1、paq8pxd\_v5的代码解释：

（1）Squash函数，返回p = 1/(1 + exp(-d))，d的范围-2047到2047，p的范围为0到4095；p是12位的数表示概率；

（2）Stretch函数，与Squash函数互为反函数，返回d = ln(p/(1-p))，d的范围是-2047到2047，p的取值范围是0到4095；

（3）Mixer函数：

a、初始化：N个输入概率、N\*M个节点权值（初始化为w=0）、S个上下文（n个输入中会有至多s个被选择）、pr[]为S个输出结果概率（初始化为2048）、rate[]为s个神经网络学习率(learning rates)。初始化为default。S>1时，再生成一个神经网络Mixer(S, 1, 1)，这是第2层；



b、add：将概率装到概率数组；

c、set：将上下文装入上下文数组，上下文范围不能超过M；

e、update()：每个bit的概率到达之前先调用，对节点w[i]进行训练，首先根据ncxt个pr[i]值计算得到err，根据ncxt个上下文的值选择更新的节点，其中w[i]的更新与t[i]有关，即与概率数组有关；更新在装概率之前，下一个bit到来之后；更新w之后，概率表与上下文表清空，数目均为0；

（关于bit 与 bit 之间解耦问题：前一个bit的输出概率pr数组会更新当前w数组，但是我们预取下一个bit y，那么就可以在算出混合概率后，直接利用当前pr数组更新w，那么当前bit与下一个bit的mixer就没有任何关系）



d、p()：第一层神经网络：根据ncxt个上下文依次选择起使的nx的节点进行乘法计算，计算得到ncxt结果，放入pr数组，放入第二层神经网络，与第二层的ncxt节点进行乘法计算得出一个结果；



（4）SmallStationaryContextMap函数：

a、初始化：维护一个16位的表t，大小为2的幂（m/2），初始值均为32768，有一个上下文cxt，一个初始化指向t[0]的指针cp；



b、set：将类中的cxt更改为输入的cx计算得到的一个值，其中cxt不超过m/512；

c、mix：使用当前bit y更新cp指向的值，cp依据当前上下文cxt和当前byte c0将指向的位置更新，计算出一个概率stretch((\*cp)>>4)，将其使用add操作加入mixer；



（一个bit只需要一个上下文，上下文不超过n/256，高位会被丢弃，pos为其中一个上下文，如果pos相同+c0相同，则cxt相同，一个很奇怪的地方：用当前y更新上一个查找位置，如果预取下一个bit y’，那么本次就可以在查找之后更新）

（5）RunContextMap函数：

a、维护一个BH表，其中BH表的详细说明：BH表维护了一个N个B字节的表（N是2的幂），其中对于bh[i]返回指向第i个元素的指针，bh[i][0]是第一个byte为校验和，其中bh[i][1]是优先级，bh[i][2..B-1]是其他值(0-255)；RunContextMap维护着每个元素都是4byte的BH表（表大小为2的幂），以及一个指向单个byte的指针，初始指向第一个元素的第2个byte；

b、set：如果cp指针指向的第一个元素为0，将其改为1，将buf(1)装入cp[1]即bh表中元素的第4位，如果cp[1]存在，则cp[1]++；更改cp指针，将cp指针指向索引为cx的第一个可用值；

c、mix：如果cp指向的值cp[1]与当前c0相同，则算出一个概率p，否则概率为0，并将概率加入mixer；

（6）StateMap函数：

a、初始化：维护一个N个元素的t表初始化为1<<31（U32，这相当于一个计算表），一个最后预测的上下文cxt，cxt<N；



b、update：根据输入数limit调整计算表t的大小，t[cxt]根据当前bit y更新；

c、p()：根据输入上下文cx，计算出概率t[cxt=cx]>>20（取高12位）;



