

```
using gtsam::symbol_shorthand::X; // Pose3 (x,y,z,r,p,y)
namespace symbol_shorthand {
    inline Key A(std::uint64_t j) { return Symbol('a', j); }
    inline Key B(std::uint64_t j) { return Symbol('b', j); }
    inline Key C(std::uint64_t j) { return Symbol('c', j); }
    inline Key D(std::uint64_t j) { return Symbol('d', j); }
    inline Key E(std::uint64_t j) { return Symbol('e', j); }
    inline Key F(std::uint64_t j) { return Symbol('f', j); }
    inline Key G(std::uint64_t j) { return Symbol('g', j); }
    inline Key H(std::uint64_t j) { return Symbol('h', j); }
    inline Key I(std::uint64_t j) { return Symbol('i', j); }
    inline Key J(std::uint64_t j) { return Symbol('j', j); }
    inline Key K(std::uint64_t j) { return Symbol('k', j); }
    inline Key L(std::uint64_t j) { return Symbol('l', j); }
    inline Key M(std::uint64_t j) { return Symbol('m', j); }
    inline Key N(std::uint64_t j) { return Symbol('n', j); }
    inline Key O(std::uint64_t j) { return Symbol('o', j); }
    inline Key P(std::uint64_t j) { return Symbol('p', j); }
    inline Key Q(std::uint64_t j) { return Symbol('q', j); }
    inline Key R(std::uint64_t j) { return Symbol('r', j); }
    inline Key S(std::uint64_t j) { return Symbol('s', j); }
    inline Key T(std::uint64_t j) { return Symbol('t', j); }
    inline Key U(std::uint64_t j) { return Symbol('u', j); }
    inline Key V(std::uint64_t j) { return Symbol('v', j); }
    inline Key W(std::uint64_t j) { return Symbol('w', j); }
    inline Key X(std::uint64_t j) { return Symbol('x', j); }
    inline Key Y(std::uint64_t j) { return Symbol('y', j); }
    inline Key Z(std::uint64_t j) { return Symbol('z', j); }
}
```

这个x帮我理解下



下面这段代码的核心作用，是给 **GTSAM** 的 `Symbol` 生成一组 **简短好用的辅助函数**，让你能够用类似 `X(0)`、`A(1)` 这种更清晰的方式创建 **Key**（每个因子图变量的唯一 ID）。



## 重点解释：这个 x 是什么？

你问的是：

cpp

 Copy code

```
using gtsam::symbol_shorthand::X; // Pose3 (x, y, z, r, p, y)
```

但下面你又定义了另一组 shorthand 函数，其中包括：

cpp

 Copy code

```
inline Key X(std::uint64_t j) { return Symbol('x', j); }
```

所以这里实际上出现了两个 X：

---



## 1. GTSAM 自带的 X — 大写 'X'，表示 Pose3 变量

在 gtsam/symbol\_shorthand.h 中，官方定义了：

cpp

 Copy code

```
inline Key X(size_t j) { return Symbol('x', j); } // Pose3
```

它的含义是：

- 用于表示 机器人位姿 (Pose3) 的键
- 使用字符 'x' 做前缀
- 所以  $x(\theta)$  表示  $\text{Symbol}('x', \theta)$ ，例如机器人第 0 个 pose

GTSAM 里面默认用 X 表示 Pose：

- $X_0 \rightarrow$  第0个位姿
- $X_1 \rightarrow$  第1个位姿
- ...

注释里的  $(x, y, z, r, p, y)$  是说 Pose3 的 DOF (位置 + 欧拉角)。

---



## 2. 你自定义的 X — 和 GTSAM 的 X 完全一样

你自己的 namespace 里写了：

cpp

 Copy code

```
inline Key X(std::uint64_t j) { return Symbol('x', j); }
```

这和官方的功能 **完全一样**。

也就是说：

- 你正在 **重新定义一套完整的字母 shorthand**
- 但你的 X 会和 GTSAM 自带的 X 冲突（名字一样）

如果你写：

cpp

Copy code

```
using gtsam::symbol_shorthand::X;
```

同时又定义了自己的：

cpp

Copy code

```
inline Key X(...)
```

那么根据 C++ 规则，这会导致 **名字冲突 / 屏蔽 (shadowing)**。

---



## 那应该怎么理解你的 x ?

你定义的 X(j) 本质上就是：

生成一个以 'x' 字符开头、索引为 j 的变量 Key。

也就是说：

cpp

Copy code

```
X(0) → Key("x0")
X(15) → Key("x15")
```

这个 key 用于标识因子图中的变量。

无论你是：

- 表示 Pose,

- 表示 Landmark，
- 表示某种自定义变量，

本质上都是一样的，只是你使用 'x' 字母作为分类。

## 总结

名称	来源	前缀字符	用途（约定）
gtsam::symbol_shorth and::X(j)	GTSAM 自带	'x'	Pose3
你自定义的 x(j)	你的 namespace	'x'	任意用途（但会和官方 X 冲突）

