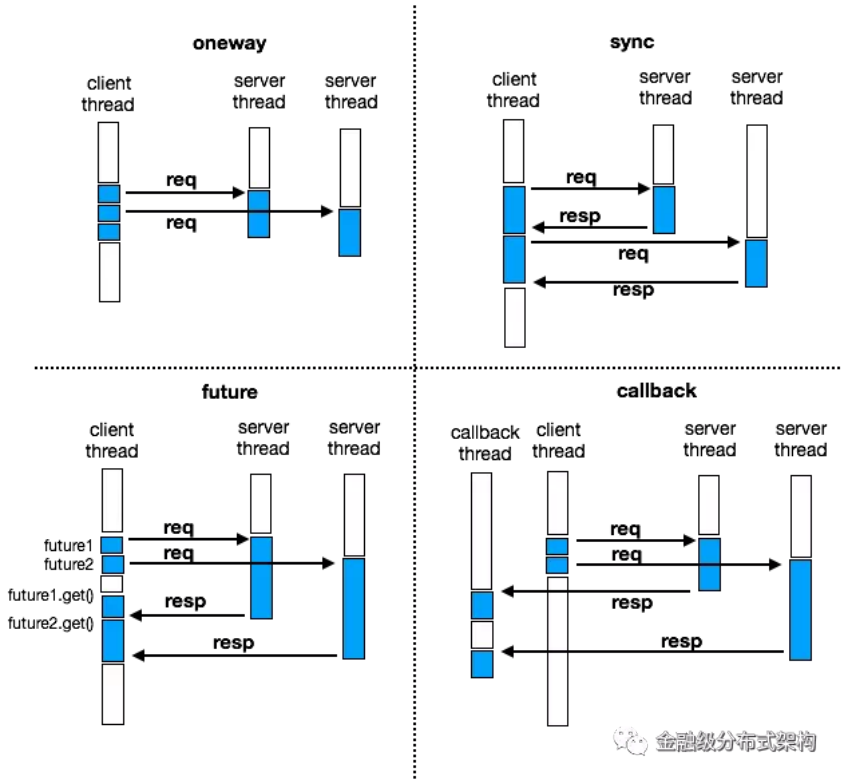
# 通信模型




图中都是ping/pong模式的通信，蓝色部分表示线程正在执行任务

oneway 不关心响应，请求线程不会被阻塞，但使用时需要注意控制调用节奏，防止压垮接收方；

sync 调用会阻塞请求线程，待响应返回后才能进行下一个请求。这是最常用的一种通信模型；

future 调用，在调用过程不会阻塞线程，但获取结果的过程会阻塞线程；

callback 是真正的异步调用，永远不会阻塞线程，结果处理是在异步线程里执行。

参考：https://mp.weixin.qq.com/s/JRsbK1Un2av9GKmJ8DK7IQ 蚂蚁通信框架实践

mpush通信模型，以及实现代码：

*通信模型：future

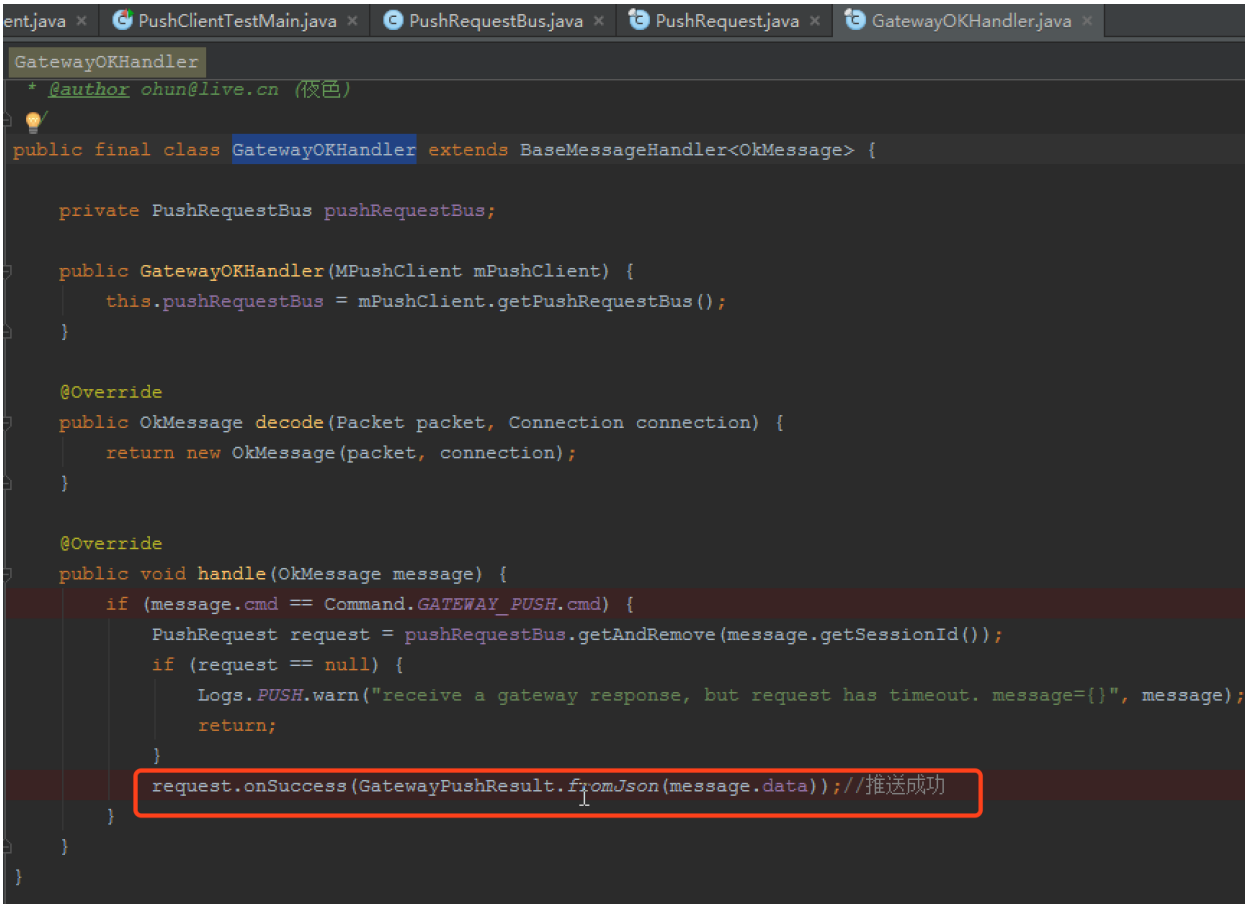
*代码实现：

PushClientTestMain#testPush()

```
FutureTask<PushResult> future = sender.send(context);
System.err.println("\n\n" + future.get());
```

*通信模型：callback

*代码实现：




```
public void onSuccess(GatewayPushResult result) {
    if (result != null) timeLine.addTimePoints(result.timePoints);
    submit(Status.success);
}
```

```java
private void submit(Status status) {
    if (this.status.compareAndSet(Status.init, status)) {//防止重复调用
        boolean isTimeoutEnd = status == Status.timeout;//任务是否超时结束

        if (future != null && !isTimeoutEnd) {//是超时结束任务不用再取消一次
            future.cancel(true);//取消超时任务
        }

        this.timeLine.end();//结束时间流统计
        super.set(getResult());//设置同步调用的返回结果

        if (callback != null) {//回调callback
            if (isTimeoutEnd) {//超时结束时，当前线程已经是线程池里的线程，直接调用callback
                callback.onResult(getResult());
            } else {//非超时结束时，当前线程为Netty线程池，要异步执行callback
                mPushClient.getPushRequestBus().asyncCall(this);//会执行run方法
            }
        }
    }
    LOGGER.info("push request {} end, {}, {}, {}", status, userId, location, timeLine);
}
```

