是都是可重入锁。 除了两个接口方法以外，ReentrantReadWriteLock还提供了一些便于外界监控其内部工作状态的方法，这里就不一一展开。

九、回到悲观锁和乐观锁

> 这篇文章经历过一次修改，我之前认为偏向锁和轻量级锁是乐观锁，重量级锁和Lock实现类为悲观锁，网上很多资料对这些概念的表述也很模糊，各执一词。

先抛出我的结论：

我们在Java里使用的各种锁，**几乎全都是悲观锁**。synchronized从偏向锁、轻量级锁到重量级锁，全是悲观锁。JDK提供的Lock实现类全是悲观锁。其实只要有“锁对象”出现，那么就一定是悲观锁。因为**乐观锁不是锁，而是一个在循环里尝试CAS的算法。**

那JDK并发包里到底有没有乐观锁呢?

有。java.util.concurrent.atomic包里面的**原子类**都是利用乐观锁实现的。

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

原子类AtomicInteger的自增方法为乐观锁策略

为什么网上有些资料认为偏向锁、轻量级锁是乐观锁？理由是它们底层用到了CAS？或者是把“乐观/悲观”与“轻量/重量”搞混了？其实，线程在抢占这些锁的时候，确实是循环+CAS的操作，感觉好像是乐观锁。但问题的关键是，我们说一个锁是悲观锁还是乐观锁，总是应该站在应用层，看它们是如何锁住应用数据的，而不是站在底层看抢占锁的过程。如果一个线程尝试获取锁时，发现已经被占用，它是否继续读取数据，等后续要更新时再决定要不要重试？对于偏向锁、轻量级锁来说，显然答案是否定的。无论是挂起还是忙等，对应用数据的读取操作都被“挡住”了。从这个角度看，它们确实是悲观锁。

退一步讲，也没有必要在这些术语上狠钻牛角尖，最重要的是理解它们的运行机制。想写得尽量简单一些，却发现洋洋洒洒近万字，只讲了个皮毛。深知自己水平有限，不敢保证完全正确，只能说路漫漫其修远兮，望指正。

垃圾回收

引用计数算法（Reference Counting Collector）

堆中每个对象（不是引用）都有一个引用计数器。当一个对象被创建并初始化赋值后，该变量计数设置为1。每当有一个地方引用它时，计数器值就加1（a = b， b被引用，则b引用的对象计数+1）。当引用失效时（一个对象的某个引用超过了生命周期（出作用域后）或者被设置为一个新值时），计数器值就减1。任何引用计数为0的对象可以被当作垃圾收集。当一个对象被垃圾收集时，它引用的任何对象计数减1。

优点：引用计数收集器执行简单，判定效率高，交织在程序运行中。对程序不被长时间打断的实时环境比较有利（OC的内存管理使用该算法）。

缺点： 难以检测出对象之间的循环引用。同时，引用计数器增加了程序执行的开销。所以Java语言并没有选择这种算法进行垃圾回收。

早期的JVM使用引用计数，现在大多数JVM采用对象引用遍历（***根搜索算法***）。

根搜索算法（Tracing Collector）

首先了解一个概念：***根集(Root Set)***

所谓根集(Root Set)就是正在执行的Java程序可以访问的引用变量（注意：不是对象）的集合(包括局部变量、参数、类变量)，程序可以使用引用变量访问对象的属性和调用对象的方法。

这种算法的基本思路：

（1）通过一系列名为“GC Roots”的对象作为起始点，寻找对应的引用节点。

（2）找到这些引用节点后，从这些节点开始向下继续寻找它们的引用节点。

（3）重复（2）。

（4）搜索所走过的路径称为引用链，**当一个对象到GC Roots没有任何引用链相连时，就证明此对象是不可的。**

`Java和C#中都是采用根搜索算法来判定对象是否存活的。`

标记可达对象：

JVM中用到的所有现代GC算法在回收前都会先找出所有仍存活的对象。根搜索算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图。下图3.0中所展示的JVM中的内存布局可以用来很好地阐释这一概念：

图

3.0 标记（marking）对象

首先，垃圾回收器将某些特殊的对象定义为GC根对象。所谓的GC根对象包括：

（1）虚拟机栈中引用的对象（栈帧中的本地变量表）；

（2）方法区中的常量引用的对象；

（3）方法区中的类静态属性引用的对象；

（4）本地方法栈中JNI（Native方法）的引用对象。

（5）活跃线程。

接下来，垃圾回收器会对内存中的整个对象图进行遍历，它先从GC根对象开始，然后是根对象引用的其它对象，比如实例变量。回收器将访问到的所有对象都标记为存活。

存活对象在上图中被标记为蓝色。当标记阶段完成了之后，所有的存活对象都已经被标记完了。其它的那些（上图中灰色的那些）也就是GC根对象不可达的对象，也就是说你的应用不会再用到它们了。这些就是垃圾对象，回收器将会在接下来的阶段中清除它们。