## 建议

如果你想扩展 shorthand，而不和 GTSAM 的冲突：

### 方法 1：换个 namespace

cpp

 Copy code

```
namespace my_symbols {
    inline Key X(std::uint64_t j) { return Symbol('x', j); }
}
```

然后用：

cpp

 Copy code

```
my_symbols::X(0)
```

### 方法 2：不要重新定义官方已经定义的字母

GTSAM 官方已提供：

- X, V, B, L 等常用 shorthand

只为未覆盖的字母添加新的即可。

## 如果你愿意，我可以帮你：

- ✓ 重写这套 shorthand，使其不会冲突
- ✓ 解释每个字母在 SLAM 中常见的语义
- ✓ 给出最佳实践命名方式

要继续吗？



using gtsam::symbol\_shorthand::X;这个用法我比理解，类似于python的导入包吗，x可以直接用？

是的，你理解得非常准确！

using gtsam::symbol\_shorthand::X; 在 C++ 中的作用和 Python 的 from xxx import x 几乎一样。

下面给你一个非常清晰的类比 👇

✓ using gtsam::symbol\_shorthand::X; 相当于 Python 的：

python

Copy code

```
from gtsam.symbol_shorthand import X
```

作用：

- 允许你直接写 `X(θ)`
- 而不用写全名 `gtsam::symbol_shorthand::X(θ)`

## ✳️ 举例说明 (C++)

cpp

 Copy code

```
using gtsam::symbol_shorthand::X;  
  
Key k = X(0); // 直接用, 不需要加 namespace
```

如果没有 `using`，你需要写：

cpp

 Copy code

```
Key k = gtsam::symbol_shorthand::X(0);
```

## ✿ 对应的 Python 用法

python

 Copy code

```
from gtsam.symbol_shorthand import X  
  
k = X(0) # 同样可以直接用
```

## 📌 总结（一句话）

`using gtsam::symbol_shorthand::X;` 就是把名字 `X` 引入当前作用域，之后可以像普通函数一样直接用。

## 如果你想，我还能进一步解释：

- `using namespace` vs `using something::name`
- C++ 的名字查找规则（为什么会冲突）
- 为什么 GTSAM 推荐只 `using` 少数字母而不是整个 `namespace`

想继续吗？

⏪ ⏹ ...

## using namespace vs using something::name

下面给你一个清晰、不啰嗦、非常实用的解释，帮助你真正理解：

### ⭐ 总对比：

写法	作用范围	安全性	常用场景
<b>using namespace</b> <b>XXX;</b>	引入整个命名空间	✗ 最危险，容易冲突	只在 .cpp 内的小范围用
<b>using XXX::name;</b>	只引入指定名字	✓ 安全稳定	推荐在头文件和库接口里使用

### 🔍 1. using namespace xxx；

**把整个命名空间包含进来**

例如：

cpp

Copy code

```
using namespace gtsam;
```

效果：

- gtsam 里所有类、函数全部**直接可见**
- 比如 Symbol、Pose3、Point3、Values 等统统都进来了

### ❗ 缺点（非常严重）：

- 名字冲突风险极大
- 尤其是在头文件 .h 中使用，会把污染扩散到所有 include 它的文件
- 大型库（STL、Eigen、GTSAM）中非常容易重名

### ✗ 一般不允许在头文件中使用

## 🔍 2. using something::name;

只引入一个名字，非常安全

例如：

cpp

Copy code

```
using gtsam::symbol_shorthand::X;
```

效果：

- 只引入 X
- 不能访问 symbol\_shorthand 中的其他东西
- 不会和别的命名空间大规模冲突

✓ 优点：

- 精准控制
- 安全、可维护
- 最推荐的方式（尤其在头文件中）

---

## 试管 例子对比

### ✗ using namespace (危险示例)

cpp

Copy code

```
using namespace std;
using namespace gtsam;

vector<int> v;    // 来自 std
Symbol s;          // 来自 gtsam
```

如果将来某个库也定义了 vector 或 Symbol，你就惨了。

---

✓ using name (推荐示例)

cpp

Copy code

```
using std::vector;
using gtsam::Symbol;
```

即使其他库也有 `vector`，你引入的是 `std::vector` 就不会冲突。

## 小总结（非常实用）

场景	推荐写法	原因
头文件 .h	✗ 不要用 <code>using namespace</code> ✓ 可以用 <code>using ns::name</code>	避免污染所有包含它的文件
实现文件 .cpp	✓ 可以用 <code>using namespace xxx</code> (适量)	让代码更简洁，影响范围只在当前 .cpp
大型库（如 GTSAM、Eigen）	强烈推荐 <code>using ns::name</code>	避免与其他库的类型冲突

## 最后一句话总结

`using namespace` 导入整个库，方便但危险；  
`using ns::name` 仅导入单个符号，最安全、最推荐。

如果你愿意，我还能解释：

- C++ 名字查找 (ADL) 为何让 `using namespace` 更危险
- 为什么标准库 (STL) 强烈禁止在 header 中 `using namespace std`
- 如何在 GTSAM 中优雅地使用 shorthand (最佳实践)