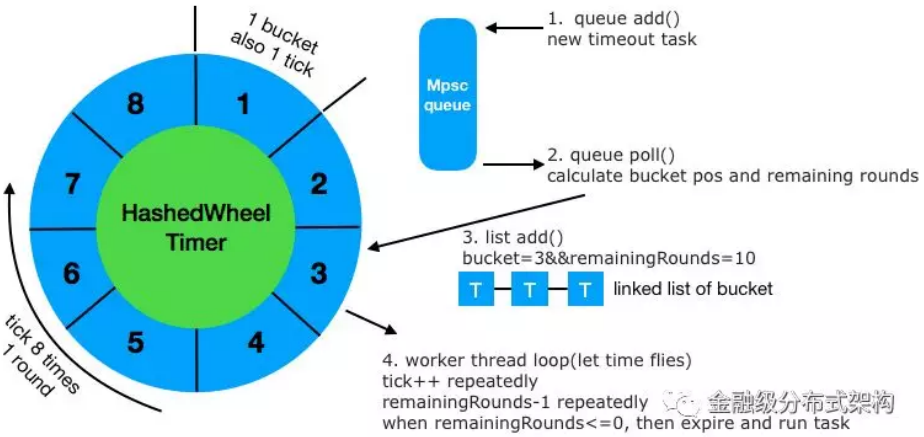
netty处理成功, 将当前对象用线程池执行

```java
/**
 * run方法会有两个地方的线程调用
 * 1. 任务超时时会调用，见PushRequestBus.I.put(sessionId, PushRequest.this);
 * 2. 异步执行callback的时候，见PushRequestBus.I.asyncCall(this);
 */
@Override
public void run() {
    //判断任务是否超时，如果超时了此时状态是init，否则应该是其他状态，因为从submit方法过来的状态都不是init
    if (status.get() == Status.init) {
        timeout();
    } else {
        callback.onResult(getResult());
    }
}
```

成功，则调用回调方法

# 超时控制

### 时间轮：（无锁队列）



除了 oneway 模式，其他三种通信模型都需要进行超时控制，我们同样采用 Netty 里针对超时机制，所设计的高效方案 HashedWheelTimer 。如图所示，其原理是首先在发起调用前，我们会新增一个超时任务 timeoutTask 到 MpscQueue （ Netty 实现的一种高效的无锁队列）里，然后在循环里，会不断的遍历 Queue 里的这些超时任务（每次最多10万），针对每个任务，会根据其设置的超时时间，来计算该任务所属于的 bucket 位置与剩余轮数 remainingRounds ，然后加入到对应 bucket 的链表结构里。随着 tick++ 的进行，时间在不断的增长，每 tick 8 次，就是 1 个时间轮 round。当对应超时任务的 remainingRounds减到 0 时，就是触发这个超时任务的时候，此时再执行其 run() 方法，做超时逻辑处理。

最佳实践：通常一个进程使用一个HashedWheelTimer实例，采用单例模型即可。

参考：https://mp.weixin.qq.com/s/JRsbK1Un2av9GKmJ8DK7IQ 蚂蚁通信框架实践

```
//NettyHttpClient.java

private Timer timer;
@Override
protected void doStart(Listener listener) throws Throwable {
 timer = new HashedWheelTimer(new NamedThreadFactory(T_HTTP_TIMER), 1, TimeUnit.SECONDS, 64);
}
@Override
public void request(RequestContext context) throws Exception {
  //2.添加请求超时检测队列
  timer.newTimeout(context, context.readTimeout, TimeUnit.MILLISECONDS);
 }
```

