**首先查看Event的类型：**

/\*\* Type Event type.\*/

enum class Type

{

TOUCH,

KEYBOARD,

ACCELERATION,

MOUSE,

FOCUS,

GAME\_CONTROLLER,

CUSTOM

};

Type \_type; ///< Event type

bool \_isStopped; ///< whether the event has been stopped.

Node\* \_currentTarget; ///< Current target

**查看EventTouch类，它继承于Event类**

enum class EventCode

{

BEGAN,

MOVED,

ENDED,

CANCELLED

};

EventCode \_eventCode;

std::vector<Touch\*> \_touches;

**查看Touch（每个EventTouch包含多个Touch指针）**

**注意: touch->getLocation();这里获取的是designedResolution中的坐标（已验证）**

class CC\_DLL Touch : public Ref

{

public:

/\*\*

\* Dispatch mode, how the touches are dispatched.

\* @js NA

\*/

enum class DispatchMode {

ALL\_AT\_ONCE, /\*\* All at once. \*/

ONE\_BY\_ONE, /\*\* One by one. \*/

};

/\*\* Constructor.

\* @js ctor

\*/

Touch()

: \_id(0),

\_startPointCaptured(false),

\_curForce(0.f),

\_maxForce(0.f)

{}

/\*\* Returns the current touch location in OpenGL coordinates.

\*

\* @return The current touch location in OpenGL coordinates.

\*/

Vec2 getLocation() const;

/\*\* Returns the previous touch location in OpenGL coordinates.

\*

\* @return The previous touch location in OpenGL coordinates.

\*/

Vec2 getPreviousLocation() const;

/\*\* Returns the start touch location in OpenGL coordinates.

\*

\* @return The start touch location in OpenGL coordinates.

\*/

Vec2 getStartLocation() const;

/\*\* Returns the delta of 2 current touches locations in screen coordinates.

\*

\* @return The delta of 2 current touches locations in screen coordinates.

\*/

Vec2 getDelta() const;

/\*\* Returns the current touch location in screen coordinates.

\*

\* @return The current touch location in screen coordinates.

\*/

Vec2 getLocationInView() const;

/\*\* Returns the previous touch location in screen coordinates.

\*

\* @return The previous touch location in screen coordinates.

\*/

Vec2 getPreviousLocationInView() const;

/\*\* Returns the start touch location in screen coordinates.

\*

\* @return The start touch location in screen coordinates.

\*/

Vec2 getStartLocationInView() const;

/\*\* Set the touch information. It always used to monitor touch event.

\*

\* @param id A given id

\* @param x A given x coordinate.

\* @param y A given y coordinate.

\*/

void setTouchInfo(int id, float x, float y)

{

\_id = id;

\_prevPoint = \_point;

\_point.x = x;

\_point.y = y;

\_curForce = 0.0f;

\_maxForce = 0.0f;

if (!\_startPointCaptured)

{

\_startPoint = \_point;

\_startPointCaptured = true;

\_prevPoint = \_point;

}

}

/\*\* Set the touch information. It always used to monitor touch event.

\*

\* @param id A given id

\* @param x A given x coordinate.

\* @param y A given y coordinate.

\* @param force Current force for 3d touch.

\* @param maxForce maximum possible force for 3d touch.

\*/

void setTouchInfo(int id, float x, float y, float force, float maxForce)

{

\_id = id;

\_prevPoint = \_point;

\_point.x = x;

\_point.y = y;

\_curForce = force;

\_maxForce = maxForce;

if (!\_startPointCaptured)

{

\_startPoint = \_point;

\_startPointCaptured = true;

\_prevPoint = \_point;

}

}

/\*\* Get touch id.

\* @js getId

\* @lua getId

\*

\* @return The id of touch.

\*/

int getID() const

{

return \_id;

}

/\*\* Returns the current touch force for 3d touch.

\*

\* @return The current touch force for 3d touch.

\*/

float getCurrentForce() const;

/\*\* Returns the maximum touch force for 3d touch.

