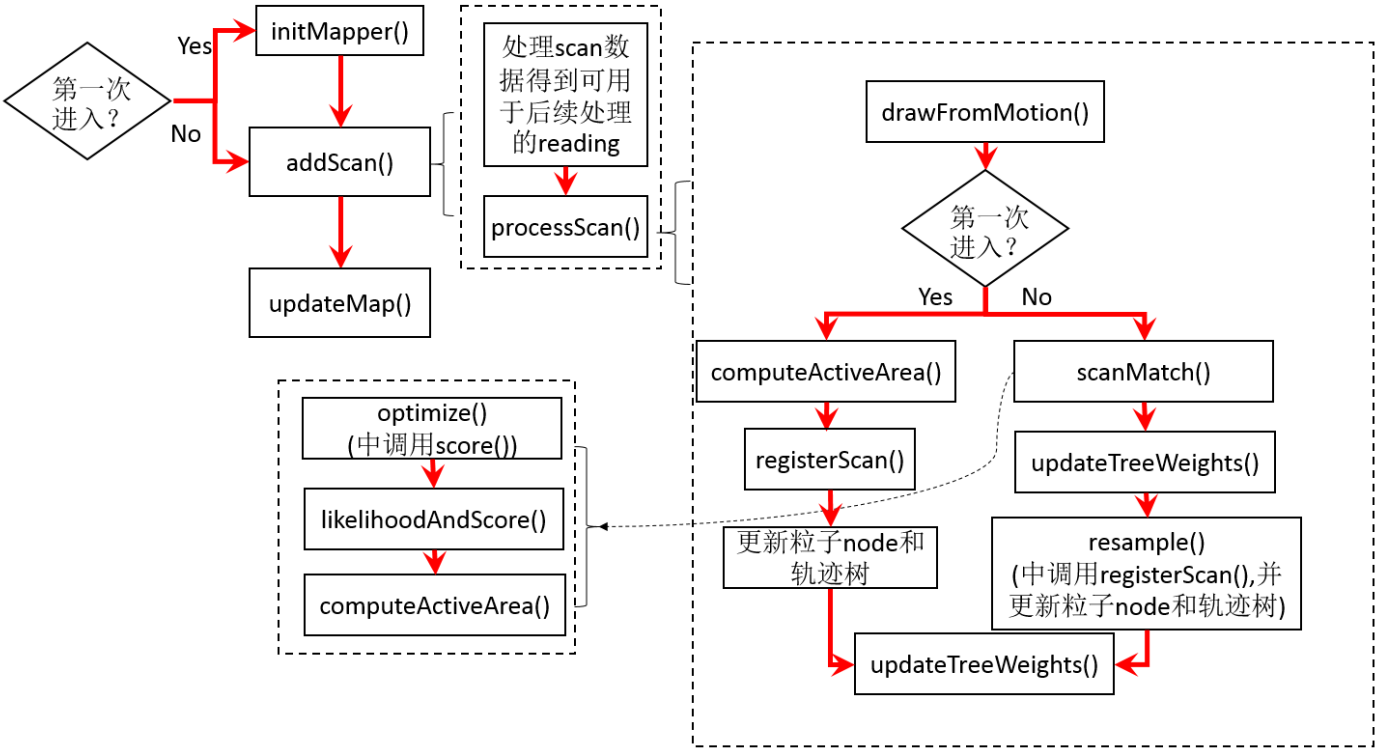
Slam_gmaping 代码阅读笔记：

总体流程说明

slam_gmapping 的主要代码段是“slam_gmapping.cpp”中的 laserCallback()。每收到一次扫描数据就会进入这个函数。laserCallback()代码执行流程大致如下图所示：



在第一次进入 laserCallback()时，会调用 initMapper()对地图进行一次初始化。之后每次进入 laserCallback()，则会首先进入 addScan()，再执行 updateMap().

粒子（Particle）与轨迹树（trajectory tree）及地图的实现：

Gmapping 中粒子的定义如下：

```
struct Particle{
  /**constructs a particle, given a map
     @param map: the particle map
  */
  Particle(const ScanMatcherMap& map);

  /** @returns the weight of a particle */
  inline operator double() const {return weight;}
  /** @returns the pose of a particle */
  inline operator OrientedPoint() const {return pose;}
  /** sets the weight of a particle
      @param w the weight
  */
  inline void setWeight(double w) {weight=w;}
  /** The map */
  ScanMatcherMap map;
  /** The pose of the robot */
  OrientedPoint pose;

  /** The pose of the robot at the previous time frame (used for computing thr odometry displacements) */
  OrientedPoint previousPose;

  /** The weight of the particle */
  double weight;

  /** The cumulative weight of the particle */
  double weightSum;

  double gweight;

  /** The index of the previous particle in the trajectory tree */
  int previousIndex;

  /** Entry to the trajectory tree */
  TNode* node;
};
```

每个粒子均包含一幅地图，粒子的位姿，上一时刻的位姿，粒子的权重，粒子的累加权重，轨迹树中上个粒子的下标，以及轨迹树的入口节点等。

轨迹树的节点由 TNode 定义，包含该节点粒子的位姿 pose，该节点粒子的权重 weight，该节点所有子节点的累加权重 accweight，该节点的父节点指针 parent，此时的激光数据 reading，子节点个数 childs。

```
struct TNode{
  /**Constructs a node of the trajectory tree.
   @param pose:      the pose of the robot in the trajectory
   @param weight:    the weight of the particle at that point in the trajectory
   @param accWeight: the cumulative weight of the particle
   @param parent:    the parent node in the tree
   @param childs:    the number of childs
  */
  TNode(const OrientedPoint& pose, double weight, TNode* parent=0, unsigned int childs=0);

  /**Destroys a tree node, and consistently updates the tree. If a node whose parent has only one child is deleted,
   also the parent node is deleted. This because the parent will not be reacheable anymore in the trajectory tree.*/
  ~TNode();

  /**The pose of the robot*/
  OrientedPoint pose;

  /**The weight of the particle*/
  double weight;

  /**The sum of all the particle weights in the previous part of the trajectory*/
  double accWeight;

  double gweight;


  /**The parent*/
  TNode* parent;

  /**The range reading to which this node is associated*/
  const RangeReading* reading;

  /**The number of childs*/
  unsigned int childs;

  /**counter in visiting the node (internally used)*/
  mutable unsigned int visitCounter;

  /**visit flag (internally used)*/
  mutable bool flag;
};
```

