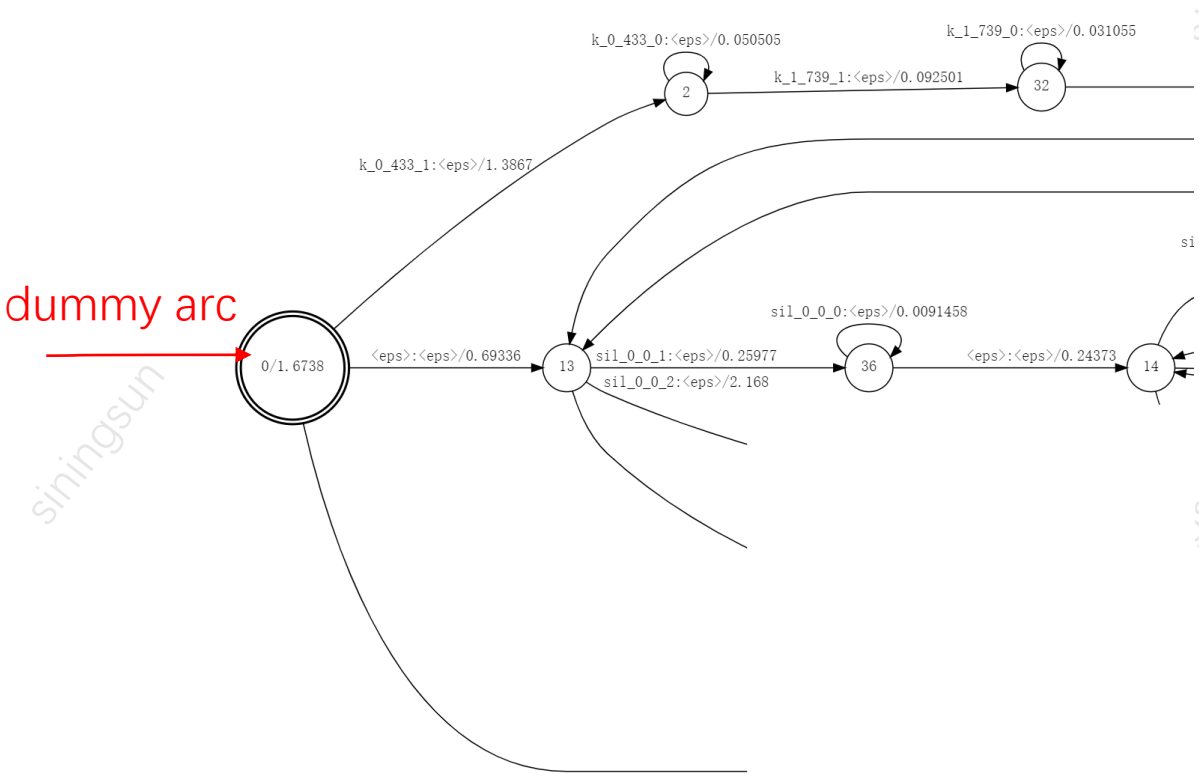
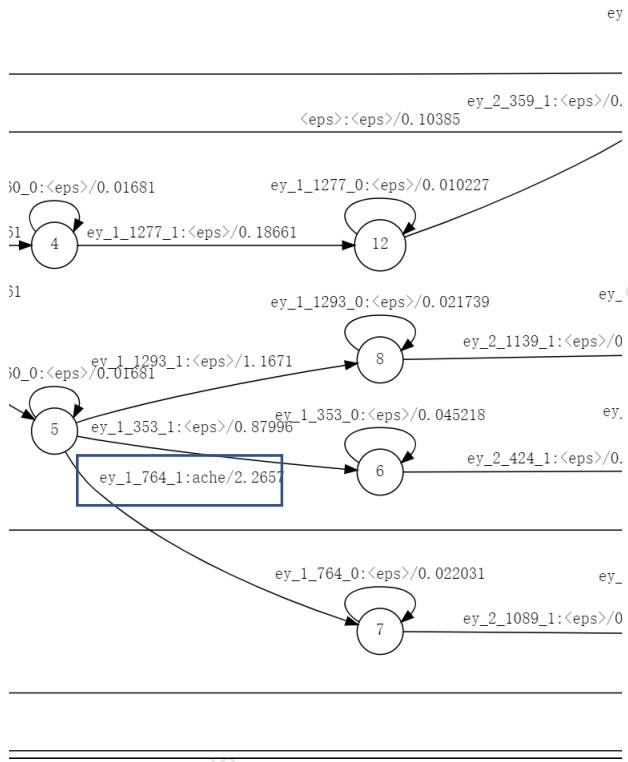
Kaldi SimpleDecoder 代码解析+逐步实例