\*

\* @return The maximum touch force for 3d touch.

\*/

float getMaxForce() const;

private:

int \_id;

bool \_startPointCaptured;

Vec2 \_startPoint;

Vec2 \_point;

Vec2 \_prevPoint;

float \_curForce;

float \_maxForce;

};

**eventDispatcher的工作由三部分组成：**

1 事件分发器EventDispatcher

2 事件类型 EventTouch EventKeyboard

3 事件监听器 EventListernerTouch EventListenerKeyboard

4 监听器实现了各种触发后的逻辑，在适当的时候由事件分发器分发事件类型，然后调用相应类型的监听器

**首先查看EventListenerVector**

class EventListenerVector

{

public:

EventListenerVector();

~EventListenerVector();

size\_t size() const;

bool empty() const;

void push\_back(EventListener\* item);

void clearSceneGraphListeners();

void clearFixedListeners();

void clear();

inline std::vector<EventListener\*>\* getFixedPriorityListeners() const { return \_fixedListeners; };

inline std::vector<EventListener\*>\* getSceneGraphPriorityListeners() const { return \_sceneGraphListeners; };

inline ssize\_t getGt0Index() const { return \_gt0Index; };

inline void setGt0Index(ssize\_t index) { \_gt0Index = index; };

private:

**std::vector<EventListener\*>\* \_fixedListeners;**

**std::vector<EventListener\*>\* \_sceneGraphListeners;**

ssize\_t \_gt0Index;

};

ListenerMap -- listener🡪ListenrVector（fixListeners，sceneGraphListeners）

实际上每次添加一个Listener，首先通过ListnerID（string类型）在ListenerMap中查找，如果找到就返回对应的EventListenerVector，如果没有找到，则创建一个EventListenerVector，同时将listenerID与新创建的EventListenerVector插入到listnerMap中

**重要的EventDispatcher：**

其中包含两个优先级数组：

FixedPriority：整形值，低权值的事件监听器将优于高权值的事件监听器

SceneGraphPriority：Node的z顺序高（绘制与顶部）节点将优于z顺序低的节点，这将保证诸如触碰事件的自顶向下传播

首先对上面两个数组进行排序，如上面所述，然后进行事件传递，传递的原则是首先把事件传递到fixedListener中优先级<0的Listeners中，如果事件被处理则break终止，如果不被处理则传递到sceneListener中，同理不被处理继续传递到fixedListener中优先级>0的listeners中，直到事件被处理或者被丢弃（没有任何节点处理）

/\*\* Adds a event listener for a specified event with the priority of scene graph.

\* @param listener The listener of a specified event.

\* @param node The priority of the listener is based on the draw order of this node.

\* @note The priority of scene graph will be fixed value 0. So the order of listener item

\* in the vector will be ' <0, scene graph (0 priority), >0'.

\*/

void addEventListenerWithSceneGraphPriority(EventListener\* listener, Node\* node);

/\*\* Adds a event listener for a specified event with the fixed priority.

\* @param listener The listener of a specified event.

\* @param fixedPriority The fixed priority of the listener.

\* @note A lower priority will be called before the ones that have a higher value.

\* **0 priority is forbidden for fixed priority since it's used** **for scene graph based priority.**

\*/

void addEventListenerWithFixedPriority(EventListener\* listener, int fixedPriority);

/\*\* Adds a Custom event listener.

**It will use a fixed priority of 1.**

\* @param eventName A given name of the event.

\* @param callback A given callback method that associated the event name.