（7）mix2函数：

mix2(Mixer& m, int s, StateMap& sm)：s为bit历史状态，关于Statetable的描述：a、nex(state，0)=下一个状态，如果y是0, 0 <=状态< 256

b、nex(state，1)=下一个状态，如果y是1

c、nex(state, 2)=状态表示的位历史中0的个数

d、nex(state, 3)=1的个数

首先根据s，在StateMap中算出1个概率，加入mixer，根据n0，n1另外计算出4个概率加入mixer；



（8）ContextMap函数：

a、hash桶E：一个hash桶有64byte元素，其中7个2byte为校验和，7个7byte的元素，last为最近2次访问的标志索引，关于函数get(U16 chk)：首先检查，last的低4位的索引对应chk值，如果校验和相同，返回bh[last&15][0]的地址，否则，依次检查桶中的校验和，若命中，则返回该bh[i]地址，如果未命中则找到优先级最小的且最近2次均未访问过的，将chk写上去并将7byte值置为0；





b、ContextMap初始化：维护每个元素都是hash桶的一个数组t，其中上下文个数未C，C个指向当前位历史的指针cp（U8\*），C个指向包含cp[i]的7个元素数组的第一个元素的指针cp0（U8\*），C个指向（count, value, unused, unused）的指针runp（U8\*），C个上下文数组cxt（U32），C个StateMap，其中初始化时，cp0[i]和cp[i]指针指向t[0]的bh[0][0]，runp[i]指向t[0]的bh[0][3]；



c、set：将cx放入cxt表中（置换而非hash，使得cx以分散分布）；

d、mix：根据当前byte c0、上一个byte c1、当前c0的bit数bpos、当前bit y来计算概率写入mixer，上下文数组cxt中一共有cn个上下文，对于每一个上下文：

首先根据当前bit y和cp[i]，算出ns=nex(\*cp[i], y1)，更新\*cp[i]=ns；

根据bpos值来更新cp[i]和cp0[i]；

根据(cxt[i]+cc)&(t.size()-1)找到t表的索引，在t表那64byte中，根据cxt[i]>>16找到了与其相同的校验和的块；

根据runp更新，即将bh[i][3]、runp[i][4]更新；

根据上下文中的最后一个字节进行预测，如果runp[i][1]与c0相等，则计算概率加入mixer；

根据cp[i]计算mix2(m, \*cp[i], sm[i])，加入了5个概率到mixer；

最后，bp==7，cn=0，清空cxt表；





**9. StateTable()**

维护了一张256x4表，表中记录了bit history，可通过next(uint8\_t const state, const int y)直接查询，y取0，1，2，3时分别表示，上一个bit=0时的bit history、=1时的bit history、bit history中0的数量、1的数量。还维护了一个256的stateGroup数组，用于对状态进行scale。next(uint8\_t const oldState, const int y, Random &rnd)，先通过旧状态查表stateTable[oldState][y]得到新状态newState，如果在stateGroup中newState>205且oldState<249且(rnd()<<((460-newState)>>3U))!=0，则newState要减4，然后返回；若不满足条件则直接返回newState。

**10. StateMap**

stateMap也是维护了映射一张表，对于不同的mapType，表有不同的初始化值。状态映射是基于状态表的(based on StateTable)，基本目的是从状态表中取出状态，根据状态和映射类型(mapType)，用不同的方式计算比值【bit1的数量/bit0的数量】，并填入表中。用状态计算概率时，先计算索引，再根据索引从这对应的表中取比值，右移20位，作为概率返回

* **表的初始化**

初始化时，s个上下文集，每个里有n个上下文。cxt是大小为s的上下文数组，数组内装着每个上下文集的最后一次预测的上下文索引，数组的检索用numContexts（0-s）记录。根据Maptype的不同种类(bitHistory, run,generic)，用不同的方式去计算历史中0和1的数量，再将1/0数量比填入数组的对应位置。