要继续吗？



```
inline Key X(size_t j) { return Symbol('x', j); } // Pose3帮我  
展开下，为什么还有return,这是宏定义吗
```

下面我来一步一步拆开解释，保证你完全理解：

---

## 🌟 这不是宏定义！不是宏！

这是一个普通的 C++ `inline` 函数。

绝对不是：

cpp

Copy code

```
#define X(j) ...
```

而是一个真正的函数，只是写得很短。

---

## 🔍 原代码：

cpp

Copy code

```
inline Key X(size_t j) { return Symbol('x', j); }
```

## ✅ 逐字展开解释：

### ① `inline`

告诉编译器：

这个函数非常小，希望编译器把它直接展开，不要真的生成函数调用。

比如：

cpp

```
Key k = X(0);
```

编译器可能会替换成：

cpp

Copy code

```
Key k = Symbol('x', 0);
```

作用：

- 加快速度
  - 避免函数调用开销
  - 可以写在头文件里而不产生重复定义
- 

## ② Key

表示函数的返回类型（GTSAM 中就是一个 uint64\_t 类型的包装）。

也就是：

scss

Copy code

X(j) 返回一个 Key

---

## ③ X(size\_t j)

这是一个函数名叫 X，带一个参数 j。

意思是：

用字符 'x' 和索引 j 构造一个变量 Key

---

## ④ { return Symbol('x', j); }

这一行：

- 调用 GTSAM 的 `Symbol` 构造出一个 key
- 然后返回这个 key

`Symbol('x', j)` 本质上就是构造一个：

arduino

 Copy code

'`x`' 作为前缀，`j` 作为编号的 Key

例如：

cpp

 Copy code

```
X(0) → Key("x0")
X(1) → Key("x1")
```

## 🔍 为什么需要 `return` ?

因为这是一个普通 C++ 函数，它必须 `return` 一个值。

`Symbol('x', j)` 会创建一个 Key (实质是 `uint64_t`)，所以你必须返回它：

cpp

 Copy code

```
return Symbol('x', j);
```

这和其他普通函数写法完全一样，例如：

cpp

 Copy code

```
int add(int a, int b) {
    return a + b;
}
```

一样是：函数计算一个值 → `return` 给调用者。

## ❗ 为什么你会误以为是宏？

因为 GTSAM 的 shorthand 用法像这样：

cpp

 Copy code

```
Key k = X(0);
```

看起来很像宏：

cpp

 Copy code

```
#define X(n) ...
```

但是实际上，它就是一个 **inline 函数**，只是写得非常简洁。

---



## 把函数完全展开给你看看

原作者写得很短，我们来写成常规长版本：

cpp

 Copy code

```
// 这是完全展开后的完整版
inline Key X(size_t j) {
    // 用字符 'x' 和索引 j 构造一个 Symbol
    Key key = Symbol('x', j);

    // 把这个 Key 返回给调用者
    return key;
}
```

你会发现：

这就是一个最普通的 C++ 函数，没有任何宏的痕迹。

---



## 一句话总结

- `inline Key X(size_t j)` 是一个普通 C++ 函数
- 它返回一个 GTSAM 的 `Key`
- 不是宏定义
- `return` 是正常函数的返回值
- 作用是创建 '`x`' + `j` 这样命名的图变量

如果你想，我还能进一步解释：

- 为什么用 'x' 而不是 "x"
- Symbol 是如何把字符和数字打包成一个 64-bit Key 的
- 为什么 shorthand 会极大提升 SLAM 代码可读性

需要继续吗？



流程：

1. 订阅雷达里程计

- \* 1.1 如果系统没有初始化，则初始化系统，包括因子图、优化器、预积分器等
  - \* 1.2 每100帧雷达里程计之后重置优化器。清空因子图优化器，用优化出的结果作为先验
  - \* 1.3 将imuQueOpt队列中，所有早于当前雷达里程计的数据进行积分，获取最新的IMU bias
  - \* 1.4 使用预积分器构造ImuFactor，并加入因子图
  - \* 1.5 添加Imu的BetweenFactor（偏差的相对差别）
  - \* 1.6 将雷达里程计平移对齐到IMU（只做平移）
  - \* 1.7 构建雷达里程计因子，并加入因子图
  - \* 1.8 使用IMU预积分器的预测作为当前因子图的变量初始值
  - \* 1.9 将新的因子图和变量初始值加入优化器，并更新
  - \* 1.10 清空因子图和变量初始值缓存，为下一次加入因子准备
  - \* 1.11 从因子图中获取当前时刻优化后的各个变量
  - \* 1.12 重置预积分器
  - \* 1.13 检查优化结果，优化结果有问题时重置优化器
  - \* 1.14 将偏差优化器的结果传递到里程计优化器
  - \* 1.15 对里程计队列中剩余的数据进行积分
- 为什么要每100帧雷达里程计之后重置优化器。清空因子图优化器，用优化出的结果作为先验

下面我给你一个 通俗但专业 的解释，让你完全理解 **为什么很多 VIO/LIOSAM 系统每 N 帧（比如 100 帧）要重置优化器、清空因子图。**

## ⭐ 一句话回答

因为因子图会无限增长，会导致优化速度越来越慢、内存越来越大，最终系统变得不可实时。

定期重置 = 用当前优化结果作为新的先验，从而“紧缩”问题规模，保持实时性与稳定性。

## 📌 更深入解释：为什么要“每100帧重置一次优化器”？

因子图是不断增长的：

- 每来一帧雷达 → 加一次关键帧因子
- 每来一段 IMU 数据 → 加 IMU 因子
- 加边缘化因子、偏差因子、先验因子...

