c/c++ 面试题库 v1.0

版本	修订内容	日期	作者
V1.0	创建	2019/7/14	SHANE

1. 基础部分

1.1.语言

一、 C 和 C++的特点与区别？

参考答案：

答：（1）C 语言特点：
1） 作为一种面向过程的结构化语言，易于调试和维护；
2） 表现能力和处理能力极强，可以直接访问内存的物理地址；
3） C 语言实现了对硬件的编程操作，也适合于应用软件的开发；
4） C 语言还具有效率高，可移植性强等特点。

（2）C++语言特点：
1） 在 C 语言的基础上进行扩充和完善，使 C++兼容了 C 语言的面向过程特点，又成为了一种面向对象的程序设计语言；
2） 可以使用抽象数据类型进行基于对象的编程；
3） 可以使用多继承、多态进行面向对象的编程；
4） 可以担负起以模版为特征的泛型化编程。

二、 阐述 C++的多态

参考答案：
编译时多态：主要指泛型编程

运行时多态：

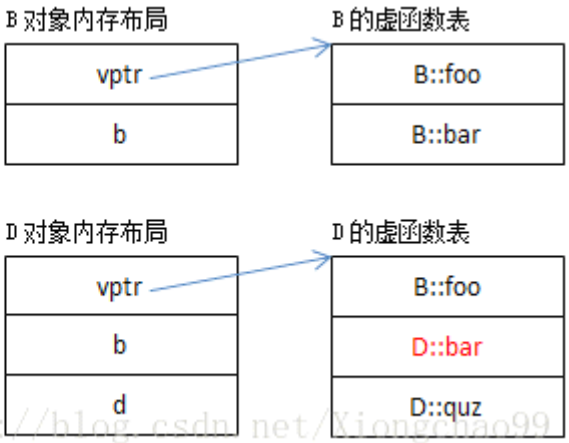
C++的多态性用一句话概括：在基类的函数前加上 virtual 关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数。

1） 用 virtual 关键字申明的函数叫做虚函数，虚函数肯定是类的成员函数；
2） 存在虚函数的类都有一个一维的虚函数表叫做虚表，类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的；
3） 多态性是一个接口多种实现，是面向对象的核心，分为类的多态性和函数的多态性。；
4） 多态用虚函数来实现，结合动态绑定.；
5） 纯虚函数是虚函数再加上 = 0；
6） 抽象类是指包括至少一个纯虚函数的类；
纯虚函数：virtual void fun()=0;即抽象类，必须在子类实现这个函数，即先有名称，没有内容，在派生类实现内容。

三、 阐释 c++虚函数的实现

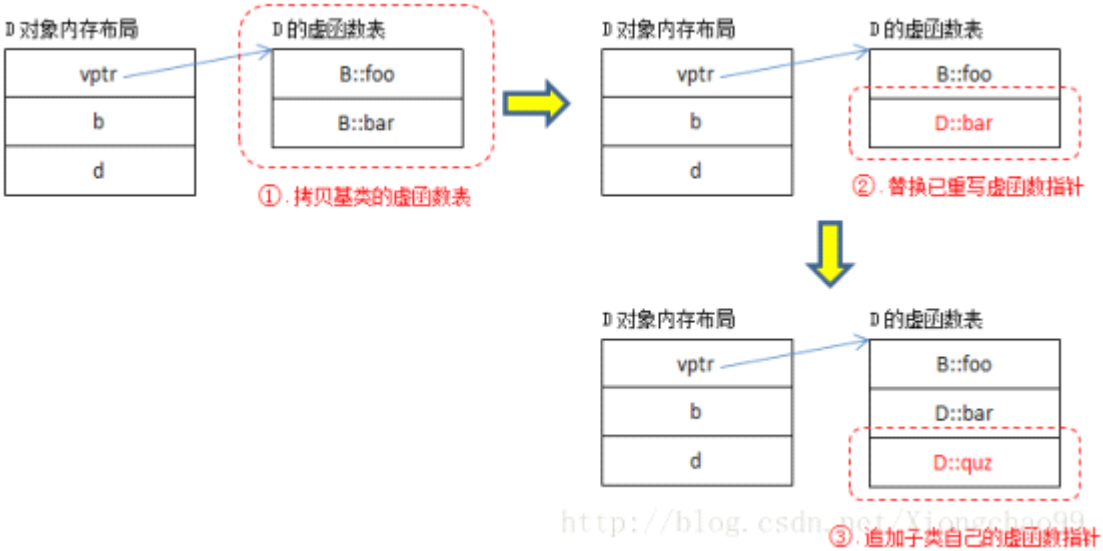
参考答案：
简单地说，每一个含有虚函数（无论是其本身的，还是继承而来的）的类都至少有一个与之对应的虚函数表，其中存放着该类所有的虚函数对应的函数指针。
例：

```
struct B {
    long b;
    virtual void foo() {}
    virtual void bar() {}
};
struct D : public B {
    long d;
    virtual void bar() {}
    virtual void quz() {}
};
```

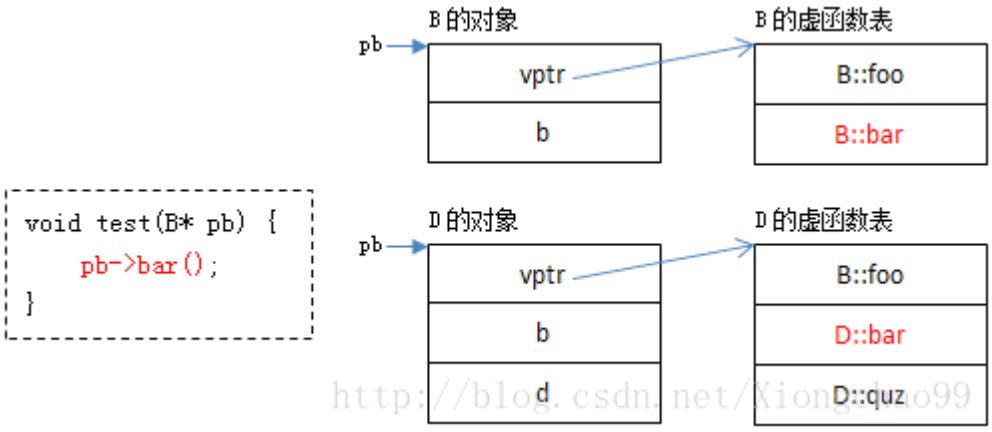


其中：
B 的虚函数表中存放着 B::foo 和 B::bar 两个函数指针。
D 的虚函数表中存放的既有继承自 B 的虚函数 B::foo，又有重写（override）了基类虚函数 B::bar 的 D::bar，还有新增的虚函数 D::quz。

虚函数表构造过程：
从编译器的角度来说，B 的虚函数表很好构造，D 的虚函数表构造过程相对复杂。下面给出了构造 D 的虚函数表的一种方式（仅供参考）：



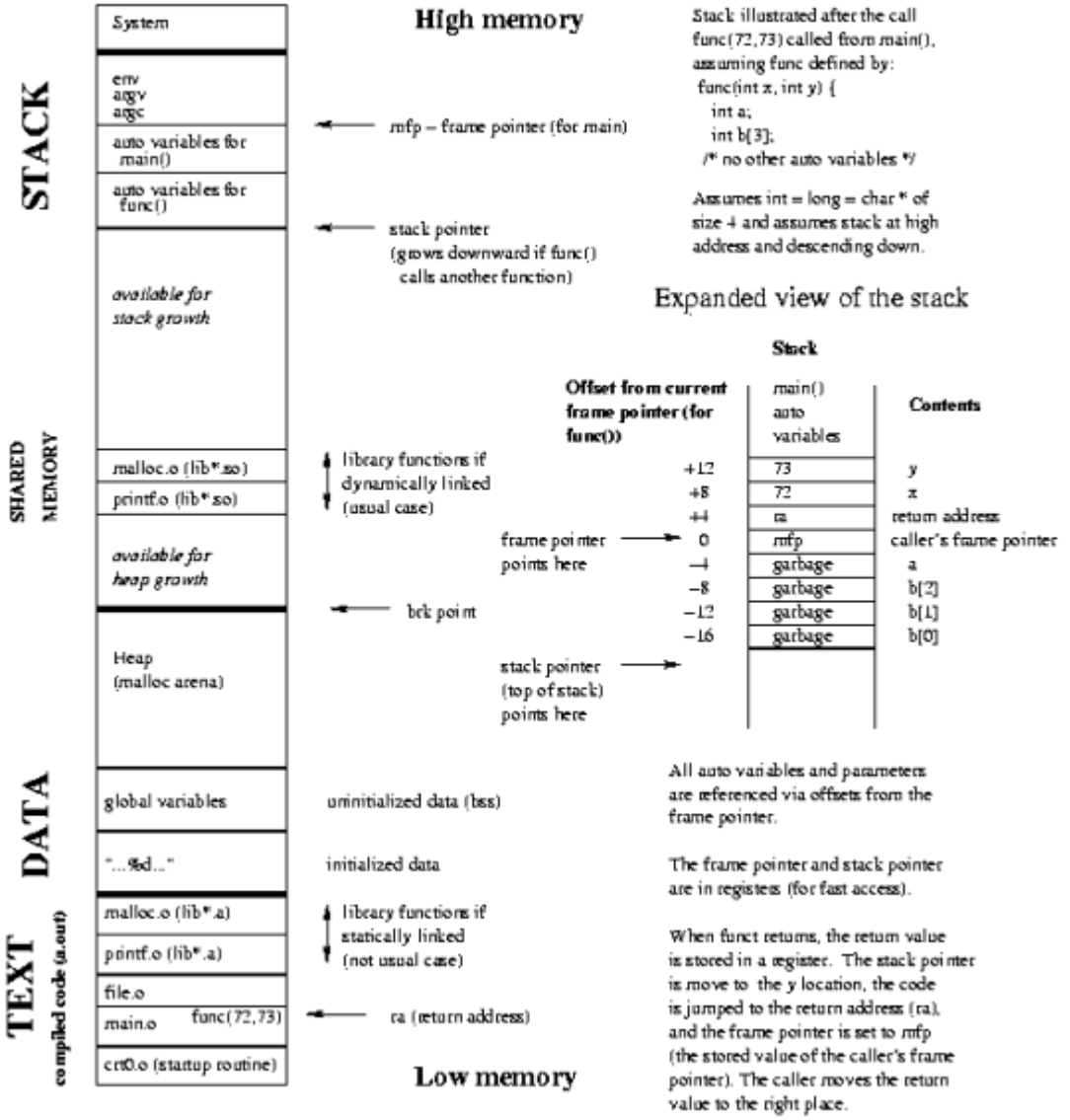
虚函数调用过程
以下面的程序为例：

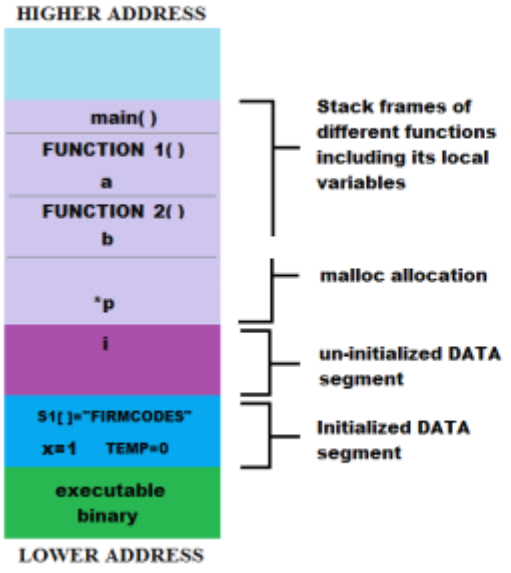


四、 C和C++内存分配问题

参考答案：

Memory Layout (Virtual address space of a C process)






```c
#include<stdio.h>
#include<malloc.h>

void FUNCTION_1();
void FUNCTION_2();

char S1[]="FIRMCODES";  //initialized read-write area of DATA segment
int i;                  //uninitialized DATA segment
const int x=1;          //initialized read-only area of DATA segment

int main()
{
    static int TEMP=0;  //uninitialized DATA segment

    char *p=(char*)malloc(sizeof(char)); //Heap segment

    FUNCTION_1();       //FUNCTION_1 stack frame

    return 0;
}

void FUNCTION_1()
{
    int a;          //initialized in stack frame of FUNCTION_1

    FUNCTION_2();   //FUNCTION_2 stack frame
}

void FUNCTION_2()
{
    int b;          //initialized in stack frame of FUNCTION_2
}
```

（1）C 语言编程中的内存基本构成

C 的内存基本上分为 4 部分：静态存储区、堆区、栈区以及常量区。他们的功能不同，对他们使用方式也就不同。

1.栈 ——由编译器自动分配释放；
2.堆 ——一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收；
3.全局区（静态区）——全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域（C++中已经不再这样划分），程序结束释放；
4.另外还有一个专门放常量的地方，程序结束释放；
(a)函数体中定义的变量通常是在栈上；
(b)用 malloc, calloc, realloc 等分配内存的函数分配得到的就是在堆上；
(c)在所有函数体外定义的是全局量；
(d)加了 static 修饰符后不管在哪里都存放在全局区（静态区）；
(e)在所有函数体外定义的 static 变量表示在该文件中有效，不能 extern 到别的文件用；
(f)在函数体内定义的 static 表示只在该函数体内有效；
(g)另外，函数中的"adgfdf"这样的字符串存放在常量区。

（2）C++编程中的内存基本构造

在 C++中内存分成 5 个区，分别是堆、栈、全局/静态存储区、常量存储区和代码区；
1、栈，就是那些由编译器在需要的时候分配，在不需要的时候自动清楚的变量的存储区，里面的变量通常是局部变量、函数参数等。
2、堆，就是那些由 new 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个 new 就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。

3、全局/静态存储区，全局变量和静态变量被分配到同一块内存中，在以前的 C 语言中，全局变量又分为初始化的和未初始化的，在 C++里面没有这个区分了，他们共同占用同一块内存区。

4、常量存储区，这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改（当然，你要通过非正当手段也可以修改）。

5、代码区 （.text 段），存放代码（如函数），不允许修改（类似常量存储区），但可以执行（不同于常量存储区）。

内存模型组成部分：自由存储区，动态区、静态区；
根据 c/c++对象生命周期不同，c/c++的内存模型有三种不同的内存区域，即：自由存储区，动态区、静态区。
自由存储区：局部非静态变量的存储区域，即平常所说的栈；
动态区： 用 new ，malloc 分配的内存，即平常所说的堆；
静态区：全局变量，静态变量，字符串常量存在的位置；
注：代码虽然占内存，但不属于 c/c++内存模型的一部分；

一个正在运行着的 C 编译程序占用的内存分为 5 个部分：代码区、初始化数据区、未初始化数据区、堆区 和栈区；

（1）代码区（text segment）：代码区指令根据程序设计流程依次执行，对于顺序指令，则只会执行一次（每个进程），如果反复，则需要使用跳转指令，如果进行递归，则需要借助栈来实现。注意：代码区的指令中包括操作码和要操作的对象（或对象地址引用）。如果是立即数（即具体的数值，如 5），将直接包含在代码中；
（2）全局初始化数据区/静态数据区（Data Segment）：只初始化一次。
（3）未初始化数据区（BSS）：在运行时改变其值。
（4）栈区（stack）：由编译器自动分配释放，存放函数的参数值、局部变量的值等，其操作方式类似于数据结构中的栈。
（5）堆区（heap）：用于动态内存分配。

为什么分成这么多个区域?
主要基于以下考虑：

#代码是根据流程依次执行的，一般只需要访问一次，而数据一般都需要访问多次，因此单独开辟空间以方便访问和节约空间。
#未初始化数据区在运行时放入栈区中，生命周期短。
#全局数据和静态数据有可能在整个程序执行过程中都需要访问，因此单独存储管理。
#堆区由用户自由分配，以便管理。

五、 数据模型（LP32 ILP32 LP64 LLP64 ILP64 ）

32 位环境涉及"ILP32"数据模型，是因为 C 数据类型为 32 位的 int、long、指针。而 64 位环境使用不同的数据模型，此时的 long 和指针已为 64 位，故称作"LP64"数据模型。

现今所有 64 位的类 Unix 平台均使用 LP64 数据模型，而 64 位 Windows 使用 LLP64 数据模型，除了指针是 64 位，其他基本类型都没有变。

Data Type	ILP32	LP32	ILP64	LP64	LLP64
宏定义	_	_	_	__LP64__	__LLP64__
平台	Win32 API / Unix 和 Unix 类的系统（Linux，Mac OS X）	Win16 API		Unix 和 Unix 类的系统（Linux，Mac OS X)	Win64 API
char	8	8	8	8	8
short	16	16	16	16	16
int	32	32	64	32	32
long	32	32	64	64	32
long long	64	64	64	64	64
pointer	32	32	64	64	64

在这张表中，LP64，ILP64，LLP64 是 64 位平台上的字长模型，ILP32 和 LP32 是 32 位平台上的字长模型。

LP64 意思是 long 和 pointer 是 64 位，

ILP64 指 int，long，pointer 是 64 位，

LLP64 指 long long 和 pointer 是 64-bit 的。

ILP32 指 int，long 和 pointer 是 32 位的，

LP32 指 long 和 pointer 是 32 位的。

float 都是 4 字节；
double 都是 8 字节；（C 中直接写小数，默认是 double 型）

这 3 个 64 位模型（LP64、LLP64 和 ILP64）之间的区别在于非浮点数据类型。当一个或多个 C 数据类型的宽度从一种模型变换成另外一种模型时，应用程序可能会受到很多方面的影响。这些影响主要可以分为两类：

- **数据对象的大小**。编译器按照自然边界对数据类型进行对齐；换而言之，32 位的数据类型在 64 位系统上要按照 32 位边界进行对齐，而 64 位的数据类型在 64 位系统上则要按照 64 位边界进行对齐。这意味着诸如结构或联合之类的数据对象的大小在 32 位和 64 位系统上是不同的。

- **基本数据类型的大小**。通常关于基本数据类型之间关系的假设在 64 位数据模型上都已经无效了。依赖于这些关系的应用程序在 64 位平台上编译也会失败。例如，`sizeof (int) = sizeof (long) = sizeof (pointer)` 的假设对于 ILP32 数据模型有效，但是对于其他数据模型就无效了。

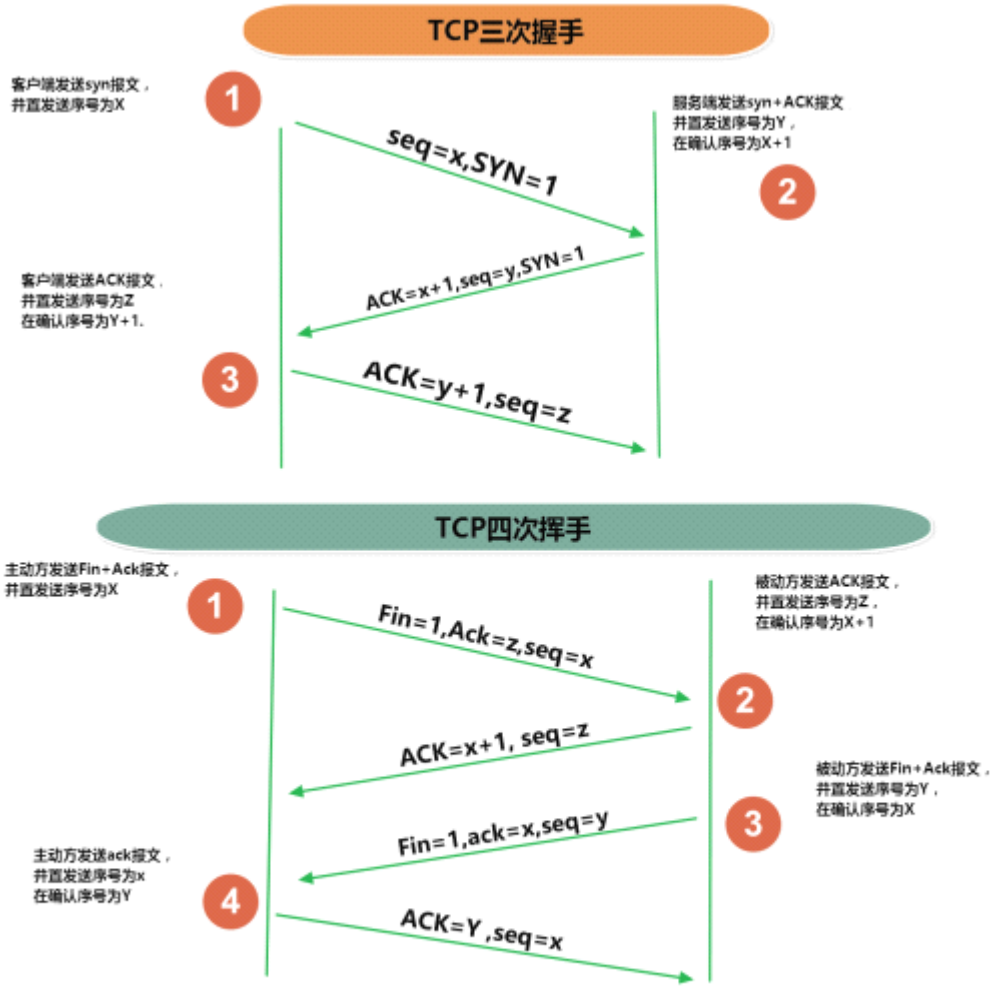
总之，编译器要按照自然边界对数据类型进行对齐，这意味着编译器会进行“填充”，从而强制进行这种方式的对齐，就像是在 C 结构和联合中所做的一样。结构或联合的成员是根据最宽的成员进行对齐的。