* **如果上下文表示的是bit history**

索引由两个for构成，外循环是集内上下文具体位置，内循环是第几个集。也就是说依次对不同的上下文集写入的同一个上下文位置cx，写入相同的1/0比。0<cx<numContextsPerSet，numContextsPerSet要求第八位全为0。对于集内的一个位置cx，取cx的后八位作为state（这么做的原因其实很简单，state的取值本来就是0-255，cx刚好是从0顺序增长）。然后通过stateTable::next(state,2or 3)查表，分别取出历史中0的数目(n0’)和1的数目(n1’)。最终的n0=n0’\*3+1, n1=n1’\*3+1。

将((n1 << 20U) / (n0 + n1)) << 12U写入所有上下文集的cx位置（也就是说不同上下文集的同一位置，写入相同的比值）。

* **如果上下文表示的是run count(数历史中0，1数量)（对应type为run）。**

For(每个上下文集内的第cx个上下文){ //0<cx<numContextPerSet

取cx第0位的bit作为predictedBit;

取cx第1位的bit作为uncertainty;

取cx>>4作为runCount;

0的数量=uncertainty\*16+16；

1的数量=runCount\*128+16;

}

* **如果通用类上下文(generic)**

直接顺序对numContextPerSet\*numComtextSets每个上下文复制2^32，这样在算概率时，右移20位，就会得到2048，也就是0.5的概率。

* 概率计算p1()、p2()

P1()、p2()均是将表中的状态转换为概率。区别在于，当只有一个上下文集时，使用p1，且不调用subscribe()；当有多个上下文集是，使用p2（）和subscribe。

* **P1（）**

将cx装入cxt数组中，因为只有一个上下文集，所以直接存入cxt[0]（也就是说cxt[0]装的是第一个上下文集的最后一次索引cx的值，cxt[1]装的是第二个上下文集的…..）。将t[cx]>>20位作为概率返回。

* **P2（）**

此时因为有多个上下文集，我们需要用numContexts确定是哪一个集。索引的计算变为idx=numContexts\*numContextsPerSet+cx; 然后将这个索引存入cxt[numContexts]; 然后numContexts+1，表示下一个集；最后t[idx]>>20作为概率返回。

* **Skip()**

对于某一个上下文集，在cxt中存储索引为 -1。也就是跳过这个上下文集。

* **Update()**

更新操作是针对t[idx]的，也就是说，每个上下文集，若之前没被跳过，就会从cxt[]中取出对应的idx，并对&t[idx]进行更新。t[idx]分成了高22位，记作n（将原本的t[idx]右移10位）和低10位记作pr。n作为count，pr作为小数部分。如果n<limit, t[idx]++。若n>count，那么t[idx]的作为count的低十位置零，并用limit的值作为低十位。

我们把得到的真实输入bit (y)右移得一个22位的数target，将((target-pr)>>3)\*dt[n]就得到概率的误差delta。 其中dt[n]=16384/(n+n+3)；然后delta的第10位置0得delta’，t[idx]+=delta’。t[idx]的更新完成。

**11. AdaptiveMap()**

这个类主要函数：初始化、update、setLimit()。上述stateMap::update用的就是这里的update函数。

* **初始化**

初始化时创建一个数组变量，这个数组和StateMap中的t[idx]是一致的，表中的每个元素为32位，高22位为预测概率，第10位为count。Limit表示最高允许的count 数。

* **Update(uint32\_t \*const p)**

这里的p就等同于stateMap中的t[idx]，概率的更新过程参照stateMap::update。值得一说的事，更新过程其实只更新了p的高22位（也就是表示预测的位），而表示count的低10位，在count<limit时是不更新的，若≥limit，则用limit 的值代替低10位。

* **SetLimit(const int lim)**

就是让limit=lim。

**12. UpdateBroadcaster&Ipredictor**

Ipredictor提供了一个纯虚函数update()，纯虚函数只能在派生类中实现，ipredictor不可被实例化。

UpdateBroadcaster类比较绕，类中有一个名为subscribers的指针数组，数组里存储的都是指向ipredictor类型的指针，大小为1024。同时，类中又有一个名为subscribe(IPredictor \*subscriber)的函数，它的参数是指向ipredictor的指针，这个指针的名字又叫subscriber。所以这个类中共有三个名字与subscribe有关的东西：1.指针数组subscribers、2.函数subscribe、3函数中的参数subscriber。注意他们的名称！！容易看混！！！！