如果一直不清理，因子图会变成巨无霸：

来自	增长速度
雷达里程计因子	1/帧
IMU BetweenFactor	多个/帧
Bias 因子	1/帧
其他约束	若干

一个小时轻松几万因子 → 优化一次至少几十毫秒甚至几百毫秒 → 无法实时运行。

## 🔥 原因 1：控制因子图大小，保证实时性

GTSAM 的优化器（特别是 ISAM2）虽然是增量式，但：

- 因子太多会引起回代反复传播、矩阵变大、求解时间变长
- IMU 因子还会耦合速度和 bias，使问题维度更高

越积越多，系统会“变卡”。

解决方案：

→ 每 100 帧把因子图“压缩”一下

- 把当前的最优状态设成新的先验
- 清空老因子
- 因子图变回很小

就像 SLAM 中常说的：

把历史信息吸收到一个新的先验里，抛弃旧因子。

这就是“重置优化器”。

---

## 🔥 原因 2：避免 IMU 预积分误差累积太长时间

IMU 预积分在两个关键帧之间积分误差：

- 积分区间越长，误差增长越快
- 偏差随时间漂移越明显
- 数值线性化点会变得不稳定

通过每 100 帧重置：

- 预积分只保留最近的区间
  - 线性化点（pose、bias）变得更接近真实值
  - 优化效果更好、更稳定
- 

## 🔥 原因 3：避免数值漂移和 Hessian 的病态（ill-conditioned）

原因更数学一点：

- 因子越多 → Hessian 矩阵变得巨大
- 长时间积累导致某些约束重复或弱化
- 容易出现 病态矩阵（ill-conditioned）
- 优化不稳定，可能突然跳值或发散（你流程里的 1.13 就是专门处理这个）

重置后：

- 重新建立先验
- 整个问题更加稠密和健康（well-conditioned）
- Hessian 不再“老化”

## 🔥 原因 4：符合真实 SLAM 流程的“边缘化（Marginalization）”思想

重置优化器实际上是在做一种“主动边缘化”：

- 你把过去所有因子全部边缘化成一个先验（PriorFactor）
- 当前状态成为新的图的起点
- 就像 VIO 里的滑动窗口，只不过窗口更大（100帧）

这是一种非常常见也很有效的 SLAM 技巧。

---

## 🔥 原因 5：ISAM2 的增量线性化点会越来越差

ISAM2 是增量优化，虽然很快，但 线性化点不会自动重置。

随着时间推移：

- 偏差估计有漂移
- 线性化点越来越偏离真实轨迹
- IMU/雷达因子的非线性越来越严重
- 优化渐渐不准，甚至发散

重置后：

- 新图的线性化点来自优化后的最新状态 → 非常准
- 优化更快、结果更稳

---

## 📌 所以总结一下

✓ 重置优化器 = 控制规模 + 重设线性化点 + 保持实时 + 保持稳定

如果你不重置，后果通常是：

- 优化耗时从 3ms → 30ms → 200ms → 不实时
- IMU bias 越积越大
- Hessian 发散 → 突然跳值
- 系统最终必崩

你在流程中这句话：

清空因子图优化器，用优化出的结果作为先验

实际上就是：

“把历史全吸收掉，从一个高质量的起点重新来过”

---

**如果你愿意，我可以继续解释：**

- 为什么选择 100 帧？是不是越小越好？
- 和滑动窗口 VIO 的区别是什么？
- 和 LOAM 的 mapping thread 的重置机制有什么关系？
- 具体代码层面 ISAM2 是怎样被 reset 的？

需要继续吗？



比如在100帧雷达中间有1000帧imu,最后优化后有几帧

我们来精确回答你问的核心问题：

“比如 100 帧雷达里程计中间有 1000 帧 IMU，最后优化后有几帧？”

你的系统流程是典型的 **LOAM + IMU (LIOSAM 思路)**：

- 雷达帧（关键帧）是优化图中的主节点
- IMU 数据做预积分，不会一个个作为节点加入因子图

所以答案非常明确：

---

**最终优化后只有 雷达帧对应的节点（比如 100 个节点）**

IMU 1000 帧不会变成 1000 个因子图节点，  
它们会被“预积分”压缩成 **100 个 IMU 预积分因子**。

也就是说：

- 雷达帧 → 每帧都是图中的一个 Pose 节点  $X(i)$
- IMU 帧 → 不会变成节点，只会以预积分方式生成 99 个 ImuFactor

所以最后图里大概是：

类型	数量
关键帧 Pose (来自 100 帧雷达)	100 个节点
IMU 预积分因子	99 个
雷达相对位姿因子	99 个
bias between 因子	~99 个
先验因子	1 个