需要注意的是 TNode 的构造函数和析构函数，在构造函数中还实现该节点的父节点的 childs 更新功能，由于每次产生新节点时，都会指定一个轨迹树中已经存

在的节点作为父节点，当存在父节点时，构造函数中自动给父节点的 childs 加1。每次删除轨迹树中一个节点的时候，会自动进入析构函数，这是程序会将父节点的 childs 减一，再判断父节点是否小于等于零，是的话连父节点一起删除。因为轨迹树中的所有节点只能由子节点向父节点反向访问，如果一个节点的唯一子节点被删除，则会导致该节点无法被访问。

```
GridSlamProcessor::TNode::TNode(const OrientedPoint& p, double w, TNode* n, unsigned int c){
        pose=p;
        weight=w;
        childs=c;
        parent=n;
        reading=0;
        gweight=0;
        if (n){
                n->childs++;
        }
        flag=0;
        accWeight=0;
}


GridSlamProcessor::TNode::~TNode(){
        if (parent && (--parent->childs)<=0)
                delete parent;
        assert(!childs);
}
```

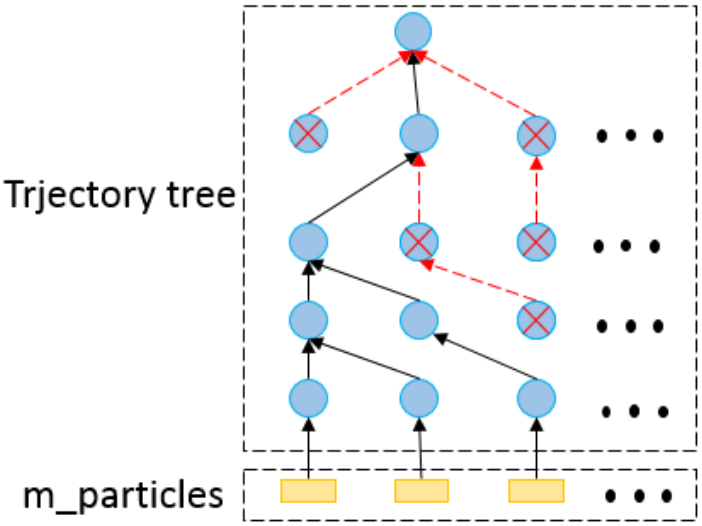
粒子的初始化在 lasercallBack()第一次执行时调用 initMapper()中实现。initMapper()中调用 gsp_->GridSlamProcessor::init()来对粒子进行初始化，粒子容器中总共 30 个粒子，初始位置均为（0,0,0）。m_neff=30.0, m_count=0, m_readingcount=0, m_linearDistance=m_angularDistance=0;
初始的轨迹树入口节点指向一个初始节点，该节点粒子位姿为（0,0,0），节点粒子权重为 0，父节点为 0，子节点数为零。reading、gweight、flag、accWeight 均为零。

```
void GridSlamProcessor::init(unsigned int size, double xmin, double ymin, double xmax, double ymax, double delta, OrientedPoint initialPose){
  m_xmin=xmin;
  m_ymin=ymin;
  m_xmax=xmax;
  m_ymax=ymax;
  m_delta=delta;
  if (m_infoStream)
    m_infoStream
      << " -xmin "<< m_xmin
      << " -xmax "<< m_xmax
      << " -ymin "<< m_ymin
      << " -ymax "<< m_ymax
      << " -delta "<< m_delta
      << " -particles "<< size << endl;


  m_particles.clear();
  TNode* node=new TNode(initialPose, 0, 0, 0);
  ScanMatcherMap lmap(Point(xmin+xmax, ymin+ymax)*.5, xmax-xmin, ymax-ymin, delta);
  for (unsigned int i=0; i<size; i++){
    m_particles.push_back(Particle(lmap));
    m_particles.back().pose=initialPose;
    m_particles.back().previousPose=initialPose;
    m_particles.back().setWeight(0);
    m_particles.back().previousIndex=0;

            // this is not needed
            //             m_particles.back().node=new TNode(initialPose, 0, node, 0);

            // we use the root directly
            m_particles.back().node= node;
  }
  m_neff=(double)size;
  m_count=0;
  m_readingCount=0;
  m_linearDistance=m_angularDistance=0;
}
```

粒子的位姿在每次调用 processScan()时，首先通过 drawFromMotion()更新一次，并在 processScan()调用结束时将上一时刻的位姿改为当前的位姿。

轨迹树更新：

每个粒子中均有一个节点指针指向轨迹树中的一个节点，轨迹树初始节点在初始化时给定，该节点粒子位姿为（0,0,0），节点粒子权重为 0，父节点为 0，所有粒子最开始均指向这个节点。随着程序运行，轨迹树按如下方式更新，在第一次进入 processScan()时，由运动模型随机出每个粒子的位置，然后根据每个粒子的位置建立 30 个（共 30 个粒子）新的节点，每个新节点的父节点为初始节点。之后每次再进入 processScan()时，轨迹树的更新发生在 resample()函数中。在重采样所需要的粒子时，同时会产生新节点,节点的位姿是被选中的粒子，父节点是 m_particle 中被选中粒子所指向的上一代节点，未被选中的粒子所指向的节点会被删除。所有的粒子更新后均指向一个最新一代的节点，每个节点通过 parent 能访问上一代的父节点，最终形成的结构如下图：



具体函数说明

addScan()---->processScan()

addscan()执行时，首先会处理扫描到的激光数据，得到一组叫 reading 的数据供下一步调用。然后函数 processScan()会调用该数据进行进一步处理。

processScan() ----> drawFromMotion()

processScan()处理的第一步是通过运动模型来更新每个粒子的位姿。通过函数 drawFromMotion()来实现。

源代码如下：

```
OrientedPoint
MotionModel::drawFromMotion(const OrientedPoint& p, const OrientedPoint& pnew, const OrientedPoint& pold) const{
        double sxy=0.3*srr;
        OrientedPoint delta=absoluteDifference(pnew, pold);
        OrientedPoint noisypoint(delta);
        noisypoint.x+=sampleGaussian(srr*fabs(delta.x)+str*fabs(delta.theta)+sxy*fabs(delta.y));
        noisypoint.y+=sampleGaussian(srr*fabs(delta.y)+str*fabs(delta.theta)+sxy*fabs(delta.x));
        noisypoint.theta+=sampleGaussian(stt*fabs(delta.theta)+srt*sqrt(delta.x*delta.x+delta.y*delta.y));
        noisypoint.theta=fmod(noisypoint.theta, 2*M_PI);
        if (noisypoint.theta>M_PI)
                noisypoint.theta-=2*M_PI;
        return absoluteSum(p,noisypoint);
}
```

算法注释：

该算法需读取*上一时刻当前粒子的位姿 p，该时刻里程计显示位姿 pnew* 以及*上一时刻里程计所读位姿 pold*。

(1)该函数调用 absoluteDifference()算出根据里程计显示的绝对偏移量，该偏移量被转化到 pold 坐标系。

(2)对该偏移量各个分量（x,y,theta）融入高斯白噪声；注：theta 融入噪声后进行了 fmod 对 2*PI 取余操作，并在取余操作后将 theta 转化到（-PI,PI）