关于标记阶段有几个关键点是值得注意的：

1. 开始进行标记前，需要先暂停应用线程，否则如果对象图一直在变化的话是无法真正去遍历它的。暂停应用线程以便JVM可以尽情地收拾家务的这种情况又被称之为**安全点**（Safe Point），这会触发一次Stop The World(STW)暂停。触发安全点的原因有许多，但最常见的应该就是垃圾回收了。
2. 暂停时间的长短并不取决于堆内对象的多少也不是堆的大小，而是存活对象的多少。因此，调高堆的大小并不会影响到标记阶段的时间长短。
3. 在根搜索算法中，要真正宣告一个对象死亡，至少要经历两次标记过程：
    1. 如果对象在进行根搜索后发现没有与GC Roots相连接的引用链，那它会被第一次标记并且进行一次筛选。筛选的条件是此对象是否有必要执行 finalize（）方法（可看作析构函数，类似于OC中的dealloc，Swift中的deinit）。当对象没有覆盖finalize（）方法，或finalize（）方法已经被虚拟机调用过，虚拟机将这两种情况都视为没有必要执行。
    2. 如果该对象被判定为有必要执行finalize（）方法，那么这个对象将会被放置在一个名为F-Queue队列中，并在稍后由一条由虚拟机自动建立的、低优先级的Finalizer线程去执行finalize（）方法。finalize（）方法是对象逃脱死亡命运的最后一次机会（因为一个对象的finalize（）方法最多只会被系统自动调用一次），稍后GC将对F-Queue中的对象进行第二次小规模的标记，如果要

在finalize（）方法中成功拯救自己，只要在finalize（）方法中让该对象重新引用链上的任何一个对象建立关联即可。而如果对象这时还没有关联到任何链上的引用，那它就会被回收掉。

4. 实际上GC判断对象是否可达看的是强引用。

   当标记阶段完成后，GC开始进入下一阶段，删除不可达对象。

Java 堆划分

1.年轻代（Young Generation）

几乎所有新生成的对象首先都是放在年轻代的。新生代内存按照**8:1:1**的比例分为**一个Eden区和两个Survivor (Survivor0, Survivor1)区**。大部分对象在Eden区中生成。当新对象生成，Eden Space申请失败（因为空间不足等），则会发起一次**GC(Scavenge GC)**。回收时先将Eden区存活对象复制到一个Survivor0区，然后清空Eden区，当这个Survivor0区也存放满了时，则将Eden区和Survivor0区存活对象复制到另一个Survivor1区，然后清空Eden和这个Survivor0区，此时Survivor0区是空的，然后将Survivor0区和Survivor1区交换，即保持Survivor1区为空， 如此往复。当Survivor1区不足以存放 Eden和Survivor0的存活对象时，就将存活对象直接存放到老年代。当对象在Survivor区躲过一次GC的话，其对象年龄便会加1，默认情况下，如果对象年龄达到15岁，就会移动到老年代中。若是老年代也满了就会触发一次Full GC，也就是新生代、老年代都进行回收。新生代大小可以由-Xmn来控制，也可以用-XX:SurvivorRatio来控制Eden和Survivor的比例。

2.年老代（Old Generation）

在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。一般来说，大对象会被直接分配到老年代。所谓的大对象是指需要大量连续存储空间的对象，最常见的一种大对象就是大数组。比如：

```
byte[] data = new byte[4*1024*1024]
```

这种一般会直接在老年代分配存储空间。

当然分配的规则并不是百分之百固定的，这要取决于当前使用的是哪种垃圾收集器组合和JVM的相关参数。

3.持久代（Permanent Generation）

用于存放静态文件（class类、方法）和常量等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。对永久代的回收主要回收两部分内容：废弃常量和无用的类。

永久代空间在Java SE8特性中已经被移除。取而代之的是元空间（MetaSpace）。因此不会再出现“java.lang.OutOfMemoryError: PermGen error”错误。

5.2 堆内存分配策略明确以下三点：

（1）对象优先在Eden分配。

（2）大对象直接进入老年代。

（3）长期存活的对象将进入老年代。

5.3 对垃圾回收机制说明以下三点：

新生代GC（Minor GC/Scavenge GC）：发生在新生代的垃圾收集动作。因为Java对象大多都具有朝生夕灭的特性，因此Minor GC非常频繁(不一定等Eden区满了才触发)，一般回收速度也比较快。在新生代中，每次垃圾收集时都会发现有大量对象死去，只有少量存活，因此可选用复制算法来完成收集。