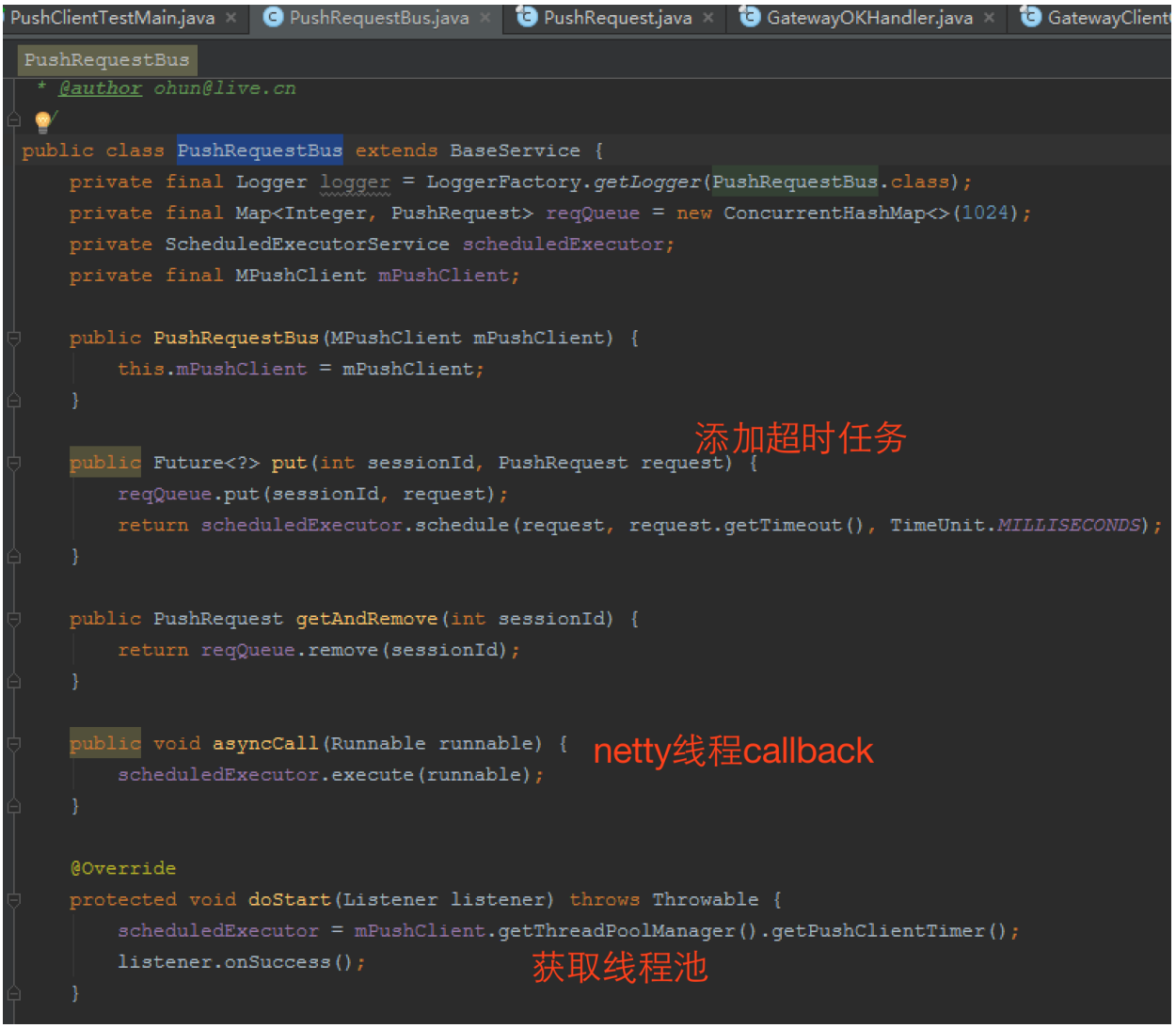
```
//RequestContext.java

import io.netty.util.Timeout;
import io.netty.util.TimerTask;
public class RequestContext implements TimerTask, HttpCallback {
 @Override
 public void run(Timeout timeout) throws Exception {
 if (tryDone()) {
 if (callback != null) {
  callback.onTimeout();
  }
  }
```