2. 高级部分

2.1.网络

六、 画出三次握手和四次挥手流程图

参考答案：



七、 请阐释 https 建立连接过程

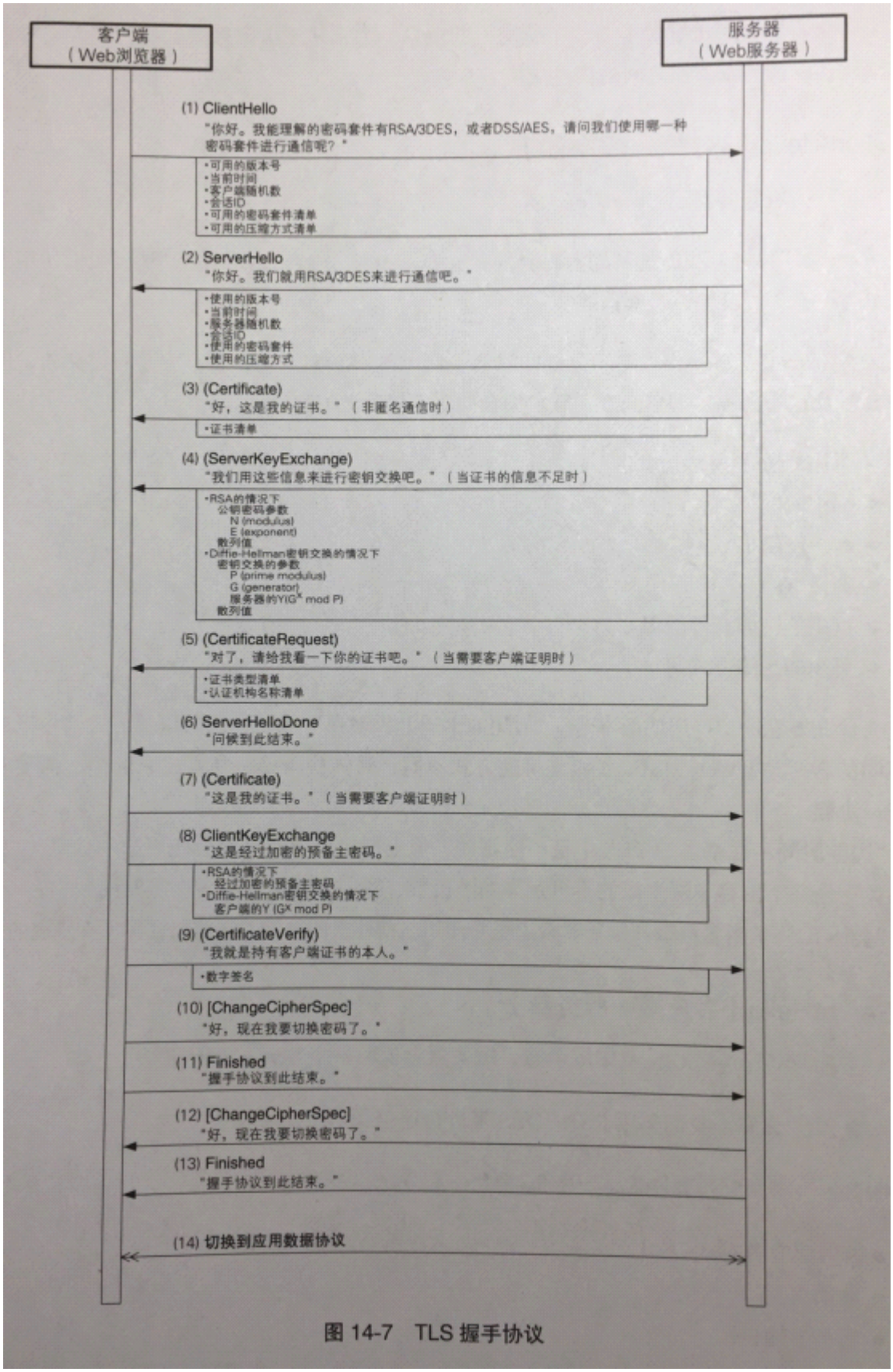
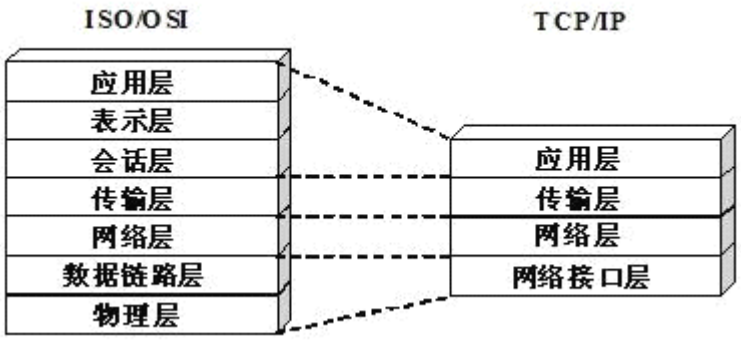


图 14-7　TLS 握手协议

参考：https://wetest.qq.com/lab/view/110.html

八、 画出 OSI 和 TCP/IP 协议栈的对应关系

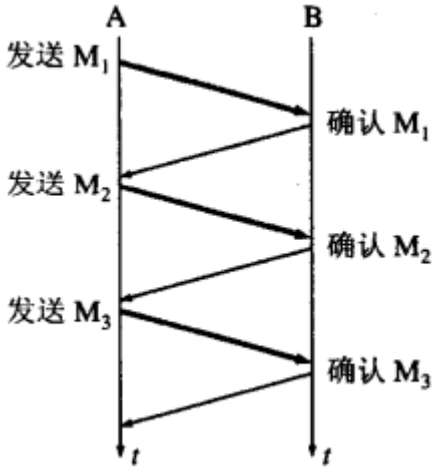


九、 请阐释 ARQ 协议的原理和过程

参考答案：
ARQ 协议，即自动重传请求（Automatic Repeat-reQuest），是 OSI 模型中数据链路层和传输层的错误纠正协议之一。它通过使用确认和超时这两个机制，在不可靠服务的基础上实现可靠的信息传输。如果发送方在发送后一段时间之内没有收到确认帧，它通常会重新发送。ARQ 包括停止等待 ARQ 协议和连续 ARQ 协议，拥有错误检测（Error Detection）、正面确认（Positive Acknowledgment）、超时重传（Retransmission after Timeout）和负面确认及重传（Negative Acknowledgment and Retransmission）等机制。

（1）停止等待 ARQ 协议
要想弄明白为什么 TCP 要使用连续 ARQ 协议，首先需要弄清楚停止等待 ARQ 协议的原理。TCP 连接是全双工的连接，也就是说在通信的时候，双方既是发送方，也是接收方。下面为了简化问题，只考虑一方发送，一方接受的情况。其中，A 作为发送方，B 作为接收方。

1.无差错情况
A 发送分组 M1，发送完就暂停发送，等待 B 的确认。B 收到 M1 就向 A 发送确认。A 在收到了对 M1 的确认后，就再发送下一个分组 M2。依次下去发送剩余的数据...如下图所示：

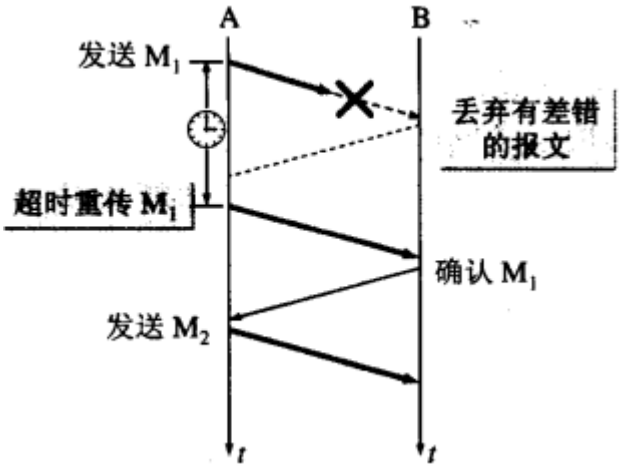


2.出现差错

如果 A 发送的过程中出现差错，B 在接收 M1 时检测出了差错，就丢弃 M1，其他什么都不做（也不会通知 A 收到有差错的分组）。又或者 A 传送的过程中分组丢失了，以上这两种情况下，B 不会发送任何信息。

既然说它是可靠传输协议，那自然有它可靠的方法：如果发生以上的情况，A 只要超过了一段时间仍然没有收到确认，就认为刚才发送的分组丢失了，所以它会重传刚刚的发送过的分组，也就是所谓的超时重传。

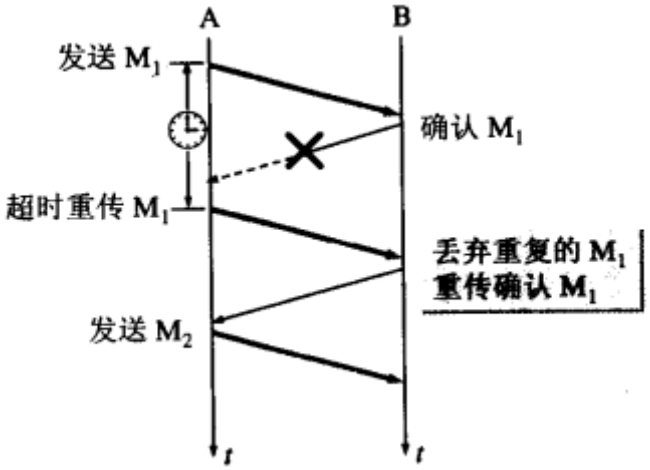
超时重传的原理也很简单：发送方发送完一个分组后，就会设置一个超时计时器，如果超时计时器到期之前没有收到接收方发来的确认信息，则会重发刚发送过的分组；如果收到确认信息，则撤销该超时计时器。如下图所示：



这里应该注意的是：
①既然发送方发送的分组可能丢失或者有差错，可能需要重传，那么它必须暂时保留已发送的分组副本，只有收到确认后，才清除这个副本。

②分组和确认分组信息都应该有各自的编号，用来标示每一个分组和确认信息。（这样才知道需要发送哪个分组，收到了哪个分组的确认信息）

③超时计时器设置的时间应该略长于分组传送往返时间。



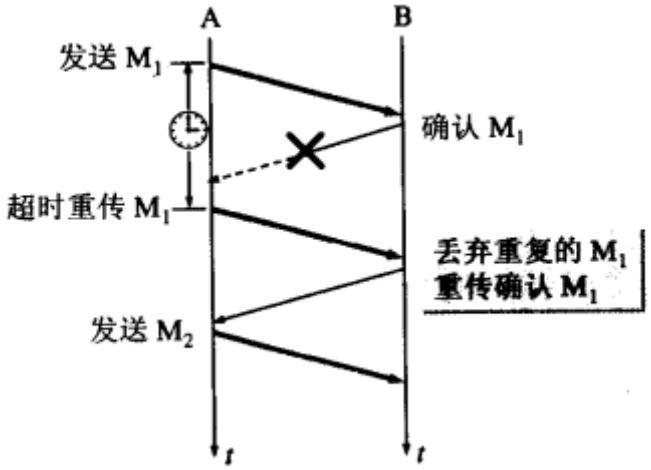
3.确认丢失和确认延迟
没有正常进行通信，除了发送方出现问题外，接收方同时也可能存在问题。

例如，如果A发送了M1分组，到达B，B发送了M1确认信息，但由于网络原因，该确认信息丢失。那么这个时候，A在超时重传时间内，没有收到B的确认信息，而且它并不知道是自己的分组有差错、丢失，还是B发生的确认丢失了。因此，A会在超时重传过后，重传M1分组。

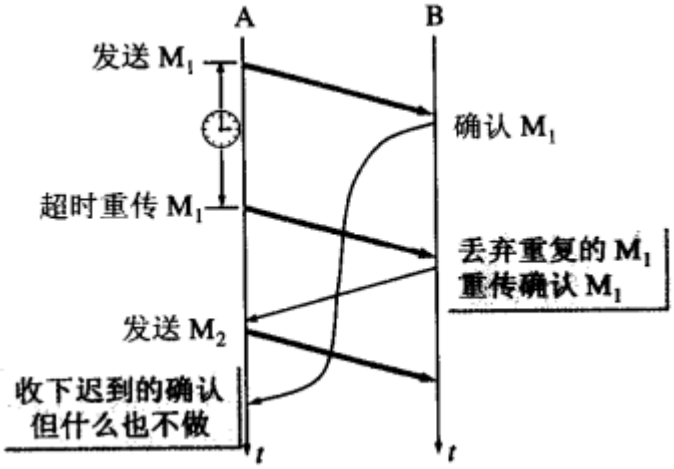
接收方B会采取这两个行动：

①B会丢弃M1分组，不向上层交付。（B之前已经收到过M1分组了）

②向A发送确认（因为A重发了，肯定重传时间内没有收到确认信息）



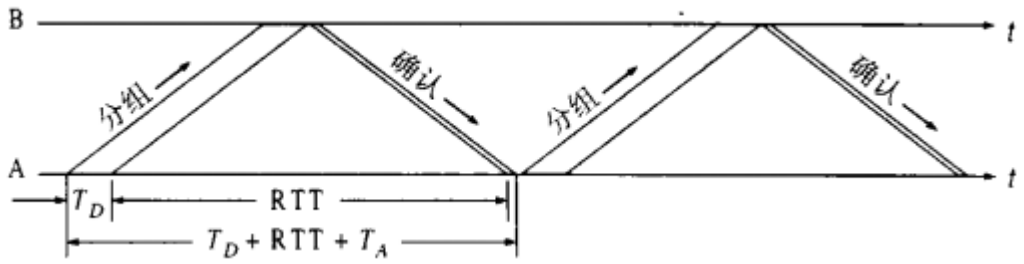
还有可能是另一种情况，就是B发送了确认，没有丢失，但是延迟了。也就是说，B发送的确认在A超时计时器过期后才到达。 这种情况下，A收到确认信息后会丢弃，然后重传刚才的分组，B收到后，丢弃重复的分组，并重传确认信息。



根据上述的确认和重传机制，我们就可以在不可靠的网络上实现可靠的传输。

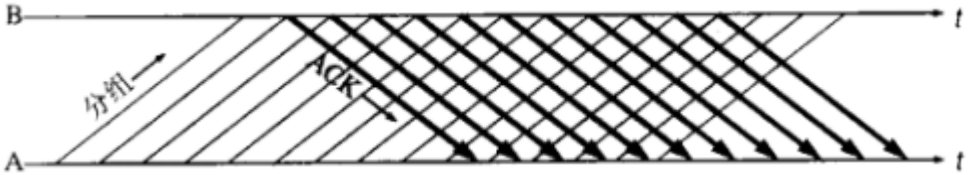
4.信道利用率

停止等待 ARQ 协议的优点是简答，但也有很严重的确定，就是信道利用率太低。如下图所示：

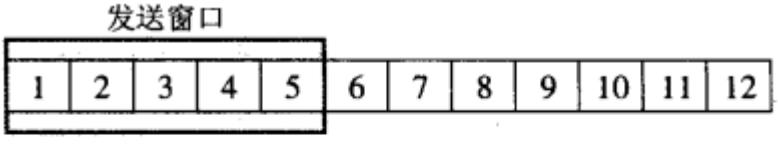


信道利用率 U = TD / (TD + RTT + TA)

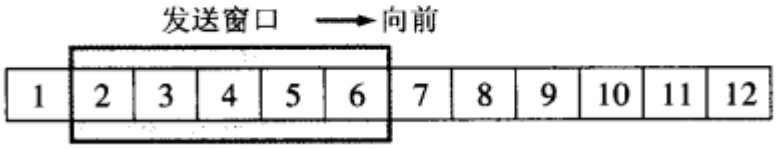
连续 ARQ 协议
由于停止等待 ARQ 协议信道利用率太低，所以需要使用连续 ARQ 协议来进行改善。这个协议会连续发送一组数据包，然后再等待这些数据包的 ACK。
发送方采用流水线传输。流水线传输就是发送方可以连续发送多个分组，不必每发完一个分组就停下来等待对方确认。如下图所示：



连续 ARQ 协议通常是结合滑动窗口协议来使用的，发送方需要维持一个发送窗口，如下图所示：



(a) 发送窗口是 5



(b) 收到一个确认后发送窗口向前滑动

图（a）是发送方维持的发送窗口，它的意义是：位于发送窗口内的 5 个分组都可以连续发送出去，而不需要等待对方的确认，这样就提高了信道利用率。

连续 ARQ 协议规定，发送方每收到一个确认，就把发送窗口向前滑动一个分组的位置。例如上面的图（b），当发送方收到第一个分组的确认，就把发送窗口向前移动一个分组的位置。如果原来已经发送了前 5 个分组，则现在可以发送窗口内的第 6 个分组。

接收方一般都是采用累积确认的方式。也就是说接收方不必对收到的分组逐个发送确认。而是在收到几个分组后，对按序到达的最后一个分组发送确认。如果收到了这个分组确认信息，则表示到这个分组为止的所有分组都已经正确接收到了。

累积确认的优点是容易实现，即使确认丢失也不必重传。但缺点是，不能正确的向发送方反映出接收方已经正确收到的所以分组的信息。比如发送方发送了前 5 个分组，而中间的第 3 个分组丢失了，这时候接收方只能对前 2 个发出确认。而不知道后面 3 个分组的下落，因此只能把后面的 3 个分组都重传一次，这种机制叫 Go-back-N（回退 N），表示需要再退回来重传已发送过的 N 个分组。

十、 请阐释滑动窗口协议原理和过程

滑动窗口协议在在发送方和接收方之间各自维持一个滑动窗口，发送发是发送窗口，接收方是接收窗口，而且这个窗口是随着时间变化可以向前滑动的。它允许发送方发送多个分组而不需等待确认。TCP 的滑动窗口是以字节为单位的。
如下图所示，发送窗口中有四个概念：：已发送并收到确认的数据（不在发送窗口和发送缓冲区之内）、已发送但未收到确认的数据（位于发送窗口之内）、允许发送但尚未发送的数据（位于发送窗口之内）、发送窗口之外的缓冲区内暂时不允许发送的数据。
接收窗口中也有四个概念：已发送确认并交付主机的数据（不在接收窗口和接收缓冲区之内）、未按序收到的数据（位于接收窗口之内）、允许的数据（位于接收窗口之内）、不允许接收的数据（位于发送窗口之内）。

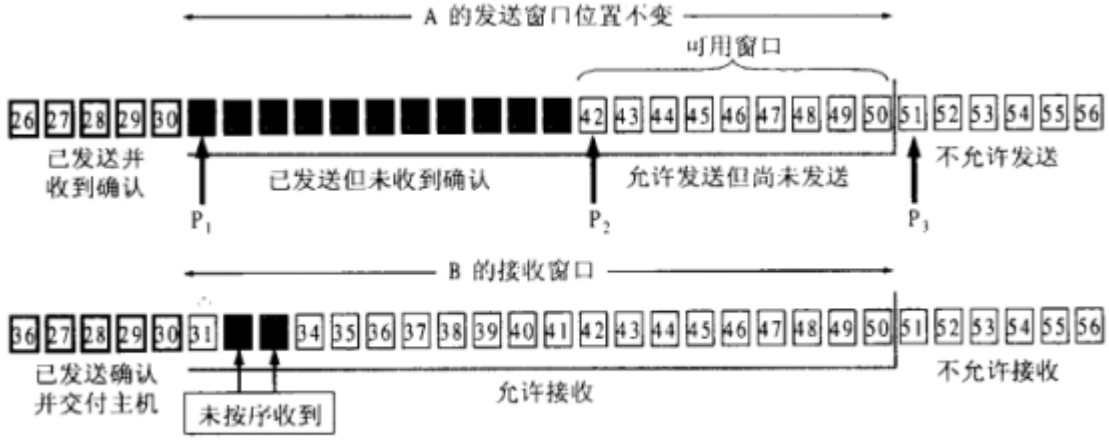


图 5-16　A 发送了 11 个字节的数据

规则：

（1）凡是已经发送过的数据，在未收到确认之前，都必须暂时保留，以便在超时重传时使用。
（2）只有当发送方 A 收到了接收方的确认报文段时，发送方窗口才可以向前滑动几个序号。
（3）当发送方 A 发送的数据经过一段时间没有收到确认（由超时计时器控制），就要使用回退 N 步协议，回到最后接收到确认号的地方，重新发送这部分数据。

