sensors_mpl.c的 `pollEvent` 函数，是SensorDevice的poll的实现。

其中有 `readEvent` 函数，是获取传感器event事件的具体函数。
`readEvent` 函数中，有 `CALL_MEMBER_FN` 的宏定义，是一个handler的指针。

```
update = CALL_MEMBER_FN(this, mHandlers[i])(mPendingEvents + i);
```

以 `rawGyroHandler` 为例子，这是原始陀螺仪数据的handler方法。

```
int MPLSensor::rawGyroHandler(sensors_event_t* s)
{
    VHANDLER_LOG;
    int update;
#if defined ANDROID_LOLLIPOP
    update = inv_get_sensor_type_gyroscope_raw(s->uncalibrated_gyro.uncalib,
                                               &s->gyro.status, (inv_time_t *)
(&s->timestamp));
#else
    update = inv_get_sensor_type_gyroscope_raw(s->uncalibrated_gyro.uncalib,
                                               &s->gyro.status, &s-
>timestamp);
#endif
    if(update) {
        memcpy(s->uncalibrated_gyro.bias, mGyroBias, sizeof(mGyroBias));
        LOGV_IF(HANDLER_DATA,"HAL:gyro bias data : %+f %+f %+f -- %lld - %d",
            s->uncalibrated_gyro.bias[0], s->uncalibrated_gyro.bias[1],
            s->uncalibrated_gyro.bias[2], s->timestamp, update);
    }
    s->gyro.status = SENSOR_STATUS_UNRELIABLE;
    LOGV_IF(HANDLER_DATA, "HAL:raw gyro data : %+f %+f %+f -- %lld - %d",
            s->uncalibrated_gyro.uncalib[0], s->uncalibrated_gyro.uncalib[1],
            s->uncalibrated_gyro.uncalib[2], s->timestamp, update);
    return update;
}
```

上诉代码中，通过 `inv_get_sensor_type_gyroscope_raw` 函数获得了未校准的陀螺仪数据。并根据是否获取到bias的值输出log。

`inv_get_sensor_type_gyroscope_raw` 的实现如下：

```
/** Gyroscope raw data (rad/s) in body frame.
* @param[out] values Rotation Rate in rad/sec.
* @param[out] accuracy Accuracy of the measurment, 0 is least accurate,
*                      while 3 is most accurate.
* @param[out] timestamp The timestamp for this sensor. Derived from the
*                       timestamp sent to inv_build_gyro().
* @return     Returns 1 if the data was updated or 0 if it was not updated.
*/
int inv_get_sensor_type_gyroscope_raw(float *values, int8_t *accuracy,
                                      inv_time_t * timestamp)
```

```
{
    long gyro[3];
    int status;

    inv_get_gyro_set_raw(gyro, accuracy, timestamp);
    values[0] = gyro[0] * GYRO_CONVERSION;
    values[1] = gyro[1] * GYRO_CONVERSION;
    values[2] = gyro[2] * GYRO_CONVERSION;
    if (hal_out.gyro_status & INV_NEW_DATA)
        status = 1;
    else
        status = 0;
    return status;
}
```

根据注释可以看出，这个函数返回了以机身为坐标的陀螺仪原始数据。

其中的核心代码就是 `inv_get_gyro_set_raw` 函数，下面进去 `inv_get_gyro_set_raw` 查看，代码如下：

```
/** Gets a whole set of gyro raw data including data, accuracy and timestamp.
 * @param[out] data Gyro Data where 1 dps = 2^16
 * @param[out] accuracy Accuracy 0 being not accurate, and 3 being most
accurate.
 * @param[out] timestamp The timestamp of the data sample.
*/
void inv_get_gyro_set_raw(long *data, int8_t *accuracy, inv_time_t *timestamp)
{
    memcpy(data, sensors.gyro.raw_scaled, sizeof(sensors.gyro.raw_scaled));
    if (timestamp != NULL) {
        *timestamp = sensors.gyro.timestamp;
    }
    if (accuracy != NULL) {
        *accuracy = 0;
    }
}
```

这个函数就是返回了一个 `sensors.gyro.raw_scaled` 的数据结构以及时间戳与精度。
sensors的结构如下：

```
struct inv_sensor_cal_t {
    struct inv_single_sensor_t gyro;
    struct inv_single_sensor_t accel;
    struct inv_single_sensor_t compass;
    struct inv_single_sensor_t temp;
    struct inv_quat_sensor_t quat;
    struct inv_soft_iron_t soft_iron;
    /** Combinations of INV_GYRO_NEW, INV_ACCEL_NEW, INV_MAG_NEW to indicate
    * which data is a new sample as these data points may have different
sample rates.
    */
```

```
        int status;
    };
```