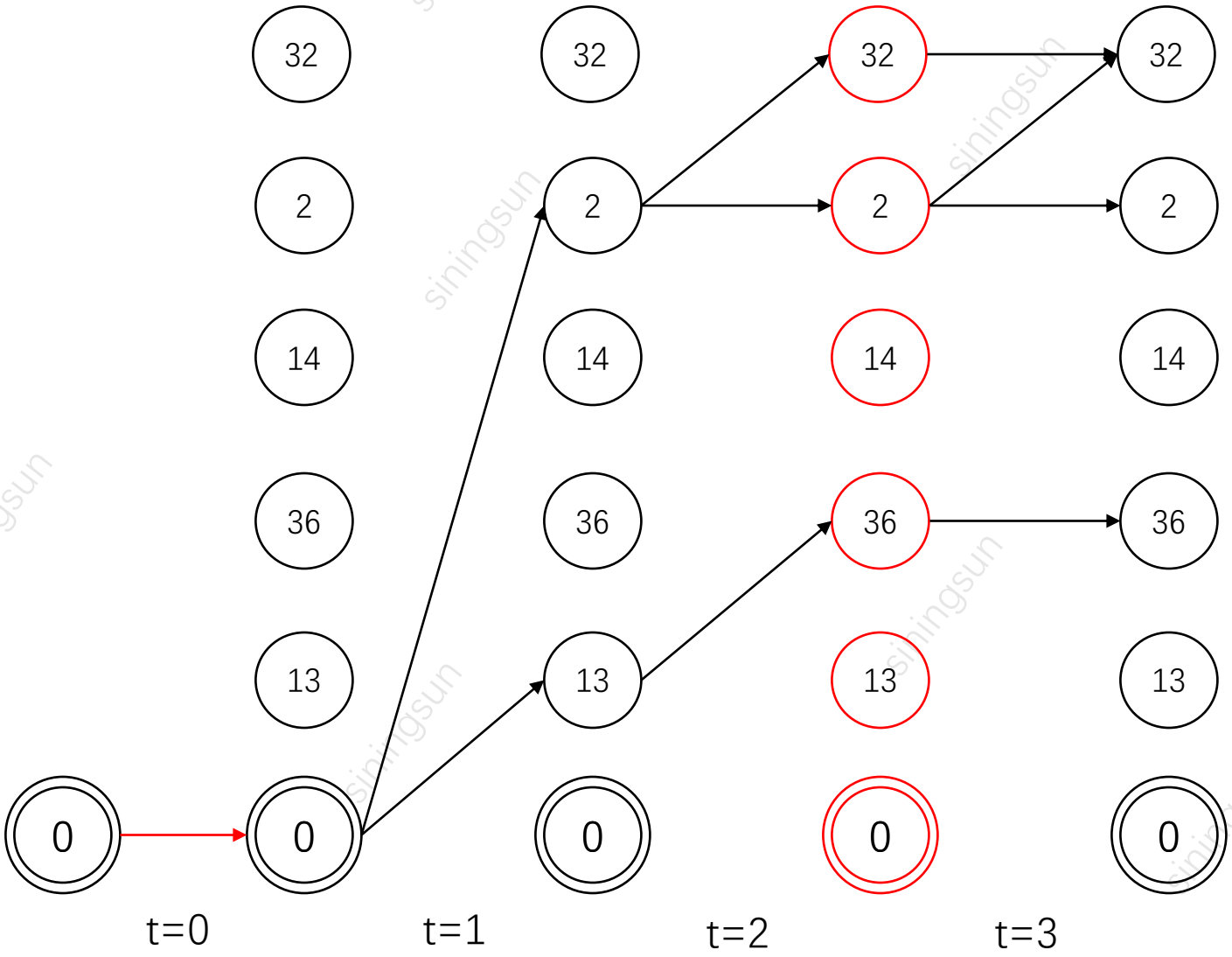
孙思宁

示例HCLG解码图

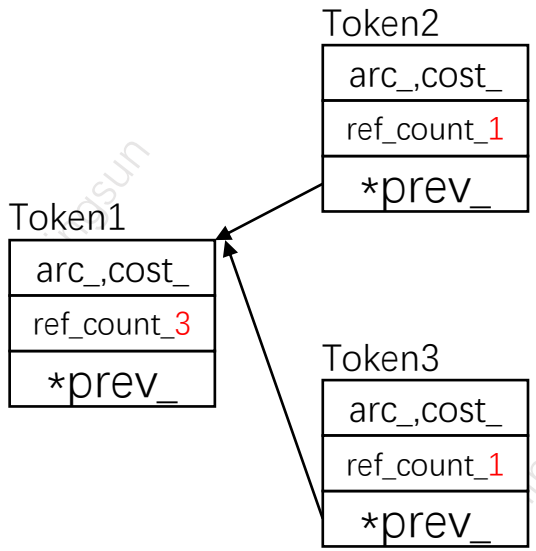

dummy arc
0/1.6738
k_0_433_0:<eps>/0.050505
2
k_1_739_0:<eps>/0.031055
k_1_739_1:<eps>/0.092501
32
k_0_433_1:<eps>/1.3867
<eps>:<eps>/0.69336
13
sil_0_0_1:<eps>/0.25977
sil_0_0_2:<eps>/2.168
sil_0_0_0:<eps>/0.0091458
36
<eps>:<eps>/0.24373
14


ey_2_359_1:<eps>/0.
<eps>:<eps>/0.10385
ey_1_1277_0:<eps>/0.010227
ey_1_1277_1:<eps>/0.18661
4
12
ey_1_1293_0:<eps>/0.021739
ey_2_1139_1:<eps>/0
ey_1_1293_1:<eps>/1.1671
8
ey_1_353_0:<eps>/0.045218
ey_1_353_1:<eps>/0.87996
ey_2_424_1:<eps>/0.
5
6
ey_1_764_1:ache/2.2657
ey_1_764_0:<eps>/0.022031
ey_2_1089_1:<eps>/0
7

将HCLG沿着时间轴展开



arc_	HCLG这个WFST上的转移弧，弧上会有声学分、语言分、输入和输出label，以及目的状态(104-107)
*prev_	指向上一个Token的回溯指针，表示当前Token是从哪个Token传递过来的
ref_count_	主要是用来垃圾回收的，Token每当被其他Token引用一次，计数器+1(109)
cost_	cost_: 当前Token中累积分数
TokenDelete	只有ref_count_为1的Token才能被删除，说明它不会被被其他Token回溯到；TokenDelete是递归回溯删除ref_count_等于1的整条路径上的Token



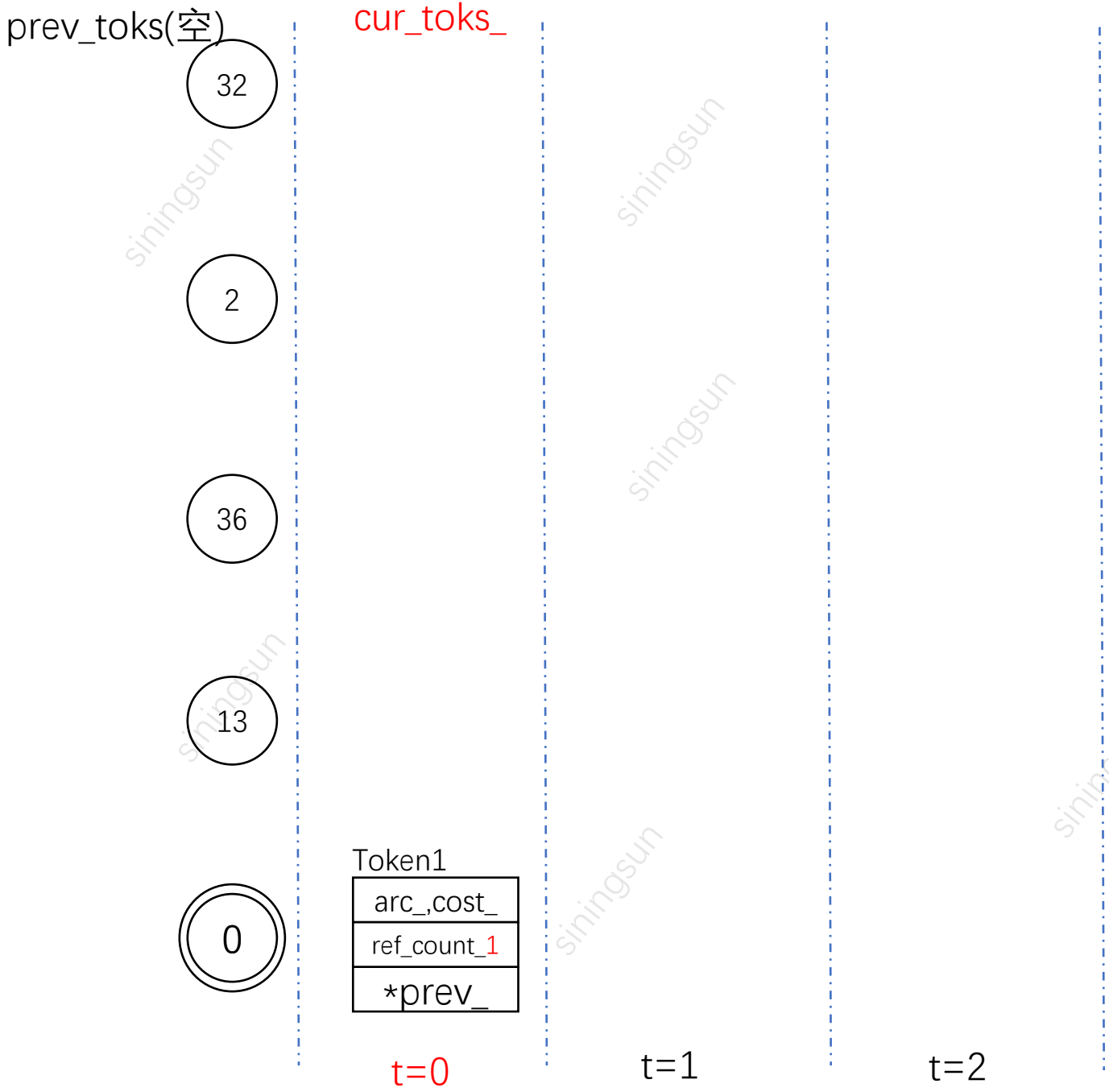
Token示例

Token结构体

InitDecoding

1. 47行，cur_toks加入第一个token，对应图中Tokne1
2. 第49行，处理从state0转移且输入label为空的所有的弧

```
93   class Token {
94    public:
95     LatticeArc arc_; // We use LatticeArc so that we can separately
96                      // store the acoustic and graph cost, in case
97                      // we need to produce lattice-formatted output.
98     Token *prev_;
99     int32 ref_count_;
100    double cost_; // accumulated total cost up to this point.
101    Token(const StdArc &arc,
102          BaseFloat acoustic_cost,
103          Token *prev): prev_(prev), ref_count_(1) {
104      arc_.ilabel = arc.ilabel;
105      arc_.olabel = arc.olabel;
106      arc_.weight = LatticeWeight(arc.weight.Value(), acoustic_cost);
107      arc_.nextstate = arc.nextstate;
108      if (prev) {
109        prev->ref_count_++;
110        cost_ = prev->cost_ + (arc.weight.Value() + acoustic_cost);
111      } else {
112        cost_ = arc.weight.Value() + acoustic_cost;
113      }
114    }
115    bool operator < (const Token &other) {
116      return cost_ > other.cost_;
117    }
118
119    static void TokenDelete(Token *tok) {
120      while (--tok->ref_count_ == 0) {
121        Token *prev = tok->prev_;
122        delete tok;
123        if (prev == NULL) return;
124        else tok = prev;
125      }
126  #ifdef KALDI_PARANOID
127      KALDI_ASSERT(tok->ref_count_ > 0);
128  #endif
129    }
130  };
131
```

