QuickVar软件设计说明书

#### 一、软件概述

QuickVar（快速突变分析）是一款快速的基因组变异分析软件。它使用一种基于*k*-mer比较的无序比对算法，能够快速鉴定单倍体基因组的变异位点。并且，该软件具有优秀的差异检测性能，能够处理大的插入和缺失变异。QuickVar是用C语言实现的，并且不带图形用户界面（GUI），能够快速地处理大量的基因组数据。QuickVar旨在提供一个快速而准确的方法来分析工业微生物的基因变异，以加速生产菌株的设计和优化。

**1.1 软件背景**

本软件面向工业微生物菌种设计改造领域。当前的工业菌株优化技术的核心是基于DBTL的循环优化过程。工业菌株的优化效果严重依赖于这个循环的通量和效率。在DBTL循环种，基因组突变的快速构建和分析是其中重要的一个环节。当前的突变分析软件均通过测序数据Mapping的机制实现。该方法需要将海量的数据与参考基因组进行序列比对，这个过程耗费大量的算力和时间。一个能够高效快速分析海量数据的突变分析软件对于提高DBTL循环的效率和速度具有重要的意义。另外，基于Mapping机制的算法，对于插入删除的分析具有较低的准确率。本软件基于Mapping-free的算法设计，面向工业菌株快速突变分析，解决了基于mapping机制算法在插入删除突变分析方面的低准确度，并且实现了比传统软件数量级的性能提升。

**1.2 软件目标**

QuickVar的主要设计目标是提高分析基因变异的效率和准确性，在加速工业菌株的设计和优化方面具有重要的应用价值。

#### 二、主要功能和特点

QuickVar软件具有以下主要功能和特点：

（1）基因组变异位点的检测：QuickVar使用Mapping-free算法对工业微生物的基因组数据进行分析，以快速检测和识别基因变异。它的差异检测性能非常优秀，能够快速地鉴定单倍体基因组中的变异位点。为了进一步提高性能，QuickVar的开发采用了纯C语言的方案。算法机制的天然优势加上编程语言的速度优势，QuickVar比基于Mapping的算法（GATK，Java）具有约50倍及以上的性能提升，同时具有高准确性的突变分析结果。

（2）插入和缺失变异的处理：基于Mapping的机制在插入删除，尤其是多碱基的插入删除方面，具有较低的准确度。因为QuickVar使用了一种基于德布莱英图（de Bruijn Graph）的路径组装方法，能够快速地准确计算插入删除的细节信息。而且即使是几十，甚至更大片段的插入删除，QuickVar都能够很好的计算处理。QuickVar在插入删除方面比基于Mapping的算法具有显著提升的准确度。

（3）提供了结果展示和导出功能，包括变异类型、位置、碱基变化等详细信息。QuickVar提供了与IGV交互的数据接口，使用户可以更好地理解分析结果。

（4）多线程支持：QuickVar的*k*-mer统计部分采用了多线程的设计，能够使用多个线程来并行处理数据来加快计算的速度。

#### 三、软件架构

**3.1 软件架构概述**

QuickVar包含两个程序：QuickVar和kcov。其中QuickVar是核心程序，kcov是辅助分析程序。QuickVar执行核心的突变分析功能，kcov是一个可以将测序数据与参考基因组的*k*-mers进行比较的工具，可以将参考基因组的每个位置的*k*-mer在测序数据中的出现频率（覆盖度）以\*.wig格式的形式输出。

如图1所示，QuickVar由两个大的模块，模块1）*k*-mer统计模块，和模块2）突变分析模块构成。其中模块1）包含两个主要的子功能模块，分别为多线程序列分析模块和*k*-mer哈希表模块。模块1）是一个多线程的*k*-mer统计模块。