每个subscriber代表订阅了一个预测器，订阅的预测器总数量用n表示，n<1024。指针数组等于说是将这些订阅了的预测器放到了一个数组里。这个更新广播的意义在于：每一个需要更新的地方，都有一个subscriber并且存在这个指针数组里，当需要更新时（bit到来时），可以统一进行更新。

* Subscribe(ipredictor \*subscriber)

将subscriber放入指针数组subscribers中，然后n++

* broadcastUpdate()

依次取出subscribers指针数组中的subscriber，并调用subscriber->update（）进行更新。

**13.RingBuffer**

创建了一个buffer数组，输入数据会被顺序的装入buffer中，每个元素代表一个byte。buffer(i)是倒数第i个byte，buffer(1)=last hole byte, buffer(0)表示当前正在读入的byte。同样，buffer[0]表示该要压缩的文件读入的第一个byte。buffer中使用offset来表示当前已输入到buffer的字节数。

* **setSize(uint32\_t newSize)**

设置buffer的大小，要求是2的幂次。且offset初始化为0.

* **getpos()**

返回offset的值

* **fill(const type B)**

将类型为type的数据B填入buffer的每个元素。Type是模板中自定的，和buffer的类型一致。

* **add(const type B)**

将B填入数组，offset++。也就是说，将一个新的byte填入buffer中。

* **set(i, B)**

将B填入到buffer[i]中。也就是将指定的上下文填入buffer的指定位置。

* **size()**

返回buffer大小

* **copyTo**

将一个buffer的所有参数完整地复制到另一个上。

**14. Shared**

该文件定义了许多可被各个模型和类访问的全局变量。首先定义了一个大结构体shared, shared中包含一个次结构体变量state，state里面包含了一些全局参数和一些小结构体。

Shared里的全局变量是：RingBuffer，level(越高表示压缩需要的RAM越大，取值0-12)，mem(65536<<level)。还设置了一个私有成员 updateBroadcaster

**State里的全局变量是**：上一个bit **y**；上0-7个bit **c0**； **c1**上一个完整的byte<即buffer(1)>； **bitPosition**当前要被预测的bit的位置(该bit在c0中的位置)；**c4**<buf(4)…buf(1)，所以c4是前4个完整的byte，包含了c0>；**c8**<buf(8)…buf(5)> ;

**Blocktype**块类型， **blockPos**块的相对位置，这两个参数用于模型中。

**State中的小结构体包括：**Match{}, Image{}, Text{}, wav{}, Audio{}, rLength{};

* init(uint8\_t level)

设置level的值，并根据level决定mem和buffer的大小，buffer的大小是min(mem\*8, 1ULL << 30)，即不会超过1GB。

* update(int y)

得到真实的输入的bit后，更新y,c0,c4,c8。更新y为该bit; c0左移1位并加上y; bitPosition+1 但要与1111111(7U)做与操作，以确保bitposition的值在0-7内。{如果bitposition==0，那么c0已经代表一个完整的byte，使c1=c0，并将c1 通过add填入buffer（也就是将这个新的完整的byte填入buffer），然后更新c8，c4，其实c8 c4的更新就是把byte往前移。（具体操作是，c8左移8位，并将c4的最高8位填入c8空出来的低8位。C4左移8位并将c0填入c4空出来的低8位），然后c0置为1。} 然后会对text结构体做一些操作(这里暂时不写，等看到textmodel再写)，然后调用updateBroadcaster进行其它的位置的统一更新。如果bitposition！=0，那么花括号内的操作略去，后面步骤相同。

* reset

将buffer，mem，c0全部重置。

* getUpdateBroadcaster

取私有成员updateBroadcaster的地址，给同类型指针updater。并返回updater。