此外，TCP 利用滑动窗口协议来进行流量控制，如下图所示

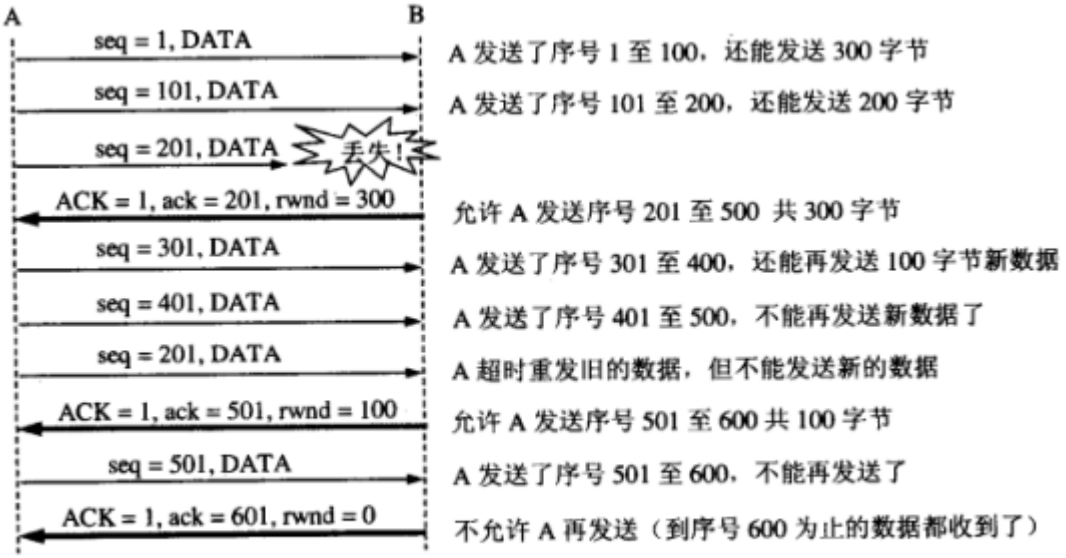


图 5-22　利用可变窗口进行流量控制举例

2.2.操作系统

十一、　　进程通信方式有哪些？哪种效率最高？

参考答案：

1)　管道(pipe)

管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

2)　有名管道 (namedpipe)

有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

3)　信号量(semophore)

信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

4)　消息队列(messagequeue)

消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

5)　信号 (sinal)

信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

6)　共享内存(shared memory)

共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

7) 套接字(socket)

套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同设备及其间的进程通信。

十二、 线程间的通信方式

1) 锁机制：包括互斥锁、条件变量、读写锁

互斥锁提供了以排他方式防止数据结构被并发修改的方法。 读写锁允许多个线程同时读共享数据，而对写操作是互斥的。 条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

2) 信号量机制(Semaphore)：包括无名线程信号量和命名线程信号量

3) 信号机制(Signal)：类似进程间的信号处理

线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制。

十三、 请分别阐释进程和线程的概念，并分析异同？

进程

进程是资源（CPU、内存等）分配的基本单位，它是程序执行时的一个实例。程序运行时系统就会创建一个进程，并为它分配资源，然后把该进程放入进程就绪队列，进程调度器选中它的时候就会为它分配 CPU 时间，程序开始真正运行。

Linux 系统函数 fork()可以在父进程中创建一个子进程，这样的话，在一个进程接到来自客户端新的请求时就可以复制出一个子进程让其来处理，父进程只需负责监控请求的到来，然后创建子进程让其去处理，这样就能做到并发处理。

多线程

线程是程序执行时的最小单位，它是进程的一个执行流，是 CPU 调度和分派的基本单位，一个进程可以由很多个线程组成，线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。线程由 CPU 独立调度执行，在多 CPU 环境下就允许多个线程同时运行。同样多线程也可以实现并发操作，每个请求分配一个线程来处理。

线程和进程各自有什么区别和优劣呢？

1) 进程是资源分配的最小单位，线程是程序执行的最小单位。

2) 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而线程是共享进程中的数据的，使用相同的地址空间，因此 CPU 切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。

3) 线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。不过如何处理好同步与互斥是编写多线程程序的难点。

4) 但是多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

2.3.数据库

十四、 MySQL 中 myisam 与 innodb 的区别

五点不同：

1) InnoDB 支持事物，而 MyISAM 不支持事物
2) InnoDB 支持行级锁，而 MyISAM 支持表级锁
3) InnoDB 支持 MVCC, 而 MyISAM 不支持
4) InnoDB 支持外键，而 MyISAM 不支持
5) InnoDB 不支持全文索引，而 MyISAM 支持。

十五、 innodb 引擎的 4 大特性

1) 插入缓冲（insert buffer),、
2) 二次写(double write),
3) 自适应哈希索引(ahi),
4) 预读(read ahead)

十六、 MyISAM 和 InnoDB selectcount(*)哪个更快，为什么

myisam 更快，因为 myisam 内部维护了一个计数器，可以直接调取。

十七、 Redis 支持的数据类型？

1) String 字符串：
格式: set key value
string 类型是二进制安全的。意思是 redis 的 string 可以包含任何数据。比如 jpg 图片或者序列化的对象 。
string 类型是 Redis 最基本的数据类型，一个键最大能存储 512MB。

2) Hash（哈希）
格式: hmset name key1 value1 key2 value2
Redis hash 是一个键值(key=>value)对集合。
Redis hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。

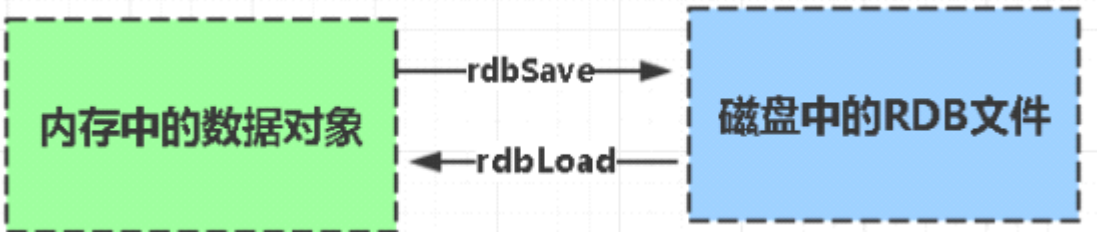
3) List（列表）
Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）

格式: lpush name value
在 key 对应 list 的头部添加字符串元素
格式: rpush name value
在 key 对应 list 的尾部添加字符串元素
格式: lrem name index
key 对应 list 中删除 count 个和 value 相同的元素
格式: llen name
返回 key 对应 list 的长度

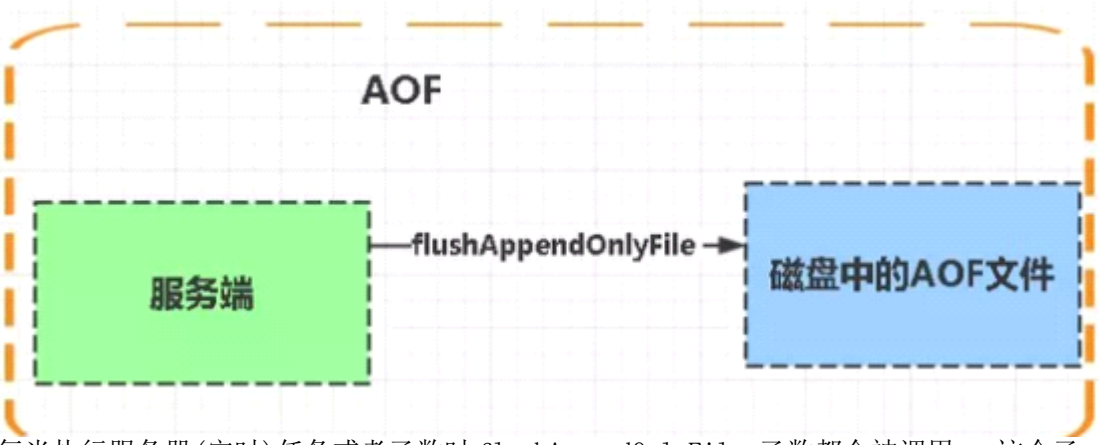
4) Set（集合）
格式: sadd name value
Redis 的 Set 是 string 类型的无序集合。
集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 O(1)。
zset(sorted set：有序集合)
格式: zadd name score value
Redis zset 和 set 一样也是 string 类型元素的集合,且不允许重复的成员。
不同的是每个元素都会关联一个 double 类型的分数。redis 正是通过分数来为集合中的成员进行从小到大的排序。
zset 的成员是唯一的,但分数(score)却可以重复。

十八、 什么是 Redis 持久化？Redis 有哪几种持久化方式？优缺点是什么？

持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。
Redis 提供了两种持久化方式:RDB（默认） 和 AOF
RDB：
rdb 是 Redis DataBase 缩写
功能核心函数 rdbSave(生成 RDB 文件)和 rdbLoad（从文件加载内存）两个函数



AOF:
Aof 是 Append-only file 缩写



每当执行服务器(定时)任务或者函数时 flushAppendOnlyFile 函数都会被调用， 这个函数执行以下两个工作

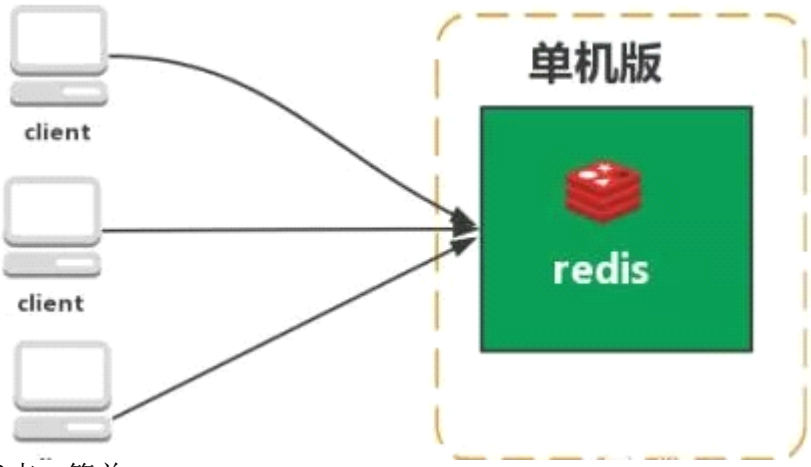
aof 写入保存：
WRITE：根据条件，将 aof_buf 中的缓存写入到 AOF 文件
SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。
存储结构:
内容是 redis 通讯协议(RESP)格式的命令文本存储。
比较：
1) aof 文件比 rdb 更新频率高，优先使用 aof 还原数据。
2) aof 比 rdb 更安全也更大
3) rdb 性能比 aof 好

十九、 redis 通讯协议(RESP)，能解释下什么是 RESP？有什么特点？

RESP 是 redis 客户端和服务端之前使用的一种通讯协议；
1) RESP 的特点：实现简单、快速解析、可读性好
2) For Simple Strings the first byte of the reply is "+" 回复
3) For Errors the first byte of the reply is "-" 错误
4) For Integers the first byte of the reply is ":" 整数
5) For Bulk Strings the first byte of the reply is "$" 字符串
6) For Arrays the first byte of the reply is "*" 数组

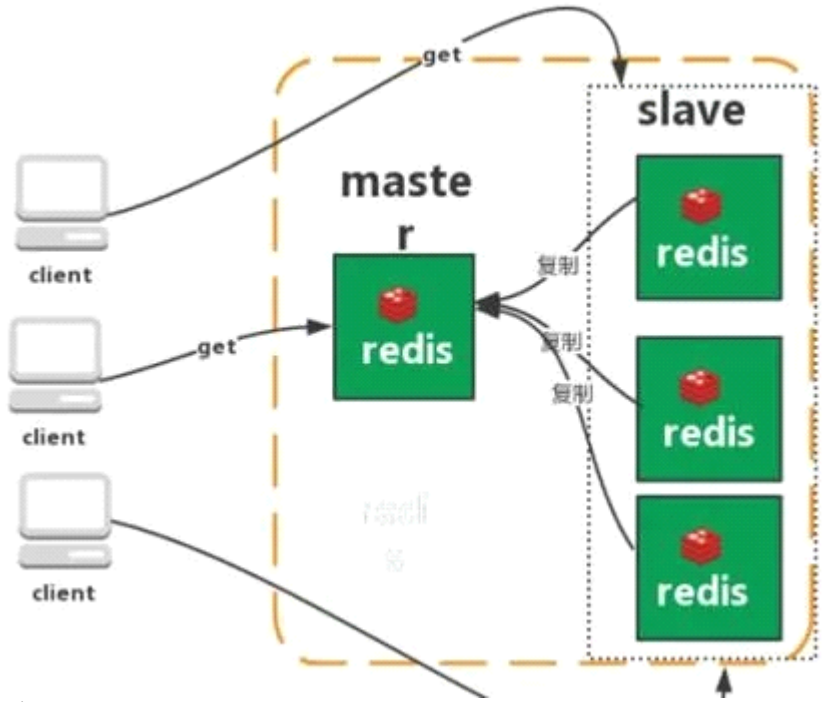
二十、 Redis 有哪些架构模式？讲讲各自的特点

单机版本：



优点：简单
问题：
1） 内存容量有限
2） 处理能力有限
3） 无法高可用。

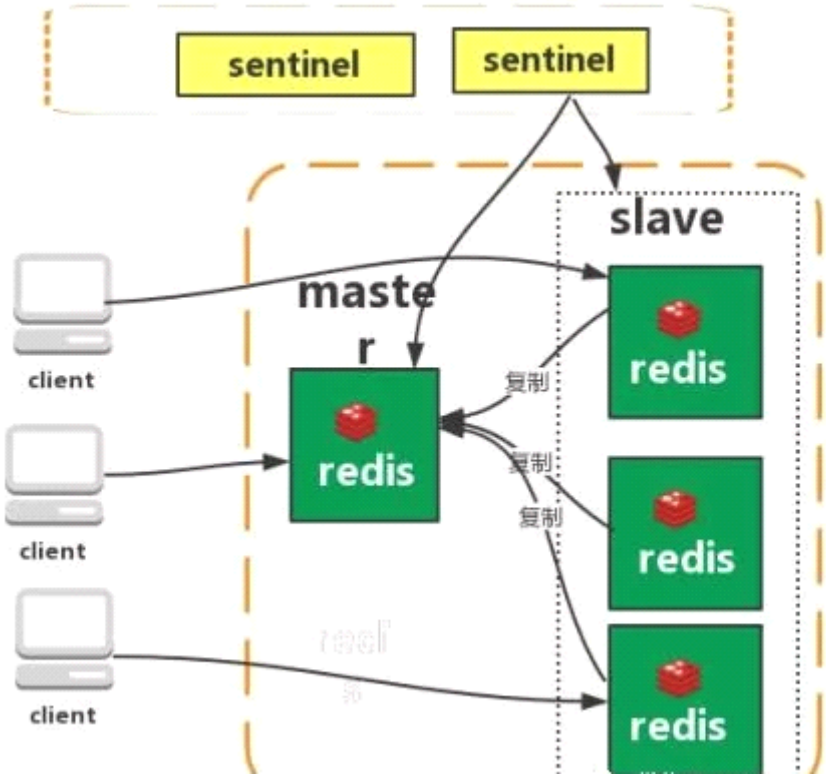
主从版本：



特点：
1） master/slave 角色
2） master/slave 数据相同
3） 降低 master 读压力在转交从库

问题：
1） 无法保证高可用
2） 没有解决 master 写的压力

哨兵模式：



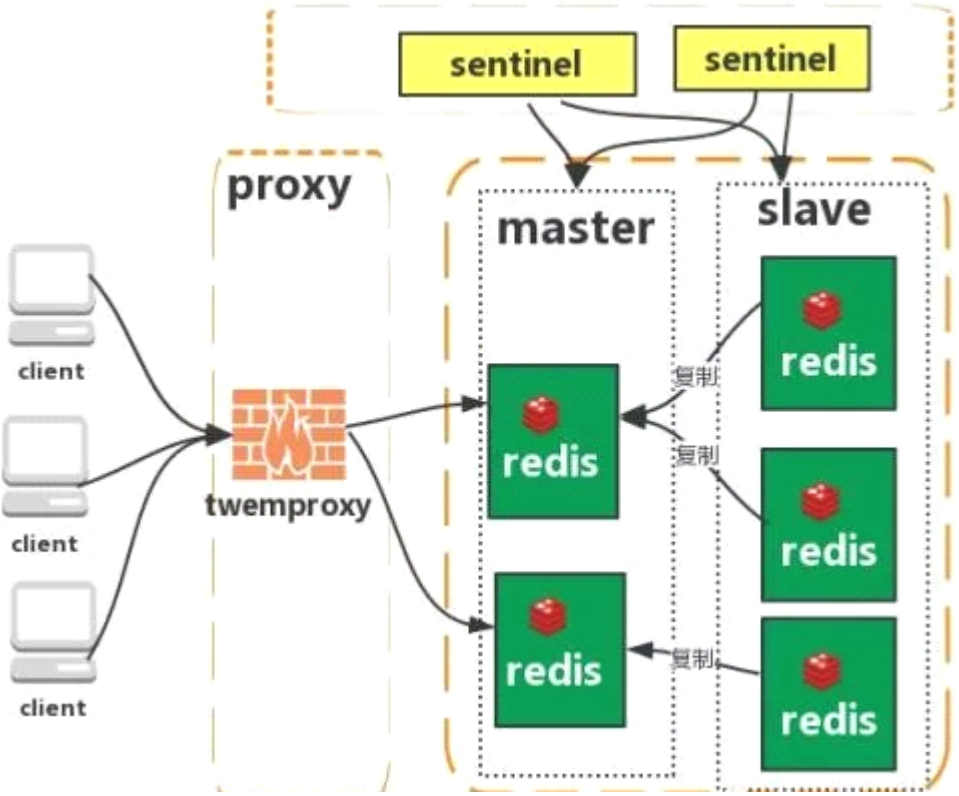
Redis sentinel 是一个分布式系统中监控 redis 主从服务器，并在主服务器下线时自动进行故障转移。其中三个特性：

监控（Monitoring）： Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。
提醒（Notification）： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。
自动故障迁移（Automatic failover）： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作。

特点：
1） 保证高可用
2） 监控各个节点
3） 自动故障迁移

缺点：主从模式，切换需要时间丢数据
没有解决 master 写的压力

集群（proxy 型）：



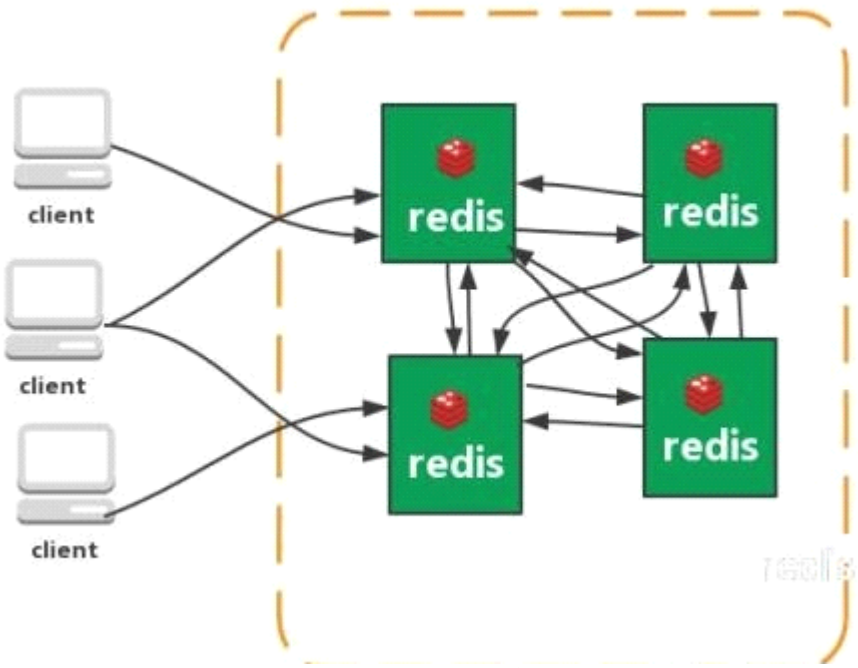
Twemproxy 是一个 Twitter 开源的一个 redis 和 memcache 快速/轻量级代理服务器；
Twemproxy 是一个快速的单线程代理程序，支持 Memcached ASCII 协议和 redis 协议。