\* @return the generated event. Needed in order to remove the event from the dispatcher

\*/

EventListenerCustom\* addCustomEventListener(const std::string &eventName, const std::function<void(EventCustom\*)>& callback);

以Touch事件为例：

在各个不同的平台中，每次收到一个Touch事件，则会调用GLView的handleTouchesBegin函数

**void GLView::handleTouchesBegin(int num, intptr\_t ids[], float xs[], float ys[])**

{

intptr\_t id = 0;

float x = 0.0f;

float y = 0.0f;

int unusedIndex = 0;

EventTouch touchEvent;

for (int i = 0; i < num; ++i)

{

id = ids[i];

x = xs[i];

y = ys[i];

**auto iter = g\_touchIdReorderMap.find(id); // 系统的id变换为游戏中的id**

// it is a new touch

if (iter == g\_touchIdReorderMap.end()) **// 没有找到表示为新的手指**

{

unusedIndex = getUnUsedIndex(); **// 寻找当前未使用的序号**

// The touches is more than MAX\_TOUCHES ?

if (unusedIndex == -1) {

CCLOG("The touches is more than MAX\_TOUCHES, unusedIndex = %d", unusedIndex);

continue;

}

Touch\* touch = g\_touches[unusedIndex] = new (std::nothrow) Touch();

touch->setTouchInfo(unusedIndex, (x - \_viewPortRect.origin.x) / \_scaleX,

(y - \_viewPortRect.origin.y) / \_scaleY);

CCLOGINFO("x = %f y = %f", touch->getLocationInView().x, touch->getLocationInView().y);

g\_touchIdReorderMap.insert(std::make\_pair(id, unusedIndex));

touchEvent.\_touches.push\_back(touch);

}

}

if (touchEvent.\_touches.size() == 0)

{

CCLOG("touchesBegan: size = 0");

return;

}

touchEvent.\_eventCode = EventTouch::EventCode::BEGAN;

auto dispatcher = Director::getInstance()->getEventDispatcher();

**dispatcher->dispatchEvent(&touchEvent);**

}

不论是Begin、Moved、End还是Cancel都会调用**dispatcher->dispatchEvent(&touchEvent);**

void EventDispatcher::dispatchEvent(Event\* event)

{

if (!\_isEnabled)

return;

updateDirtyFlagForSceneGraph();

DispatchGuard guard(\_inDispatch);

if (event->getType() == Event::Type::TOUCH)

{

**// 单独处理Touch事件**

**dispatchTouchEvent(static\_cast<EventTouch\*>(event));**

return;

}

auto listenerID = \_\_getListenerID(event);

sortEventListeners(listenerID);

auto pfnDispatchEventToListeners = &EventDispatcher::dispatchEventToListeners;

if (event->getType() == Event::Type::MOUSE) {

pfnDispatchEventToListeners = &EventDispatcher::dispatchTouchEventToListeners;

}

auto iter = \_listenerMap.find(listenerID);

if (iter != \_listenerMap.end())

{

auto listeners = iter->second;

auto onEvent = [&event](EventListener\* listener) -> bool{

event->setCurrentTarget(listener->getAssociatedNode());

listener->\_onEvent(event);

return event->isStopped();

};

(this->\*pfnDispatchEventToListeners)(listeners, onEvent);

}

updateListeners(event);

}

**void EventDispatcher::dispatchTouchEvent(EventTouch\* event)**

{

**// 对于OneByOne和AllAtOnce事件进行排序，其中这两个ListerID均对应着一个ListenerVector，对于Fix优先级按照从小到大排序，对于SceneGraph优先级按照从大到小排序**

sortEventListeners(EventListenerTouchOneByOne::LISTENER\_ID);

sortEventListeners(EventListenerTouchAllAtOnce::LISTENER\_ID);

auto oneByOneListeners = getListeners(EventListenerTouchOneByOne::LISTENER\_ID);

auto allAtOnceListeners = getListeners(EventListenerTouchAllAtOnce::LISTENER\_ID);

// If there aren't any touch listeners, return directly.

if (nullptr == oneByOneListeners && nullptr == allAtOnceListeners)

return;

bool isNeedsMutableSet = (oneByOneListeners && allAtOnceListeners);

const std::vector<Touch\*>& originalTouches = event->getTouches();

std::vector<Touch\*> mutableTouches(originalTouches.size());

std::copy(originalTouches.begin(), originalTouches.end(), mutableTouches.begin());

//

// process the target handlers 1st

//

if (oneByOneListeners)