老年代GC（Major GC/Full GC）：发生在老年代的垃圾回收动作。Major GC，经常会伴随至少一次Minor GC。由于老年代中的对象生命周期比较长，因此Major GC并不频繁，一般都是等待老年代满了后才进行Full GC，而且其速度一般会比Minor GC慢10倍以上。另外，如果分配了Direct Memory，在老年代中进行Full GC时，会顺便清理掉Direct Memory中的废弃对象。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记—清除算法或标记—整理算法来进行回收。

新生代采用空闲指针的方式来控制GC触发，指针保持最后一个分配的对象在新生代区间的位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发GC。当连续分配对象时，对象会逐渐从Eden到Survivor，最后到老年代。

用Java VisualVM来查看，能明显观察到新生代满了后，会把对象转移到旧生代，然后清空继续装载，当老年代也满了后，就会报outofmemory的异常

Java 函数引用

```
public class User {
    private String username;
    private Integer age;

    public User() {
    }

    public User(String username, Integer age) {
        this.username = username;
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" +
                "username='" + username + '\'' +
                ", age=" + age +
                '}';
    }

    // Getter&Setter
}
```

```
// Function<para1, para2, ...,  returnVal>
public static void main(String[] args) {
    // 使用双冒号::来构造静态函数引用
    Function<String, Integer> fun = Integer::parseInt;
    Integer value = fun.apply("123");
    System.out.println(value);

    // 使用双冒号::来构造非静态函数引用
    String content = "Hello JDK8";
    Function<Integer, String> func = content::substring;
    String result = func.apply(1);
    System.out.println(result);

    // 构造函数引用
    BiFunction<String, Integer, User> biFunction = User::new;
    User user = biFunction.apply("mengday", 28);
    System.out.println(user.toString());

    // 函数引用也是一种函数式接口，所以也可以将函数引用作为方法的参数
    sayHello(String::toUpperCase, "hello");
}

// 方法有两个参数，一个是
private static void sayHello(Function<String, String> func, String parameter){
    String result = func.apply(parameter);
    System.out.println(result);
}
```

比较：== 和equals（）

在我们实现自己的equals方法之前，equals等价于`==`,而`==`运算符是判断两个对象是不是同一个对象，即他们的**地址是否相等**。而覆写equals更多的是追求两个对象在**逻辑上的相等**，你可以说是**值相等**，也可说是**内容相等**。

> **自反性**：对于任何非空引用值 x，x.equals(x) 都应返回 true。

> **对称性**：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。

> **传递性**：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true， 并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。

> **一致性**：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false， 前提是对象上 equals 比较中所用的信息没有被修改。

> **非空性**：对于任何非空引用值 x，x.equals(null) 都应返回 false。

```
public boolean equals(Object anObject) {
        if (this == anObject) {
            return true;
        }
        if (anObject instanceof String) {
            String anotherString = (String)anObject;
            int n = value.length;
            if (n == anotherString.value.length) {
                char v1[] = value;
                char v2[] = anotherString.value;
                int i = 0;
                while (n-- != 0) {
                    if (v1[i] != v2[i])
                        return false;
                    i++;
                }
                return true;
            }
        }
        return false;
    }
```

上面的equals有以下几点诀窍：

- **使用==操作符检查“参数是否为这个对象的引用”**：如果是对象本身，则直接返回，拦截了对本身调用的情况，算是一种性能优化。
- **使用instanceof操作符检查“参数是否是正确的类型”**：如果不是，就返回false，正如对称性和传递性举例子中说得，不要想着兼容别的类型，很容易出错。在实践中检查的类型多半是equals所在类的类型，或者是该类实现的接口的类型，比如Set、List、Map这些集合接口。
- **把参数转化为正确的类型**： 经历了上一步的检测，基本会成功。
- **对于该类中的“关键域”，检查参数中的域是否与对象中的对应域相等**：基本类型的域就用`==`比较，float域用Float.compare方法，double域用Double.compare方法，至于别的引用域，我们一般递归调用它们的equals方法比较，加上判空检查和对自身引用的检查，一般会写成这样：`(field == o.field || (field != null && field.equals(o.field)))`,而上面的String里使用的是数组，所以只要把数组中的每一位拿出来比较就可以了。
- **编写完成后思考是否满足上面提到的对称性，传递性，一致性等等。**

还有一些注意点。

覆盖equals时一定要覆盖hashCode

equals函数里面一定要是Object类型作为参数

equals方法本身不要过于智能，只要判断一些值相等即可。

Integer == equals

object一般是不能直接比较的，但是比较两个Integer的时候如果value 在-128 - 127之间会自动拆箱使用Integer.valueOf() 来比较值的大小。若超出范围则比较引用地址，就是false了