特点：
1） 多种 hash 算法：MD5、CRC16、CRC32、CRC32a、hsieh、murmur、Jenkins
2） 支持失败节点自动删除
3） 后端 Sharding 分片逻辑对业务透明，业务方的读写方式和操作单个 Redis 一致

缺点：增加了新的 proxy，需要维护其高可用。

failover 逻辑需要自己实现，其本身不能支持故障的自动转移可扩展性差，进行扩缩容都需要手动干预

集群（直连型）：



从 redis 3.0 之后版本支持 redis-cluster 集群，Redis-Cluster 采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接。

特点：

1) 无中心架构（不存在哪个节点影响性能瓶颈），少了 proxy 层。
2) 数据按照 slot 存储分布在多个节点，节点间数据共享，可动态调整数据分布。
3) 可扩展性，可线性扩展到 1000 个节点，节点可动态添加或删除。
4) 高可用性，部分节点不可用时，集群仍可用。通过增加 Slave 做备份数据副本
5) 实现故障自动 failover，节点之间通过 gossip 协议交换状态信息，用投票机制完成 Slave 到 Master 的角色提升。

缺点：

1) 资源隔离性较差，容易出现相互影响的情况。
2) 数据通过异步复制,不保证数据的强一致性

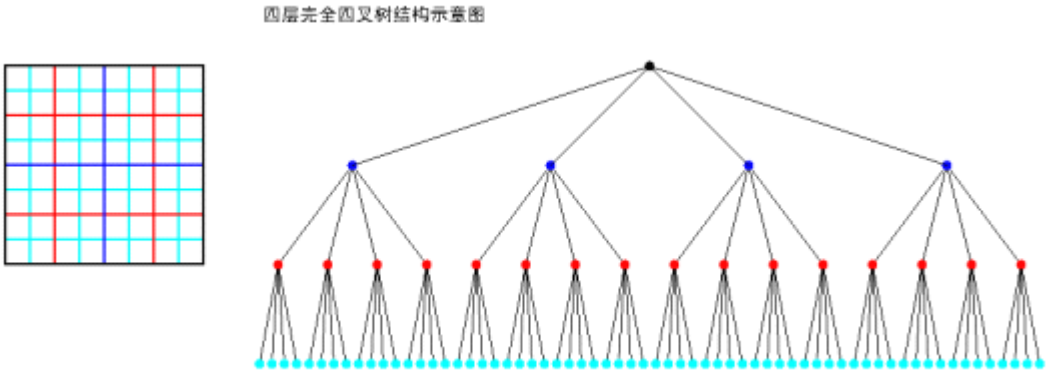
2.4.数据结构

二十一、 线性表
二十二、 二叉树
二十三、 红黑树
二十四、 平衡树
二十五、 Radix 树
二十六、 四叉树

四叉树或四元树也被称为 Q 树（Q-Tree）。**四叉树广泛应用于图像处理、空间数据索引、2D 中的快速碰撞检测、存储稀疏数据等，而八叉树（Octree）主要应用于 3D 图形处理。对游戏编程，这会很有用。**本文着重于对四叉树与八叉树的原理与结构的介绍，帮助您在脑海中建立四叉树与八叉树的基本思想。本文并不对这两种数据结构同时进行详解，而只对四叉树进行详解，因为八叉树的建立可由四叉树的建立推得。

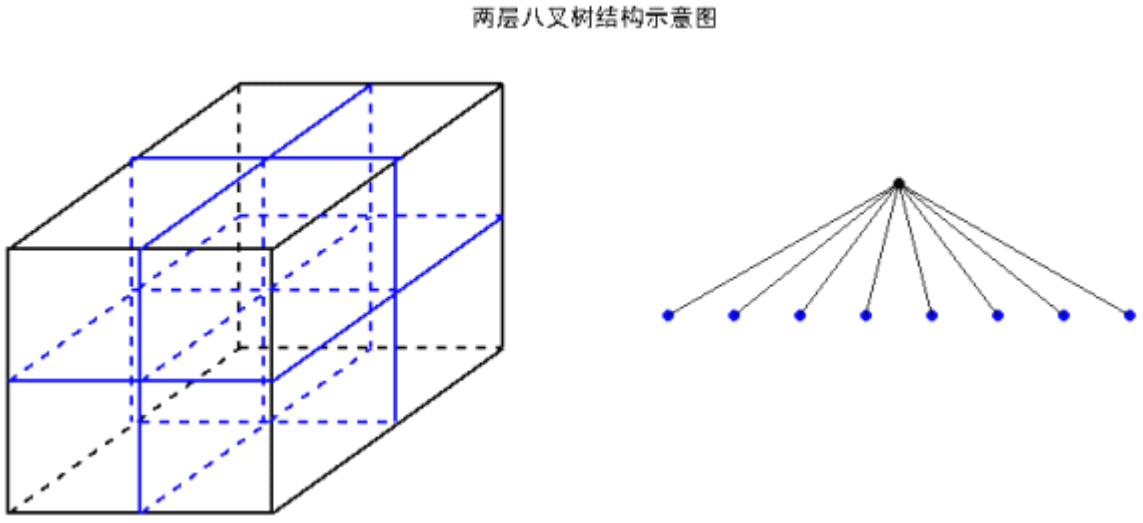
四叉树与八叉树的结构与原理

四叉树（Q-Tree）是一种树形数据结构。四叉树的定义是：它的每个节点下至多可以有四个子节点，通常把一部分二维空间细分为四个象限或区域并把该区域里的相关信息存入到四叉树节点中。这个区域可以是正方形、矩形或是任意形状。以下为四叉树的二维空间结构(左)和存储结构(右)示意图（注意节点颜色与网格边框颜色）：



四叉树的每一个节点代表一个矩形区域（如上图黑色的根节点代表最外围黑色边框的矩形区域），每一个矩形区域又可划分为四个小矩形区域，这四个小矩形区域作为四个子节点所代表的矩形区域。

较之四叉树，八叉树将场景从二维空间延伸到了三维空间。八叉树（Octree）的定义是：若不为空树的话，树中任一节点的子节点恰好只会有八个，或零个，也就是子节点不会有 0 与 8 以外的数目。那么，这要用来做什么？想象一个立方体，我们最少可以切成多少个相同等分的小立方体？答案就是 8 个。如下八叉树的结构示意图所示：



四叉树存储结构的 c 语言描述：

```
/* 一个矩形区域的象限划分：：

        UL(1)    |     UR(0)
      ----------|-----------
        LL(2)    |     LR(3)
以下对该象限类型的枚举
*/
typedef enum
{
    UR = 0,
    UL = 1,
    LL = 2,
    LR = 3
}QuadrantEnum;

/* 矩形结构 */
typedef struct quadrect_t
{
    double  left,
            top,
            right,
            bottom;
}quadrect_t;

/* 四叉树节点类型结构 */
typedef struct quadnode_t
{
    quadrect_t    rect;          //节点所代表的矩形区域
    list_t        *lst_object;   //节点数据，节点类型一般为链表，可存储多个对象
    struct  quadnode_t  *sub[4]; //指向节点的四个孩子
}quadnode_t;

/* 四叉树类型结构 */
typedef struct quadtree_t
{
    quadnode_t  *root;
    int         depth;           // 四叉树的深度
}quadtree_t;
```

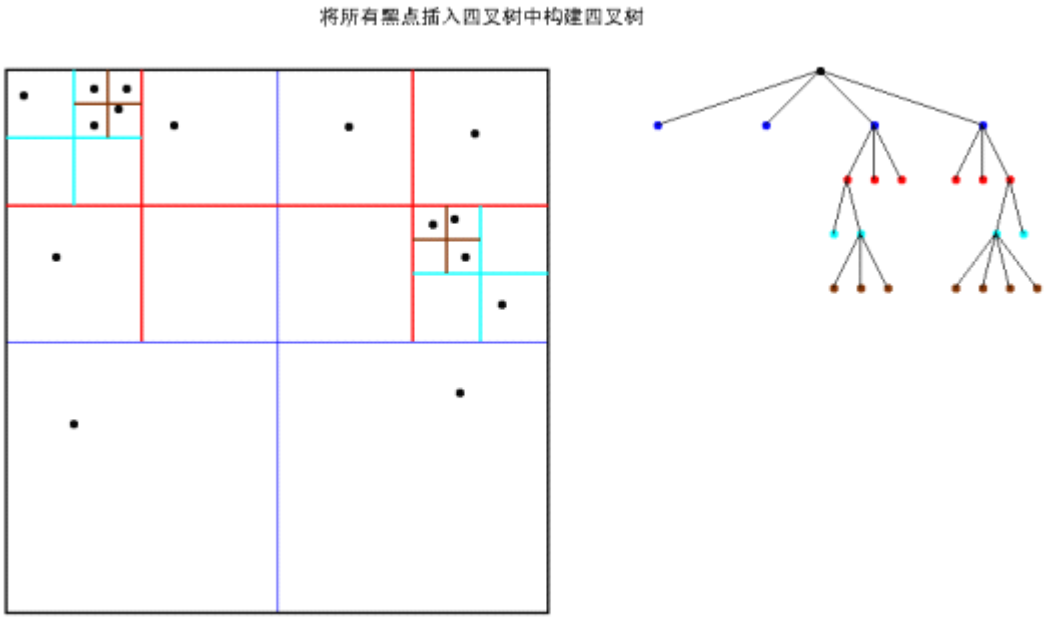
四叉树的建立

方法 1、利用四叉树分网格，如本文的第一张图<四层完全四叉树结构示意图>，根据左图的网格图形建立如右图所示的完全四叉树。

伪码：

```
Funtion QuadTreeBuild ( depth, rect )
{
    QuadTree->depth = depth;
    /*创建分支，root 树的根，depth 深度，rect 根节点代表的矩形区域*/
    QuadCreateBranch (  QuadTree->root, depth, rect );
}
Funtion QuadCreateBranch ( n, depth,rect )
{
    if ( depth!=0 )
   {
        n = new node;     //开辟新节点
        n ->rect = rect; //将该节点所代表的矩形区域存储到该节点中
        将 rect 划成四份 rect[UR], rect[UL], rect[LL], rect[LR];
        /*创建各孩子分支*/
        QuadCreateBranch ( n->sub[UR], depth-1, rect [UR] );
        QuadCreateBranch ( n->sub[UL], depth-1, rect [UL] );
        QuadCreateBranch ( n->sub[LL], depth-1, rect [LL] );
        QuadCreateBranch ( n->sub[LR], depth-1, rect [LR] );
    }
}
```

方法 2、假设在一个矩形区域里有 N 个对象，如下左图一个黑点代表一个对象，每个对象的坐标位置都是已知的，用四叉树的一个节点存储一个对象，构建成如下右图所示的四叉树。



方法也是采用递归的方法对该矩形进行划分分区块，分完后再往里分，直到每一个子矩形区域里只包含一个对象为止。

伪码：

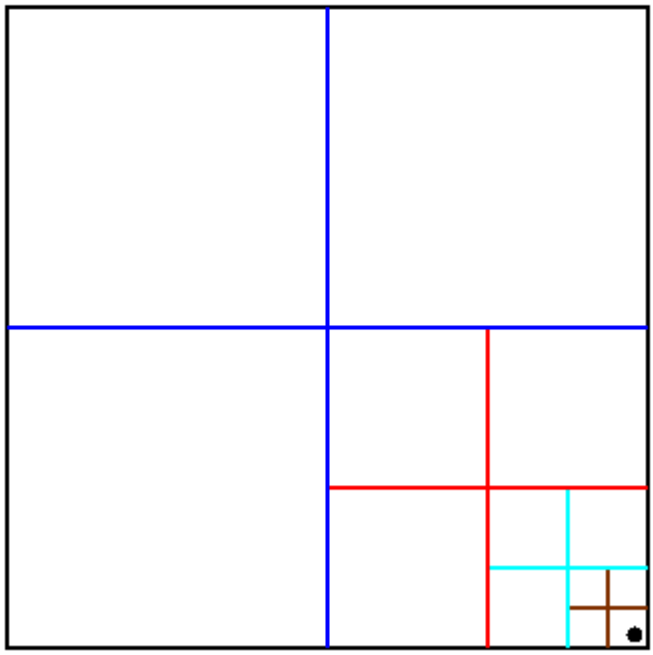
```
Funtion QuadtreeBuild()
{
     Quadtree = {empty};
     For (i = 1;i<n;i++) //遍历所有对象
     {
        QuadInsert(i, root); //将 i 对象插入四叉树
     }
    剔除多余的节点; //执行完上面循环后，四叉树中可能有数据为空的叶子节点需要剔除
}
Funtion QuadInsert(i,n)//该函数插入后四叉树中的每个节点所存储的对象数量不是 1 就是 0
{
     if（节点 n 有孩子）
     {
        通过划分区域判断 i 应该放置于 n 节点的哪一个孩子节点 c;
        QuadInsert(i,c);
     }
     else if（节点 n 存储了一个对象）
     {
        为 n 节点创建四个孩子;
        将 n 节点中的对象移到它应该放置的孩子节点中;
        通过划分区域判断 i 应该放置于 n 节点的哪一个孩子节点 c;
        QuadInsert(i,c);
     }
     else if（n 节点数据为空）
     {
        将 i 存储到节点 n 中;
     }
}
```

用四叉树查找某一对象

1、采用盲目搜索，与二叉树的递归遍历类似，可采用后序遍历或前序遍历或中序遍历对其进行搜索某一对象，时间复杂度为 0（n）。

2、根据对象在区域里的位置来搜索，采用分而治之思想，时间复杂度只与四叉树的深度有关。比起盲目搜索，这种搜索在区域里的对象越多时效果越明显

从最外层矩形区域搜索位于最右下角的黑点

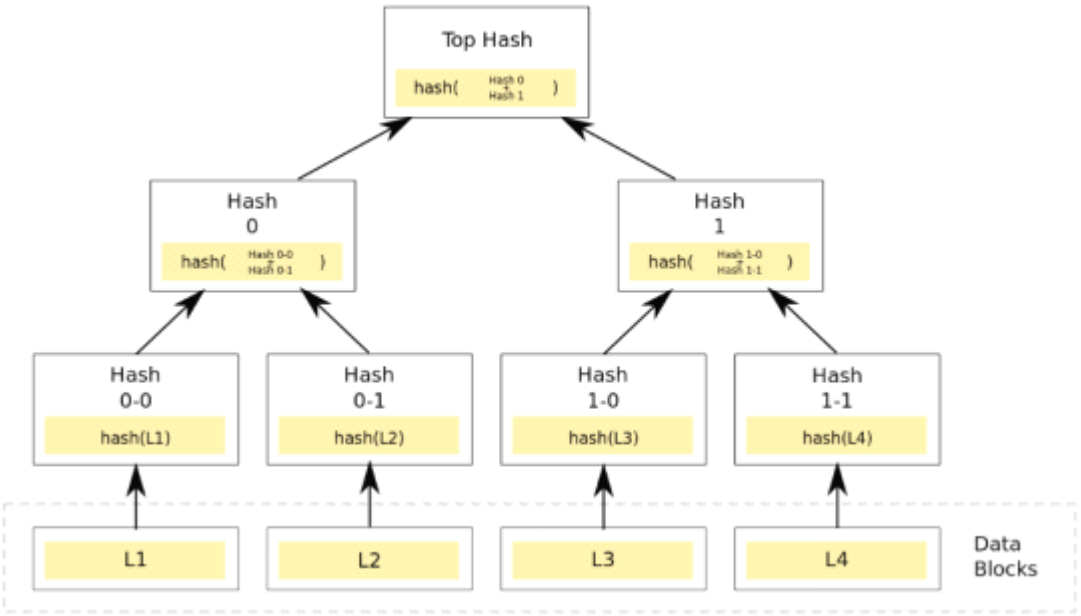


伪码：

```
Funtion find ( n, pos, )
{
      If (n 节点所存的对象位置为 pos 所指的位置 )
          Return n;
      If ( pos 位于第一象限 )
          temp = find ( n->sub[UR], pos );
      else if ( pos 位于第二象限)
          temp = find ( n->sub[UL], pos );
      else if ( pos 位于第三象限 )
          temp = find ( n->sub[LL], pos );
      else  //pos 位于第四象限
          temp = find ( n->sub[LR], pos );
      return temp;
}
```

二十七、　八叉树

二十八、　梅克尔树

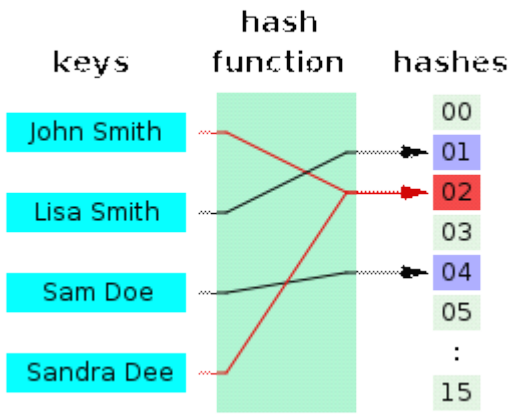


Merkle Tree

通常也被称作 Hash Tree，顾名思义，就是存储 hash 值的一棵树。Merkle 树的叶子是数据块(例如，文件或者文件的集合)的 hash 值。非叶节点是其对应子节点串联字符串的 hash。

Hash

是一个把任意长度的数据映射成固定长度数据的函数。例如，对于数据完整性校验，最简单的方法是对整个数据做 Hash 运算得到固定长度的 Hash 值，然后把得到的 Hash 值公布在网上，这样用户下载到数据之后，对数据再次进行 Hash 运算，比较运算结果和网上公布的 Hash 值进行比较，如果两个 Hash 值相等，说明下载的数据没有损坏。可以这样做是因为输入数据的稍微改变就会引起 Hash 运算结果的面目全非，而且根据 Hash 值反推原始输入数据的特征是困难的。



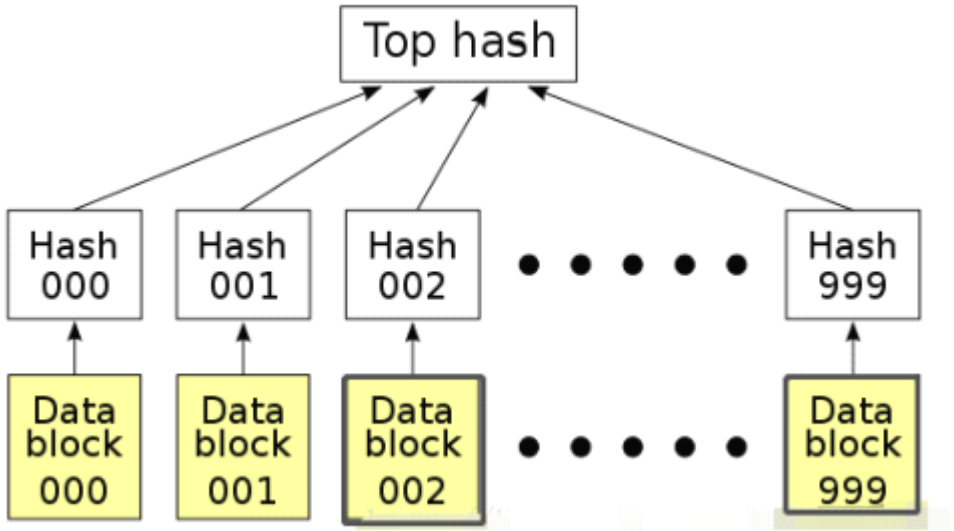
如果从一个稳定的服务器进行下载，采用单一 Hash 是可取的。但如果数据源不稳定，一旦数据损坏，就需要重新下载，这种下载的效率是很低的。