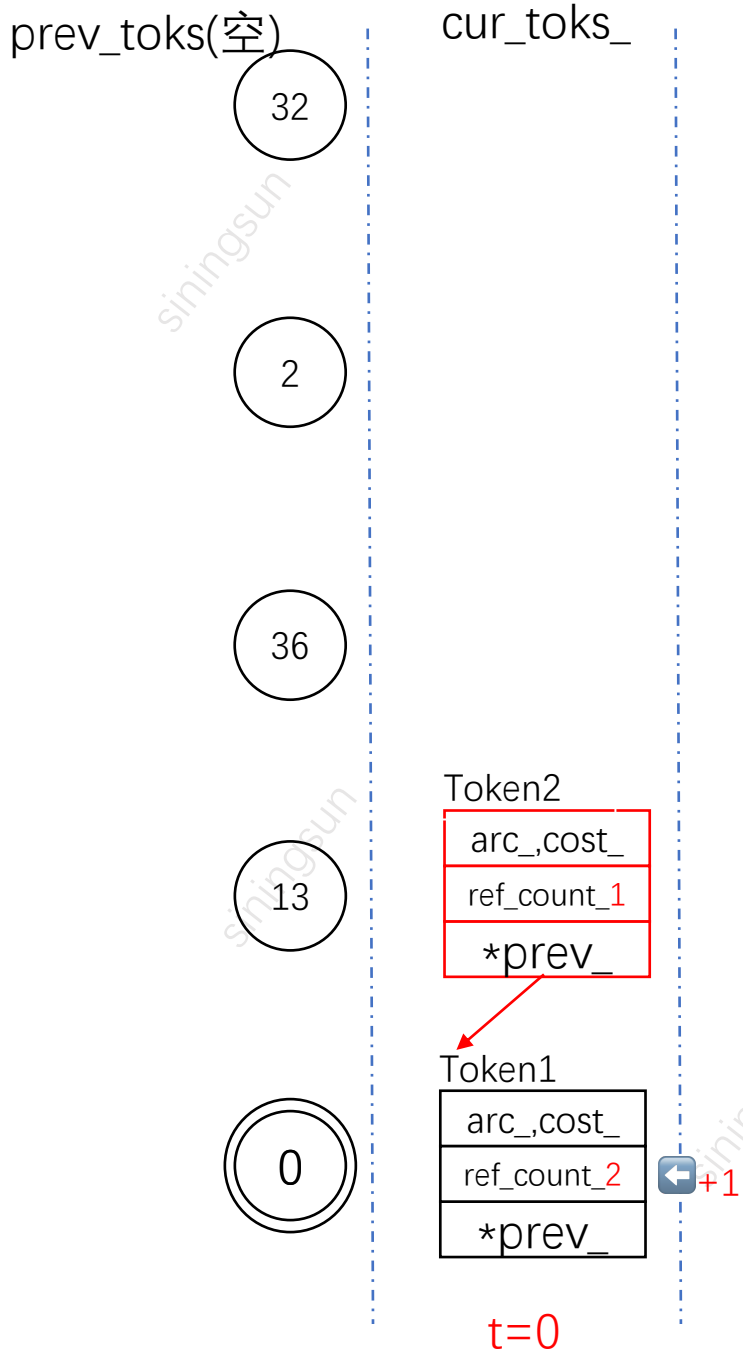


逐步分解SimpleDecoder的解码过程

InitDecoding

1. 47行，cur_toks加入第一个token，对应图中Tokne1
2. 第49行，处理从state0转移且输入label为空的所有的弧

```
39  void SimpleDecoder::InitDecoding() {
40    // clean up from last time:
41    ClearToks(cur_toks_);
42    ClearToks(prev_toks_);
43    // initialize decoding:
44    StateId start_state = fst_.Start();
45    KALDI_ASSERT(start_state != fst::kNoStateId);
46    StdArc dummy_arc(0, 0, StdWeight::One(), start_state);
47    cur_toks_[start_state] = new Token(dummy_arc, 0.0, NULL);
48    num_frames_decoded_ = 0;
49    ProcessNonemitting();
50  }
51
```



逐步分解SimpleDecoder的解码过程

ProcessNonemitting()

1. 本例子只有0->13这一个满足要求的转移
2. 新建一个状态13的Token2，加入到cur_toks_
3. Token2指向了Token1，Token1的ref_count_+1

```
208 void SimpleDecoder::ProcessNonemitting() {
209   // Processes nonemitting arcs for one frame.  Propagates within
210   // cur_toks_.
211   std::vector<StateId> queue;
212   double infinity = std::numeric_limits<double>::infinity();
213   double best_cost = infinity;
214   for (unordered_map<StateId, Token*>::iterator iter = cur_toks_.begin();
215        iter != cur_toks_.end();
216        ++iter) {
217     queue.push_back(iter->first);
218     best_cost = std::min(best_cost, iter->second->cost_);
219   }
220   double cutoff = best_cost + beam_;
221
222   while (!queue.empty()) {
223     StateId state = queue.back();
224     queue.pop_back();
225     Token *tok = cur_toks_[state];
226     KALDI_ASSERT(tok != NULL && state == tok->arc_.nextstate);
227     for (fst::ArcIterator<fst::Fst<StdArc> > aiter(fst_, state);
228          !aiter.Done();
229          aiter.Next()) {
230       const StdArc &arc = aiter.Value();
231       if (arc.ilabel == 0) {  // propagate nonemitting only.
232         const BaseFloat acoustic_cost = 0.0;
233         Token *new_tok = new Token(arc, acoustic_cost, tok);
234         if (new_tok->cost_ > cutoff) {
235           Token::TokenDelete(new_tok);
236         } else {
237           unordered_map<StateId, Token*>::iterator find_iter
238               = cur_toks_.find(arc.nextstate);
239           if (find_iter == cur_toks_.end()) {
240             cur_toks_[arc.nextstate] = new_tok;
241             queue.push_back(arc.nextstate);
242           } else {
243             if ( *(find_iter->second) < *new_tok ) {
244               Token::TokenDelete(find_iter->second);
245               find_iter->second = new_tok;
246               queue.push_back(arc.nextstate);
247             } else {
248               Token::TokenDelete(new_tok);
249             }
250           }
```