(3)将融合噪声的偏移量叠加到粒子的位姿作为粒子的新位姿。

疑问：该算法认为在 pold 坐标系中的沿 x,y 的平移与旋转量 theta 分别会对在世界坐标系下的 x,y,theta 方向的运动以固定系数产生影响。即噪声与 pold 的位姿无关，只与 pnew 与 pold 的相对位姿有关。然而，当 pold 位姿取不同值时，对噪声的影响理应不同。《概率机器人中》5.4.1 节所提供的另一种算法（ sample_motion_model_odometry)更为准确，之后可尝试用该方法进行实现。不过最终粒子位姿都会通过激光雷达扫描数据进行校准，该算法的准确性应该对结果影响不大。

processScan() ---->computeActiveArea()

通过运动模型更新每个粒子的位姿之后，接下来需要判断程序是否是第一次进入 processScan()。如果是第一次进去，则只要进行**计算有效区域 computeActiveArea()**和**计算占用概率栅格地图 registerScan()**以及**更新粒子 node 和轨迹树**几个步骤。如果不是第一次进入，则需要进行**扫描匹配 scanMatch()**,**权重计算 updateTreeWeights()**以及**重采样 resample()**三个步骤。

computeActiveArea()通过如下方式实现：

源代码如下(使用的是 scanmacher.new.cpp 下的函数)：

```
void ScanMatcher::computeActiveArea(ScanMatcherMap& map, const OrientedPoint& p, const double* readings){
	if (m_activeAreaComputed)
		return;
	HierarchicalArray2D<PointAccumulator>::PointSet activeArea;
	OrientedPoint lp=p;
	lp.x+=cos(p.theta)*m_laserPose.x-sin(p.theta)*m_laserPose.y;
	lp.y+=sin(p.theta)*m_laserPose.x+cos(p.theta)*m_laserPose.y;
	lp.theta+=m_laserPose.theta;
	IntPoint p0=map.world2map(lp);
	const double * angle=m_laserAngles;
	for (const double* r=readings; r<readings+m_laserBeams; r++, angle++)
		if (m_generateMap){
			double d=*r;
			if (d>m_laserMaxRange)
				continue;
			if (d>m_usableRange)
				d=m_usableRange;

			Point phit=lp+Point(d*cos(lp.theta+*angle),d*sin(lp.theta+*angle));
			IntPoint p1=map.world2map(phit);

			d+=map.getDelta();
			//Point phit2=lp+Point(d*cos(lp.theta+*angle),d*sin(lp.theta+*angle));
			//IntPoint p2=map.world2map(phit2);
			IntPoint linePoints[20000] ;
			GridLineTraversalLine line;
			line.points=linePoints;
			//GridLineTraversal::gridLine(p0, p2, &line);
			GridLineTraversal::gridLine(p0, p1, &line);
			for (int i=0; i<line.num_points-1; i++){
				activeArea.insert(map.storage().patchIndexes(linePoints[i]));
			}
			if (d<=m_usableRange){
				activeArea.insert(map.storage().patchIndexes(p1));
				//activeArea.insert(map.storage().patchIndexes(p2));
			}
		} else {
			if (*r>m_laserMaxRange||*r>m_usableRange) continue;
			Point phit=lp;
			phit.x+=*r*cos(lp.theta+*angle);
			phit.y+=*r*sin(lp.theta+*angle);
			IntPoint p1=map.world2map(phit);
			assert(p1.x>=0 && p1.y>=0);
			IntPoint cp=map.storage().patchIndexes(p1);
			assert(cp.x>=0 && cp.y>=0);
			activeArea.insert(cp);

		}
	//this allocates the unallocated cells in the active area of the map
	//cout << "activeArea::size() " << activeArea.size() << endl;
	map.storage().setActiveArea(activeArea, true);
	m_activeAreaComputed=true;
}
```

算法注释：

该算法需读取上一时刻**粒子现有地 map，粒子位姿 p 以及当前激光雷达数据 reading**。

(1)在每次调用 computeActiveArea()之前，都会先调用 invalidateActiveArea()，在其中将 m_activeareaComputed 赋值为 false。computeActiveArea()首先计算出激光雷达在世界坐标系中的位姿，并转化到地图坐标系，记为 p0。

m_laserpose 应该是指激光雷达安装位置相对于机器人坐标系的位姿。

```
lp.x+=cos(p.theta)*m_laserPose.x-sin(p.theta)*m_laserPose.y;
lp.y+=sin(p.theta)*m_laserPose.x+cos(p.theta)*m_laserPose.y;
lp.theta+=m_laserPose.theta;
IntPoint p0=map.world2map(lp);
```

(2)对于每一条激光数据，首先判断 m_generateMap 是否为真，为真，则执行如下操作：

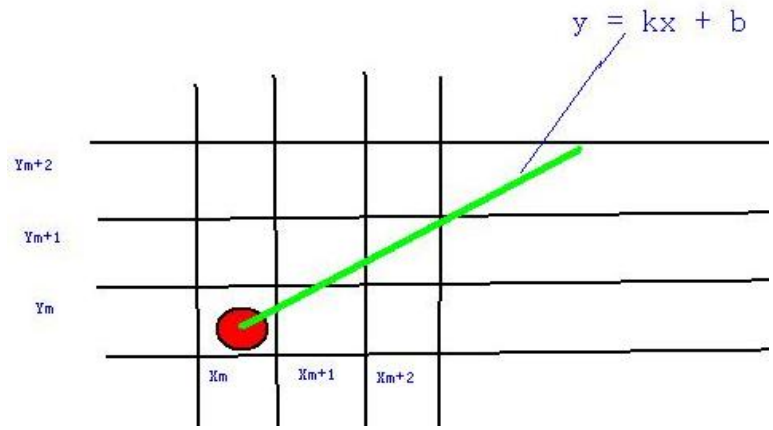
i. 计算出激光探测点在地图中的位置，记为 p1。

```
if (m_generateMap){
        double d=*r;
        if (d>m_laserMaxRange)
                continue;
        if (d>m_usableRange)
                d=m_usableRange;

        Point phit=lp+Point(d*cos(lp.theta+*angle),d*sin(lp.theta+*angle));
        IntPoint p1=map.world2map(phit);
```

ii. 然后调用 gridLine()计算激光束通过的地图上的点，将算出的点标记为 activeArea。

```
d+=map.getDelta();
//Point phit2=lp+Point(d*cos(lp.theta+*angle),d*sin(lp.theta+*angle));
//IntPoint p2=map.world2map(phit2);
IntPoint linePoints[20000] ;
GridLineTraversalLine line;
line.points=linePoints;
//GridLineTraversal::gridLine(p0, p2, &line);
GridLineTraversal::gridLine(p0, p1, &line);
for (int i=0; i<line.num_points-1; i++){
        activeArea.insert(map.storage().patchIndexes(linePoints[i]));
}
if (d<=m_usableRange){
        activeArea.insert(map.storage().patchIndexes(p1));
        //activeArea.insert(map.storage().patchIndexes(p2));
}
```

iii. gridLine()中核心代码在 gridLineCore()中实现。gridLineCore()中核心算法注释如下。



对每一束激光束，设 start 为该激光束起点，end 为激光束端点（障碍物位置）
算法原理如下：
设 start、end 所在直线方程为 y = kx + b，我们以 0<k<1 的情况为例进行推导。记 dy=|end.y-start.y|, dx=|end.x-start.x|。由于斜率小于 1，选择在 x 方向进行步进。我们的目的就是要计算该束激光所经过的所有网格的坐标。假设当前 x = x_m 时，y = y_m。那么当 x 执行一个单位步进时（即 x = x_m +1 时），y 等于 y_m 还是等于 y_m +1 更符合这个直线方程呢？我们可以通过比较 y 方向上两个网格与直线上真实坐标的差值来确定哪个更符合。