Hash List

在点对点网络中作数据传输的时候，会同时从多个机器上下载数据，而且很多机器可以认为是不稳定或者不可信的。为了校验数据的完整性，更好的办法是把大的文件分割成小的

数据块（例如，把分割成 2K 为单位的数据块）。这样的好处是，如果小块数据在传输过程中损坏了，那么只要重新下载这一快数据就行了，不用重新下载整个文件。

怎么确定小的数据块没有损坏哪？只需要为每个数据块做 Hash。BT 下载的时候，在下载到真正数据之前，我们会先下载一个 Hash 列表。那么问题又来了，怎么确定这个 Hash 列表本事是正确的哪？答案是把每个小块数据的 Hash 值拼到一起，然后对这个长字符串在作一次 Hash 运算，这样就得到 Hash 列表的根 Hash(Top Hash or Root Hash)。下载数据的时候，首先从可信的数据源得到正确的根 Hash，就可以用它来校验 Hash 列表了，然后通过校验后的 Hash 列表校验数据块。



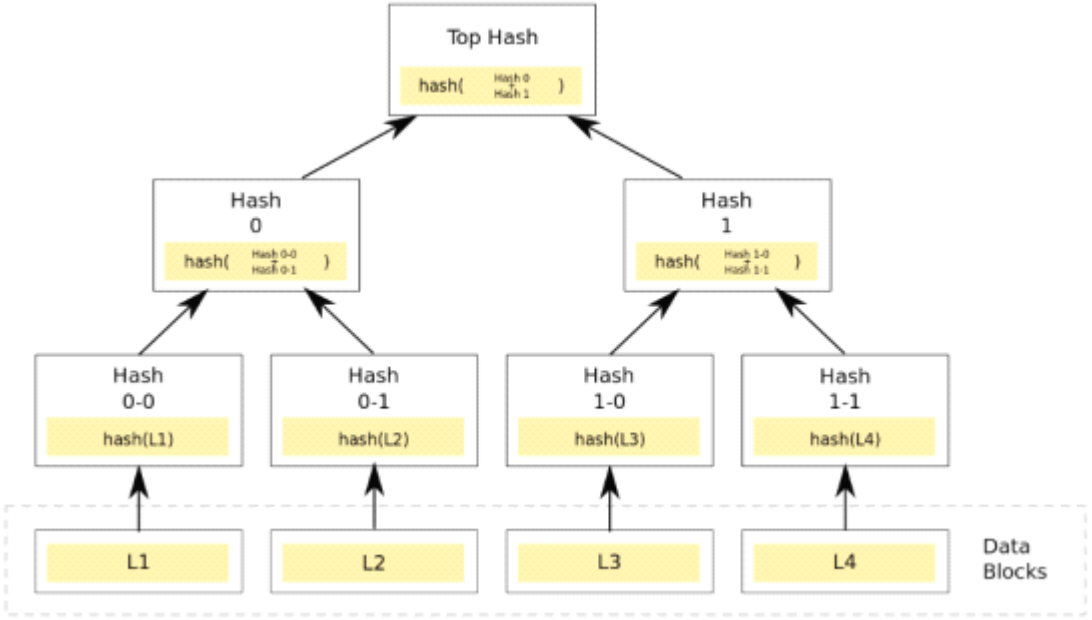
Merkle Tree

Merkle Tree 可以看做 Hash List 的泛化（Hash List 可以看作一种特殊的 Merkle Tree，即树高为 2 的多叉 Merkle Tree）。

在最底层，和哈希列表一样，我们把数据分成小的数据块，有相应地哈希和它对应。但是往上走，并不是直接去运算根哈希，而是把相邻的两个哈希合并成一个字符串，然后运算这个字符串的哈希，这样每两个哈希就结婚生子，得到了一个”子哈希“。如果最底层的哈希总数是单数，那到最后必然出现一个单身哈希，这种情况就直接对它进行哈希运算，所以也能得到它的子哈希。于是往上推，依然是一样的方式，可以得到数目更少的新一级哈希，最终必然形成一棵倒挂的树，到了树根的这个位置，这一代就剩下一个根哈希了，我们把它叫做 Merkle Root。

在 p2p 网络下载网络之前，先从可信的源获得文件的 Merkle Tree 树根。一旦获得了树根，就可以从其他从不可信的源获取 Merkle tree。通过可信的树根来检查接受到的 Merkle Tree。如果 Merkle Tree 是损坏的或者虚假的，就从其他源获得另一个 Merkle Tree，直到获得一个与可信树根匹配的 Merkle Tree。

Merkle Tree 和 Hash List 的主要区别是，可以直接下载并立即验证 Merkle Tree 的一个分支。因为可以将文件切分成小的数据块，这样如果有一块数据损坏，仅仅重新下载这个数据块就行了。如果文件非常大，那么 Merkle tree 和 Hash list 都很到，但是 Merkle tree 可以一次下载一个分支，然后立即验证这个分支，如果分支验证通过，就可以下载数据了。而 Hash list 只有下载整个 hash list 才能验证。



Merkle Tree 的特点

1. MT 是一种树，大多数是二叉树，也可以多叉树，无论是几叉树，它都具有树结构的所有特点；
2. Merkle Tree 的叶子节点的 value 是数据集合的单元数据或者单元数据 HASH。
3. 非叶子节点的 value 是根据它下面所有的叶子节点值，然后按照 Hash 算法计算而得出的。

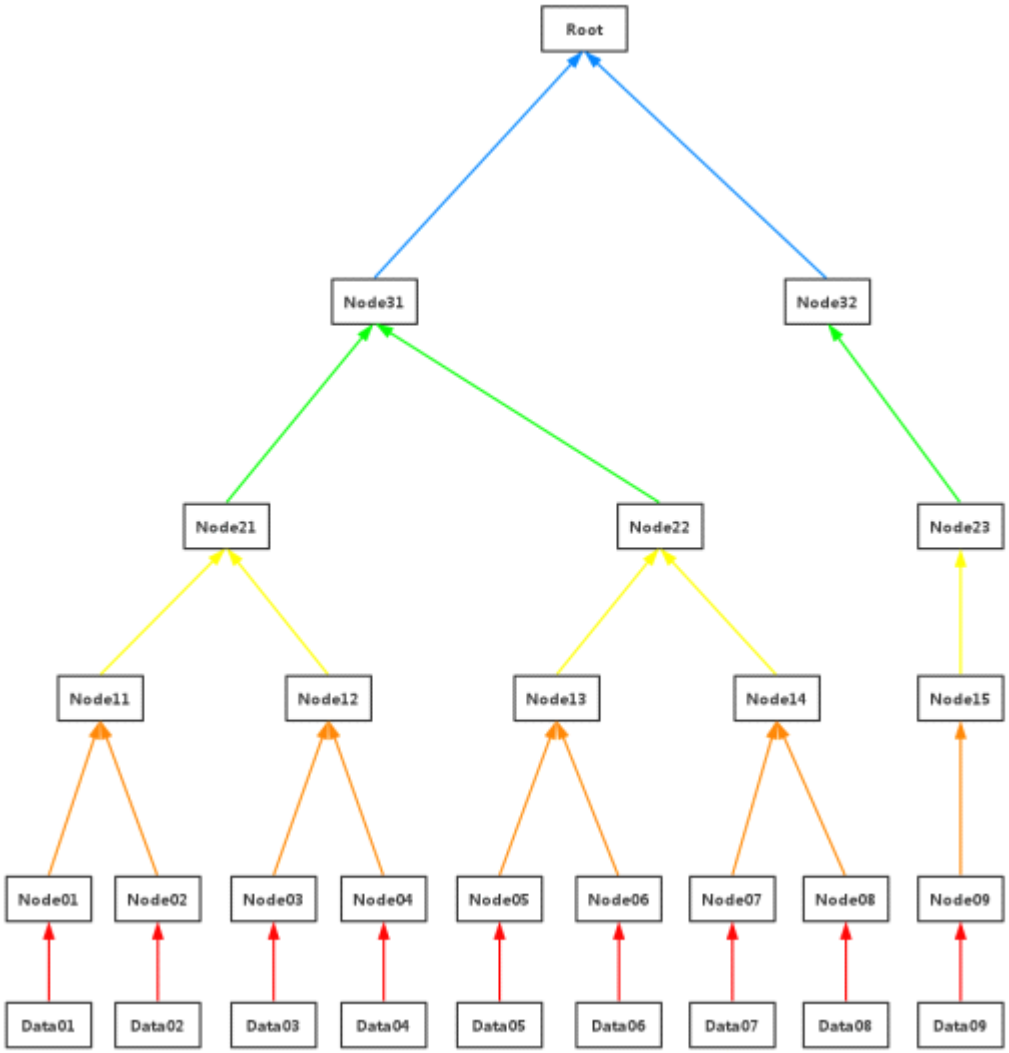
通常，加密的 hash 方法像 SHA-2 和 MD5 用来做 hash。但如果仅仅防止数据不是蓄意的损坏或篡改，可以改用一些安全性低但效率高的校验和**算法**，如 CRC。
Second Preimage Attack: Merkle tree 的树根并不表示树的深度，这可能会导致 second-preimage attack，即攻击者创建一个具有相同 Merkle 树根的虚假文档。一个简单的解决方法在 Certificate Transparency 中定义：当计算叶节点的 hash 时，在 hash 数据前加 0x00。当计算内部节点是，在前面加 0x01。另外一些实现限制 hash tree 的根，通过在 hash 值前面加深度前缀。因此，前缀每一步会减少，只有当到达叶子时前缀依然为正，提取的 hash 链才被定义为有效。

Merkle Tree 的操作

创建 Merckle Tree

加入最底层有 9 个数据块。

step1：（红色线）对数据块做 hash 运算，Node0i = hash(Data0i)， i=1,2,…,9
step2：（橙色线）相邻两个 hash 块串联，然后做 hash 运算，Node1((i+1)/2) = hash(Node0i+Node0(i+1))， i=1,3,5,7;对于 i=9， Node1((i+1)/2) = hash(Node0i)
step3：（黄色线）重复 step2
step4：（绿色线）重复 step2
step5：（蓝色线）重复 step2，生成 Merkle Tree Root



易得，创建 Merkle Tree 是 O(n)复杂度(这里指 O(n)次 hash 运算)，n 是数据块的大小。得到 Merkle Tree 的树高是 log(n)+1。

2.5.算法

二十九、　　排序算法

选择/冒泡/快速/堆排等

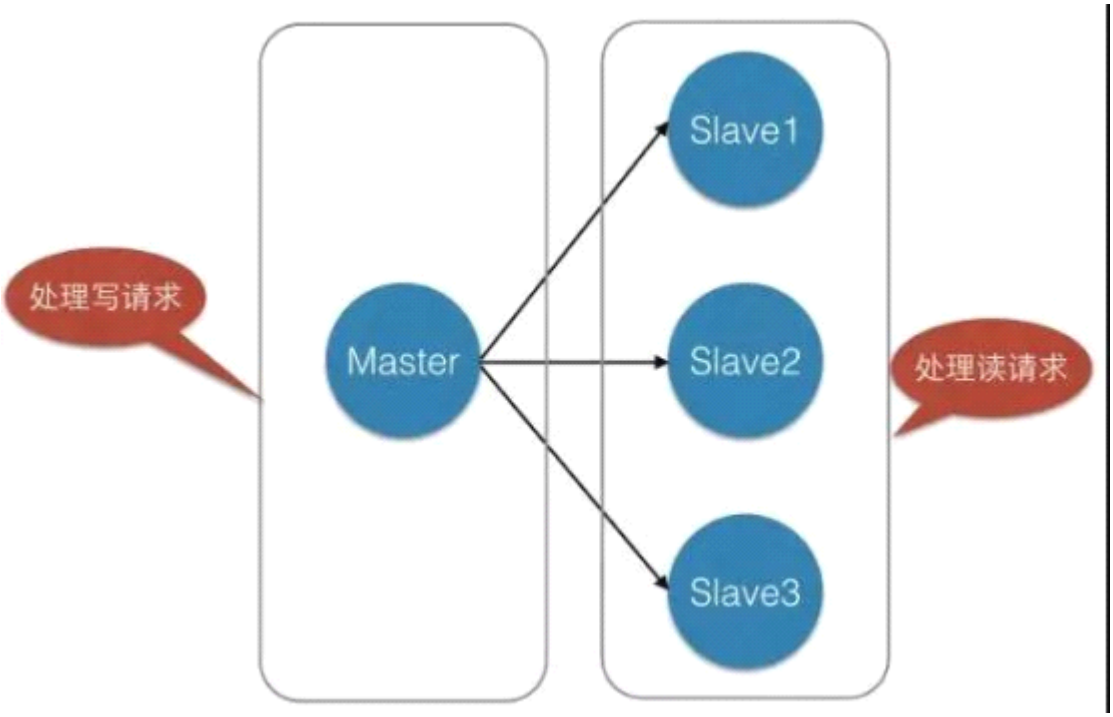
三十、　　一致性哈希算法

算法背景：
一致性哈希算法在 1997 年由麻省理工学院的 Karger 等人在解决分布式 Cache 中提出的，设计目标是为了解决因特网中的热点(Hot spot)问题，初衷和 CARP 十分类似。一致性哈希修正了 CARP 使用的简单哈希算法带来的问题，使得 DHT 可以在 P2P 环境中真正得到应用。

评判算法好坏标准：

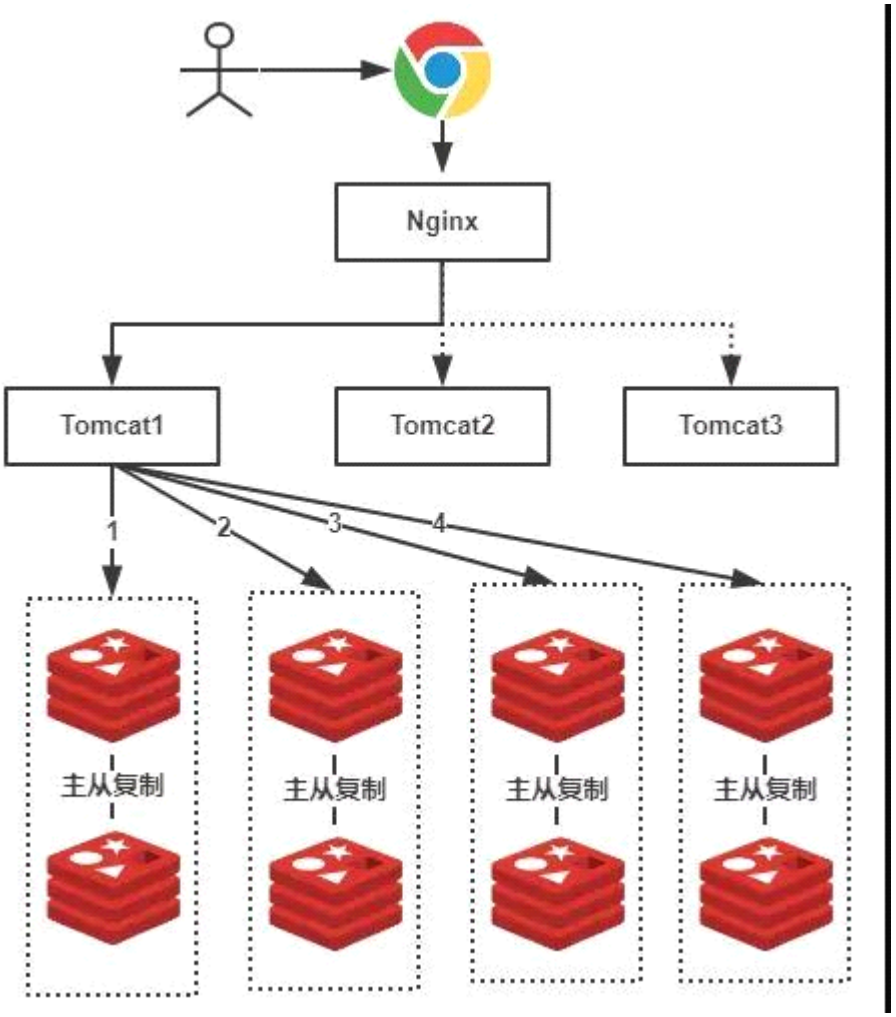
一致性 Hash 算法提出了在动态变化的 Cache 环境中，判定哈希算法好坏的四个定义：
1、平衡性（Balance）：平衡性是指哈希的结果能够尽可能分布在所有的缓冲(Cache)中去，这样可以使得所有的缓冲空间得到利用。很多哈希算法都能够满足这一条件。
2、单调性（Monotonicity）：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应该能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去，而不会映射到旧的缓冲集合中的其他缓冲区。
3、分散性（Spread）：在分布式环境中，终端有可能看不到所有的缓冲，而只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上去，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应该能够尽量避免不一致的情况发生，也就是尽量降低分散性。
4、负载（Load）：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射到不同的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

应用场景：
现在一致性 hash 算法在分布式系统中也得到了广泛应用，分布式系统中涉及到集群部署，包括缓存 Redis 集群，数据库集群，我们在使用 Redis 的时候，为了保证 Redis 的高可用，提高 Redis 的读写性能，最简单的方式我们会做主从复制，组成 Master-Master 或者 Master-Slave 的形式，或者搭建 Redis 集群，进行数据的读写分离，类似于数据库的主从复制和读写分离。如下所示



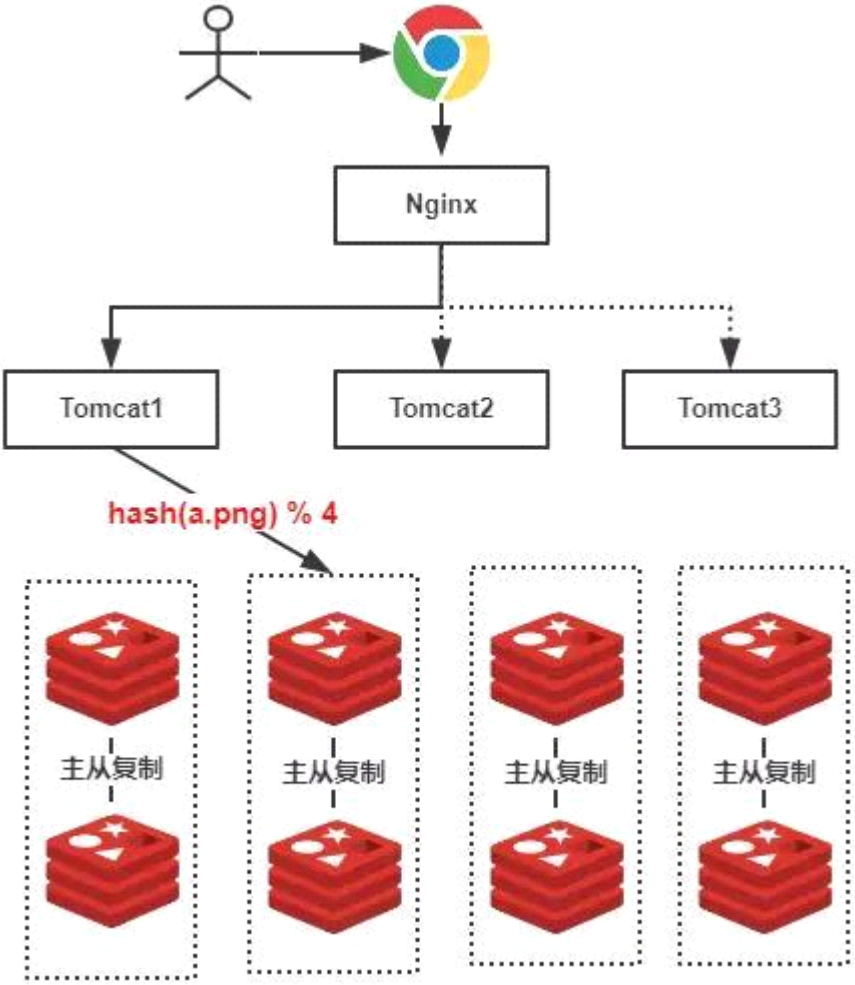
同样数据库中也是，当单表数据大于 500W 的时候需要对其进行分库分表，当数据量很大的时候（标准可能不一样，要看 Redis 服务器容量）我们同样可以对 Redis 进行类似的操作，就是分库分表。
假设，我们有一个社交网站，需要使用 Redis 存储图片资源，存储的格式为键值对，key 值为图片名称，value 为该图片所在文件服务器的路径，我们需要根据文件名查找该文件所在文件服务器上的路径，数据量大概有 2000W 左右，按照我们约定的规则进行分库，规

则就是随机分配，我们可以部署 8 台缓存服务器，每台服务器大概含有 500W 条数据，并且进行主从复制，示意图如下



由于规则是随机的，所有我们的一条数据都有可能存储在任何一组 Redis 中，例如上图我们用户查找一张名称为”a.png”的图片，由于规则是随机的，我们不确定具体是在哪一个 Redis 服务器上的，因此我们需要进行 1、2、3、4，4 次查询才能够查询到（也就是遍历了所有的 Redis 服务器），这显然不是我们想要的结果，有了解过的小伙伴可能会想到，随机的规则不行，可以使用类似于数据库中的分库分表规则：按照 Hash 值、取模、按照类别、按照某一个字段值等等常见的规则就可以出来了！好，按照我们的主题，我们就使用 Hash 的方式。