{

auto mutableTouchesIter = mutableTouches.begin();

auto touchesIter = originalTouches.begin();

for (; touchesIter != originalTouches.end(); ++touchesIter)

{

bool isSwallowed = false;

**// 一个匿名函数**

**auto onTouchEvent = [&](EventListener\* l) -> bool** { // Return true to break

EventListenerTouchOneByOne\* listener = static\_cast<EventListenerTouchOneByOne\*>(l);

// Skip if the listener was removed.

if (!listener->\_isRegistered)

return false;

event->setCurrentTarget(listener->\_node);

bool isClaimed = false;

std::vector<Touch\*>::iterator removedIter;

EventTouch::EventCode eventCode = event->getEventCode();

**if (eventCode == EventTouch::EventCode::BEGAN)**

**{**

**if (listener->onTouchBegan)**

**{**

**isClaimed = listener->onTouchBegan(\*touchesIter, event);**

**// 如果监听器的onTouchBegan返回true在讲该Touch放入listener中，主要是为了用于后续的Moved、End与Cancel事件判断**

**if (isClaimed && listener->\_isRegistered)**

**{**

**listener->\_claimedTouches.push\_back(\*touchesIter);**

**}**

**}**

**}**

**// 如果该监听器有确认的Touch事件，同时发现这个事件在Listener中存在，则调用相应的Moved、End和Cancel函数**

**else if (listener->\_claimedTouches.size() > 0**

**&& ((removedIter = std::find(listener->\_claimedTouches.begin(), listener->\_claimedTouches.end(), \*touchesIter)) != listener->\_claimedTouches.end()))**

{

isClaimed = true;

switch (eventCode)

{

case EventTouch::EventCode::MOVED:

if (listener->onTouchMoved)

{

listener->onTouchMoved(\*touchesIter, event);

}

break;

case EventTouch::EventCode::ENDED:

if (listener->onTouchEnded)

{

listener->onTouchEnded(\*touchesIter, event);

}

if (listener->\_isRegistered)

{

listener->\_claimedTouches.erase(removedIter);

}

break;

case EventTouch::EventCode::CANCELLED:

if (listener->onTouchCancelled)

{

listener->onTouchCancelled(\*touchesIter, event);

}

if (listener->\_isRegistered)

{

listener->\_claimedTouches.erase(removedIter);

}

break;

default:

CCASSERT(false, "The eventcode is invalid.");

break;

}

}

// If the event was stopped, return directly.

if (event->isStopped())

{

updateListeners(event);

return true;

}

CCASSERT((\*touchesIter)->getID() == (\*mutableTouchesIter)->getID(),

"touchesIter ID should be equal to mutableTouchesIter's ID.");

if (isClaimed && listener->\_isRegistered && listener->\_needSwallow)

{

if (isNeedsMutableSet)

{

mutableTouchesIter = mutableTouches.erase(mutableTouchesIter);

isSwallowed = true;

}

return true;

}

return false;

};

// 发送给OneByOneListener进行处理

**dispatchTouchEventToListeners(oneByOneListeners, onTouchEvent);**

if (event->isStopped())

{

return;

}

if (!isSwallowed)

++mutableTouchesIter;

}

}

//

// process standard handlers 2nd

//

if (allAtOnceListeners && mutableTouches.size() > 0)

{

auto onTouchesEvent = [&](EventListener\* l) -> bool{

EventListenerTouchAllAtOnce\* listener = static\_cast<EventListenerTouchAllAtOnce\*>(l);

// Skip if the listener was removed.

if (!listener->\_isRegistered)

return false;

event->setCurrentTarget(listener->\_node);

switch (event->getEventCode())

{

case EventTouch::EventCode::BEGAN:

if (listener->onTouchesBegan)

{

listener->onTouchesBegan(mutableTouches, event);

}

break;

case EventTouch::EventCode::MOVED:

if (listener->onTouchesMoved)

{

listener->onTouchesMoved(mutableTouches, event);

}

break;

case EventTouch::EventCode::ENDED:

if (listener->onTouchesEnded)