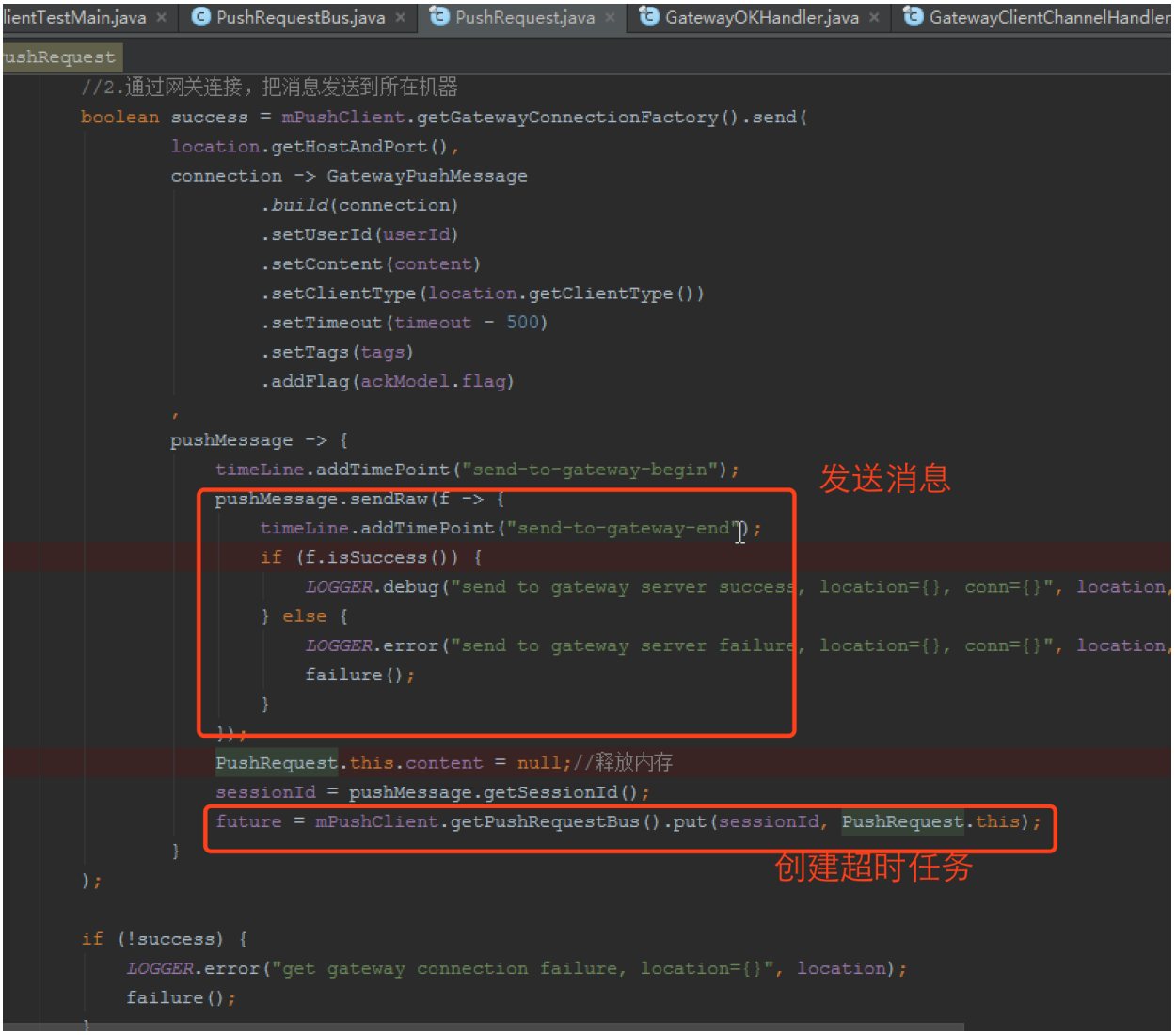
```
13    }
14    /**
15    * 由于检测请求超时的任务存在，为了防止多线程下重复处理
16    * @return
17    */
18    public boolean tryDone() {
19    return cancelled.compareAndSet(false, true);
20    }
21  }
```