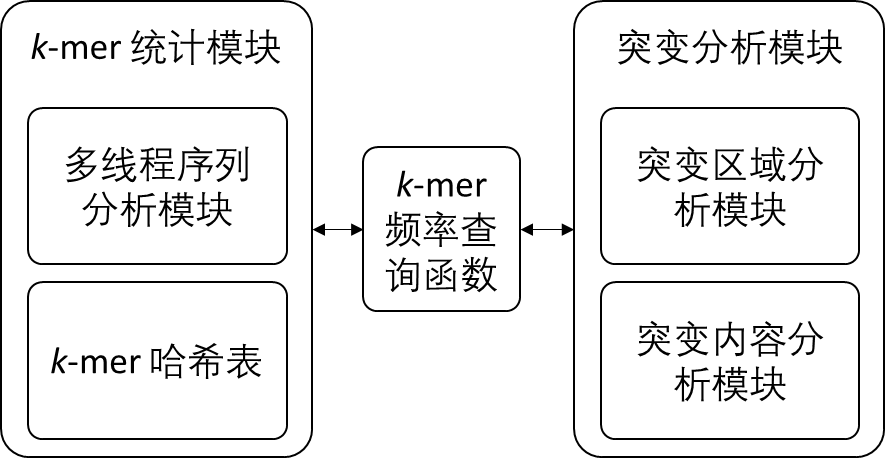


图 1 软件架构图

在*k*-mer统计过程，每个*k*-mer序列按照A-00，C-01，G-10，T-11的碱基-比特映射机制映射为一个二进制比特串，然后将这个二进制比特串进行哈希存储。图2展示了*k*-mer哈希数据结构的整体设计。为了支持多线程的设计，采用1024大小的数组存储了1024个*k*-mer哈希表。按照末尾10个比特的数值大小分别将相应的*k*-mer存储不同地址的*k*-mer哈希表中。例如，如果*k*-mer的末尾10位比特位0000001010，该二进制比特的十进制数值为12，则该*k*-mer将在地址为12-1=11的*k*-mer哈希表中存储。同时，由于该*k*-mer的末尾10位比值可以由所在哈希表的数字地址表示，末尾10位比特将作为该*k*-mer的频率统计信息位使用。这样的机制支持了多线程和频率统计的同时，没有增加任何的信息位。作为频率统计信息位使用的比特位数在程序的起始由宏常量KC\_BITS定义，默认值为10。该默认值支持最高为210-1=1023覆盖度*k*-mer统计，如果需要更高的*k*-mer覆盖度统计需求，可以根据需求在编译前通过调整该宏变量实现。