遍历cur_toks_里面所有的元素，将state id放入队列，并且得到cur_toks_里面最优的分数

最优分数+beam，计算出cutoff

循环处理cur_toks_里面所有的stateid

处理从当前stateid发射出的所有弧

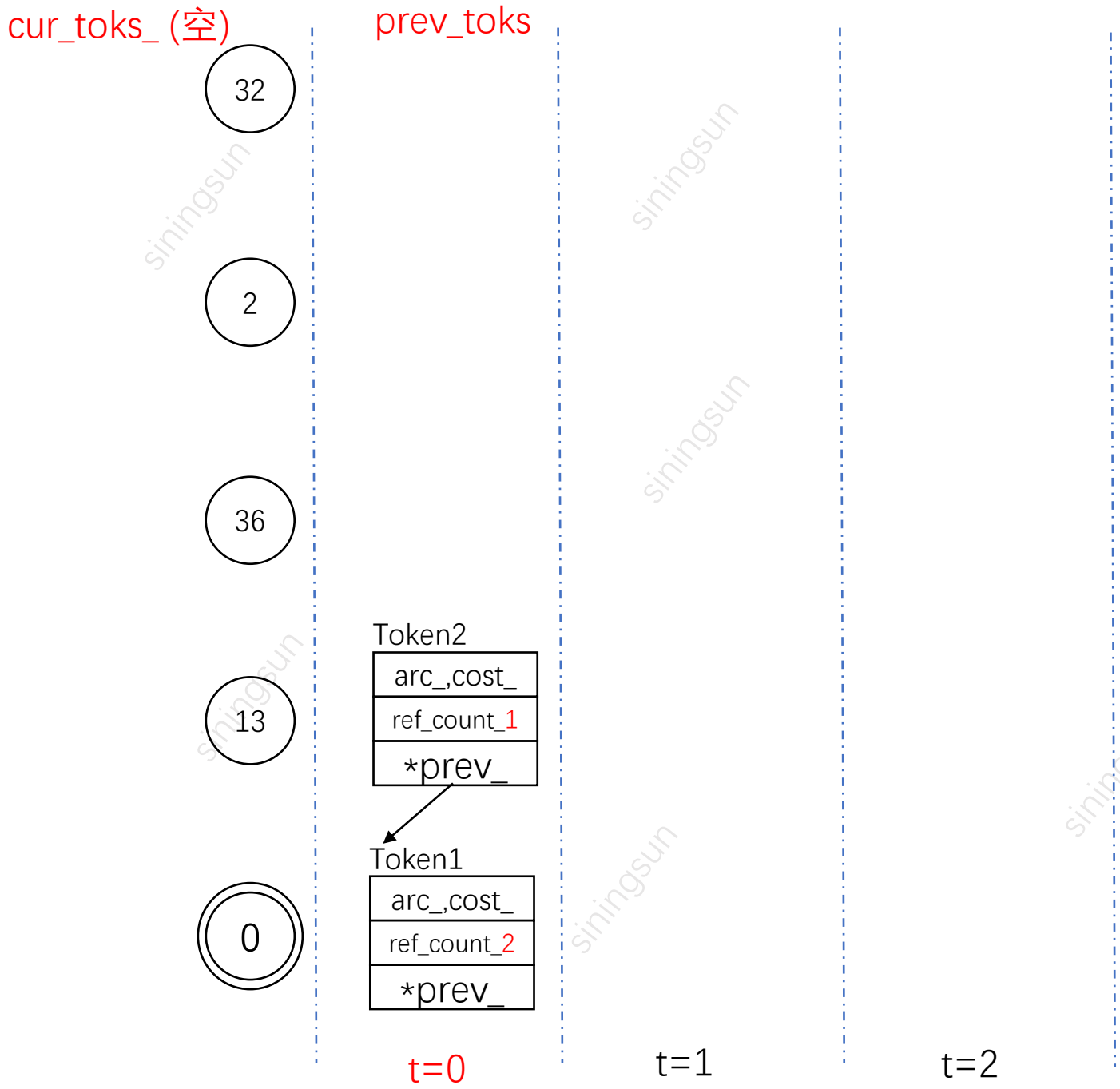
只处理ilabel为空的弧，空转移没有AM分数

如果新token的得分超过cutoff，直接丢弃

查找新token对应的stateid是不是已经在cur_toks_

如果不在，将<stateid, new_tok>加入到cur_toks_，同时这里还会把新的stateid加入到处理队列中，也就是如果有连续的空转移，会一起被处理，一直到达没有空转移状态

如果新token对应的stateid中已经存在于cur_toks_，那么根据维特比的要求，保留得分最优的token，非最优的被递归删除

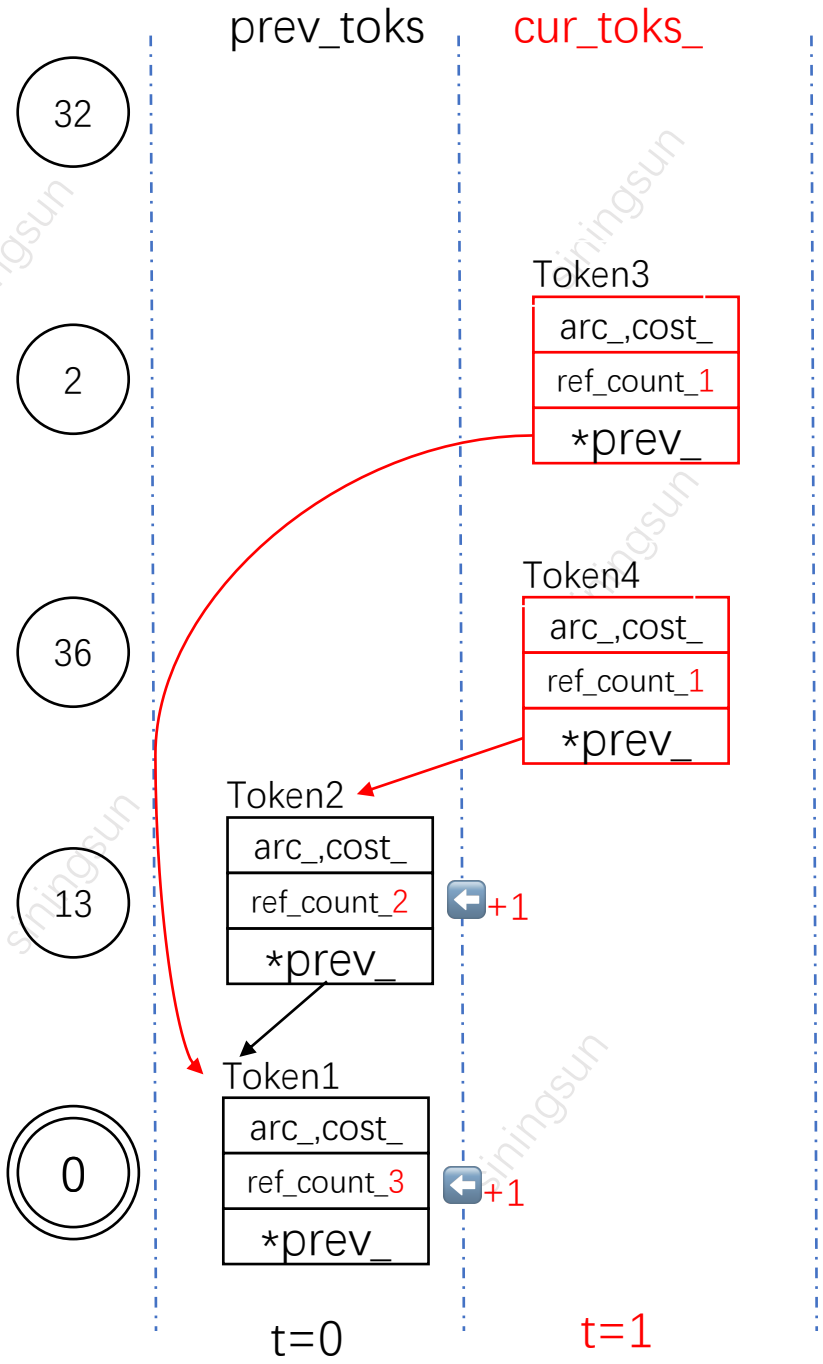


逐步分解SimpleDecoder的解码过程

AdcanceDecoding

1. 解码器开始逐帧处理decodable中的后验概率
2. while循环中，首先ClearToks，此时prev_toks_本身就是空；
3. 将cur_toks_和prev_toks_交换，然后70行开始核心代码

```
52  void SimpleDecoder::AdvanceDecoding(DecodableInterface *decodable,
53                                      int32 max_num_frames) {
54    KALDI_ASSERT(num_frames_decoded_ >= 0 &&
55                 "You must call InitDecoding() before AdvanceDecoding()");
56    int32 num_frames_ready = decodable->NumFramesReady();
57    // num_frames_ready must be >= num_frames_decoded, or else
58    // the number of frames ready must have decreased (which doesn't
59    // make sense) or the decodable object changed between calls
60    // (which isn't allowed).
61    KALDI_ASSERT(num_frames_ready >= num_frames_decoded_);
62    int32 target_frames_decoded = num_frames_ready;
63    if (max_num_frames >= 0)
64      target_frames_decoded = std::min(target_frames_decoded,
65                                       num_frames_decoded_ + max_num_frames);
66    while (num_frames_decoded_ < target_frames_decoded) {
67      // note: ProcessEmitting() increments num_frames_decoded_
68      ClearToks(prev_toks_);
69      cur_toks_.swap(prev_toks_);
70      ProcessEmitting(decodable);
71      ProcessNonemitting();
72      PruneToks(beam_, &cur_toks_);
73    }
74  }
75
```