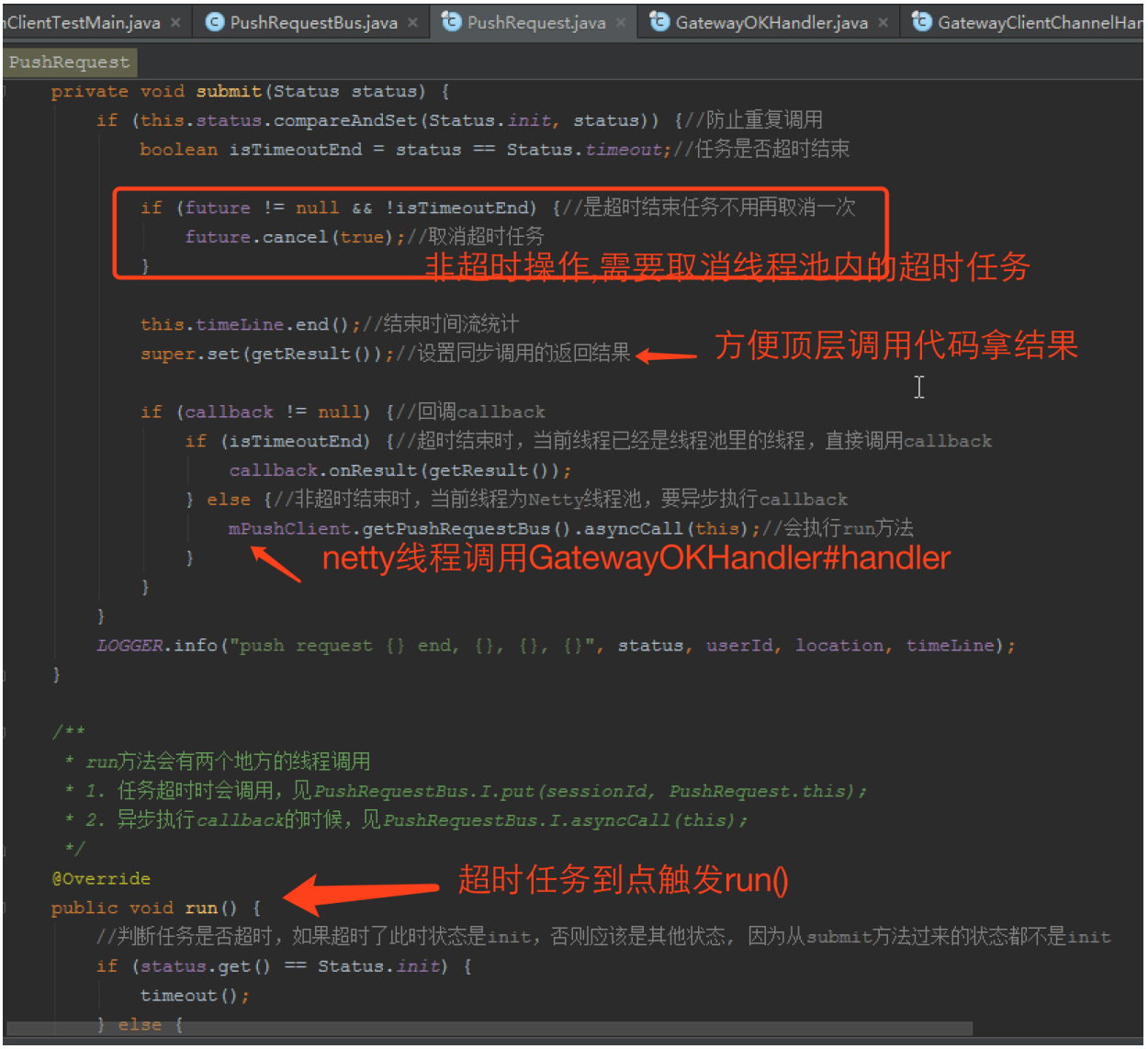
**线程池任务调度：（有锁队列）**

ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(push_client
, new NamedPoolThreadFactory(T_PUSH_CLIENT_TIMER), (r, e) -> r.run() // run caller thread
);
executor.setRemoveOnCancelPolicy(true);//取消任务时，同时删除队列中的任务
//schedule方法执行一次，内部为延迟队列
executor.schedule(request, request.getTimeout(), TimeUnit.MILLISECONDS);