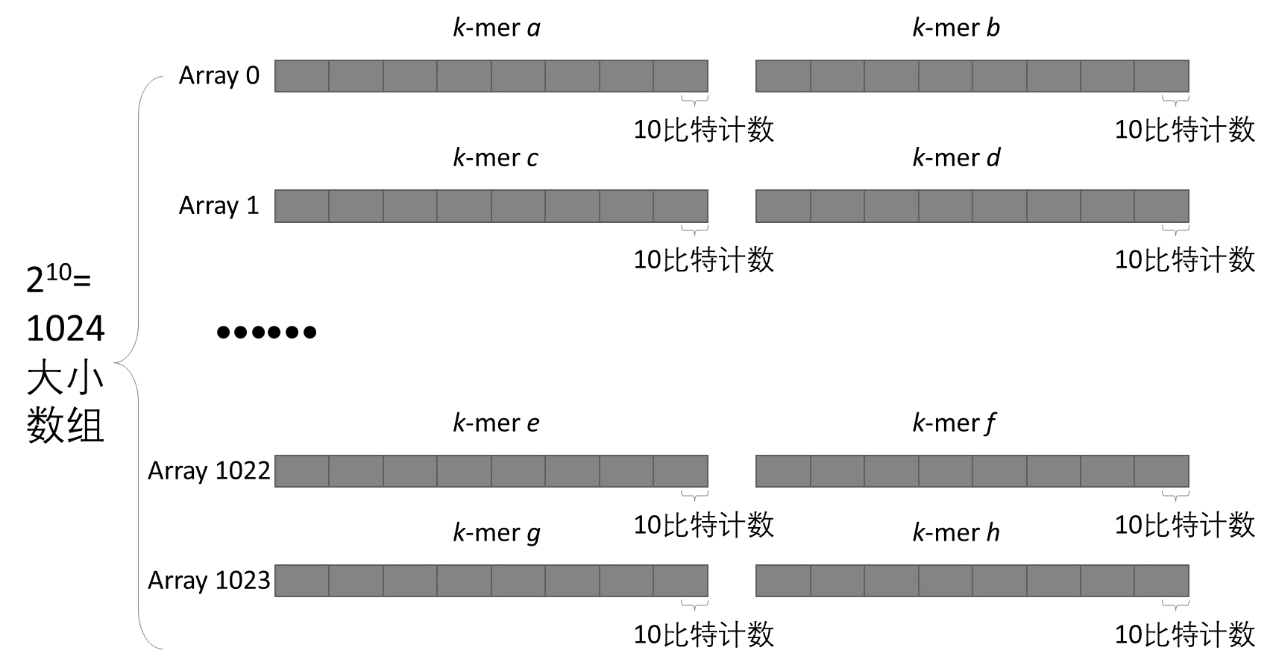


图 2 *k*-mer哈希表的结构示意图（KC\_BITS=10）

**3.2 核心函数名称和功能介绍**

----------------------------------------------------------------------

*k*-mer统计调用函数，该函数可打开一个序列文件并统计其*k*-mer信息，然后将统计信息以kc\_c4x\_t的形式返回。

static kc\_c4x\_t \*count\_file(const char \*fn, int k, int p, int block\_size, int n\_thread)

{

    pldat\_t pl;

    gzFile fp;

    if ((fp = gzopen(fn, "r")) == 0) return 0;

    pl.ks = kseq\_init(fp);

    pl.k = k;

    pl.n\_thread = n\_thread;

    pl.h = c4x\_init(p);

    pl.block\_len = block\_size;

    kt\_pipeline(3, worker\_pipeline, &pl, 3);

    kseq\_destroy(pl.ks);

    gzclose(fp);

    return pl.h;

}

----------------------------------------------------------------------

*k*-mer频率查询函数，该函数可查看kc\_c4x\_t \*h中储存的*k*-mer的频率信息。它接受uint64\_t 作为kmer信息输入，然后查询kc\_c4x\_t \*h中该kmer的统计信息，然后将该信息以整数的形式返回。

int kmer\_cov(uint64\_t kmer, uint64\_t mask, kc\_c4x\_t \*h){

    int j, x, cov=0, a\_key;

    uint64\_t hash\_key = hash64(kmer, mask);

    j = hash\_key & ((1<<KC\_BITS) - 1);

    if(kh\_size(h->h[j]) < 1){return 0;}

    hash\_key = hash\_key >> KC\_BITS<< KC\_BITS;

    x = kc\_c4\_get(h->h[j], hash\_key);

    if(kh\_exist(h->h[j], x)){

        a\_key = kh\_key(h->h[j], x);

        cov = a\_key & KC\_MAX;

    }else{

        return 0;

    }

    return cov;

}

----------------------------------------------------------------------

基于贪婪路径搜索的路径组装函数。该函数通过序列组装的方式来分析var\_loc\_p位置的差异区域具体的序列差异信息。搜索过程的路径信息以链表的形式存储，链表的具体设计在后续的部分在本说明的后续部分有详细的描述。分析结束后，返回满足条件的末尾*k*-mer节点。

int var\_path\_search\_ref(evaluation\_t \*eva, int var\_loc\_p, int max\_path\_len, int min\_cov){

    kc\_c4x\_t \*h = eva->h;

    int k = eva->k;

    uint64\_t mask = (1ULL<<k\*2) - 1; //

    var\_location a\_var = eva->var\_locs[var\_loc\_p];

    uint64\_t kmer\_s = a\_var.kmer\_s;

    uint64\_t kmer\_t = a\_var.kmer\_t;

    int start\_pos = a\_var.pos\_s;

    int term\_pos = a\_var.pos\_t;

    all\_nodes[0].kmer = kmer\_s;

    all\_nodes[0].cov = kmer\_cov(min\_hash\_key(kmer\_s, k),mask,h);

    int a\_cov = 1;

    if(a\_cov < min\_cov ){a\_cov = min\_cov ;}

    term\_nodes[0] = &all\_nodes[0];

    int fresh\_terms = 1, all\_node\_num = 1, p = 0;

    path\_node \*p\_node;

    path\_node next\_nodes[4];

    next\_nodes[0].cov = 0; next\_nodes[1].cov = 0; next\_nodes[2].cov = 0; next\_nodes[3].cov = 0;

    path\_node \*pre\_term\_nodes[Max\_Path\_Num+1];

    int good\_term\_node\_num = 0;

    while(p <= max\_path\_len && fresh\_terms > 0 && fresh\_terms <= Max\_Path\_Num){

        int i;