Dupper = $y_m + 1 - Yreal = y_m + 1 - (k*(x_m+1)+b)$；表示 y_m +1 和 y 真实值的差

Ddown = Yreal - $y_m = k*(x_m+1)+b-y_m$；表示 y_m 和 y 真实值的差

记 Diff = Ddown - Dupper

$$= (k*(x_m+1)+b-y_m)-(y_m+1-(k*(x_m+1)+b))$$

我们就是要判断 Diff 的正负，如果 Diff 大于零，说明 y_m + 1 更接近真实值，反之则 y_m 更接近。由于 k=dy/dx，在方程两边同乘以 dx，得到

$$p_m = dx*Diff = 2dy*x_m - 2dx*y_m + 2dy + dx*(2b-1)$$

那么
$$p_{m+1} = p_m + 2dy - 2dx \quad (p_m > 0)$$
$$p_{m+1} = p_m + 2dy \quad (p_m < 0)$$

算法循环判断 p_m，如果大于 0，则下一个点的 y 坐标加 1。在第一个点，带入初始点 (x_0, y_0)，及 $b = y_0 - k*x_0$，可得

$$p_0 = 2dy - dx$$

K 取其他范围值时，计算思路与上述过程类似。在|k|>1 时，程序需选取 y 方向执行单位步进。k<0 时，非步进方向每次改变时应该是减一而非加一。
源代码如下：

```
void GridLineTraversal::gridLineCore( IntPoint start, IntPoint end, GridLineTraversalLine *line )
{
  int dx, dy, incr1, incr2, d, x, y, xend, yend, xdirflag, ydirflag;
  int cnt = 0;

  dx = abs(end.x-start.x); dy = abs(end.y-start.y);

  if (dy <= dx) {
    d = 2*dy - dx; incr1 = 2 * dy; incr2 = 2 * (dy - dx);
    if (start.x > end.x) {
      x = end.x; y = end.y;
      ydirflag = (-1);
      xend = start.x;
    } else {
      x = start.x; y = start.y;A
      ydirflag = 1;
      xend = end.x;
    }
    line->points[cnt].x=x;
    line->points[cnt].y=y;
    cnt++;
    if (((end.y - start.y) * ydirflag) > 0) {
      while (x < xend) {
        x++;
        if (d <0) {
          d+=incr1;
        } else {
          y++; d+=incr2;
        }
        line->points[cnt].x=x;
        line->points[cnt].y=y;
        cnt++;
      }
    } else {
      while (x < xend) {
        x++;
        if (d <0) {
          d+=incr1;
        } else {
          y--; d+=incr2;
        }
        line->points[cnt].x=x;
        line->points[cnt].y=y;
        cnt++;

      }
    }
  } else {
    d = 2*dx - dy;
    incr1 = 2*dx; incr2 = 2 * (dx - dy);
    if (start.y > end.y) {
      y = end.y; x = end.x;
      yend = start.y;
      xdirflag = (-1);
    } else {
      y = start.y; x = start.x;
      yend = end.y;
      xdirflag = 1;
    }
    line->points[cnt].x=x;
    line->points[cnt].y=y;
    cnt++;
    if (((end.x - start.x) * xdirflag) > 0) {
      while (y < yend) {
        y++;
        if (d <0) {
          d+=incr1;
        } else {
          x++; d+=incr2;
        }
        line->points[cnt].x=x;
        line->points[cnt].y=y;
        cnt++;
      }
    } else {
      while (y < yend) {
        y++;
        if (d <0) {
          d+=incr1;
        } else {
          x--; d+=incr2;
        }
        line->points[cnt].x=x;
        line->points[cnt].y=y;
        cnt++;
      }
    }
  }
  line->num_points = cnt;
}
```

如果 m_generateMap 非真，则只将探测点标记为 activeArea.

```
if (*r>m_laserMaxRange||*r>m_usableRange) continue;
Point phit=lp;
phit.x+=*r*cos(lp.theta+*angle);
phit.y+=*r*sin(lp.theta+*angle);
IntPoint p1=map.world2map(phit);
assert(p1.x>=0 && p1.y>=0);
IntPoint cp=map.storage().patchIndexes(p1);
assert(cp.x>=0 && cp.y>=0);
activeArea.insert(cp);
```

m_generateMap 在 gsp_地图初始化时被设为 false,扫描匹配时一直是 false。在 updatemap()函数中新定义一个 matcher,其中的 m_generateMap 被置为 true。所以粒子中保存的地图和 updateMap()中输出的地图中包含信息不同。

processScan() ---->registerScan()

registerScan 源码如下：

```
void ScanMatcher::registerScan(ScanMatcherMap& map, const OrientedPoint& p, const double* readings){
	if (!m_activeAreaComputed)
		computeActiveArea(map, p, readings);

	//this operation replicates the cells that will be changed in the registration operation
	map.storage().allocActiveArea();

	OrientedPoint lp=p;
	lp.x+=cos(p.theta)*m_laserPose.x-sin(p.theta)*m_laserPose.y;
	lp.y+=sin(p.theta)*m_laserPose.x+cos(p.theta)*m_laserPose.y;
	lp.theta+=m_laserPose.theta;
	IntPoint p0=map.world2map(lp);
	const double * angle=m_laserAngles;
	for (const double* r=readings; r<readings+m_laserBeams; r++, angle++)
		if (m_generateMap){
			double d=*r;
			if (d>m_laserMaxRange)
				continue;
			if (d>m_usableRange)
				d=m_usableRange;
			Point phit=lp+Point(d*cos(lp.theta+*angle),d*sin(lp.theta+*angle));
			IntPoint p1=map.world2map(phit);

			d+=map.getDelta();
			//Point phit2=lp+Point(d*cos(lp.theta+*angle),d*sin(lp.theta+*angle));
			//IntPoint p2=map.world2map(phit2);
			IntPoint linePoints[20000] ;
			GridLineTraversalLine line;
			line.points=linePoints;
			//GridLineTraversal::gridLine(p0, p2, &line);
			GridLineTraversal::gridLine(p0, p1, &line);
			for (int i=0; i<line.num_points-1; i++){
				map.cell(line.points[i]).update(false, Point(0,0));
			}
			if (d<=m_usableRange){
				map.cell(p1).update(true,phit);
			//	map.cell(p2).update(true,phit);
			}
		} else {
			if (*r>m_laserMaxRange||*r>m_usableRange) continue;
			Point phit=lp;
			phit.x+=*r*cos(lp.theta+*angle);
			phit.y+=*r*sin(lp.theta+*angle);
			map.cell(phit).update(true,phit);
		}
}
```