{

listener->onTouchesEnded(mutableTouches, event);

}

break;

case EventTouch::EventCode::CANCELLED:

if (listener->onTouchesCancelled)

{

listener->onTouchesCancelled(mutableTouches, event);

}

break;

default:

CCASSERT(false, "The eventcode is invalid.");

break;

}

// If the event was stopped, return directly.

if (event->isStopped())

{

updateListeners(event);

return true;

}

return false;

};

dispatchTouchEventToListeners(allAtOnceListeners, onTouchesEvent);

if (event->isStopped())

{

return;

}

}

updateListeners(event);

}

**void EventDispatcher::dispatchTouchEventToListeners**(EventListenerVector\* listeners, const std::function<bool(EventListener\*)>& onEvent)

{

bool shouldStopPropagation = false;

auto fixedPriorityListeners = listeners->getFixedPriorityListeners();

auto sceneGraphPriorityListeners = listeners->getSceneGraphPriorityListeners();

ssize\_t i = 0;

// priority < 0

if (fixedPriorityListeners)

{

CCASSERT(listeners->getGt0Index() <= static\_cast<ssize\_t>(fixedPriorityListeners->size()), "Out of range exception!");

if (!fixedPriorityListeners->empty())

{

for (; i < listeners->getGt0Index(); ++i)

{

auto l = fixedPriorityListeners->at(i);

if (l->isEnabled() && !l->isPaused() && l->isRegistered() && **onEvent(l)**)

{

**shouldStopPropagation = true; // 停止传播**

break;

}

}

}

}

auto scene = Director::getInstance()->getRunningScene();

if (scene && sceneGraphPriorityListeners)

{

if (!shouldStopPropagation)

{

// priority == 0, scene graph priority

// first, get all enabled, unPaused and registered listeners

std::vector<EventListener\*> sceneListeners;

for (auto& l : \*sceneGraphPriorityListeners)

{

if (l->isEnabled() && !l->isPaused() && l->isRegistered())

{

sceneListeners.push\_back(l);

}

}

// second, for all camera call all listeners

// get a copy of cameras, prevent it's been modified in listener callback

// if camera's depth is greater, process it earlier

auto cameras = scene->getCameras();

Camera\* camera;

for (int j = int(cameras.size()) - 1; j >= 0; --j)

{

camera = cameras[j];

if (camera->isVisible() == false)

{

continue;

}

Camera::\_visitingCamera = camera;

auto cameraFlag = (unsigned short)camera->getCameraFlag();

for (auto& l : sceneListeners)

{

if (nullptr == l->getAssociatedNode() || 0 == (l->getAssociatedNode()->getCameraMask() & cameraFlag))

{

continue;

}

**if (onEvent(l))**

**{**

**shouldStopPropagation = true;**

**break;**

**}**

}

if (shouldStopPropagation)

{

break;

}

}

Camera::\_visitingCamera = nullptr;

}

}

if (fixedPriorityListeners)

{

if (!shouldStopPropagation)

{

// priority > 0

ssize\_t size = fixedPriorityListeners->size();

for (; i < size; ++i)

{

auto l = fixedPriorityListeners->at(i);

if (l->isEnabled() && !l->isPaused() && l->isRegistered() && onEvent(l))

{

shouldStopPropagation = true;

break;

}

}

}

}

}

实际上所有的后续Moved、Ended与Cancel事件是否需要传递给该节点，关键点在于：

listener->\_claimedTouches.push\_back(\*touchesIter);

也就是每个监听器保存了当前的Touch，后面的事件通过监听器保存的进行对比判断是否应该发送给该节点，实际上对于多个事件，虽然是不同的实例，但是在只是在GLView的onTouchBegan创建了Touch实例并保存，在OnTouchMoved、onTouchEnd和onTouchCancel中只是在保存的列表中寻找，而并没有重新创建，因此上面的匿名函数用于保存保存的Touch进行比较是正确的，也就是虽然Event不是同一个，但是保存的Touch都是相同的指针