        //Copying old term nodes pointer

        for(i=0; i < fresh\_terms; i++){pre\_term\_nodes[i] = term\_nodes[i]; }

        int new\_fresh\_terms = 0;

        int n\_p = 0;

        while(n\_p < fresh\_terms && new\_fresh\_terms <=Max\_Path\_Num ){

            p\_node = pre\_term\_nodes[n\_p]; //Obtain term node one by one

            uint64\_t kp = (\*p\_node).kmer<<2;

            next\_nodes[0].kmer = kp & mask;

            next\_nodes[1].kmer = (kp + 1) & mask;

            next\_nodes[2].kmer = (kp + 2) & mask;

            next\_nodes[3].kmer = (kp + 3) & mask;

            next\_nodes[0].cov = kmer\_cov(min\_hash\_key(next\_nodes[0].kmer, k),mask,h);

            next\_nodes[1].cov = kmer\_cov(min\_hash\_key(next\_nodes[1].kmer, k),mask,h);

            next\_nodes[2].cov = kmer\_cov(min\_hash\_key(next\_nodes[2].kmer, k),mask,h);

            next\_nodes[3].cov = kmer\_cov(min\_hash\_key(next\_nodes[3].kmer, k),mask,h);

            int i;

            for(i=0;i<4;i++){

                if(next\_nodes[i].cov >=a\_cov){

                    if( next\_nodes[i].kmer == kmer\_t){ // if the k-mer is equal to kmer\_t, then

                        all\_nodes[all\_node\_num].pre\_node = p\_node;

                        all\_nodes[all\_node\_num].kmer = next\_nodes[i].kmer;

                        all\_nodes[all\_node\_num].cov = next\_nodes[i].cov;

                        good\_term\_nodes[good\_term\_node\_num] = &all\_nodes[all\_node\_num];

                        good\_term\_node\_num++;

                    }else{

                        all\_nodes[all\_node\_num].pre\_node = p\_node;

                        all\_nodes[all\_node\_num].kmer = next\_nodes[i].kmer;

                        all\_nodes[all\_node\_num].cov = next\_nodes[i].cov;

                        term\_nodes[new\_fresh\_terms] = &all\_nodes[all\_node\_num];

                        new\_fresh\_terms++;

                    }

                    all\_node\_num++;

                }

            }

            n\_p++;

        }

        fresh\_terms = new\_fresh\_terms;

        p++;

    }

    if(good\_term\_node\_num == 1){

        if(path\_node\_num(good\_term\_nodes[good\_term\_node\_num-1]) < k ){

            good\_term\_node\_num = 0;

        }

    }

    return good\_term\_node\_num;

}

----------------------------------------------------------------------

序列差异分析调用主函数。该函数分析特定差异位点的具体差异信息并返回。路径组装过程通过调用var\_path\_search\_ref()函数实现。

int var\_analysis\_ref(evaluation\_t \*eva, int var\_loc\_p, int max\_path\_len, int min\_cov){

    uint64\_t mask = (1ULL<<eva->k\*2) - 1;

    var\_location ori\_var;

    ori\_var.kmer\_s = eva->var\_locs[var\_loc\_p].kmer\_s;

    ori\_var.kmer\_t = eva->var\_locs[var\_loc\_p].kmer\_t;

    ori\_var.pos\_s = eva->var\_locs[var\_loc\_p].pos\_s;

    ori\_var.pos\_t = eva->var\_locs[var\_loc\_p].pos\_t;

    int good\_term\_node\_num = var\_path\_search\_ref(eva, var\_loc\_p, max\_path\_len, min\_cov);

    int ext\_times = 0 ;

    while( good\_term\_node\_num < 1 && ext\_times < 5){

        extent\_var\_loc(eva, eva->kms, &eva->var\_locs[var\_loc\_p], min\_cov, eva->k);

        good\_term\_node\_num = var\_path\_search\_ref(eva, var\_loc\_p, max\_path\_len, min\_cov);

        ext\_times = ext\_times + 1;