算法注释：

该算法需读取上一时刻**粒子现有地 map，粒子位姿 p 以及当前激光雷达数据 reading**。

(1)registerScan()函数进入后会首先调用 computeActiveArea()。然后计算出激光雷达在世界坐标系中的位姿，并转化到地图坐标系，算法处理和 computeActiveArea()基本类似。通过调用 update()来对相应的网格特性进行更新。

```
void PointAccumulator::update(bool value, const Point& p){
        if (value) {
                acc.x+= static_cast<float>(p.x);
                acc.y+= static_cast<float>(p.y);
                n++;
                visits+=SIGHT_INC;
        } else
                visits++;
}
```

(2)首先判断 m_generateMap 是否为真，为真，则执行如下操作，将光束经过的网格 visits 全部加一，如果量到的距离在有效量程之内，还需将障碍物所在网格 p1 的性质 acc.x,acc.y,n,visits 做相应变化。每扫到一次障碍物，障碍物在地图上的网格 cell 里的 acc 会将障碍物在世界坐标系中的位置累加进来，并且 n 加一，后面用到位置的时候取均值 mean()const{return1./n*Point(acc.x,acc.y);p1 处的 visits 会被加两次，n 加一次.

```
for (int i=0; i<line.num_points-1; i++){
        map.cell(line.points[i]).update(false, Point(0,0));
}
if (d<=m_usableRange){
        map.cell(p1).update(true,phit);
//      map.cell(p2).update(true,phit);
}
```

如果 m_generateMap 值为 false，则只将障碍物所在网格 phit 的性质 acc.x,acc.y,n,visits 做相应变化。(phit 是在世界坐标系，不在地图坐标系，应该需要转换吧？这个是 scanmatcher.new.cpp 中的实现，在另一处实现 scanmatcher.cpp 中的用的是 p1 而不是 phit,此处应该是错误的。)

```
if (*r>m_laserMaxRange||*r>m_usableRange) continue;
Point phit=lp;
phit.x+=*r*cos(lp.theta+*angle);
phit.y+=*r*sin(lp.theta+*angle);
map.cell(phit).update(true,phit);
```

processScan() ----> scanMatch()

当程序不是第一次进入，则需要进行扫描匹配 scanMatch(),权重计算 updateTreeWeights()以及重采样 resample()三个步骤。scanMach()函数则会依次调用 optimize(),likelihoodAndScore(),computeActiveArea()。

对于每一个粒子 scanMatch()首先定义一个位姿 corrected,然后调用 optimize()得到校正后的粒子位姿并返回 bestscore.

scanMatch()---->optimize()

optimize()源码如下：

```
double ScanMatcher::optimize(OrientedPoint& pnew, const ScanMatcherMap& map, const OrientedPoint& init, const double* readings) const{
        double bestScore=-1;
        OrientedPoint currentPose=init;
        double currentScore=score(map, currentPose, readings);
        double adelta=m_optAngularDelta, ldelta=m_optLinearDelta;
        unsigned int refinement=0;
        enum Move{Front, Back, Left, Right, TurnLeft, TurnRight, Done};
/*      cout << __PRETTY_FUNCTION__<<  " readings: ";
        for (int i=0; i<m_laserBeams; i++){
                cout << readings[i] << " ";
        }
        cout << endl;
*/      int c_iterations=0;
        do{
                if (bestScore>=currentScore){
                        refinement++;
                        adelta*=.5;
                        ldelta*=.5;
                }
                bestScore=currentScore;
//              cout <<"score="<< currentScore << " refinement=" << refinement;
//              cout <<  "pose=" << currentPose.x  << " " << currentPose.y << " " << currentPose.theta << endl;
                OrientedPoint bestLocalPose=currentPose;
                OrientedPoint localPose=currentPose;

                Move move=Front;
                do {
                        localPose=currentPose;
                        switch(move){
                                case Front:
                                        localPose.x+=ldelta;
                                        move=Back;
                                        break;
                                case Back:
                                        localPose.x-=ldelta;
                                        move=Left;
                                        break;
                                case Left:
                                        localPose.y-=ldelta;
                                        move=Right;
                                        break;
                                case Right:
                                        localPose.y+=ldelta;
                                        move=TurnLeft;
                                        break;
                                case TurnLeft:
                                        localPose.theta+=adelta;
                                        move=TurnRight;
                                        break;
                                case TurnRight:
                                        localPose.theta-=adelta;
                                        move=Done;
                                        break;
                                default:;
                        }

                        double odo_gain=1;
                        if (m_angularOdometryReliability>0.){
                                double dth=init.theta-localPose.theta;  dth=atan2(sin(dth), cos(dth));  dth*=dth;
                                odo_gain*=exp(-m_angularOdometryReliability*dth);
                        }
                        if (m_linearOdometryReliability>0.){
                                double dx=init.x-localPose.x;
                                double dy=init.y-localPose.y;
                                double drho=dx*dx+dy*dy;
                                odo_gain*=exp(-m_linearOdometryReliability*drho);
                        }
                        double localScore=odo_gain*score(map, localPose, readings);
                        if (localScore>currentScore){
                                currentScore=localScore;
                                bestLocalPose=localPose;
                        }
                        c_iterations++;
                } while(move!=Done);
                currentPose=bestLocalPose;
//              cout << "currentScore=" << currentScore<< endl;
                //here we look for the best move;
        }while (currentScore>bestScore || refinement<m_optRecursiveIterations);
        //cout << __PRETTY_FUNCTION__ << "bestScore=" << bestScore<< endl;
        //cout << __PRETTY_FUNCTION__ << "iterations=" << c_iterations<< endl;
        pnew=currentPose;
        return bestScore;
}
```