## ！ IMU 的 1000 帧去哪了？

它们都被融入预积分器中：

nginx

Copy code

IMU \*1000 帧 → 预积分 → 100 个雷达关键帧之间的 IMU 因子

所以实际上图中一点都不会变大，因为：

✓ 预积分的本质：

将大量 IMU 测量压缩成一个 IMU 因子（一种积分结果 + 噪声协方差）



再具体点（假设 100 个雷达关键帧）：

假设雷达帧编号：

Copy code

0, 1, 2, ..., 99

IMU：

yaml

Copy code

每两个雷达帧之间大约 10 帧 IMU (共 1000 IMU)

那么最终因子图中节点是：

python-repl

Copy code

```
X0  
X1  
X2  
...  
X99
```

IMU 因子是：

scss

Copy code

```
IMU(X0 → X1)  
IMU(X1 → X2)  
...  
IMU(X98 → X99)
```

雷达因子：

scss

Copy code

```
LidarFactor(X0, X1)  
LidarFactor(X1, X2)  
...
```

偏差因子：

scss

Copy code

```
Bias(0 → 1)  
Bias(1 → 2)  
...
```

## ！所以最终有多少“帧”？

- ✓ 节点（关键帧）数量 = 雷达帧数量（100）

IMU 数据无论多少，都不会增加节点数量。

---

## 🔥 更直白的类比

IMU 是辅助数据，不是关键帧。

它只让关键帧之间的约束变得更精确，但不会自己变成关键帧。

就像拍视频：

- 雷达：每秒 10 张主要画面 → 看得清楚，所以它们当节点
  - IMU：每秒 100 张模糊中间帧 → 不保存，只用来推测运动轨迹
- 

## 📌 你所描述的“每100帧重置优化器”对数量的影响？

- ✓ 重置之后，图仍然只保留一个最新先验 + 最新关键帧节点
- ✓ 过去的节点被边缘化，不会继续存在

所以在普通运行中，图中的节点数量永远不会超过 100，多数情况下 < 50。

---

## ✳️ 如果你愿意，我可以帮你进一步：

- 画一个图说明 100 雷达 + 1000 IMU 是如何变成 100 节点的
- 解释 IMU 预积分数学原理
- 展示 LIO-SAM / VINS-Fusion / LIO-Livox 中如何管理节点数量
- 给出伪代码或 GTSAM 示例实现

需要继续吗？



先验因子是什么，我不理解，最后清空后就只剩一个先验因子吗，这个结果会传递给后端图优化吗，还是会作为关键帧呢

下面我给你一次性彻底解释清楚“先验因子（PriorFactor）是什么、为什么只剩它、它的作用是什么、是否算关键帧、怎么传递给后端优化”。

这是 SLAM / 固定窗口 / 图优化里最容易困惑的地方，我帮你画清楚整个逻辑。

---



## 1. 什么是“先验因子”(PriorFactor)？

先验因子 = 给某个变量加一个固定的约束 / 已知值。

它的形式像：

nginx

Copy code

$x_0 \approx x_{initial}$

数学上就是：

Copy code

$\| x_0 - \text{初始值} \|$  约束很强（协方差小）

---

✓ 你可以把先验因子理解成：

“我知道这个节点现在应该在这里，你必须靠近这个结果。”

在图优化里，它的作用是提供

- 红旗 坐标系固定点 (anchor)
  - 红旗 把历史全部压缩成一个约束
  - 红旗 作为新的图的起点
-

## 📌 2. 为什么清空因子图后只剩一个先验因子？

因为你把所有历史信息浓缩到了这个先验因子中。

例如：

你的系统经过 100 帧优化后得到了：

Copy code

$X(99)$  = 优化后的当前姿态

速度 =  $v$

偏差 =  $b$

当你 **重置优化器** 的时候，就是：

🔥 把当前优化出来的  $X(99)$  作为新的先验：

SCSS

Copy code

```
PriorFactor( X0_new = X(99)_optimized )
```

这样：

- 旧的 100+ 个因子全部丢掉
- 它们的信息已经“吸收”进先验里
- 新图从这个先验的状态重新开始

所以只会留下：

nginx

Copy code

```
PriorFactorPose
```

```
PriorFactorVel
```

```
PriorFactorBias
```