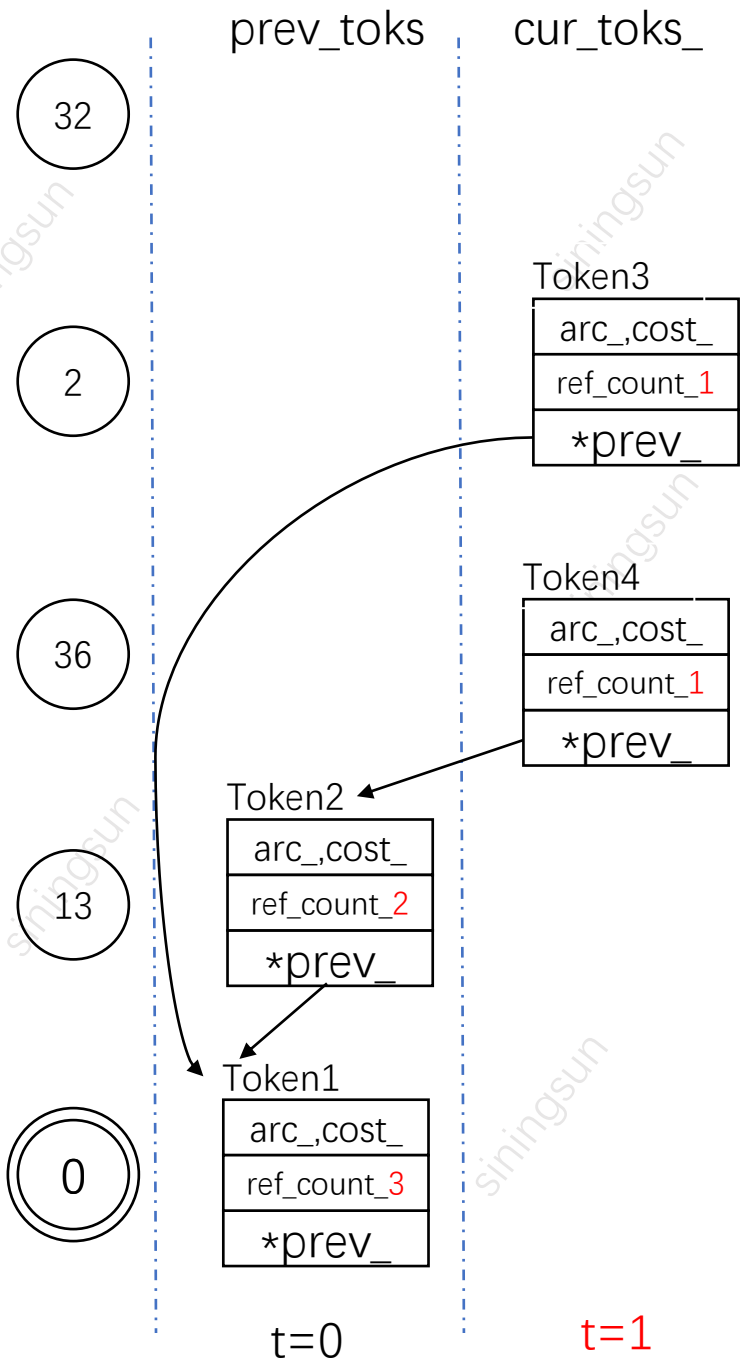


逐步分解SimpleDecoder的解码过程

ProcessEmitting()

1. line 172-204，遍历prev_toks_，而从178-203，遍历每个prev_toks_中的所有可以到达的非空转移
2. 本例子中，S13->S36, S0->S2两个转移产生新的Token3和4，放入了cur_toks，注意此时Token1，2的计数器都增加了1

```
167  void SimpleDecoder::ProcessEmitting(DecodableInterface *decodable) {
168    int32 frame = num_frames_decoded_;
169    // Processes emitting arcs for one frame.  Propagates from
170    // prev_toks_ to cur_toks_.
171    double cutoff = std::numeric_limits<BaseFloat>::infinity();
172    for (unordered_map<StateId, Token*>::iterator iter = prev_toks_.begin();
173         iter != prev_toks_.end();   遍历上一帧所有的stateid-token映射表
174         ++iter) {
175      StateId state = iter->first;
176      Token *tok = iter->second;   获取token对应的state id和token
177      KALDI_ASSERT(state == tok->arc_.nextstate);
178      for (fst::ArcIterator<fst::Fst<StdArc> > aiter(fst_, state);
179           !aiter.Done();   遍历从每个状态出发的所有弧
180           aiter.Next()) {
181        const StdArc &arc = aiter.Value();   只处理输入为非空的转移，
182        if (arc.ilabel != 0) {  // propagate..   空转移交给ProcessNonEmitting处理.
183          BaseFloat acoustic_cost = -decodable->LogLikelihood(frame, arc.ilabel);
184          double total_cost = tok->cost_ + arc.weight.Value() + acoustic_cost;
185                                                   原始token中的得分和新转移的声学、图上的得分累计
186          if (total_cost >= cutoff) continue;   超过cutoff门限就无需进行传递
187          if (total_cost + beam_  < cutoff)   如果当前cost+beam比cufoff更小，那么更新cutoff为
188            cutoff = total_cost + beam_;   一个更加严格的下限
189          Token *new_tok = new Token(arc, acoustic_cost, tok);
190          unordered_map<StateId, Token*>::iterator find_iter   新建一个Token，确定
191              = cur_toks_.find(arc.nextstate);   当前stateid是否已经存在，如果
192          if (find_iter == cur_toks_.end()) {   不存在(192-194),直接将新的token
193            cur_toks_[arc.nextstate] = new_tok;   加入。如果存在，再判断新的token
194          } else {   和已经存在的token哪个更好。如果
195            if ( *(find_iter->second) < *new_tok ) {   new_tok比现有的更好，那么更新
196              Token::TokenDelete(find_iter->second);   当前stateid的Token为new_tok，并且
197              find_iter->second = new_tok;   递归回溯删除原来的token(195-198)；
198            } else {   如果当前token不如先有token，
199              Token::TokenDelete(new_tok);   直接删除(199)
200            }
201          }
202        }
203      }
204    }
205    num_frames_decoded_++;
206  }
```

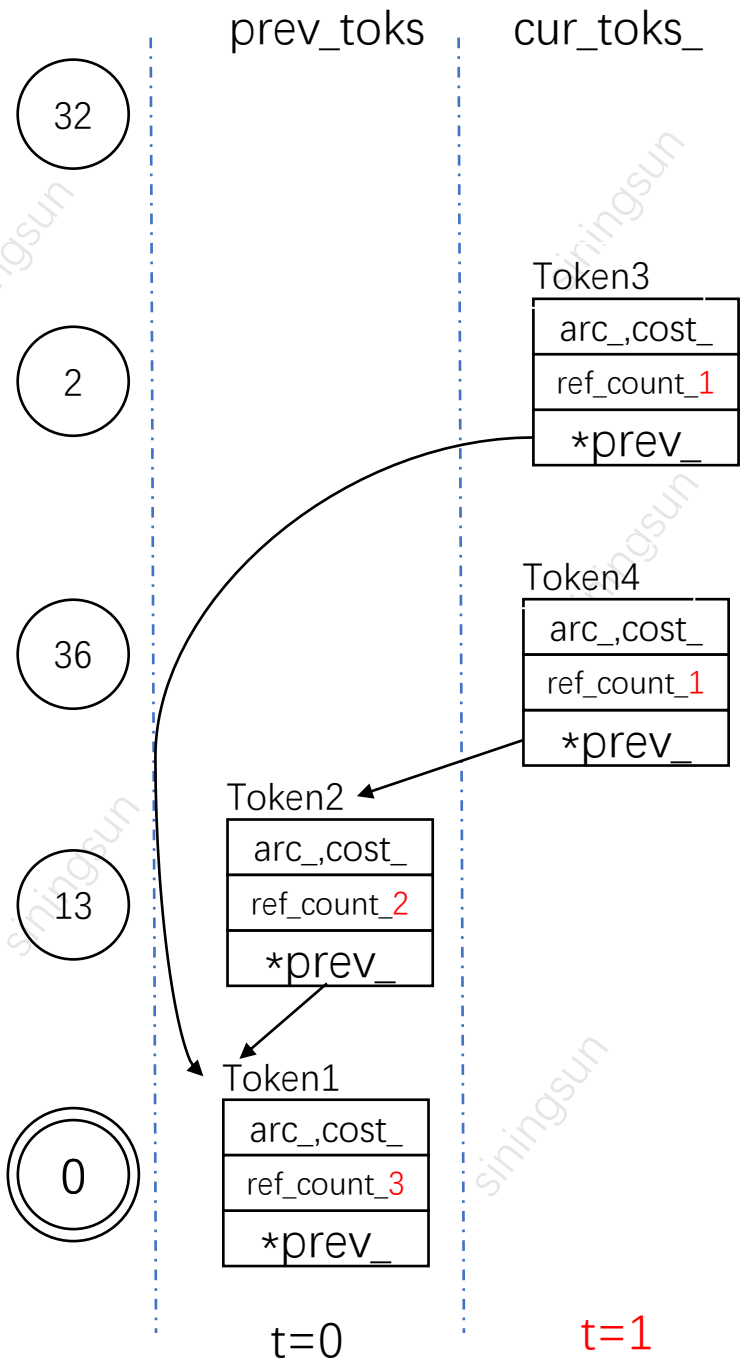


逐步分解SimpleDecoder的解码过程

AdcanceDecoding

1. 返回到AdvanceDecoding里面，继续第71行处理非发射弧。
2. 比例中cur_toks_里的stateid是2，36，都有没有非发射弧，因此这一步没有进行任务操作
3. 最后第72行进行剪枝，

```
52  void SimpleDecoder::AdvanceDecoding(DecodableInterface *decodable,
53                                      int32 max_num_frames) {
54    KALDI_ASSERT(num_frames_decoded_ >= 0 &&
55                 "You must call InitDecoding() before AdvanceDecoding()");
56    int32 num_frames_ready = decodable->NumFramesReady();
57    // num_frames_ready must be >= num_frames_decoded, or else
58    // the number of frames ready must have decreased (which doesn't
59    // make sense) or the decodable object changed between calls
60    // (which isn't allowed).
61    KALDI_ASSERT(num_frames_ready >= num_frames_decoded_);
62    int32 target_frames_decoded = num_frames_ready;
63    if (max_num_frames >= 0)
64      target_frames_decoded = std::min(target_frames_decoded,
65                                       num_frames_decoded_ + max_num_frames);
66    while (num_frames_decoded_ < target_frames_decoded) {
67      // note: ProcessEmitting() increments num_frames_decoded_
68      ClearToks(prev_toks_);
69      cur_toks_.swap(prev_toks_);
70      ProcessEmitting(decodable);
71      ProcessNonemitting();
72      PruneToks(beam_, &cur_toks_);
73    }
74  }
75
```