optimize()代码注释：

optimize()首先调用函数 score()计算当前激光雷达采集数据的“概率”。score()实现方式如下：

```
inline double ScanMatcher::score(const ScanMatcherMap& map, const OrientedPoint& p, const double* readings) const{
        double s=0;
        const double * angle=m_laserAngles+m_initialBeamsSkip;
        OrientedPoint lp=p;
        lp.x+=cos(p.theta)*m_laserPose.x-sin(p.theta)*m_laserPose.y;
        lp.y+=sin(p.theta)*m_laserPose.x+cos(p.theta)*m_laserPose.y;
        lp.theta+=m_laserPose.theta;
        unsigned int skip=0;
        double freeDelta=map.getDelta()*m_freeCellRatio;
        for (const double* r=readings+m_initialBeamsSkip; r<readings+m_laserBeams; r++, angle++){
                skip++;
                skip=skip>m_likelihoodSkip?0:skip;
                if (skip||*r>m_usableRange||*r==0.0) continue;
                Point phit=lp;
                phit.x+=*r*cos(lp.theta+*angle);
                phit.y+=*r*sin(lp.theta+*angle);
                IntPoint iphit=map.world2map(phit);
                Point pfree=lp;
                pfree.x+=(*r-map.getDelta()*freeDelta)*cos(lp.theta+*angle);
                pfree.y+=(*r-map.getDelta()*freeDelta)*sin(lp.theta+*angle);
                pfree=pfree-phit;
                IntPoint ipfree=map.world2map(pfree);
                bool found=false;
                Point bestMu(0.,0.);
                for (int xx=-m_kernelSize; xx<=m_kernelSize; xx++)
                for (int yy=-m_kernelSize; yy<=m_kernelSize; yy++){
                        IntPoint pr=iphit+IntPoint(xx,yy);
                        IntPoint pf=pr+ipfree;
                        //AccessibilityState s=map.storage().cellState(pr);
                        //if (s&Inside && s&Allocated){
                                const PointAccumulator& cell=map.cell(pr);
                                const PointAccumulator& fcell=map.cell(pf);
                                if (((double)cell )> m_fullnessThreshold && ((double)fcell )<m_fullnessThreshold){
                                        Point mu=phit-cell.mean();
                                        if (!found){
                                                bestMu=mu;
                                                found=true;
                                        }else
                                                bestMu=(mu*mu)<(bestMu*bestMu)?mu:bestMu;
                                }
                        //}
                }
                if (found)
                        s+=exp(-1./m_gaussianSigma*bestMu*bestMu);
        }
        return s;
}
```

score()首先计算出雷达在世界坐标系下的位姿。

//m_laserAngles 是每一个波束对应角度的数组

//p 是机器人当前位姿（x, y, theta）

//lp 代表雷达位姿，以下计算雷达在世界坐标系下的位姿

//mfreeCellRatio=sqrt(2.)

```
double s=0;
const double * angle=m_laserAngles+m_initialBeamsSkip;
OrientedPoint lp=p;
lp.x+=cos(p.theta)*m_laserPose.x-sin(p.theta)*m_laserPose.y;
lp.y+=sin(p.theta)*m_laserPose.x+cos(p.theta)*m_laserPose.y;
lp.theta+=m_laserPose.theta;
unsigned int skip=0;
double freeDelta=map.getDelta()*m freeCellRatio;
```

对于每一束激光，要计算出波束的探测点位 iphit 置和 ipfree。（ipfree 目的应该是要将激光探测点沿当前方向向里挪一定距离，但目前的这个取值为 $-\sqrt{2}*Delta$）不知道代表什么含义，如果是 $-\sqrt{2}*Delta$ 则比较好理解，是向里

挪一个网格。以下是计算 iphit 和 ipfree 的代码。

```
Point phit=lp;
phit.x+=*r*cos(lp.theta+*angle);
phit.y+=*r*sin(lp.theta+*angle);
IntPoint iphit=map.world2map(phit);
Point pfree=lp;
pfree.x+=(*r-map.getDelta()*freeDelta)*cos(lp.theta+*angle);
pfree.y+=(*r-map.getDelta()*freeDelta)*sin(lp.theta+*angle);
pfree=pfree-phit;
IntPoint ipfree=map.world2map(pfree);
bool found=false;
Point bestMu(0.,0.);
```

然后在以 iphit 为中心的九宫格中遍历，当一个网格满足以下条件：该网格被标记为障碍物（ ((double)cell）>m_fullnessThreshold ）且该网格沿激光方向（ ((double)fcell）< m_fullnessThreshold ）往回走一段距离的网格没有被标记为障碍物时，满足上述条件并且离 phit 最近的点则被认为是真实障碍物的坐标点。(注：和当前障碍物坐标距离最近的以前(double)cell 在被访问过的网格返回值是 n/visits,未访问过的网格返回值是-1.）然后计算出该真实点和 phit 的距离，“概率”计算类似于高斯分布公式为 $p=e^{(-\frac{d^2}{0.05})}$ 。（注：m_gausianSigma=0.05）然后将每束激光的概率叠加得到该扫描的“概率”。当在九宫格中没有找到符合条件的点时，该束激光被略过，其“概率”不被叠加到总的概率中。在该算法中,只有当扫描到的点附近有之前被标记的障碍物时才会计算其概率，在激光扫描到未探索区域时，概率就会一直为零。

```
for (int xx=-m_kernelSize; xx<=m_kernelSize; xx++)
for (int yy=-m_kernelSize; yy<=m_kernelSize; yy++){
        IntPoint pr=iphit+IntPoint(xx,yy);
        IntPoint pf=pr+ipfree;
        //AccessibilityState s=map.storage().cellState(pr);
        //if (s&Inside && s&Allocated){
                const PointAccumulator& cell=map.cell(pr);
                const PointAccumulator& fcell=map.cell(pf);
                if (((double)cell )> m_fullnessThreshold && ((double)fcell )<m_fullnessThreshold){
                        Point mu=phit-cell.mean();
                        if (!found){
                                bestMu=mu;
                                found=true;
                        }else
                                bestMu=(mu*mu)<(bestMu*bestMu)?mu:bestMu;
                }
        //}
}
if (found)
        s+=exp(-1./m_gaussianSigma*bestMu*bestMu);
```

在通过 score()计算完当前采集数据的“概率”后，optimize()通过如下方式确定该粒子的最优位置。通过让粒子前后左右平移以及左转右转来寻找附近使采集数据概率最大化的位置。最终得到粒子的最优位置 pnew 并返回采集数据的最优(大)“概率”bestscore.

scanMatch()---->likelihoodAndScore ()

调用 optimize()可得到粒子通过扫描数据校正后最优位置 corrected 以及该位置扫描数据的“概率”打分 score。当 score 大于所规定的最小可信概率时，便将粒子

的位置更新为 corrected，否则仍采用原来里程计所给出的位置。实现如下：

```
if (score>m_minimumScore){
  it->pose=corrected;
} else {
    if (m_infoStream){
      m_infoStream << "Scan Matching Failed, using odometry. Likelihood=" << l <<std::endl;
      m_infoStream << "lp:" << m_lastPartPose.x << " "  << m_lastPartPose.y << " "<< m_lastPartPose.theta <<std::endl;
      m_infoStream << "op:" << m_odoPose.x << " " << m_odoPose.y << " "<< m_odoPose.theta <<std::endl;
    }
}
```