gyro的结构如下：

```
    struct inv_single_sensor_t {
        /** Orientation Descriptor. Describes how to go from the mounting frame to
    the body frame when
        * the rotation matrix could be thought of only having elements of 0,1,-1.
        * 2 bits are used to describe the column of the 1 or -1 and the 3rd bit is
    used for the sign.
        * Bit 8 is sign of +/- 1 in third row. Bit 6-7 is column of +/-1 in third
    row.
        * Bit 5 is sign of +/- 1 in second row. Bit 3-4 is column of +/-1 in
    second row.
        * Bit 2 is sign of +/- 1 in first row. Bit 0-1 is column of +/-1 in first
    row.
        */
        int orientation;
        /** The raw data in raw data units in the mounting frame */
        short raw[3];
        /** Raw data in body frame */
        long raw_scaled[3];
        /** Calibrated data */
        long calibrated[3];
        long sensitivity;
        /** Sample rate in microseconds */
        long sample_rate_us;
        long sample_rate_ms;
        /** INV_CONTIGUOUS is set for contiguous data. Will not be set if there
    was a sample
        * skipped due to power savings turning off this sensor.
        * INV_NEW_DATA set for a new set of data, cleared if not available.
        * INV_CALIBRATED_SET if calibrated data has been solved for */
        int status;
        /** 0 to 3 for how well sensor data and biases are known. 3 is most
    accurate. */
        int accuracy;
        inv_time_t timestamp;
        inv_time_t timestamp_prev;
        /** Bandwidth in Hz */
        int bandwidth;
    };
```

从注释可以看出，`sensors.gyro.raw_scaled` 的数据就是机身的陀螺仪原始数据。那这个数据是从哪里来的，看下面的函数：

```
    /** Takes raw data stored in the sensor, removes bias, and converts it to
    * calibrated data in the body frame. Also store raw data for body frame.
    * @param[in,out] sensor structure to modify
    * @param[in] bias bias in the mounting frame, in hardware units scaled by
```

```
*                 2^16. Length 3.
*/
void inv_apply_calibration(struct inv_single_sensor_t *sensor, const long
*bias)
{
    long raw32[3];

    // Convert raw to calibrated
    raw32[0] = (long)sensor->raw[0] << 15;
    raw32[1] = (long)sensor->raw[1] << 15;
    raw32[2] = (long)sensor->raw[2] << 15;

    inv_convert_to_body_with_scale(sensor->orientation, sensor->sensitivity <<
1, raw32, sensor->raw_scaled);

    raw32[0] -= bias[0] >> 1;
    raw32[1] -= bias[1] >> 1;
    raw32[2] -= bias[2] >> 1;

    inv_convert_to_body_with_scale(sensor->orientation, sensor->sensitivity <<
1, raw32, sensor->calibrated);

    sensor->status |= INV_CALIBRATED;
}
```

注释说明了将陀螺仪原始数据储存在一个 inv_single_sensor_t 的结构体中，消除偏差，并且转换成机身的校准数据，同时，未校准数据也储存下来。这个函数中输入和输出都是snesors这个结构体，inv_convert_to_body_with_scale 函数是核心的转换函数，第一个函数的输出是 sensor->raw_scaled ,原始数据。之后减去bias偏差，并再次使用 inv_convert_to_body_with_scale 函数转换输出到 sensor->calibrated 中储存。

下面看一下 inv_convert_to_body_with_scale 函数做了些什么：

```
/** Uses the scalar orientation value to convert from chip frame to body frame
and
* apply appropriate scaling.
* @param[in] orientation A scalar that represent how to go from chip to body
frame
* @param[in] sensitivity Sensitivity scale
* @param[in] input Input vector, length 3
* @param[out] output Output vector, length 3
*/
void inv_convert_to_body_with_scale(unsigned short orientation, long
sensitivity, const long *input, long *output)
{
    output[0] = inv_q30_mult(input[orientation & 0x03] *
                         SIGNSET(orientation & 0x004), sensitivity);
    output[1] = inv_q30_mult(input[(orientation>>3) & 0x03] *
                         SIGNSET(orientation & 0x020), sensitivity);
    output[2] = inv_q30_mult(input[(orientation>>6) & 0x03] *
                         SIGNSET(orientation & 0x100), sensitivity);
}
```