为 Redis 集群使用 Hash：
可想而知，如果我们使用 Hash 的方式 hash(图片名称) % N ，每一张图片在进行分库的时候都可以定位到特定的服务器，示意图如下：



因为图片的名称是不重复的，所以，当我们对同一个图片名称做相同的哈希计算时，得出的结果应该是不变的，如果我们有 4 台服务器，使用哈希后的结果对 4 求余，那么余数一定是 0、1、2 或 3，没错，正好与我们之前的服务器编号相同。

如果求余的结果为 0， 我们就把当前图片名称对应的图片缓存在 0 号服务器上；如果余数为 1，就把当前图片名对应的图片缓存在 1 号服务器上；如果余数为 2，同理。那么，当我们访问任意一个图片的时候，只要再次对图片名称进行上述运算，即可得出对应的图片应该存放在哪一台缓存服务器上，我们只要在这一台服务器上查找图片即可，如果图片在对应的服务器上不存在，则证明对应的图片没有被缓存，也不用再去遍历其他缓存服务器了，通过这样的方法，即可将 3 万张图片随机的分布到 3 台缓存服务器上了，而且下次访问某张图片时，直接能够判断出该图片应该存在于哪台缓存服务器上，这样就能满足我们的需求了，我们暂时称上述算法为 HASH 算法或者取模算法。

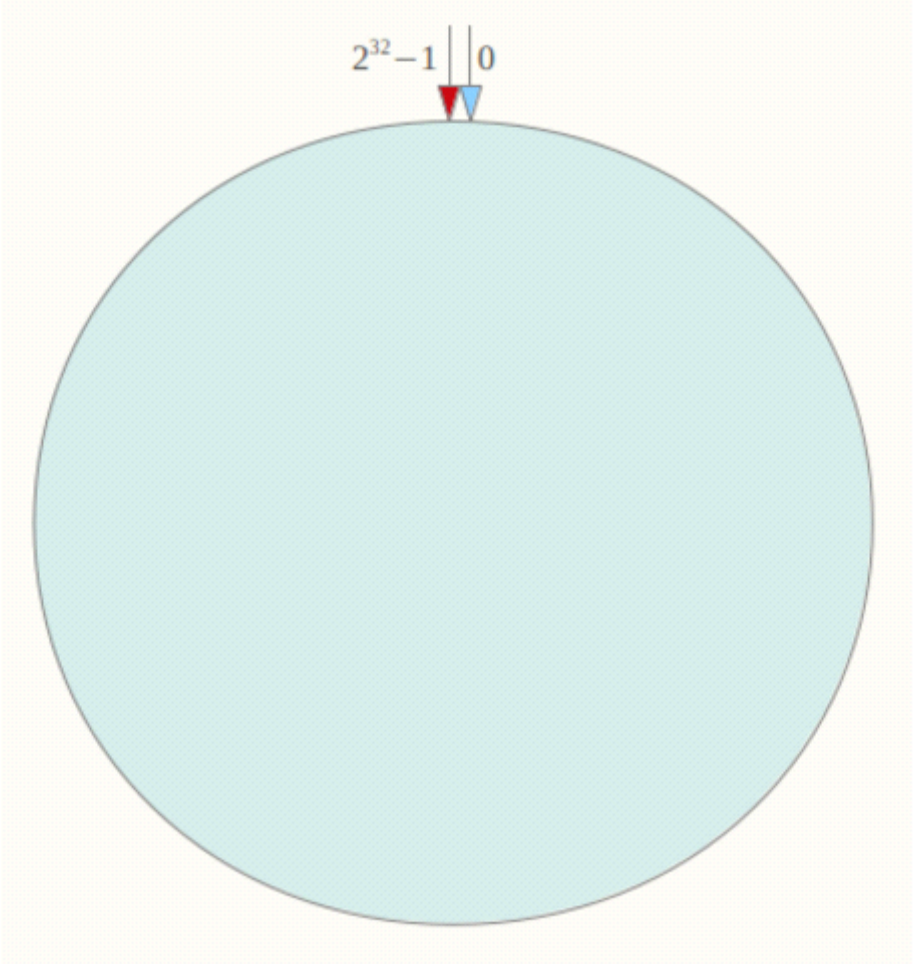
上图中，假设我们查找的是”a.png”，由于有 4 台服务器（排除从库），因此公式为 hash(a.png) % 4 = 2
，可知定位到了第 2 号服务器，这样的话就不会遍历所有的服务器，大大提升了性能！

使用 Hash 的问题：
上述的方式虽然提升了性能，我们不再需要对整个 Redis 服务器进行遍历！但是，使用上述 HASH 算法进行缓存时，会出现一些缺陷，主要体现在服务器数量变动的时候，所有缓存的位置都要发生改变！

试想一下，如果 3 台缓存服务器已经不能满足我们的缓存需求，那么我们应该怎么做呢？没错，很简单，多增加两台缓存服务器不就行了，假设，我们增加了一台缓存服务器，那么缓存服务器的数量就由 4 台变成了 5 台，此时，如果仍然使用上述方法对同一张图片进行缓存，那么这张图片所在的服务器编号必定与原来 4 台服务器时所在的服务器编号不同，因为除数由 4 变为了 5，被除数不变的情况下，余数肯定不同，这种情况带来的结果就是当服务器数量变动时，所有缓存的位置都要发生改变，换句话说，当服务器数量发生改变时，所有缓存在一定时间内是失效的，当应用无法从缓存中获取数据时，则会向后端服务器请求数据，同理，假设 4 台缓存中突然有一台缓存服务器出现了故障，无法进行缓存，那么我们则需要将故障机器移除，但是如果移除了一台缓存服务器，那么缓存服务器数量从 4 台变为 3 台，如果想要访问一张图片，这张图片的缓存位置必定会发生改变，以前缓存的图片也会失去缓存的作用与意义，由于大量缓存在同一时间失效，造成了缓存的雪崩，此时前端缓存已经无法起到承担部分压力的作用，后端服务器将会承受巨大的压力，整个系统很有可能被压垮，所以，我们应该想办法不让这种情况发生，但是由于上述 HASH 算法本身的缘故，使用取模法进行缓存时，这种情况是无法避免的。

我们来回顾一下使用上述算法会出现的问题。
问题 1：当缓存服务器数量发生变化时，会引起缓存的雪崩，可能会引起整体系统压力过大而崩溃（大量缓存同一时间失效）。
问题 2：当缓存服务器数量发生变化时，几乎所有缓存的位置都会发生改变，怎样才能尽量减少受影响的缓存呢？
其实，上面两个问题是一个问题，那么，一致性哈希算法能够解决上述问题吗？解决这些问题，一致性哈希算法诞生了。

一致性哈希的基本概念：
一致性 Hash 算法也是使用取模的方法，只是，刚才描述的取模法是对服务器的数量进行取模，而一致性 Hash 算法是对 2^32 取模，什么意思呢？简单来说，一致性 Hash 算法将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数 H 的值空间为 0-2^32-1（即哈希值是一个 32 位无符号整形），整个哈希环如下：



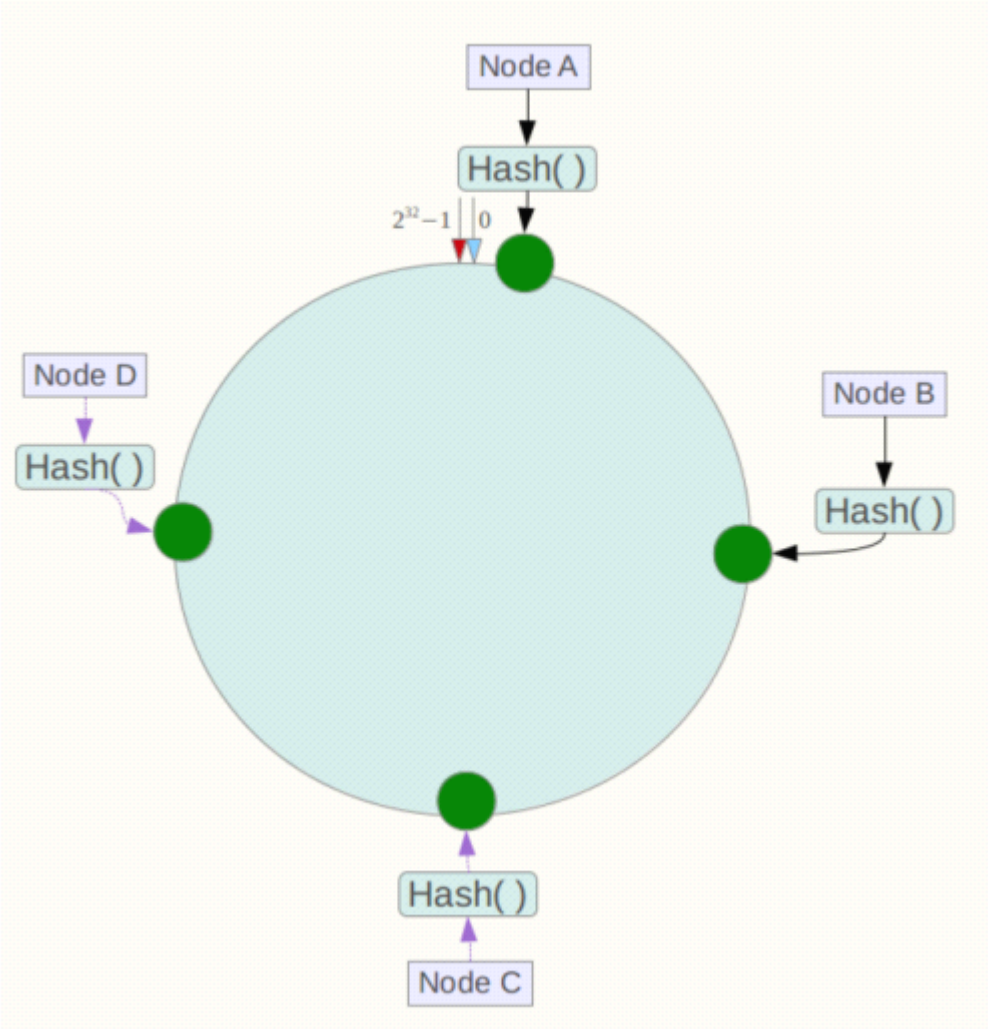
整个空间按顺时针方向组织，圆环的正上方的点代表 0，0 点右侧的第一个点代表 1，以此类推，2、3、4、5、6……直到 2^32-1，也就是说 0 点左侧的第一个点代表 2^32-1， 0 和 2^32-1 在零点中方向重合，我们把这个由 2^32 个点组成的圆环称为 Hash 环。

那么，一致性哈希算法与上图中的圆环有什么关系呢？我们继续聊，仍然以之前描述的场景为例，假设我们有 4 台缓存服务器，服务器 A、服务器 B、服务器 C，服务器 D，那么，在生产环境中，这 4 台服务器肯定有自己的 IP 地址或主机名，我们使用它们各自的 IP 地址或主机名作为关键字进行哈希计算，使用哈希后的结果对 2^32 取模，可以使用如下公式示意：

hash（服务器 A 的 IP 地址） % 2^32

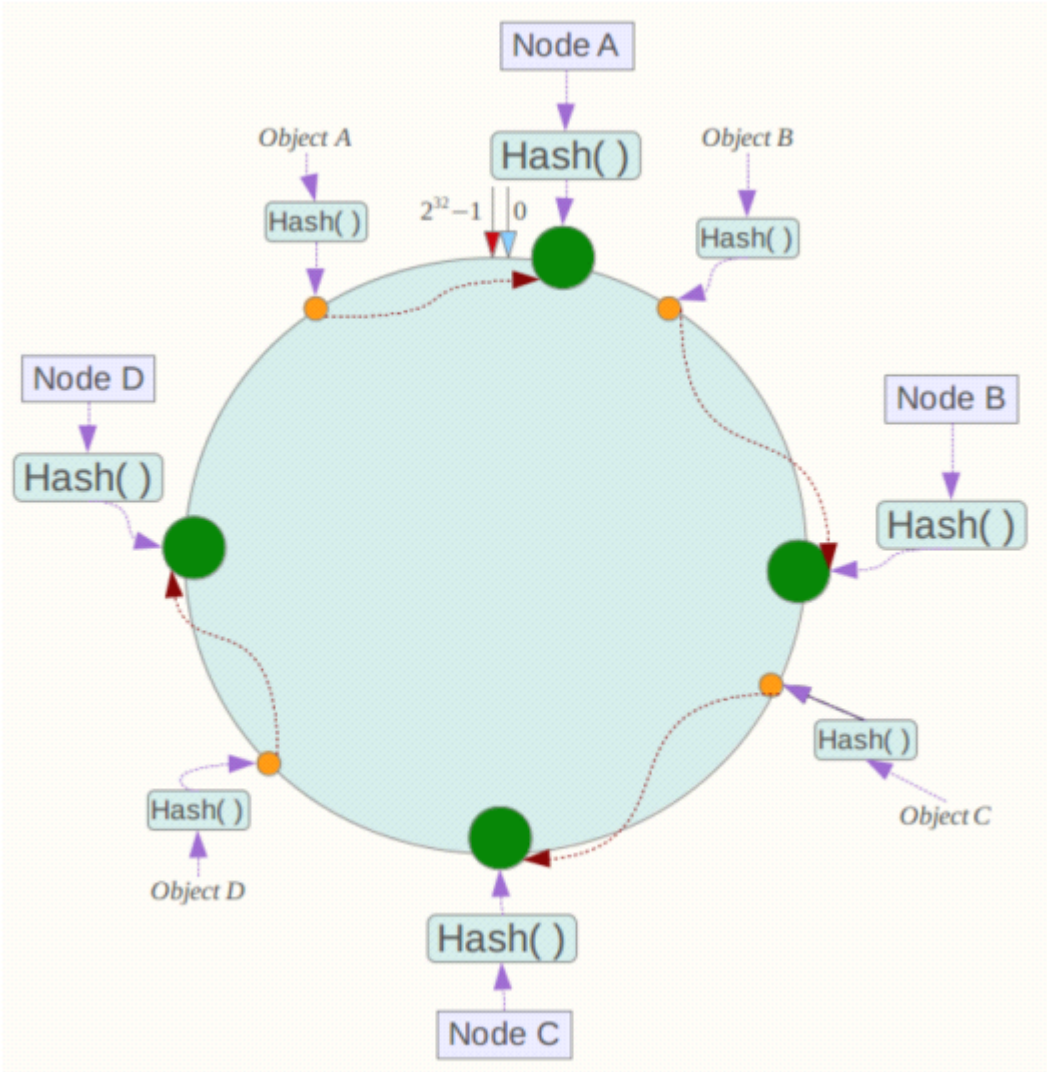
通过上述公式算出的结果一定是一个 0 到 2^32-1 之间的一个整数，我们就用算出的这个整数，代表服务器 A，既然这个整数肯定处于 0 到 2^32-1 之间，那么，上图中的 hash 环上必定有一个点与这个整数对应，而我们刚才已经说明，使用这个整数代表服务器 A，那么，服务器 A 就可以映射到这个环上。

以此类推，下一步将各个服务器使用类似的 Hash 算式进行一个哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中四台服务器使用 IP 地址哈希后在环空间的位置如下：



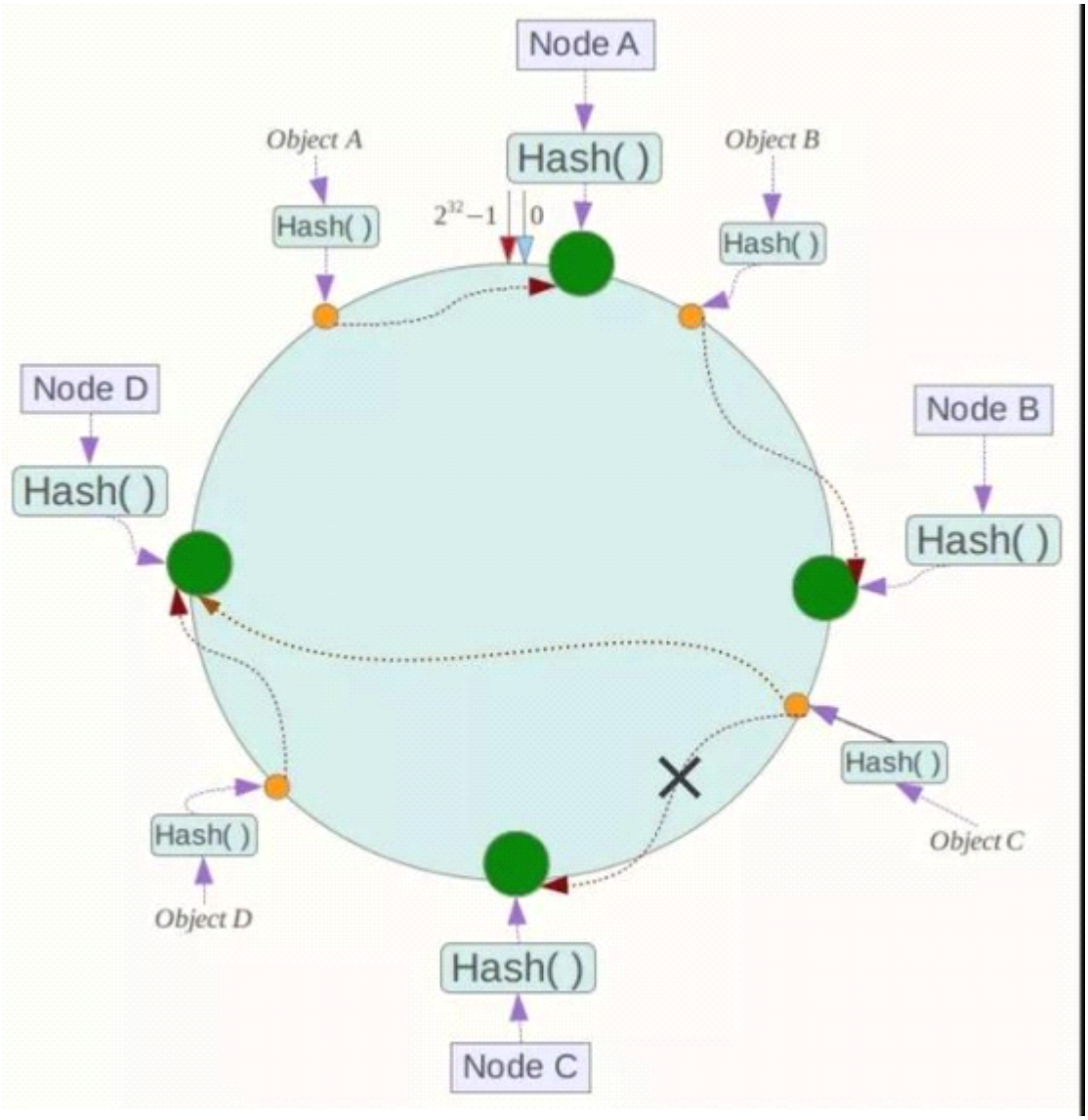
接下来使用如下算法定位数据访问到相应服务器：　将数据 key 使用相同的函数 Hash 计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器！