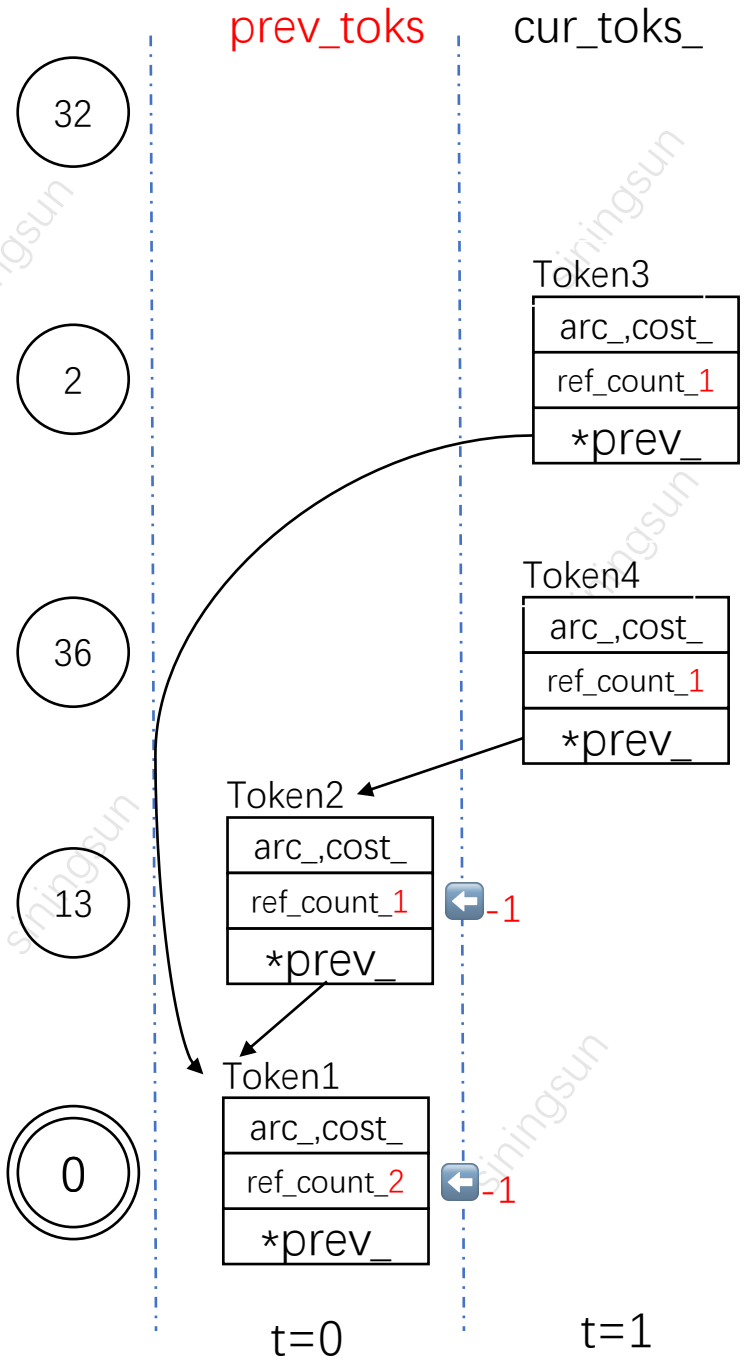


逐步分解SimpleDecoder的解码过程

ProneToks()

1. SimpleDecoder里面的剪枝操作特别简单，FasterDecoder稍微复杂一下，引入了max和min的active token，并动态调整beam
2. 本例子中，我们认为所有的token都在beam以内，没有进行剪枝，cur_toks_没有变化，只是对代码进行简单注释

```
266  // static
267  void SimpleDecoder::PruneToks(BaseFloat beam, unordered_map<StateId, Token*> *toks) {
268    if (toks->empty()) {
269      KALDI_VLOG(2) <<  "No tokens to prune.\n";
270      return;
271    }
272    double best_cost = std::numeric_limits<double>::infinity();
273    for (unordered_map<StateId, Token*>::iterator iter = toks->begin();
274         iter != toks->end(); ++iter)                          遍历cur_toks_里面所有的元素，或得
275      best_cost = std::min(best_cost, iter->second->cost_);    最优的分数存入best_cost_
276    std::vector<StateId> retained;
277    double cutoff = best_cost + beam;
278    for (unordered_map<StateId, Token*>::iterator iter = toks->begin();
279         iter != toks->end(); ++iter) {                        再一遍历cur_toks_，如果token得分在
280      if (iter->second->cost_ < cutoff)                       beam以内，则将stateid放入retained中，
281        retained.push_back(iter->first);                      否则此token就会被剪枝删除。
282      else
283        Token::TokenDelete(iter->second);
284    }
285    unordered_map<StateId, Token*> tmp;
286    for (size_t i = 0; i < retained.size(); i++) {  将保留的tokens存在tmp中，后面在swap
287      tmp[retained[i]] = (*toks)[retained[i]];      到toks_中，也就是cur_toks_里面
288    }
289    KALDI_VLOG(2) <<  "Pruned to " << (retained.size()) << " toks.\n";
290    tmp.swap(*toks);
291  }
```