    }

    int best\_termnode\_index = best\_term\_node(eva, eva->var\_locs[var\_loc\_p].pos\_t - eva->var\_locs[var\_loc\_p].pos\_s, good\_term\_node\_num);

    output\_path(eva, var\_loc\_p, best\_termnode\_index);

    if(good\_term\_node\_num > 1){

        fprintf(stdout, "\tMultiple-path");

    }

    fprintf(stdout, "\n");

    return good\_term\_node\_num;

}

----------------------------------------------------------------------

参考基因组差异区域分析函数。该函数打开一个基因组序列文件，将该基因组的k-mer信息依次存放于数组中。然后，通过分析该基因组序列文件与已经统计好的k-mer信息进行比较，将潜在的差异区域返回。

static evaluation\_t \*analysis\_ref\_seq(evaluation\_t \*eva, const char \*fn, int max\_len, int min\_cov)

{

    pldat\_t pl;

    gzFile fp;

    int k = eva->k;

    int p = eva->p;

    if ((fp = gzopen(fn, "r")) == 0) return 0;

    pl.ks = kseq\_init(fp);

    pl.k = eva->k;

    fprintf(stderr, "Analyzing the locations with variations ...... \n");

    uint64\_t mask = (1ULL<<k\*2) - 1;

    int ret;

    stepdat\_t \*s;

    CALLOC(s, 1);

    s->p = &pl;

    int i = 0;

    var\_location \*var\_locs;

    while ((ret = kseq\_read(pl.ks)) >= 0) {// Reading one seq for each loop

        i = i + 1;

        uint64\_t \*kms;  //all k-mers in array, all forward direction

        int l = pl.ks->seq.l, km\_num = 0, miss\_km\_num = 0;

        if (l < k) continue;

        MALLOC(kms, l);

        km\_num = seq\_kmers(kms, k, l, pl.ks->seq.s);

        MALLOC(var\_locs, (l-k + 1));

        int j = 0;

        int var\_loc\_num = 0;

        var\_location one\_var\_loc;

        one\_var\_loc.kmer\_s = 0;

        one\_var\_loc.pos\_s = -1;

        //one\_var\_loc.name = pl.ks->name;

        pl.ks->name.s;

        int cov=0;

        while(j < km\_num){ // Obtaining the var\_locations for present seq

            cov = kmer\_cov( min\_hash\_key(kms[j],k), mask, eva->h);

            if(cov < min\_cov){ //k-mer NOT present

                if(one\_var\_loc.pos\_s < 0 && j > k){

                    one\_var\_loc.kmer\_s = kms[j-1];

                    one\_var\_loc.pos\_s = j - 1;

                }

            }else{

                if(one\_var\_loc.pos\_s > 0){ //

                    int rep\_km\_num = kmer\_cov(min\_hash\_key(kms[j],k), mask, eva->hr);

                    int s\_km\_cov = kmer\_cov(min\_hash\_key(one\_var\_loc.kmer\_s,k), mask, eva->h);

                        var\_locs[var\_loc\_num].kmer\_s = one\_var\_loc.kmer\_s;

                        var\_locs[var\_loc\_num].pos\_s = one\_var\_loc.pos\_s;

                        var\_locs[var\_loc\_num].kmer\_t = kms[j];

                        var\_locs[var\_loc\_num].pos\_t = j;

                        memcpy(var\_locs[var\_loc\_num].name, pl.ks->name.s, pl.ks->name.l);

                        if(var\_locs[var\_loc\_num].pos\_t - var\_locs[var\_loc\_num].pos\_s <= k){// shift term k-mer

                        //     // fprintf(stdout, "\tshift\t");

                            var\_locs[var\_loc\_num].pos\_t = var\_locs[var\_loc\_num].pos\_s + k + 1;

                            var\_locs[var\_loc\_num].kmer\_t = kms[var\_locs[var\_loc\_num].pos\_t];

                        }

                        one\_var\_loc.kmer\_s = 0;