然后程序调用 likelihoodAndScore ()，likelihoodAndScore 实现原理和 score 基本一致，后面有些许不同。在 score()中，当在九宫格中没有找到符合条件的点时，该束激光被略过，其“概率”不被叠加到总的概率中。而在 likelihoodAndScore()中，定义了两个变量 s 和 l，s 与 score()中的 s 相同，l 的计算中则是在九宫格中没有找到符合条件的点时，该束激光的概率被认为是一个常值 noHit。

```
if (found){
        s+=exp(-1./m_gaussianSigma*bestMu*bestMu);
        c++;
}
if (!skip){
        double f=(-1./m_likelihoodSigma)*(bestMu*bestMu);
        l+=(found)?f:noHit;
}
```

调用完 likelihoodAndScore()后，l 被赋给粒子的 weight。scanMatch()最终调用 computeActiveArea()更新地图有效区域。

```
m_matcher.likelihoodAndScore(s, l, it->map, it->pose, plainReading);
sumScore+=score;
it->weight+=l;
it->weightSum+=l;

//set up the selective copy of the active area
//by detaching the areas that will be updated
m_matcher.invalidateActiveArea();
m_matcher.computeActiveArea(it->map, it->pose, plainReading);
```

processScan() ---->updateTreeWeights ()

```
void  GridSlamProcessor::updateTreeWeights(bool weightsAlreadyNormalized){

  if (!weightsAlreadyNormalized) {
    normalize();
  }
  resetTree();
  propagateWeights();
}
```

updateTreeWeights() 分三个部分：normalize()，resetTree() 和 propagateWeights()。

updateTreeWeights ()---->normalize()

normalize()首先寻找出各个粒子的最大 weight,记为 lmax，然后进行如下转换

$m\_weight = e^{-(weight-l\max)}; w = \frac{m\_weight}{sum(m\_weight)}$，将每个粒子的权重归一化。并计算出 m_neff 来确定是否应该进行重采样。

```
inline void GridSlamProcessor::normalize(){
  //normalize the log m_weights
  double gain=1./(m_obsSigmaGain*m_particles.size());//m_obsSigmaGain=1
  double lmax= -std::numeric_limits<double>::max(); //-1.7976e+308
  for (ParticleVector::iterator it=m_particles.begin(); it!=m_particles.end(); it++){
    lmax=it->weight>lmax?it->weight:lmax;
  }
  //cout << "!!!!!!!!!!! maxwaight= "<< lmax << endl;

  m_weights.clear();
  double wcum=0;
  m_neff=0;
  for (std::vector<Particle>::iterator it=m_particles.begin(); it!=m_particles.end(); it++){
    m_weights.push_back(exp(gain*(it->weight-lmax)));
    wcum+=m_weights.back();
    //cout << "l=" << it->weight<< endl;
  }

  m_neff=0;
  for (std::vector<double>::iterator it=m_weights.begin(); it!=m_weights.end(); it++){
    *it=*it/wcum;
    double w=*it;
    m_neff+=w*w;
  }
  m_neff=1./m_neff;

}
```

resetTree()将轨迹树中所有的节点的 accweight 和 visitCounter 全部赋为零。

```
void GridSlamProcessor::resetTree(){
  // don't calls this function directly, use updateTreeWeights(..) !

        for (ParticleVector::iterator it=m_particles.begin(); it!=m_particles.end(); it++){
                TNode* n=it->node;
                while (n){
                        n->accWeight=0;
                        n->visitCounter=0;
                        n=n->parent;
                }
        }
}
```

propagateWeights()中更新轨迹树中的 accweight 和 visitCounter，accweight 是指该节点所有子节点的权重的累加值，visitCounter 是为了更新 accweight 时轨迹树向上遍历是内部使用，visitCounter 等于 childs 是表示所有的子节点权重均已累加，就会向上遍历。

```
double propagateWeight(GridSlamProcessor::TNode* n, double weight){
        if (!n)
                return weight;
        double w=0;
        n-> ++;
        n->accWeight+=weight;
        if (n->visitCounter==n->childs){
                w=propagateWeight(n->parent,n->accWeight);
        }
        assert(n->visitCounter<=n->childs);
        return w;
}

double GridSlamProcessor::propagateWeights(){
  // don't calls this function directly, use updateTreeWeights(..) !

        // all nodes must be resetted to zero and weights normalized

        // the accumulated weight of the root
        double lastNodeWeight=0;
        // sum of the weights in the leafs
        double aw=0;

        std::vector<double>::iterator w=m_weights.begin();
        for (ParticleVector::iterator it=m_particles.begin(); it!=m_particles.end(); it++){
                double weight=*w;
                aw+=weight;
                TNode * n=it->node;
                n->accWeight=weight;
                lastNodeWeight+=propagateWeight(n->parent,n->accWeight);
                w++;
        }

        if (fabs(aw-1.0) > 0.0001 || fabs(lastNodeWeight-1.0) > 0.0001) {
          cerr << "ERROR: ";
          cerr << "root->accWeight=" << lastNodeWeight << "    sum_leaf_weights=" << aw << endl;
          assert(0);
        }
        return lastNodeWeight;
}
```

processScan() ----> resample()

代码说明：

首先 resample()定义变量 hasResampled 并定义为 false，这个变量作为函数返回值，在进行重采样之后会被赋为 true。然后函数将所有当前粒子存入 oldGeneration 中。

```
bool hasResampled = false;

TNodeVector oldGeneration;
for (unsigned int i=0; i<m_particles.size(); i++){
  oldGeneration.push_back(m_particles[i].node);
}
```

然后程序利用 m_neff 判断是否应该进行重采样，m_neff 是在 normalize()中求得的。这个判断是用来防止好的粒子被替换，具体原理参见[1,2]。在本程序实现中，当 m_neff < 0.5*m_particles.size()时，就进行重采样。

```
if (m_neff<m_resampleThreshold*m_particles.size()){
```

重采样的实现方法在函数 resampleIndexes()中实现。

```
template <class Particle, class Numeric>
std::vector<unsigned int> uniform_resampler<Particle, Numeric>:: resampleIndexes(const std::vector<Particle>& particles, int nparticles) const{
        Numeric cweight=0;

        //compute the cumulative weights
        unsigned int n=0;
        for (typename std::vector<Particle>::const_iterator it=particles.begin(); it!=particles.end(); ++it){
                cweight+=(Numeric)*it;
                n++;
        }

        if (nparticles>0)
                n=nparticles;

        //compute the interval
        Numeric interval=cweight/n;

        //compute the initial target weight
        Numeric target=interval*::drand48();
        //compute the resampled indexes

        cweight=0;
        std::vector<unsigned int> indexes(n);
        n=0;
        unsigned int i=0;
        for (typename std::vector<Particle>::const_iterator it=particles.begin(); it!=particles.end(); ++it, ++i){
                cweight+=(Numeric)* it;
                while(cweight>target){
                        indexes[n++]=i;
                        target+=interval;
                }
        }
        return indexes;
}
```