我们再通过t=2的时候跑一遍整个过程，重新回到AdvanceDecoding这个函数里面

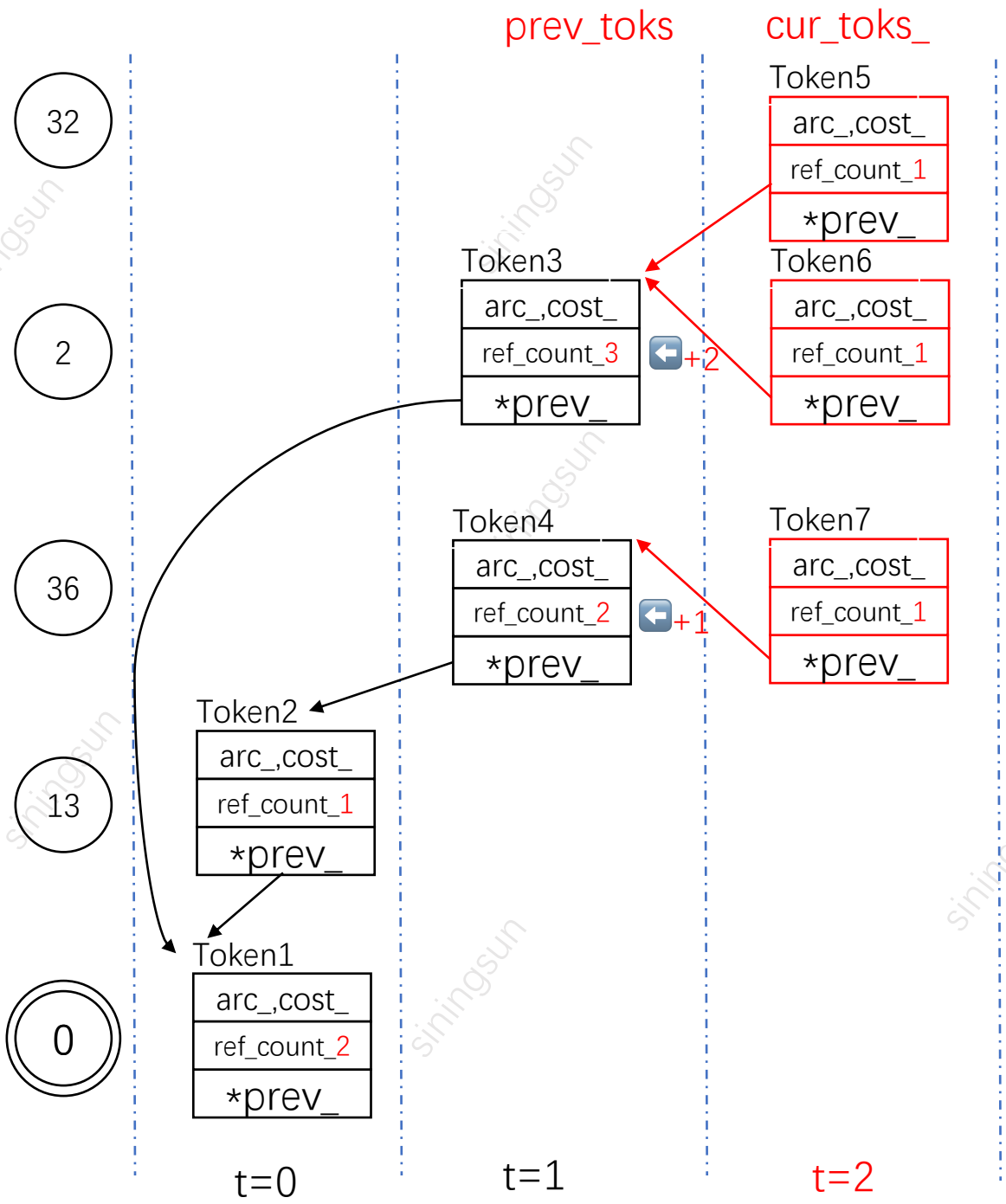
逐步分解SimpleDecoder的解码过程

AdcanceDecoding

1. 首先进行ClearToks(prev_toks_)，此时prev_toks_不再是空
2. ClearToks遍历prev_toks_，递归的删除ref_count_为1的Token，然后将map清空；注意第261行代码，这里prev_toks_的每个token的ref_count_都减去1！这样保证了ref_count_就等于回指的Token个数

```
52  void SimpleDecoder::AdvanceDecoding(DecodableInterface *decodable,
53                                      int32 max_num_frames) {
54    KALDI_ASSERT(num_frames_decoded_ >= 0 &&
55                 "You must call InitDecoding() before AdvanceDecoding()");
56    int32 num_frames_ready = decodable->NumFramesReady();
57    // num_frames_ready must be >= num_frames_decoded, or else
58    // the number of frames ready must have decreased (which doesn't
59    // make sense) or the decodable object changed between calls
60    // (which isn't allowed).
61    KALDI_ASSERT(num_frames_ready >= num_frames_decoded_);
62    int32 target_frames_decoded = num_frames_ready;
63    if (max_num_frames >= 0)
64      target_frames_decoded = std::min(target_frames_decoded,
65                                       num_frames_decoded_ + max_num_frames);
66    while (num_frames_decoded_ < target_frames_decoded) {
67      // note: ProcessEmitting() increments num_frames_decoded_
68      ClearToks(prev_toks_);
69      cur_toks_.swap(prev_toks_);
70      ProcessEmitting(decodable);
71      ProcessNonemitting();
72      PruneToks(beam_, &cur_toks_);
73    }
74  }
75

258  void SimpleDecoder::ClearToks(unordered_map<StateId, Token*> &toks) {
259    for (unordered_map<StateId, Token*>::iterator iter = toks.begin();
260         iter != toks.end(); ++iter) {
261      Token::TokenDelete(iter->second);
262    }
263    toks.clear();
264  }
```

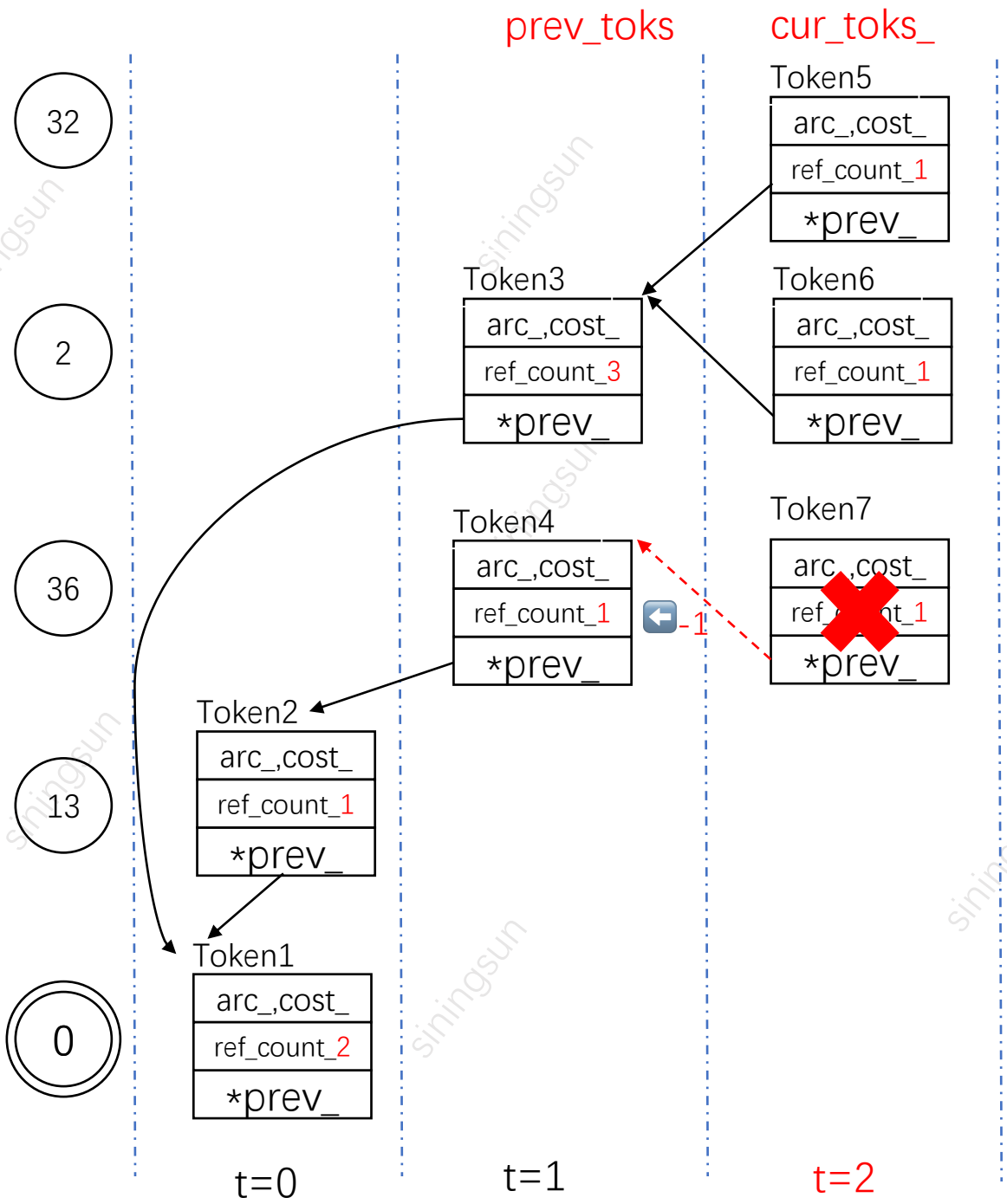


逐步分解SimpleDecoder的解码过程

AdcanceDecoding

1. 69行，把prev_toks_和cur_toks_交换，此时prev_toks_存放的是t=1的所有Token，cur_toks_被清空，准备存储t=2的Tokens
2. 70行，处理prev_toks_里面所有的stateid的发射转移，此时pre_toks_里面只有<2,Token3>和<36,Token4>,此时可能的转移是2->2, 2->32, 36->36，产生了新的Token5-7
3. 我们假设此时没有非发射弧需要处理，并且假设此此时Token7的cost_满足剪枝的要求，然后模拟一下Token7->4->2这条路径是如何在t=2到t=3被剪枝的

```
52  void SimpleDecoder::AdvanceDecoding(DecodableInterface *decodable,
53                                      int32 max_num_frames) {
54    KALDI_ASSERT(num_frames_decoded_ >= 0 &&
55                 "You must call InitDecoding() before AdvanceDecoding()");
56    int32 num_frames_ready = decodable->NumFramesReady();
57    // num_frames_ready must be >= num_frames_decoded, or else
58    // the number of frames ready must have decreased (which doesn't
59    // make sense) or the decodable object changed between calls
60    // (which isn't allowed).
61    KALDI_ASSERT(num_frames_ready >= num_frames_decoded_);
62    int32 target_frames_decoded = num_frames_ready;
63    if (max_num_frames >= 0)
64      target_frames_decoded = std::min(target_frames_decoded,
65                                       num_frames_decoded_ + max_num_frames);
66    while (num_frames_decoded_ < target_frames_decoded) {
67      // note: ProcessEmitting() increments num_frames_decoded_
68      ClearToks(prev_toks_);
69      cur_toks_.swap(prev_toks_);
70      ProcessEmitting(decodable);
71      ProcessNonemitting();
72      PruneToks(beam_, &cur_toks_);
73    }
74  }
75
```