                        one\_var\_loc.pos\_s = -1;

                        var\_loc\_num++;

                    }

                }

            }

            j++;

        } //var\_locations obtained.

        eva->var\_locs = var\_locs;

        eva->kms = kms;

        fprintf(stderr, "#Total areas with variations detected: %d \n", var\_loc\_num);

        fprintf(stderr, "Analyzing the variation details ...... \n");

        int var\_loc\_p = 0;

        while(var\_loc\_p < var\_loc\_num){

            var\_analysis\_ref(eva, var\_loc\_p, max\_len, min\_cov);

            var\_loc\_p++;

        }

    }

    // Cleaning

    kseq\_destroy(pl.ks);

    return eva;

}

----------------------------------------------------------------------

**3.3 QuickVar运行与交互设计**

QuickVar采用了单个C程序的架构，其运行于命令行界面，没有图形用户界面（GUI）。主程序的main()主函数作为程序的执行入口，各模块功能通过在主函数中调用相应的函数实现。其主要功能通过解析输入文件、执行基因变异分析算法、以及输出分析结果来实现。QuickVar需要一个FASTA格式的基因组序列文件和一个或多个测序数据文件作为输入。测序数据格式支持fastq，fasta以及相应的经gzip压缩后的格式。QuickVar将输出一个包含所有检测到的基因变异信息的文本文件。

QuickVar的运行截图如图3所示。

QuickVar运行参数说明：

QuickVar [参考基因组序列文件] [测序数据文件1] [测序数据文件2] ……[更多的测序序列文件]

可选参数：

-k 可选参数，*k*-mer长度，默认 31

-c 突变分析时设置的最小*k*-mer频率，所有频率小于该值的*k*-mer在突变分析时将不被考虑，默认值8

-l QuickVar采用组装的策略分析差异内容，该参数为组装过程的最大组装长度，默认为500（bp）。

-t *k*-mer统计模块的线程数，默认值3

文本

描述已自动生成

图 3 QuickVar运行截图

此外，QuickVar还提供了一个专门的工具**kcov**用于统计参考基因组中的*k*-mer在测序文件中的频率统计\*.wig文件。该工具的使用与QuickVar主程序类似，不同的是该程序没有-c和-l参数。

kcov的运行截图如图4所示。

kcov运行参数说明：

kcov[参考基因组序列文件] [测序数据文件1] [测序数据文件2] ……[更多的测序序列文件]

可选参数：

-k 可选参数，*k*-mer长度，默认 31

-t *k*-mer统计模块的线程数，默认值3

文本

描述已自动生成

图 4 kcov运行截图

该工具生成的\*.wig文件可直接导入IGV（Integrative Genomics Viewer）进行可视化浏览和分析。如图3所示，该图是一个显示测序读长深度和*k*-mer 覆盖度，也就是kcov生成的文件可视化后的*k*-mer覆盖度（或者出现频率）曲线。

图片包含 图形用户界面

描述已自动生成

图 3 kcov生成的*k*-mer频率\*.wig文件在IGV中导入的样例图

#### 四、软件原理

如图4所示，QuickVar开发了一种无映射的机制来分析基因变异。该机制包括以下两个主要的步骤：

步骤1：基于*k*-mer比较识别变异位点，通过比较单倍体基因组中*k*-mer与参考基因组中的*k*-mer，可以找到变异位点的位置。

步骤2：基于德布莱英图的组装过程，通过组装变异位点周围的*k*-mer序列，可以计算出变异的详细信息，例如变异类型、大小和位置等。在分析过程中，QuickVar应用了多线程技术，以提高分析速度。

图示

描述已自动生成

图 4 QuickVar突变分析原理图

#### 五、数据结构设计

**1）*k*-mer哈希表设计**

如图2所示，QuickVar利用一个uint64\_t 数据来实现*k*-mer值的存储和其出现频率的统计。*k*-mer哈希表的设计基本原理是采用一个哈希函数来定位*k*-mer值在数据结构的存储地址。QuickVar采用的哈希函数如下：

static inline uint64\_t hash64(uint64\_t key, uint64\_t mask) // invertible integer hash function