通常一共 3 个先验因子（或一个联合先验）。

## 📌 3. 清空后只剩先验因子是正常的吗？

是的，这就是 **滑动窗口 / 固定窗口 VIO / 重置 ISAM2** 的标准流程。

这不是 bug，这是 正确做法。



## 4. 这个先验因子会如何作用于后端图优化？

✓ 这个先验因子是下一次优化器的 起点

下一次你再加入新的：

- 雷达因子
- IMU 预积分因子
- Bias 因子

这些因子的变量都是在 **先验因子定义的坐标系下** 优化的。

换句话说：

后端图优化永远都从这个先验因子“接着往下优化”。

它不是 keyframe，它是坐标约束。



## 5. 这个先验因子是不是关键帧？

✗ 不是关键帧。

✓ 它是一个“强约束”，不是物理测量。

关键帧是：

Copy code

$x_0, x_1, x_2, \dots$  (雷达的位姿节点)

先验因子只是：

SCSS

Copy code

**Prior(  $x_0 =$  上一次优化的最后结果 )**



## 6. 为什么需要先验因子？

如果没有先验因子，清空因子图以后：

- 图里没有任何约束
- pose、vel、bias 都变成“漂浮的”未知量
- 优化器会“发散”，结果毫无意义

先验因子用来：

- ✓ 固定当前图的起点
  - ✓ 引导新的因子（IMU + 雷达）往正确方向优化
  - ✓ 保证新图稳定收敛
- 

## 📍 7. 整个流程图示（非常重要）

假设你有 100 帧雷达：

nginx

Copy code

X0 - X1 - X2 - ... - X99

IMU 因子很多，Bias 因子很多。

优化器不断增大。

当到达 X99 时你重置：

### 🔥 重置前：

图里有几百上千个因子：

nginx

Copy code

X0 — X1 — X2 — ... — X99  
 ↗ (IMU/Bias/LiDAR 因子很多)  
 |  
 先验（最开始的）

### 🔥 重置后：

清空图 → 用最后一个结果生成新的先验：

SCSS

 Copy code

```
Prior(X0_new = X99_optimized)
```

(图里只有一个节点 + 一个先验因子)

## 📌 8. 下一次迭代 (加入 IMU + 雷达) :

新的因子不断加入：

python-repl

 Copy code

```
Prior(X0_new)
  |
  IMUFactor
  |
  X1_new
  |
  LidarFactor
  |
  X2_new
  ...
  ...
```

这样图又开始从头构建。

## 📌 9. 每次重置后对后端的意义

- ✓ 保持图规模小（实时）
- ✓ 让线性化点非常准（IMU 因子稳定）
- ✓ 避免 Hessian 病态
- ✓ 避免误差积累太久
- ✓ 保持系统几十毫秒级实时运行

也就是你的流程里说的：

## 1.14 将偏差优化器的结果传递到里程计优化器

是指

- 把重置后的先验状态（pose、vel、bias）传递给后端
  - 后端继续在这个新起点构建新图
- 

## 🎯 最后，用一句话总结你问的问题：

重置优化器后，只保留 1 个（或 3 个）先验因子，用来作为下一轮优化的起点；它不会作为关键帧，但它承载了历史所有信息，并会被后端继续使用。

---

如果你愿意，我可以继续讲：

- 为什么先验因子的协方差要调得很强？
- 为什么要同时给 pose、vel、bias 都加先验？
- LIO-SAM / VINS-Fusion 都是怎么做先验 reset 的？
- 给你画一张“因子图重置过程”的图示

你需要哪一部分？



关键帧是如何计算的，这几个因子是如何计算的

下面我把 **关键帧如何生成、IMU/LiDAR/Bias 因子如何计算** 用一步一步的方式讲清楚。  
我会用 **LIOSAM / LIO-Livox / VINS-Fusion** 中通用的方式解释，让你完全理解后端图优化从哪里来的。

---

## ⭐ 整体框架回忆

只要搞清楚图优化里：

1. **关键帧（Pose 节点）怎么产生**

2. IMU 因子怎么构造
3. 雷达因子怎么构造
4. Bias 因子怎么构造

就完整了解因子图了。

我按这个顺序给你讲👉

## 1 关键帧 (Keyframe) 如何计算？

关键帧是图优化中的节点 ( $X_0, X_1, X_2 \dots$ )，也就是 Pose3。

关键帧不是每一帧传感器数据，关键帧要“触发”才生成。

LIO / VIO 中最常见的关键帧触发条件：

### ✓ 方式 1：固定时间间隔（常见于 VINS）

比如每 0.1 秒取一个关键帧。

### ✓ 方式 2：运动量触发（常见于 LIO-SAM）

例如位移  $> 0.5m$  或旋转  $> 5^\circ$  时产生关键帧。

公式示例：

markdown

Copy code

```
if |p_curr - p_last_kf| > 0.5m or |rotation_delta| > 5°  
    生成一个关键帧
```

### ✓ 方式 3：雷达帧本身就是关键帧（LOAM 系）

例如：

- 每来一帧 LiDAR scan
- 运行 scan-to-map 估计位姿

- 这帧自动被当作关键帧加入后端

你提的系统就是这种：

**每一帧雷达里程计都对应关键帧。**

所以如果有 100 帧雷达 → 有 100 个关键帧。

---

## 2 IMU 因子 (**imuFactor**) 如何计算？

IMU 因子来源于 **预积分 (preintegration)**。

其过程如下：

### ✓ 步骤 1：IMU 原始测量插入队列 **imuQueOpt**

(加速度  $a_t$ , 角速度  $\omega_t$ )