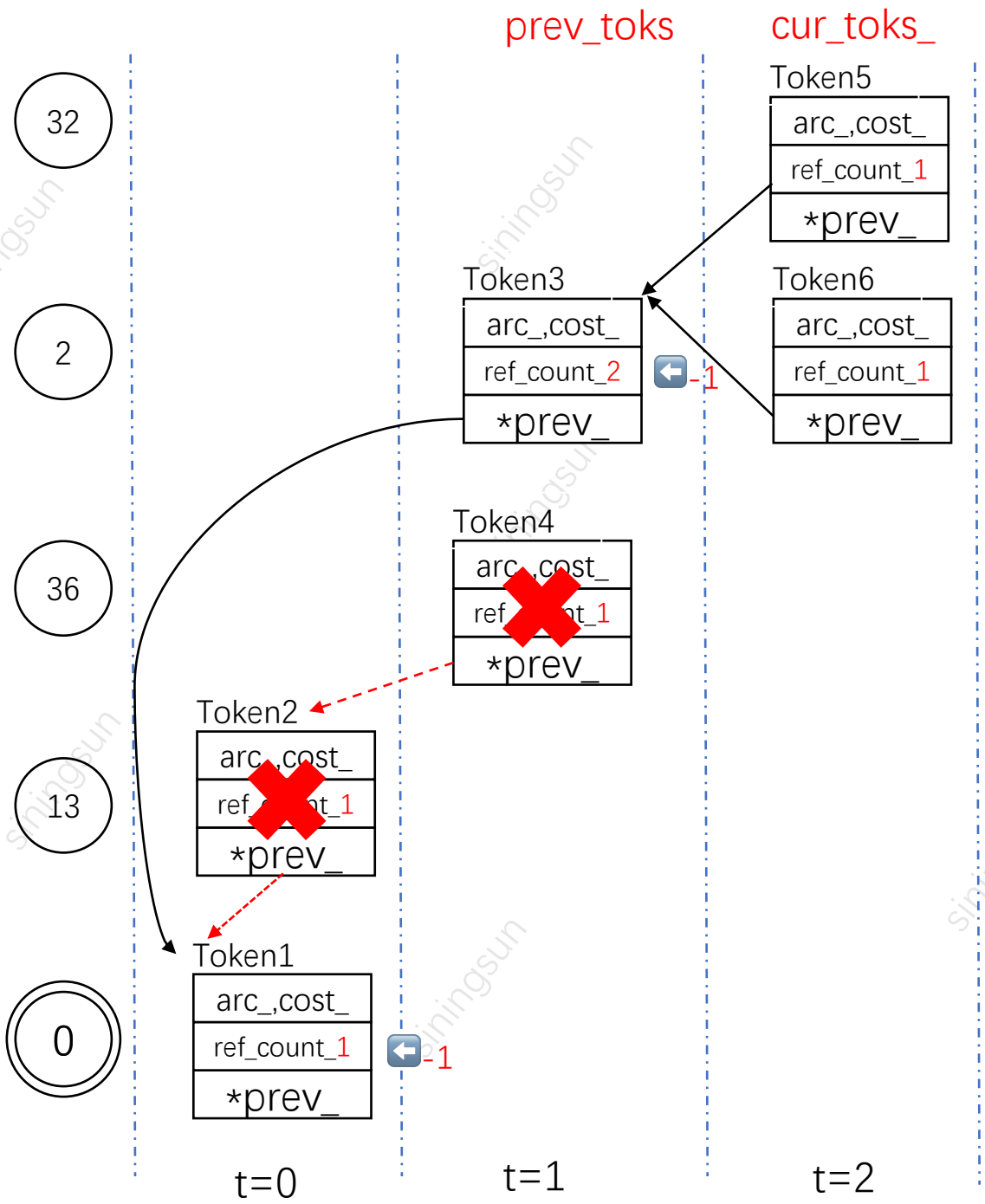


逐步分解SimpleDecoder的解码过程

ProneToks()

1. 此时Token7满足282行，调用TokenDelete递归删除
2. 此时Token7的ref_count_=1,满足while的条件，将Token7的prev_，也就是Token4赋予tok，删除Token7
3. 重新回到while的条件里之后，tok已经指向了Token4，Token4的ref_count_-1=1，不满足条件，退出剪枝
4. 此时并没有删除掉Token4，而这一步是在t=3完成

```
267  void SimpleDecoder::PruneToks(BaseFloat beam, unordered_map<StateId, Token*> *toks) {
268    if (toks->empty()) {
269      KALDI_VLOG(2) <<  "No tokens to prune.\n";
270      return;
271    }
272    double best_cost = std::numeric_limits<double>::infinity();
273    for (unordered_map<StateId, Token*>::iterator iter = toks->begin();
274         iter != toks->end(); ++iter)
275      best_cost = std::min(best_cost, iter->second->cost_);
276    std::vector<StateId> retained;
277    double cutoff = best_cost + beam;
278    for (unordered_map<StateId, Token*>::iterator iter = toks->begin();
279         iter != toks->end(); ++iter) {
280      if (iter->second->cost_ < cutoff)
281        retained.push_back(iter->first);
282      else
283        Token::TokenDelete(iter->second);
284    }
119      static void TokenDelete(Token *tok) {
120        while (--tok->ref_count_ == 0) {
121          Token *prev = tok->prev_;
122          delete tok;
123          if (prev == NULL) return;
124          else tok = prev;
125        }
126  #ifdef KALDI_PARANOID
127        KALDI_ASSERT(tok->ref_count_ > 0);
128  #endif
129      }
130    };
```



逐步分解SimpleDecoder的解码过程

AdcanceDecoding

1. t=3时，68行首先进行ClearToks(prev_toks_)此时prev_toks_还停留在t=1，此时对prev_toks_进行clear，对Token3和4进行递归删除，此时Token4，Token2都会被删除，Token1和Token3的ref_count_会减1.
2. 69行将t=2的toks_交换给prev_toks_,继续解码

```
52  void SimpleDecoder::AdvanceDecoding(DecodableInterface *decodable,
53                                      int32 max_num_frames) {
54    KALDI_ASSERT(num_frames_decoded_ >= 0 &&
55                 "You must call InitDecoding() before AdvanceDecoding()");
56    int32 num_frames_ready = decodable->NumFramesReady();
57    // num_frames_ready must be >= num_frames_decoded, or else
58    // the number of frames ready must have decreased (which doesn't
59    // make sense) or the decodable object changed between calls
60    // (which isn't allowed).
61    KALDI_ASSERT(num_frames_ready >= num_frames_decoded_);
62    int32 target_frames_decoded = num_frames_ready;
63    if (max_num_frames >= 0)
64      target_frames_decoded = std::min(target_frames_decoded,
65                                       num_frames_decoded_ + max_num_frames);
66    while (num_frames_decoded_ < target_frames_decoded) {
67      // note: ProcessEmitting() increments num_frames_decoded_
68      ClearToks(prev_toks_);
69      cur_toks_.swap(prev_toks_);
70      ProcessEmitting(decodable);
71      ProcessNonemitting();
72      PruneToks(beam_, &cur_toks_);
73    }
74  }
75
258 void SimpleDecoder::ClearToks(unordered_map<StateId, Token*> &toks) {
259   for (unordered_map<StateId, Token*>::iterator iter = toks.begin();
260        iter != toks.end(); ++iter) {
261     Token::TokenDelete(iter->second);
262   }
263   toks.clear();
264 }
```