{

    key = (~key + (key << 21)) & mask; // key = (key << 21) - key - 1;

    key = key ^ key >> 24;

    key = ((key + (key << 3)) + (key << 8)) & mask; // key \* 265

    key = key ^ key >> 14;

    key = ((key + (key << 2)) + (key << 4)) & mask; // key \* 21

    key = key ^ key >> 28;

    key = (key + (key << 31)) & mask;

    return key;

}

// The inversion of hash64(). Modified from <https://naml.us/blog/tag/invertible>

static inline uint64\_t hash64i(uint64\_t key, uint64\_t mask) //

{

    uint64\_t tmp;

    // Invert key = key + (key << 31)

    tmp = (key - (key << 31));  key = (key - (tmp << 31)) & mask;

    // Invert key = key ^ (key >> 28)

    tmp = key ^ key >> 28;  key = key ^ tmp >> 28;

    // Invert key \*= 21

    key = (key \* 14933078535860113213ull) & mask;

    // Invert key = key ^ (key >> 14)

    tmp = key ^ key >> 14;  tmp = key ^ tmp >> 14;  tmp = key ^ tmp >> 14;  key = key ^ tmp >> 14;

    // Invert key \*= 265

    key = (key \* 15244667743933553977ull) & mask;

    // Invert key = key ^ (key >> 24)

    tmp = key ^ key >> 24;  key = key ^ tmp >> 24;

    // Invert key = (~key) + (key << 21)

    tmp = ~key;     tmp = ~(key - (tmp << 21)); tmp = ~(key - (tmp << 21)); key = ~(key - (tmp << 21)) & mask;

    return key;

}

哈希表的数据结构如下：

typedef struct {

    int p; // suffix length; at least 8

    kc\_c4\_t \*\*h; // 1<<p hash tables

} kc\_c4x\_t;

static kc\_c4x\_t \*c4x\_init(int p)

{

    int i;

    kc\_c4x\_t \*h;

    CALLOC(h, 1);

    MALLOC(h->h, 1<<p);

    h->p = p;

    for (i = 0; i < 1<<p; ++i)

        h->h[i] = kc\_c4\_init();

    return h;

}

typedef struct {

    int n, m;

    uint64\_t \*a;

} buf\_c4\_t;

注：该哈希表的实现源于https://github.com/lh3/kmer-cnt

**2）路径搜索链表设计**

在路径搜索过程中，路径信息通过一个反向链表的形式实现。具体的数据结构采用如图5所示的结构体（path\_node）来实现。该结构体包含三个元素，指针pre\_node指向路径的上一节点，uint\_64 kmer是当前节点的*k*-mer信息，unsigned int cov是当前节点k-mer的覆盖度信息。如图5所示，该链表设计可以支持方便的反向计算完整的路径。