---

### ✓ 步骤 2：找到两个关键帧之间的 IMU 数据

假设关键帧时刻是  $t_i$  和  $t_j$  (一般是雷达帧对应的时间)，

你取所有 IMU：

nginx

Copy code

$t_i < t_{imu} < t_j$

---

### ✓ 步骤 3：预积分公式

对加速度和角速度进行积分 (扣掉 bias)，得到：

- $\Delta R$  (旋转增量)
- $\Delta v$  (速度增量)
- $\Delta p$  (位置增量)
- 协方差矩阵 (噪声随时间传播)

GTSAM 提供现成的预积分器：

cpp

Copy code

```
preintegrated_imu_->integrateMeasurement(acc, gyro, dt);
```

## ✓ 步骤 4：构造 IMU 因子

cpp

 Copy code

```
ImuFactor(X(i), V(i), X(j), V(j), preintegrated_imu_result);
```

这个因子连接：

- 位姿  $X(i)$ 、 $X(j)$
- 速度  $V(i)$ 、 $V(j)$

## 3 LiDAR 因子如何计算？

这取决于你使用的算法类型：

### A 方式 1：直接使用雷达里程计（外部给定）→ LIO-SAM 的后端

你在流程中提到：

1.7 构建雷达里程计因子，并加入因子图

说明你不是用 scan-to-map，而是前端已经给你一个 LiDAR 相对 pose。

通常是：

SCSS

 Copy code

```
Pose_Lidar(i) → Pose_Lidar(j)
```

然后将该相对变换  $T_{ij}$  加入因子图：

cpp

 Copy code

```
BetweenFactor<Pose3>( X(i), X(j), T_ij, LidarNoise )
```

注意你还提到：

1.6 只做平移对齐 IMU → 说明雷达坐标系需要外参转换

## B 方式 2：使用 ICP 或 Scan-to-Map

如果你的前端自己算相对位姿：

ini

Copy code

$T_{ij} = \text{ICP}(\text{Scan}_i, \text{Scan}_j)$

也一样加 BetweenFactor：

cpp

Copy code

$\text{BetweenFactor} < \text{Pose3} > (\text{X}(i), \text{X}(j), T_{ij}, \text{noise})$

## 4 Bias 因子 BetweenFactor 如何计算？

GTSAM 推荐加入：

css

Copy code

$\text{BetweenFactor} < \text{imuBias::ConstantBias} > (\text{Bias}(i), \text{Bias}(j), \text{ZeroBias}, \text{Covariance})$

作用：

- 限制偏差变化不要太剧烈 (IMU bias 随时间慢漂)
- 如果漂得太快，优化器会察觉并修正

公式是：

scss

Copy code

$\text{Bias}(j) - \text{Bias}(i) \approx 0$  (噪声很小)

所以它确保 bias 是连续的。

## 🔥 整合起来之后，因子图是这样：

关键帧（每一帧雷达）：

python-repl

Copy code

X0, V0, B0

X1, V1, B1

X2, V2, B2

...

因子：

scss

Copy code

Prior(X0, V0, B0)

IMU因子( X0, V0, B0 -> X1, V1, B1 )

LIDAR因子( X0 -> X1 )

IMU因子( X1, V1, B1 -> X2, V2, B2 )

LIDAR因子( X1 -> X2 )

BiasBetween因子( B0 -> B1 )

BiasBetween因子( B1 -> B2 )

...

## ⭐ 最终答案（你问的核心）

关键帧如何算？——由雷达里程计触发（每一帧雷达是关键帧）

IMU 因子如何算？——预积分（把  $t_i$  到  $t_j$  的 IMU 数据压缩成一个因子）

雷达因子如何算？——根据雷达里程计前端给的相对位姿，用 BetweenFactor 表示

Bias 因子？——限制 bias 连续性，用 BetweenFactor 表示零差分约束

## 🎯 如果你愿意，我还能继续解释：

- 画一张因子图示意图（你会更直观）
- 解释 IMU 预积分的数学原理  $\Delta R, \Delta v, \Delta p$
- 为什么每帧 LiDAR 直接作为关键帧？
- Keyframe 密度对精度和实时性的影响
- 给你一段完整的 GTSAM 后端代码结构

你想继续哪一部分？

□ ⏪ ...