例如我们有 Object A、Object B、Object C、Object D 四个数据对象，经过哈希计算后，在环空间上的位置如下：

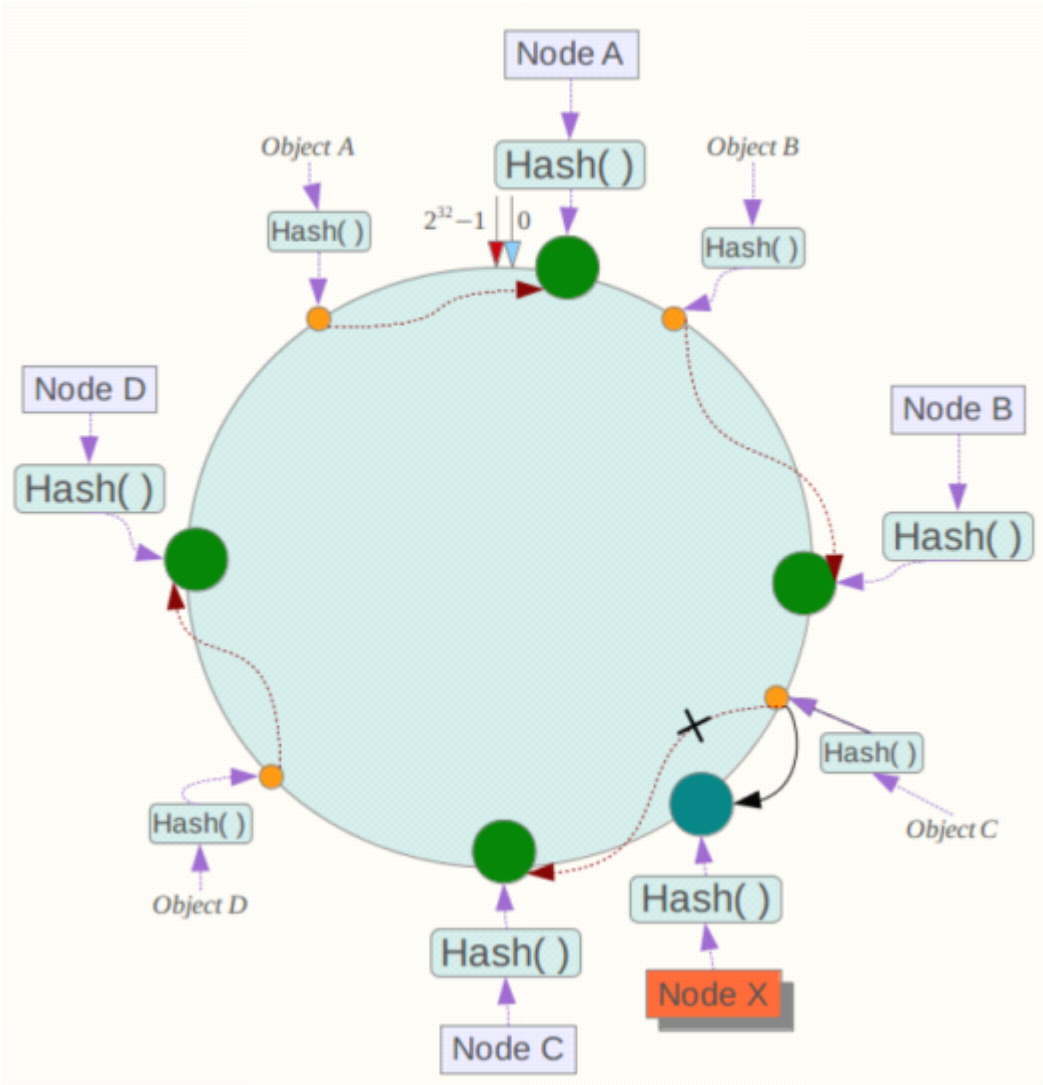


根据一致性 Hash 算法，数据 A 会被定为到 Node A 上，B 被定为到 Node B 上，C 被定为到 Node C 上，D 被定为到 Node D 上。
说到这里可能会有疑问，为什么 hash 一致性的数据空间范围是 2^32 次方？
因为，c/c++及 java 中 int 的最大值是 2^31-1 最小值是-2^31,2^32 刚好是无符号整形的最大值；
进一步追尾基础，为什么 java 中 int 的最大值是 2^31-1 最小值是-2^31？
因为，int 的最大值最小值范围设定是因为一个 int 占 4 个字节，一个字节占 8 位，二进制中刚好是 32 位。（基础忘记的需要恶补一下了）

一致性 Hash 算法的容错性和可扩展性：
现假设 Node C 不幸宕机，可以看到此时对象 A、B、D 不会受到影响，只有 C 对象被重定位到 Node D。一般的，在一致性 Hash 算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响，如下所示：

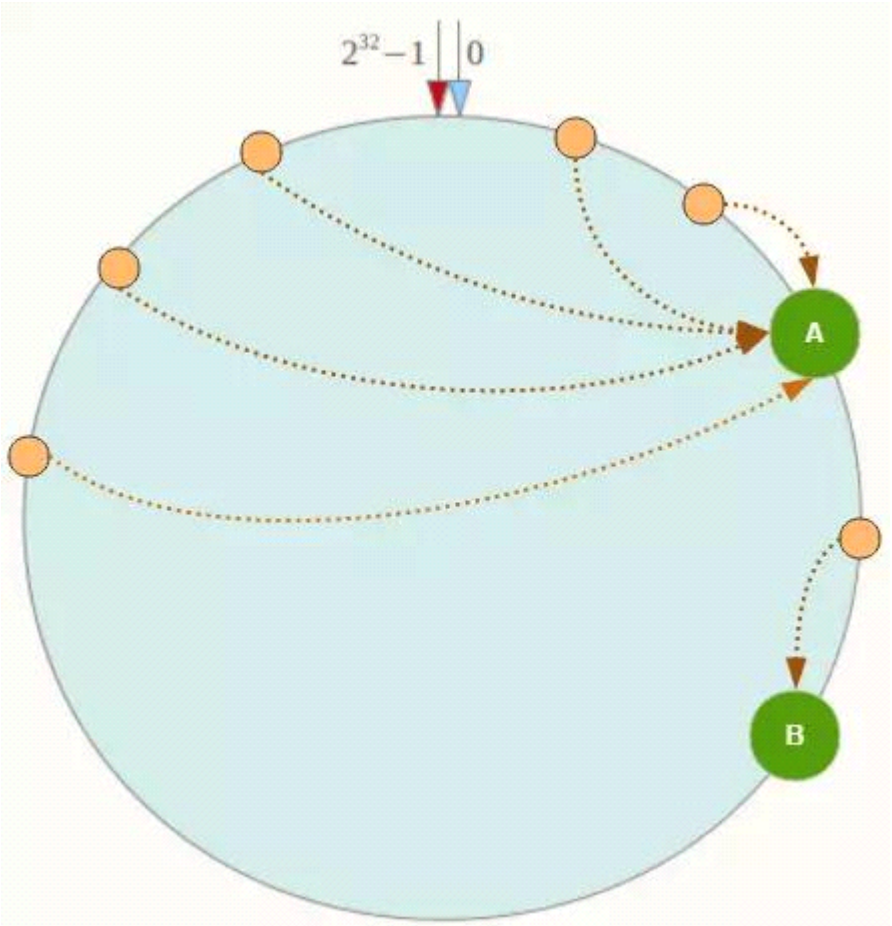


下面考虑另外一种情况，如果在系统中增加一台服务器 Node X，如下图所示：



此时对象 Object A、B、D 不受影响，只有对象 C 需要重定位到新的 Node X ！一般的，在一致性 Hash 算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。
综上所述，一致性 Hash 算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

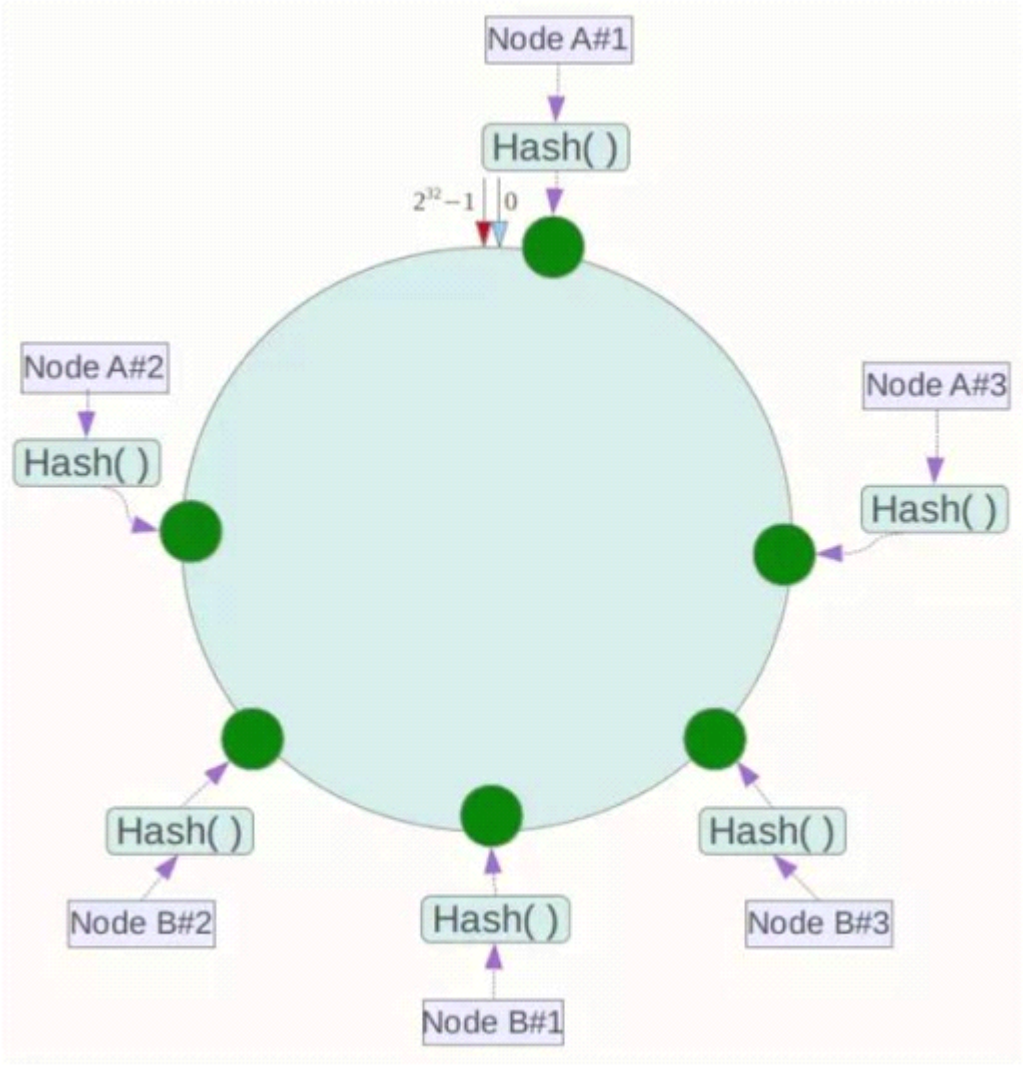
Hash 环的数据倾斜问题：
一致性 Hash 算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜（被缓存的对象大部分集中缓存在某一台服务器上）问题，例如系统中只有两台服务器，其环分布如下：



此时必然造成大量数据集中到Node A上，而只有极少量会定位到Node B上，从而出现hash环偏斜的情况，当hash环偏斜以后，缓存往往会极度不均衡的分布在各服务器上，如果想要均衡的将缓存分布到2台服务器上，最好能让这2台服务器尽量多的、均匀的出现在hash环上，但是，真实的服务器资源只有2台，我们怎样凭空的让它们多起来呢，没错，就是凭空的让服务器节点多起来，既然没有多余的真正的物理服务器节点，我们就只能将现有的物理节点通过虚拟的方法复制出来。

这些由实际节点虚拟复制而来的节点被称为"虚拟节点"，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以在服务器IP或主机名的后面增加编号来实现。

例如上面的情况，可以为每台服务器计算三个虚拟节点，于是可以分别计算 “Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：



同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到 Node A 上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为 32 甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

总结：
上文中，我们一步步分析了什么是一致性 Hash 算法，主要是考虑到分布式系统每个节点都有可能失效，并且新的节点很可能动态的增加进来的情况，如何保证当系统的节点数目发生变化的时候，我们的系统仍然能够对外提供良好的服务，这是值得考虑的！

原文链接：
https://www.cnblogs.com/study-everyday/p/8629100.html
参考：https://blog.csdn.net/cywosp/article/details/23397179/
https://baike.baidu.com/item/%E4%B8%80%E8%87%B4%E6%80%A7%E5%93%88%E5%B8%8C/2460889?fr=aladdin

三十一、 paxos 算法

在一个分布式系统中，由于节点故障、网络延迟等各种原因，根据 CAP 理论，我们只能保证一致性（Consistency）、可用性（Availability）、分区容错性（Partition Tolerance）中的两个。
对于一致性要求高的系统，比如银行取款机，就会选择牺牲可用性，故障时拒绝服务。MongoDB、Redis、MapReduce 使用这种方案。
对于静态网站、实时性较弱的查询类数据库，会牺牲一致性，允许一段时间内不一致。简单分布式协议 Gossip，数据库 CouchDB、Cassandra 使用这种方案。

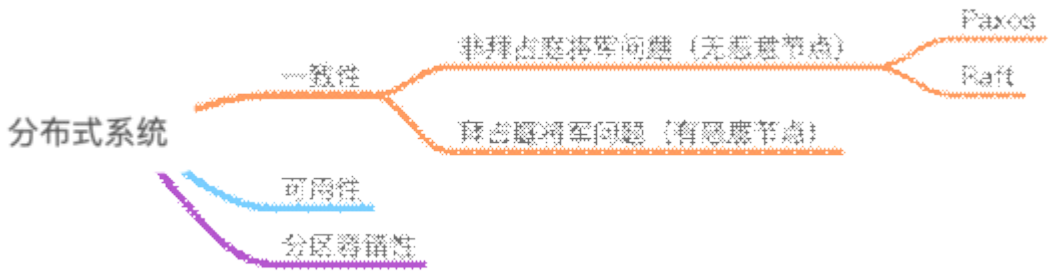


图 1

如图所示，一致性问题，可以根据是否存在恶意节点分类两类。无恶意节点，是指节点会丢失、重发、不响应消息，但不会篡改消息。而恶意节点可能会篡改消息。有恶意节点的问题称为拜占庭将军问题，不在今天的讨论范围。Paxos 很好地解决了无恶意节点的分布式一致性问题。

背景：
1990 年，Leslie Lamport 在论文《The Part-Time Parliament》中提出 Paxos 算法。由于论文使用故事的方式，没有使用数学证明，起初并没有得到重视。直到 1998 年该论文才被正式接受。后来 2001 年 Lamport 又重新组织了论文，发表了《Paxos Made Simple》。作为分布式系统领域的早期贡献者，Lamport 获得了 2013 年图灵奖。
Paxos 算法广泛应用在分布式系统中，Google Chubby 的作者 Mike Burrows 说：“这个世界上只有一种一致性算法，那就是 Paxos（There is only one consensus protocol, and that's Paxos）”。
后来的 Raft 算法、是对 Paxos 的简化和改进，变得更加容易理解和实现。

Paxos 类型：

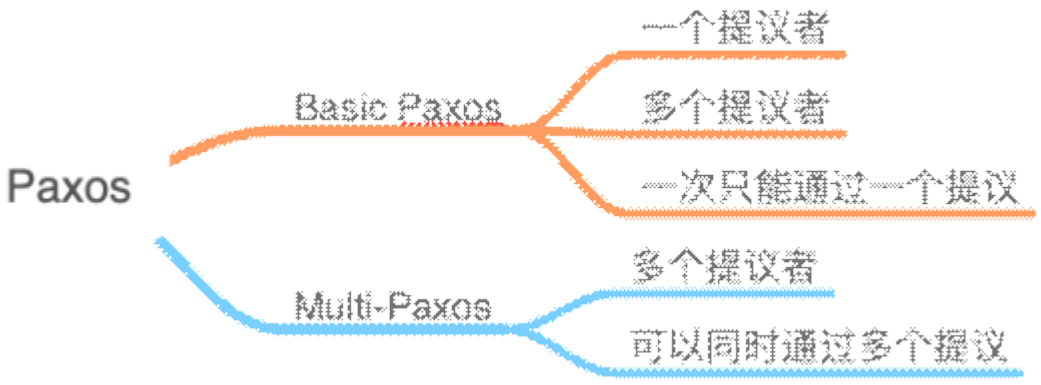


图 2

如图 2 所示，假设议员要提议中午吃什么。如果有一个或者多个人同时提议，但一次只能通过一个提议，这就是 Basic Paxos，是 Paxos 中最基础的协议。

显然 Basic Paxos 是不够高效的，如果将 Basic Paxos 并行起来，同时提出多个提议，比如中午吃什么、吃完去哪里嗨皮、谁请客等提议，议员也可以同时通过多个提议。这就是 Multi-Paxos 协议。

Basic Paxos：
角色
Paxos 算法存在 3 种角色：Proposer、Acceptor、Learner，在实现中一个节点可以担任多个角色。

图 3

- roposer 负责提出提案
- Acceptor 负责对提案进行投票
- Learner 获取投票结果，并帮忙传播

Learner 不参与投票过程，为了简化描述，我们直接忽略掉这个角色。

算法：
运行过程分为两个阶段，Prepare 阶段和 Accept 阶段。

Proposer 需要发出两次请求，Prepare 请求和 Accept 请求。Acceptor 根据其收集的信息，接受或者拒绝提案。

Prepare 阶段

- Proposer 选择一个提案编号 n，发送 Prepare(n)请求给超过半数（或更多）的 Acceptor。
- Acceptor 收到消息后，如果 n 比它之前见过的编号大，就回复这个消息，而且以后不会接受小于 n 的提案。另外，如果之前已经接受了小于 n 的提案，回复那个提案编号和内容给 Proposer。

Accept 阶段

- 当 Proposer 收到超过半数的回复时，就可以发送 Accept(n, value)请求了。 n 就是自己的提案编号，value 是 Acceptor 回复的最大提案编号对应的 value，如果 Acceptor 没有回复任何提案，value 就是 Proposer 自己的提案内容。
- Acceptor 收到消息后，如果 n 大于等于之前见过的最大编号，就记录这个提案编号和内容，回复请求表示接受。
- 当 Proposer 收到超过半数的回复时，说明自己的提案已经被接受。否则回到第一步重新发起提案。

完整算法如图 4 所示：

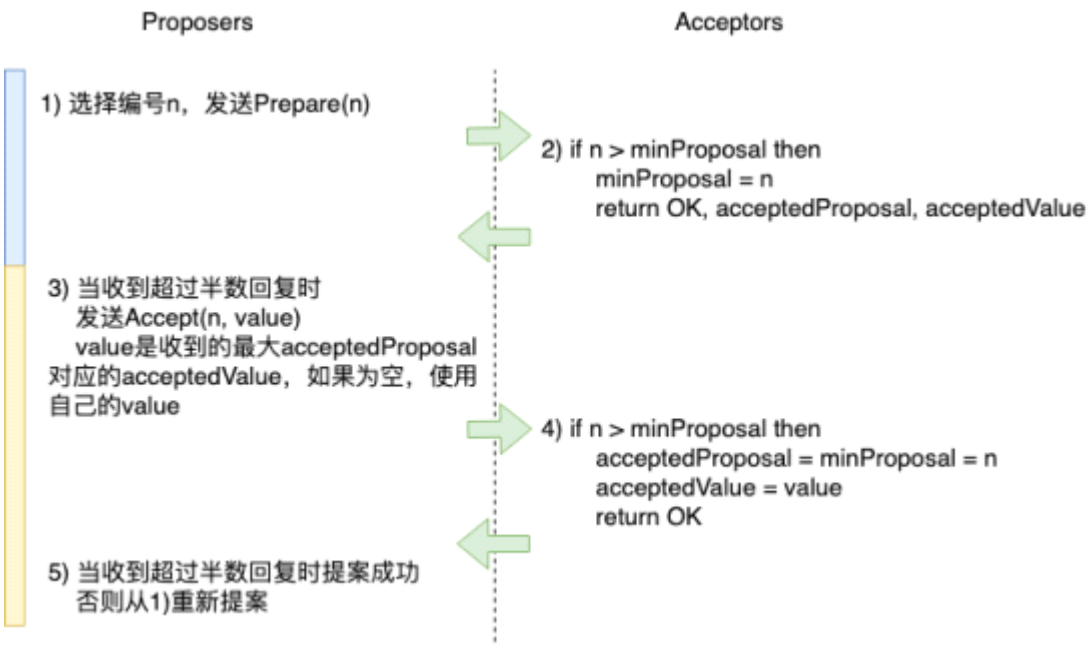


图 4

Acceptor 需要持久化存储 minProposal、acceptedProposal、acceptedValue 这 3 个值。

三种情况

Basic Paxos 共识过程一共有三种可能的情况。下面分别进行介绍。

情况 1：提案已接受

如图 5 所示。X、Y 代表客户端，S1 到 S5 是服务端，既代表 Proposer 又代表 Acceptor。为了防止重复，Proposer 提出的编号由两部分组成：