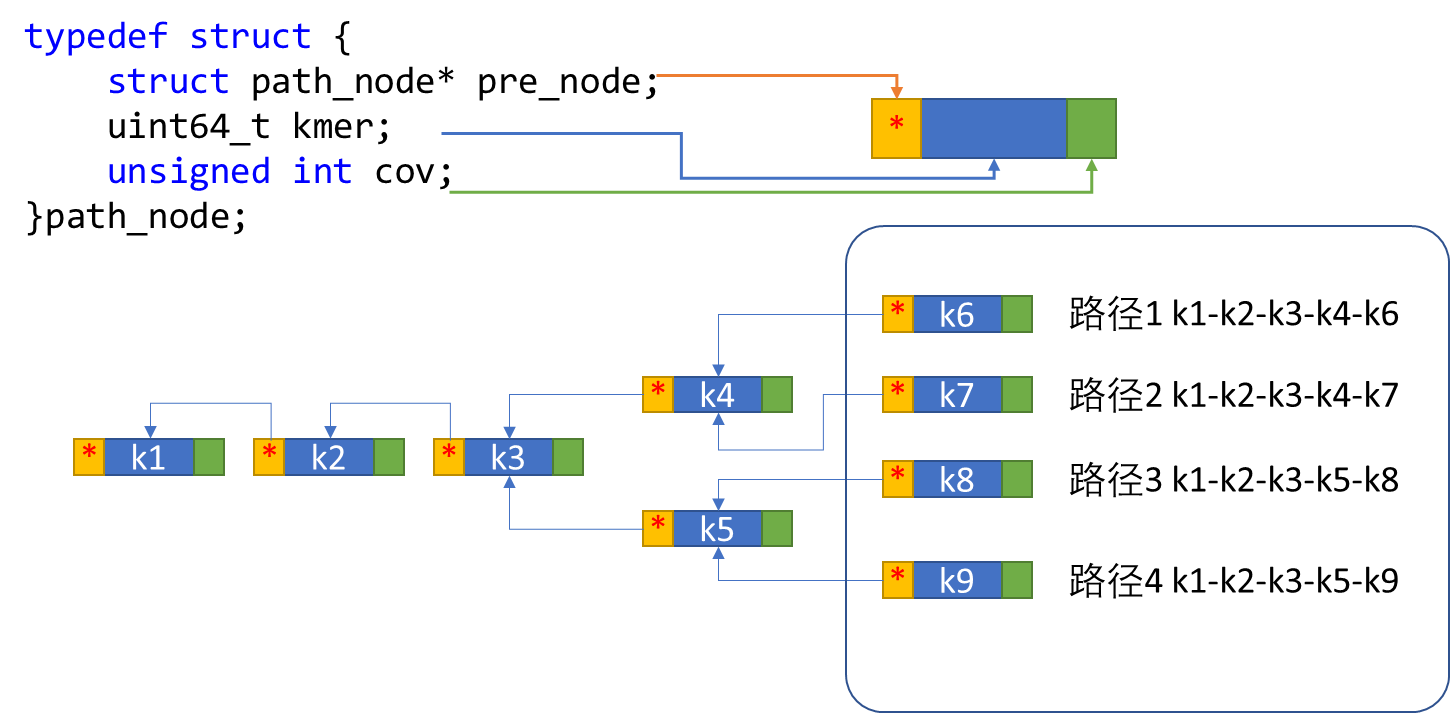


图 5 路径搜索的链表数据结构设计

#### 六. 软件特点

（1）无需read mapping：QuickVar采用一种mapping-free的算法，可以快速鉴定单倍体基因组中的变异位点。

（2）高效准确：QuickVar使用多线程技术和德布莱英图的组装过程，可以快速准确地分析单倍体基因组中的变异信息。

（3）易于使用：QuickVar具有简单易用的界面和命令行操作模式，适用于不同用户的需求。

（4）工业应用：QuickVar适用于工业微生物分析等领域，可以加快工业菌种设计-构建-测试-学习（DBTL）迭代优化的速度。

#### 七、应用场景

QuickVar是一款快速准确的基因组变异分析软件，主要用于工业微生物的分析和基因组工程领域。它能够广泛应用于工业微生物的基因变异分析，尤其是在加速工程生产者的DBTL（设计-构建-测试-学习）周期方面具有潜在的应用价值。例如，代谢工程旨在通过基因调控等手段操纵微生物产生有价值的化合物，如生物燃料和药物。在这一过程中，识别导致目标产物生产增加的基因变异是至关重要的。QuickVar可以为此提供高效和准确的解决方案。同样，合成生物学旨在设计和构建具有所需功能的新生物系统，这也需要基因组变异分析来实现所需的属性。QuickVar可以加速分析过程，协助工程微生物生产者的设计-构建-测试-学习循环。QuickVar的多线程分析策略可以加速分析过程，快速准确地分析工业微生物的基因变异，并优化生产菌株的设计和性能。除了工业微生物，QuickVar还可用于生物信息学领域中的其他基因组变异分析任务。

#### 八、操作系统

QuickVar可在Linux和macOS操作系统上运行，需要安装GCC编译器和zlib库依赖库。

#### 九、总结

QuickVar是一个快速、准确和易于使用的基因变异分析软件。QuickVar采用直接的*k*-mer比较和德布莱英图的组装算法，不需要计算复杂的测序数据 mapping步骤，从而能够快速准确地检测出基因组中的变异信息。同时，它还支持多线程分析，提高了分析速度。QuickVar在工业微生物改造、病原体检测和进化研究等领域都有广泛的应用前景。