因子的公式帮我学习下

下面我把 **IMU 因子**、**雷达 Between 因子**、**Bias 因子**、**先验因子** 的公式全部用最清晰、实用的方式整理出来，

这些公式就是 **GTSAM/LIO-SAM/VINS-Fusion** 后端的基础。

我会按照“概念 → 数学公式 → 实际方程 → 噪声模型”来讲。

## ✳️ 因子图中的 4 大因子

1. 先验因子 (**PriorFactor**)
2. IMU 预积分因子 (**ImuFactor**)
3. 雷达相对位姿因子 (**BetweenFactor**)
4. Bias 连续性因子 (**BiasBetweenFactor**)

下面逐个展开。

## ⭐ 1. 先验因子 (**PriorFactor**)

用途：固定第一个关键帧，使图不会漂浮。

数学形式：

$$e = X - X_{prior}$$

协方差：

$$R_{\text{prior}}$$

误差项：

$$\|e\|_{R_{\text{prior}}}^2 = (X - X_{\text{prior}})^T R^{-1} (X - X_{\text{prior}})$$

GTSAM 中写法：

cpp

Copy code

```
graph.emplace_shared<PriorFactor<Pose3>>(X(0), X_prior, noisePrior);
```

---

## ★ 2. IMU 预积分因子 (ImuFactor)

IMU 因子的核心是预积分结果：

- $\Delta R$  旋转增量
- $\Delta v$  速度增量
- $\Delta p$  位置增量

它们来自 IMU 加速度  $a$  和角速度  $\omega$ ：

---

### 2.1 原理：

IMU 单帧公式（原始）：

$$\omega_m = \omega + b_g + n_g$$

$$a_m = R^T(\ddot{p} - g) + b_a + n_a$$

其中：

- $b_g, b_a$  = gyro 和 accel 的 bias
- $n_g, n_a$  = 传感器噪声

---

### 2.2 预积分定义

IMU 在关键帧  $i \rightarrow j$  之间连续积分，得到：

$$\Delta R_{ij}, \Delta v_{ij}, \Delta p_{ij}$$

它们满足：

$$R_j = R_i \Delta R_{ij} \cdot \exp(J_r^g \delta b_g)$$

$$v_j = v_i + g \Delta t + R_i \Delta v_{ij} + J_v^b \delta b$$

$$p_j = p_i + v_i \Delta t + \frac{1}{2} g \Delta t^2 + R_i \Delta p_{ij} + J_p^b \delta b$$

GTSAM 用线性化的 Jacobian (J) 自动处理 bias 的影响。

---

## ★ IMU 因子的误差项 (GTSAM 中最关键)

IMU 因子的误差项是：

$$e = \begin{bmatrix} e_R \\ e_v \\ e_p \end{bmatrix}$$


---

✓ 旋转误差：

$$e_R = \text{Log}(\Delta R_{ij}^{-1} (R_i^{-1} R_j) \cdot \text{CorrectBiasTerm})$$


---

✓ 速度误差：

$$e_v = R_i^T (v_j - v_i - g \Delta t) - \Delta v_{ij} - J_v \delta b$$


---

✓ 位置误差：

$$e_p = R_i^T (p_j - p_i - v_i \Delta t - \frac{1}{2} g \Delta t^2) - \Delta p_{ij} - J_p \delta b$$

噪声：

$$(e)^T \Sigma^{-1} e$$

这里的协方差  $\Sigma$  是预积分器传递过来的 IMU 噪声传播结果。

## ★ 3. 雷达 BetweenFactor (LiDAR Pose Factor)

你提到的雷达里程计因子一般是相邻雷达帧之间的相对位姿：

$$T_{ij} = T_i^{-1}T_j$$

误差项：

$$e = \text{Log}(T_{ij}^{-1}(T_i^{-1}T_j))$$

协方差：

$$e^T R_{\text{lidar}}^{-1} e$$

GTSAM 写法：

cpp

 Copy code

```
graph.emplace_shared<BetweenFactor<Pose3>>(<span style="color: orange;>X(i), X(j), T_ij, lidarNoise </span>)
```

## ★ 4. Bias 连续性因子 (BiasBetweenFactor)

假设 bias 是一个随机游走：

$$b_j = b_i + n_{bias}$$

误差：

$$e_b = (b_j - b_i)$$

协方差：

$$e_b^T R_{bias}^{-1} e_b$$

GTSAM 写法：

cpp

 Copy code

```
graph.emplace_shared<BetweenFactor<imuBias::ConstantBias>>(<span style="color: orange;>B(i), B(j), imuBias::ConstantBias(), biasRwNoise);</span>)
```

## ★ 因子图整体误差函数（最终优化目标）

所有因子的误差项加起来：

$$\min_{X,V,B} \sum e_{\text{prior}} + \sum e_{\text{lidar}} + \sum e_{\text{imu}} + \sum e_{\text{bias}}$$

这就是 Bundle Adjustment / MAP 优化。

GTSAM 把这个目标函数自动转为稀疏矩阵求解。

## 🧠 总结（非常重要）

因子	公式结构	含义
先验因子	$X - X_{\text{prior}}$	固定起点
LiDAR 因子	$\text{Log}(T_{ij}^{-1} (T_i^{-1} T_j))$	雷达相对位姿约束
IMU 因子	$\Delta R, \Delta v, \Delta p$ 的预积分约束	提供高频速度/加速度信息
Bias 因子	$b_j - b_i$	限制 bias 平滑变化