该函数用一个scalar orientation计算了芯片框架到机身框架的向量转换。

下面回到 `inv_apply_calibration` 函数，看看哪些地方使用了这个函数。
1) `inv_set_gyro_bias` 未使用
2) `inv_build_gyro`。
接着详细看一下2)的使用：

```
/** Record new gyro data and calls inv_execute_on_data() if previous
* sample has not been processed.
* @param[in] gyro Data is in device units. Length 3.
* @param[in] timestamp Monotonic time stamp, for Android it's in nanoseconds.
* @param[out] executed Set to 1 if data processing was done.
* @return Returns INV_SUCCESS if successful or an error code if not.
*/
inv_error_t inv_build_gyro(const short *gyro, inv_time_t timestamp)
{
#ifdef INV_PLAYBACK_DBG
    if (inv_data_builder.debug_mode == RD_RECORD) {
        int type = PLAYBACK_DBG_TYPE_GYRO;
        fwrite(&type, sizeof(type), 1, inv_data_builder.file);
        fwrite(gyro, sizeof(gyro[0]), 3, inv_data_builder.file);
        fwrite(&timestamp, sizeof(timestamp), 1, inv_data_builder.file);
    }
#endif

    memcpy(sensors.gyro.raw, gyro, 3 * sizeof(short));
    sensors.gyro.status |= INV_NEW_DATA | INV_RAW_DATA | INV_SENSOR_ON;
    sensors.gyro.timestamp_prev = sensors.gyro.timestamp;
    sensors.gyro.timestamp = timestamp;
    inv_apply_calibration(&sensors.gyro, inv_data_builder.save.gyro_bias);

    return INV_SUCCESS;
}
```

可以看出这个函数的第一个参数，一个const *short 类型的变量就是数据的来源。
接着看哪里使用了 `inv_build_gyro` 函数：

正是MPLSensor类的buildMpuEvent成员函数，函数中有语句
`inv_build_gyro(mCachedGyroData, mSensorTimestamp);`
`mCachedGyroData` 是来源。
往上翻，看到

```
for (i = 0; i < 3; i++) {
        if (mLocalSensorMask & INV_THREE_AXIS_ACCEL) {
            mCachedAccelData[i] = *((short *) (rdata + i * 2));
        }
        if (mLocalSensorMask & INV_THREE_AXIS_GYRO) {
            mCachedGyroData[i] = *((short *) (rdata + i * 2 +
                ((mLocalSensorMask & INV_THREE_AXIS_ACCEL)? 6: 0)));
        }
        if ((mLocalSensorMask & INV_THREE_AXIS_COMPASS)
```

```
                && mCompassSensor->isIntegrated()) {
            mCachedCompassData[i] =
                *((short *)(rdata + i * 2 + 6 * (sensors - 1)));
        }
    }
```

mCacheGyroData从rdata中来，继续看rdata的来源。
rdata的实际空间是一个字符数组，如下定义

```
char mIIOBuffer[(16 + 8 * 3 + 8) * IIO_BUFFER_LENGTH];//IIO_BUFFER_LENGTH的长度
为480
```

rdata是使用了read从设备文件中读进，其中的 `nbyte=(8 * sensors + 8) * 1;//sensors是传感器的数量`

```
ssize_t rsize = read(iio_fd, rdata, nbyte);
```

而 `iio_fd = open(iio_device_node, O_RDONLY);` ，是 `iio_device_node` 文件的一个文件句柄。这句打开文件的语句在 `enable_iio_sysfs` 函数中，并被MPLSensor的构造函数所调用。
下面看看iio_device_node是什么： `inv_get_iio_device_node(iio_device_node);` 获得了这个字符串，进去 `inv_get_iio_device_node` 看一下：

```
/**
 *  @brief   return iio device node. If iio is not initialized, return false.
 *           So the return must be checked to make sure the numeber is valid.
 *  @unsigned char *name: This should be array big enough to hold the device
 *            node. It should be zeroed before calling this function.
 *            Or it could have unpredicable result.
 */
inv_error_t inv_get_iio_device_node(const char *name)
{
    if (process_sysfs_request(CMD_GET_DEVICE_NODE, (char *)name) < 0)
        return INV_ERROR_NOT_OPENED;
    else
        return INV_SUCCESS;
}
```

重点在 `process_sysfs_request(CMD_GET_DEVICE_NODE, (char *)name)` 中
进入 `process_sysfs_request` 中，可以看到：

```
case CMD_GET_DEVICE_NODE:
        //data就是传入的参数name
        sprintf(data, "/dev/iio:device%d", iio_dev_num);
        break;
```

就是把iio_dev_num连接在/dev/iio:device后面了， `iio_dev_num` 是通过一个 `find_type_by_name` 函数获取的， `find_type_by_name` 怎么运作的不在深究。

有点太过深入了，我们回到 buildMpuEvent 成员函数中，我们说这个函数调用了
inv_build_gyro(mCachedGyroData, mSensorTimestamp);
来获取数据，那么哪里调用了 buildMpuEvent 函数呢：

在 sensors_poll_context_t::pollEvents() 中我们找到了这样一句 ((MPLSensor*) mSensor)->buildMpuEvent();

而 pollEvents 被 poll__poll 所调用，然后 poll__poll 被 open_sensors 处理到一个 sensors_poll_context_t 类型的dev中：
dev->device.poll= poll__poll;

这个函数最终会被SensorDevice类所使用，最终SensorService会使用SensorDevice类进行数据的监听和获取。
到此，整个流程逻辑分析完毕，从SensorDevice出发又回到了SensorDevice。