序列号.Server ID

例如 S1 提出的提案编号，就是 1.1、2.1、3.1……

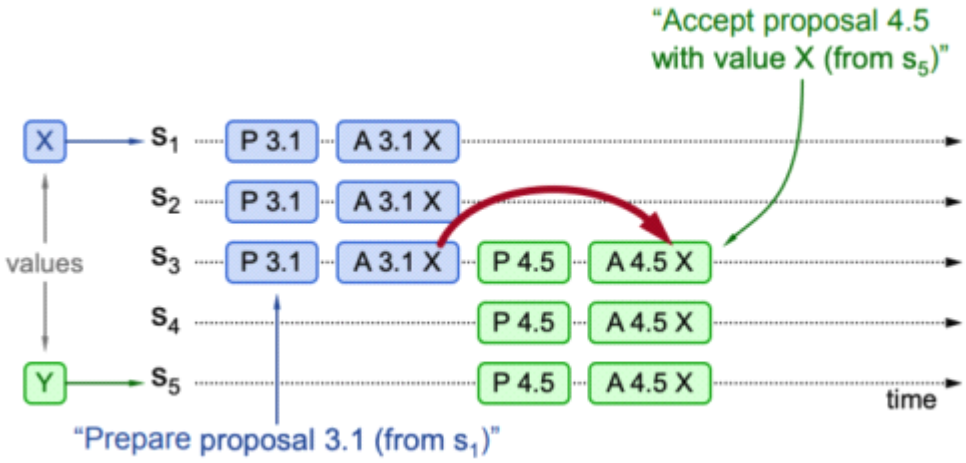


图 5

这个过程表示，S1 收到客户端的提案 X，于是 S1 作为 Proposer，给 S1-S3 发送 Prepare(3.1)请求，由于 Acceptor S1-S3 没有接受过任何提案，所以接受该提案。然后 Proposer S1-S3 发送 Accept(3.1, X)请求，提案 X 成功被接受。

在提案 X 被接受后，S5 收到客户端的提案 Y，S5 给 S3-S5 发送 Prepare(4.5)请求。对 S3 来说，4.5 比 3.1 大，且已经接受了 X，它会回复这个提案 (3.1, X)。S5 收到 S3-S5 的回复后，使用 X 替换自己的 Y，于是发送 Accept(4.5, X)请求。S3-S5 接受提案。最终所有 Acceptor 达成一致，都拥有相同的值 X。
这种情况的结果是：新 Proposer 会使用已接受的提案

情况 2：提案未接受，新 Proposer 可见

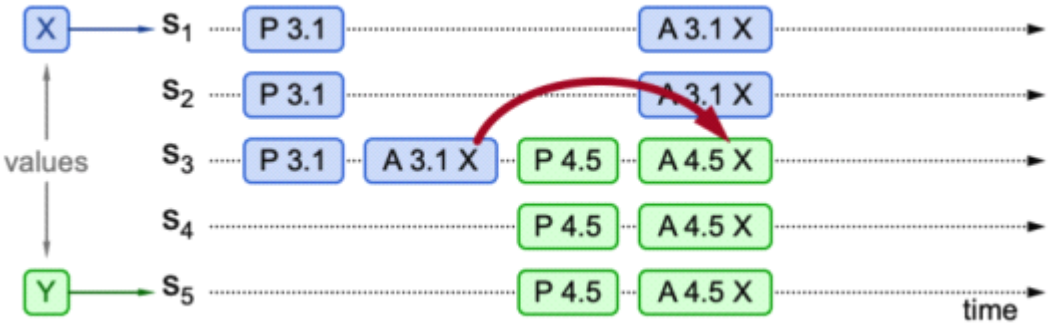


图 6

如图 6 所示，S3 接受了提案(3.1, X)，但 S1-S2 还没有收到请求。此时 S3-S5 收到 Prepare(4.5)，S3 会回复已经接受的提案(3.1, X)，S5 将提案值 Y 替换成 X，发送 Accept(4.5, X)给 S3-S5，对 S3 来说，编号 4.5 大于 3.1，所以会接受这个提案。
然后 S1-S2 接受 Accept(3.1, X)，最终所有 Acceptor 达成一致。
这种情况的结果是：新 Proposer 会使用已提交的值，两个提案都能成功

情况 3：提案未接受，新 Proposer 不可见

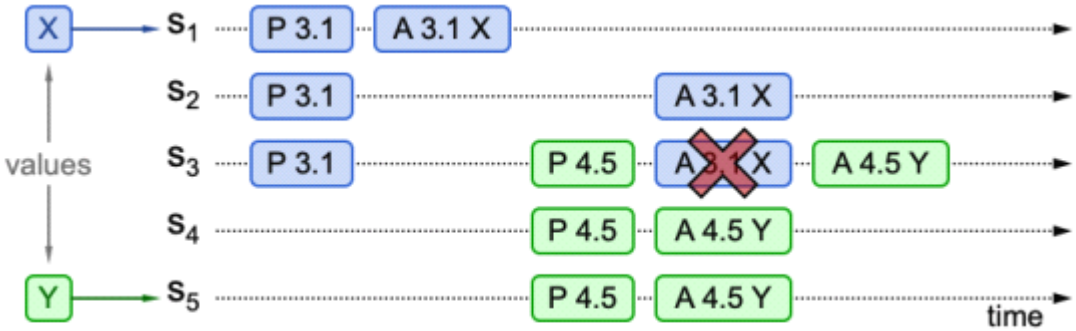


图 7

如图 7 所示，S1 接受了提案(3.1, X)，S3 先收到 Prepare(4.5)，后收到 Accept(3.1, X)，由于 3.1 小于 4.5，会直接拒绝这个提案。所以提案 X 无法收到超过半数的回复，这个提案就被阻止了。提案 Y 可以顺利通过。
这种情况的结果是：新 Proposer 使用自己的提案，旧提案被阻止

活锁 (livelock)
活锁发生的几率很小，但是会严重影响性能。就是两个或者多个 Proposer 在 Prepare 阶段发生互相抢占的情形。

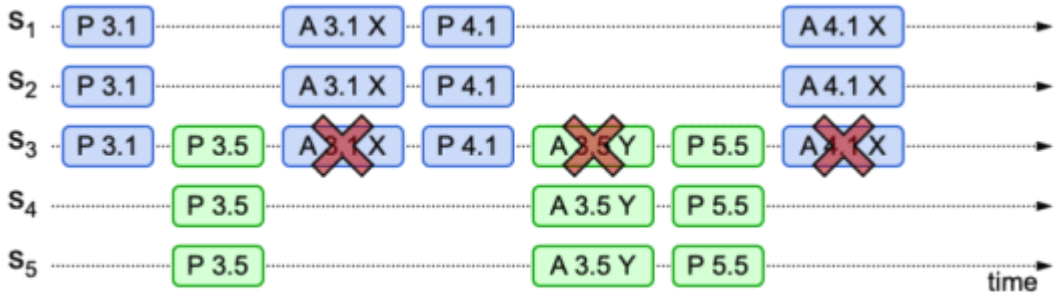


图 8

解决方案是 Proposer 失败之后给一个随机的等待时间，这样就减少同时请求的可能。

Multi-Paxos

上一小节提到的活锁，也可以使用 Multi-Paxos 来解决。它会从 Proposer 中选出一个 Leader，只由 Leader 提交 Proposal，还可以省去 Prepare 阶段，减少了性能损失。当然，直接把 Basic Paxos 的多个 Proposer 的机制搬过来也是可以的，只是性能不够高。将 Basic Paxos 并行之后，就可以同时处理多个提案了，因此要能存储不同的提案，也要保证提案的顺序。
Acceptor 的结构如图 9 所示，每个方块代表一个 Entry，用于存储提案值。用递增的 Index 来区分 Entry。

图 9

Multi-Paxos 需要解决几个问题，我们逐个来看。

1. Leader 选举
一个最简单的选举方法，就是 Server ID 最大的当 Leader。

每个 Server 间隔 T 时间向其他 Server 发送心跳包，如果一个 Server 在 2T 时间内没有收到来自更高 ID 的心跳，那么它就成为 Leader。

其他 Proposer，必须拒绝客户端的请求，或将请求转发给 Leader。

当然，还可以使用其他更复杂的选举方法，这里不再详述。

2. 省略 Prepare 阶段
Prepare 的作用是阻止旧的提案，以及检查是否有已接受的提案值。

当只有一个 Leader 发送提案的时候，Prepare 是不会产生冲突的，可以省略 Prepare 阶段，这样就可以减少一半 RPC 请求。

Prepare 请求的逻辑修改为：
- Acceptor 记录一个全局的最大提案编号

- 回复最大提案编号，如果当前 entry 以及之后的所有 entry 都没有接受任何提案，回复 noMoreAccepted

当 Leader 收到超过半数的 noMoreAccepted 回复，之后就不需要 Prepare 阶段了，只需要发送 Accept 请求。直到 Accept 被拒绝，就重新需要 Prepare 阶段。

3. 完整信息流

目前为止信息是不完整的。

- Basic Paxos 只需超过半数的节点达成一致。但是在 Multi-Paxos 中，这种方式可能会使一些节点无法得到完整的 entry 信息。我们希望每个节点都拥有全部的信息。
- 只有 Proposer 知道一个提案是否被接受了（根据收到的回复），而 Acceptor 无法得知此信息。

第 1 个问题的解决方案很简单，就是 Proposer 给全部节点发送 Accept 请求。

第 2 个问题稍微复杂一些。首先，我们可以增加一个 Success RPC，让 Proposer 显式地告诉 Acceptor，哪个提案已经被接受了，这个是完全可行的，只不过还可以优化一下，减少请求次数。

我们在 Accept 请求中，增加一个 firstUnchosenIndex 参数，表示 Proposer 的第一个未接受的 Index，这个参数隐含的意思是，对该 Proposer 来说，小于 Index 的提案都已经被接受了。因此 Acceptor 可以利用这个信息，把小于 Index 的提案标记为已接受。另外要注意的是，只能标记该 Proposer 的提案，因为如果发生 Leader 切换，不同的 Proposer 拥有的信息可能不同，不区分 Proposer 直接标记的话可能会不一致。

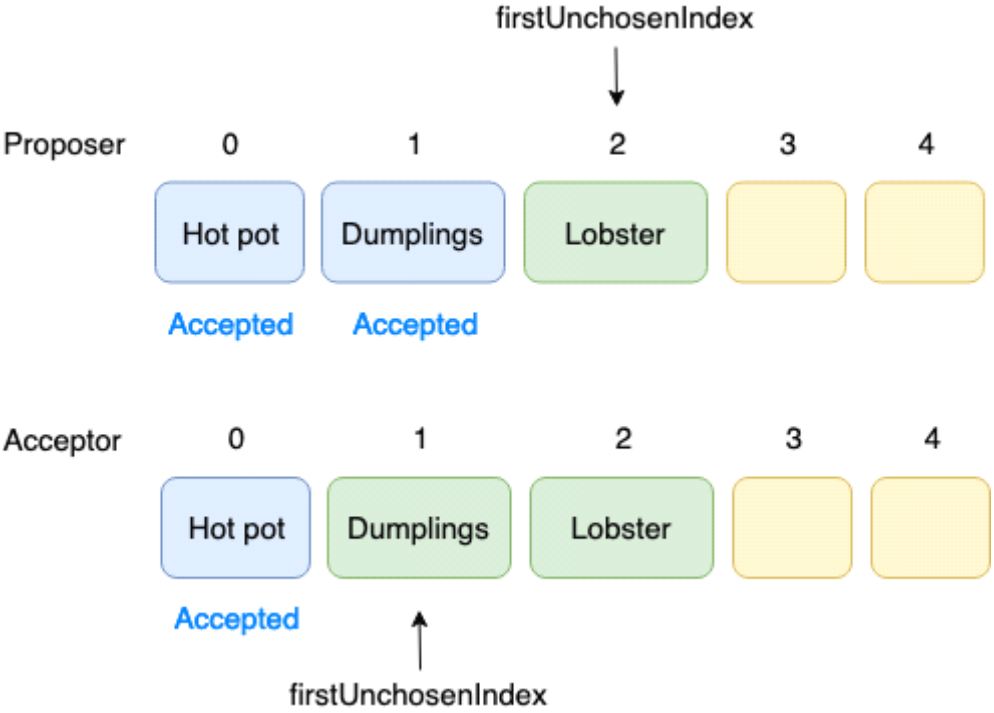


图 10

如图 10 所示，Proposer 正在准备提交 Index=2 的 Accept 请求，0 和 1 是已接受的提案，因此 firstUnchosenIndex=2。当 Acceptor 收到请求后，比较 Index，就可以将 Dumplings 提案标记为已接受。

由于之前提到的 Leader 切换的情况，仍然需要显式请求才能获得完整信息。在 Acceptor 回复 Accept 消息时，带上自己的 firstUnchosenIndex。如果比 Proposer 的小，那么就需

要发送 Success(index, value)，Acceptor 将收到的 index 标记为已接受，再回复新的 firstUnchosenIndex，如此往复直到两者的 index 相等。

总结
Paxos 是分布式一致性问题中的重要共识算法。这篇文章分别介绍了最基础的 Basic Paxos，和能够并行的 Multi-Paxos。

在 Basic Paxos 中，介绍了 3 种基本角色 Proposer、Acceptor、Learner，以及提案时可能发生的 3 种基本情况。在 Multi-Paxos 中，介绍了 3 个需要解决的问题：Leader 选举、Prepare 省略、完整信息流。

在下一篇文章中，我们将实现一个简单的 demo 来验证这个算法，实现过程将会涉及到更多的细节。

原文链接：
https://segmentfault.com/a/1190000018844326

三十二、 raft 算法

https://www.jianshu.com/p/8e4bbe7e276c

三十三、 椭圆曲线算法

为什么使用椭圆曲线加密算法？
RSA 的解决分解整数问题需要亚指数时间复杂度的算法，而目前已知计算椭圆曲线离散对数问题(ECDLP)的最好方法都需要全指数时间复杂度。这意味着在椭圆曲线系统中我们只需要使用相对于 RSA 短得多的密钥就可以达到与其相同的安全强度。例如，一般认为 160 比特的椭圆曲线密钥提供的安全强度与 1024 比特 RSA 密钥相当。使用短的密钥的好处在于加解密速度快、节省能源、节省带宽、存储空间。

比特币以及中国的二代身份证都使用了 256 比特的椭圆曲线密码算法。

实数椭圆曲线

什么是椭圆曲线，想必大家都会首先想到的是高中时学到的标准椭圆曲线方程。

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1(a>b，焦点在x轴，a<b，焦点在y轴)$$

其实本文提到的椭圆曲线，跟这个高中时代的椭圆曲线方程基本无关。椭圆曲线的椭圆一词来源于椭圆周长积分公式。(这个命名深究起来比较复杂，了解一下就可以了[1])

一条椭圆曲线是在射影平面上满足威尔斯特拉斯方程（Weierstrass）所有点的集合，

$$Y^2Z+a_1XYZ+a_3YZ^2=X^3+a_2X^2Z+a_4XZ^2+a_6Z^3$$

对普通平面上点(x,y)，令x=X/Z，y=Y/Z，Z≠0，得到如下方程

$$y^2Z^3+a_1xyZ^3+a_3yZ^3=x^3Z^3+a_2x^2Z^3+a_4xZ^3+a_6Z^3$$

约掉Z^3可以得到：

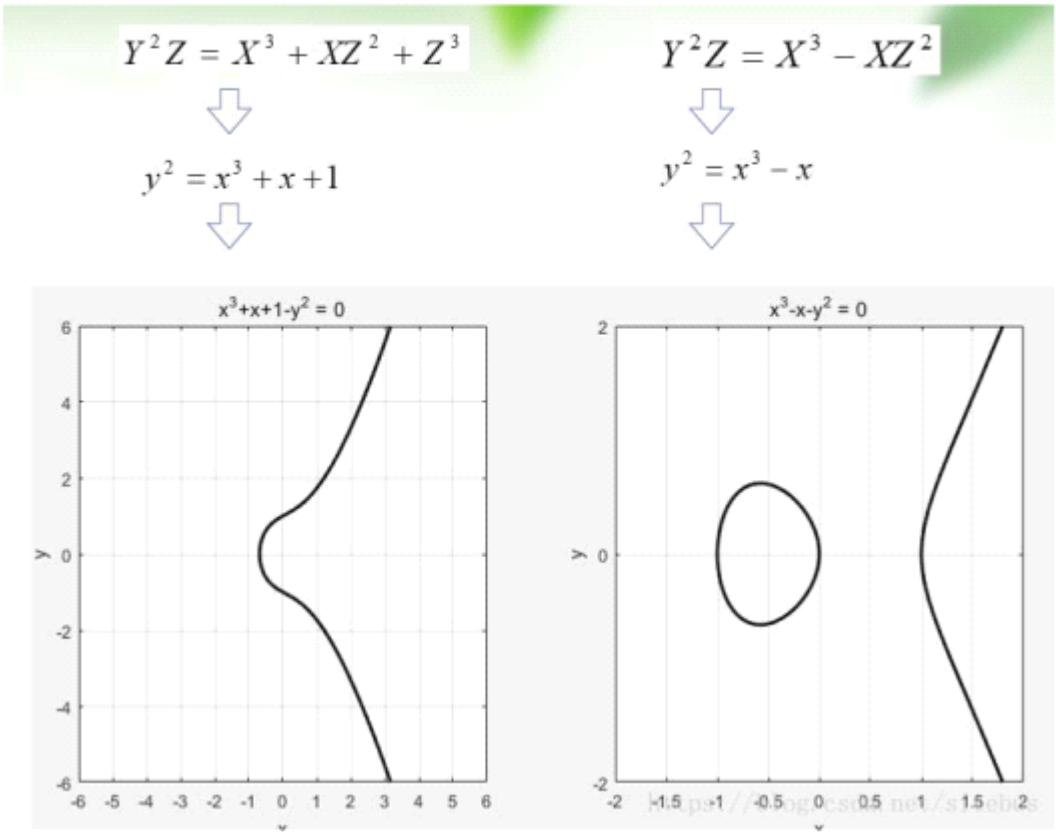
$$y^2+a_1xy+a_3y=x^3+a_2x^2+a_4x+a_6$$

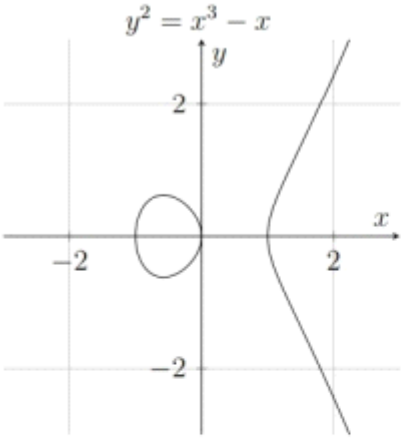
简化版的Weierstrass方程：

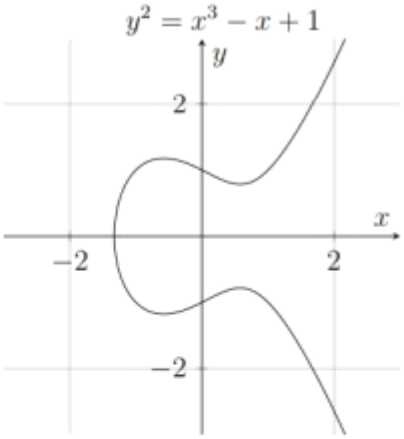
$$E:y^2=x^3+ax+b$$

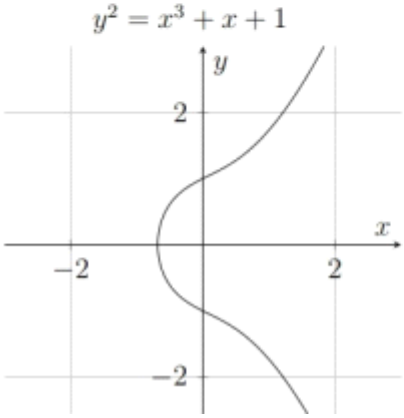
其中，
(1) $\Delta=-16(4a^3+27b)'=0$,用来保证曲线是光滑的，即曲线的所有点都没有两个或者两个以上的不同的切线。
(2) $a,b\in K,K$ 为E的基础域。
(3) 点 O_∞ 是曲线的唯一的无穷远点。

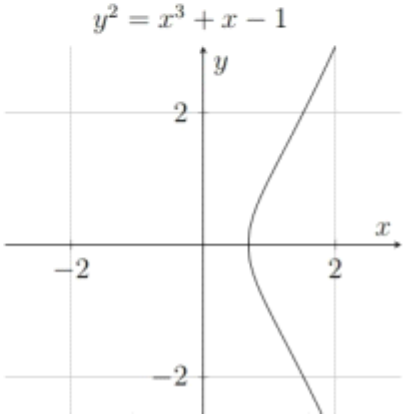
椭圆曲线示例











3. 参考

https://www.cnblogs.com/wellp/p/8536745.html
https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/
https://blog.csdn.net/hunandexingkong/article/details/73188945
https://blog.csdn.net/sitebus/article/details/82835492