算法思路如下，先算出所有粒子权重的累加值记为 cweight,再用 cweight 除以粒子数 n 得到粒子权重的平均值 interval。之后随即产生一个 0 到 1 之间的随机数再乘以 interval,记为 target，target 的取值则是在 0 到 interval 之间。进行 n 次抽签，定义容器 indexes 来存储被抽中的粒子下标。方法如下：

1. 将 cweight 置零
2. 进行如下循环：则得到的 indexes 为被抽中的粒子下标

```
for (typename std::vector<Particle>::const_iterator it=particles.begin(); it!=particles.end(); ++it, ++i){
        cweight+=(Numeric)* it;
        while(cweight>target){
                indexes[n++]=i;
                target+=interval;
        }
```

这个方法可以举例说明：假设总共三个粒子，权重分别为 2.5,0.2,0.3。权重平均值为 1，这时，我们产生的随机目标 target 在 0 到 1 之间随机分布。按照算法，indexes[1]=0,indexes[2]=0，若 0<target<0.5，则 indexes[3]=0 。0.5<target<0.7 时，indexes[3]=1。0.7<target<1 时，indexes[3]=2。该算法保证如果一个粒子权重大于 1 倍粒子权重均值时，至少被抽中一次，大于 2 倍均值时，至少被抽中 2 次，以此类推。小于 1 倍均值，最多被抽中 1 次，小于 2 倍均值，最多被抽中 2 次，以此类推。

最终被抽中的粒子下标最终被存储到 indexes 中，通过一下算法找出需要删除的粒子的下标，存储在 deletedParticles 中,并将所有新选中的粒子存储在 temp 中。同时会产生新节点,节点的位姿是被选中的粒子，父节点是 m_particle 中被选中粒子所指向的上一代节点，然后 temp 中的新的粒子的 node 指针会指向新产生的节点。

```
for (unsigned int i=0; i<m_indexes.size(); i++){
  //                          cerr << " " << m_indexes[i];
  while(j<m_indexes[i]){
    deletedParticles.push_back(j);
    j++;
                    }
  if (j==m_indexes[i])
    j++;
  Particle & p=m_particles[m_indexes[i]];
  TNode* node=0;
  TNode* oldNode=oldGeneration[m_indexes[i]];
  //                          cerr << i << "->" << m_indexe
  node=new  TNode(p.pose, 0, oldNode, 0);
  //node->reading=0;
  node->reading=reading;
  //                          cerr << "A("<<node->parent->c

  temp.push_back(p);
  temp.back().node=node;
  temp.back().previousIndex=m_indexes[i];
}
while(j<m_indexes.size()){
  deletedParticles.push_back(j);
  j++;
}
```

最后是更新 m_particles 中存储的粒子，首先对于被淘汰的 m_particle 中的粒子，删除其对应的轨迹树上的节点。然后清空 m_particle,temp 中存储的新粒子存入 m_particle 中，并将每个粒子的 weight 重置为 0。

```
//          cerr << endl;
std::cerr <<  "Deleting Nodes:";
for (unsigned int i=0; i<deletedParticles.size(); i++){
  std::cerr <<" " << deletedParticles[i];
  delete m_particles[deletedParticles[i]].node;
  m_particles[deletedParticles[i]].node=0;
}
std::cerr  << " Done" <<std::endl;

//END: BUILDING TREE
std::cerr << "Deleting old particles..." ;
m_particles.clear();
std::cerr << "Done" << std::endl;
std::cerr << "Copying Particles and  Registering  scans...";
for (ParticleVector::iterator it=temp.begin(); it!=temp.end(); it++){
  it->setWeight(0);
  m_matcher.invalidateActiveArea();
  m_matcher.registerScan(it->map, it->pose, plainReading);
  m_particles.push_back(*it);
}
```

updateMap()

updateMap()在 slam_gmapping.cpp 中实现，首先程序调用找出历史累加权重最大的粒子，记为 best。

```
GMapping::GridSlamProcessor::Particle best =
        gsp_->getParticles()[gsp_->getBestParticleIndex()];//找出累加权重最大的粒子
```

然后找到这个累加权重最大的粒子所指向的轨迹树的节点，并沿轨迹树向上遍历，并根据每个节点的位置与激光数据更新一幅新定义的地图 smap。粒子中本身存储了一幅地图，但是 gsp_中的 m_generateMap 是 false,所以粒子更新地图调用

computeActiveArea()和 registerScan()时只更新障碍物所在网格。而updateMap()中重新定义了一个 matcher 并将其 m_generateMap 置为 true,所以更新 smap 时会将障碍物和非障碍物区域一起更新。

```
GMapping::ScanMatcherMap smap(center, xmin_, ymin_, xmax_, ymax_,
                              delta_);

ROS_DEBUG("Trajectory tree:");
for(GMapping::GridSlamProcessor::TNode* n = best.node;
    n;
    n = n->parent)
{
  ROS_DEBUG("  %.3f %.3f %.3f",
            n->pose.x,
            n->pose.y,
            n->pose.theta);
  if(!n->reading)
  {
    ROS_DEBUG("Reading is NULL");
    continue;
  }
  matcher.invalidateActiveArea();
  matcher.computeActiveArea(smap, n->pose, &((*n->reading)[0]));
  matcher.registerScan(smap, n->pose, &((*n->reading)[0]));
}
```

最终输出的地图只区分地图上每个网格是一下哪种情况：unknow vs.obstacle vs.free。对于之前定义的地图 smap，遍历地图上的每一个网格，通过调用“doubleocc=smap.cell(p);”（返回值应该是 PointAccumulator 中的inlineoperatordouble()const{returnvisits?(double)n*SIGHT_INC/(double)visits:-1;}）得到每个网格的占有率，占有率小于零代表未被访问，输出的地图中相应网格值被赋为-1，占有率大于 occ_tresh 小于 1 零代表障碍，输出的地图中相应网格值被赋为 100，占有率小于 occ_tresh 代表无障碍，输出的地图中相应网格值被赋为 0。

```
for(int x=0; x < smap.getMapSizeX(); x++)
{
  for(int y=0; y < smap.getMapSizeY(); y++)
  {
    /// @todo Sort out the unknown vs. free vs. obstacle thresholding
    GMapping::IntPoint p(x, y);
    double occ=smap.cell(p);
    assert(occ <= 1.0);
    if(occ < 0)
      map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = -1;
    else if(occ > occ_thresh_)
    {
      //map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = (int)round(occ*100.0);
      map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = 100;
    }
    else
      map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = 0;
  }
}
```

参考文献

[1]Grisetti G, Stachniss C, Burgard W. Improved techniques for grid mapping with rao-blackwellized particle filters[J]. IEEE transactions on Robotics, 2007, 23(1): 34-46.

[2] A. Doucet, N. de Freitas, and N. Gordan, editors. Sequential Monte-Carlo Methods in Practice. Springer Verlag, 2001