mpush超时控制，以及实现代码：
*超时控制：线程池任务调度
*代码实现
PushRequestBus.java
ThreadPoolManager.java
ClientExecutorFactory.java
PushRequest.java
GatewayOKHandler.java


PushClientTestMain.java
PushRequestBus.java
PushRequest.java
GatewayOKHandler.java
PushRequestBus
* @author ohun@live.cn
public class PushRequestBus extends BaseService {
private final Logger logger = LoggerFactory.getLogger(PushRequestBus.class);
private final Map<Integer, PushRequest> reqQueue = new ConcurrentHashMap<>(1024);
private ScheduledExecutorService scheduledExecutor;
private final MPushClient mPushClient;
public PushRequestBus(MPushClient mPushClient) {
this.mPushClient = mPushClient;
}
添加超时任务
public Future<?> put(int sessionId, PushRequest request) {
reqQueue.put(sessionId, request);
return scheduledExecutor.schedule(request, request.getTimeout(), TimeUnit.MILLISECONDS);
}
public PushRequest getAndRemove(int sessionId) {
return reqQueue.remove(sessionId);
}
public void asyncCall(Runnable runnable) {
netty线程callback
scheduledExecutor.execute(runnable);
}
@Override
protected void doStart(Listener listener) throws Throwable {
scheduledExecutor = mPushClient.getThreadPoolManager().getPushClientTimer();
listener.onSuccess();
获取线程池
}


lientTestMain.java
PushRequestBus.java
PushRequest.java
GatewayOKHandler.java
GatewayClientChannelHandler
ushRequest
//2.通过网关连接，把消息发送到所在机器
boolean success = mPushClient.getGatewayConnectionFactory().send(
location.getHostAndPort(),
connection -> GatewayPushMessage
.build(connection)
.setUserId(userId)
.setContent(content)
.setClientType(location.getClientType())
.setTimeout(timeout - 500)
.setTags(tags)
.addFlag(ackModel.flag)
,
pushMessage -> {
timeLine.addTimePoint("send-to-gateway-begin");
pushMessage.sendRaw(f -> {
timeLine.addTimePoint("send-to-gateway-end");
if (f.isSuccess()) {
LOGGER.debug("send to gateway server success, location={}, conn={}", location,
} else {
LOGGER.error("send to gateway server failure, location={}, conn={}", location,
failure();
}
});
PushRequest.this.content = null;//释放内存
sessionId = pushMessage.getSessionId();
future = mPushClient.getPushRequestBus().put(sessionId, PushRequest.this);
}
);
if (!success) {
LOGGER.error("get gateway connection failure, location={}", location);
failure();
}
发送消息
创建超时任务


ClientTestMain.java
PushRequestBus.java
PushRequest.java
GatewayOKHandler.java
GatewayClientChannelHar
PushRequest
private void submit(Status status) {
if (this.status.compareAndSet(Status.init, status)) {//防止重复调用
boolean isTimeoutEnd = status == Status.timeout;//任务是否超时结束
if (future != null && !isTimeoutEnd) {//是超时结束任务不用再取消一次
future.cancel(true);//取消超时任务
}
非超时操作,需要取消线程池内的超时任务
this.timeLine.end();//结束时间流统计
super.set(getResult());//设置同步调用的返回结果
方便顶层调用代码拿结果
if (callback != null) {//回调callback
if (isTimeoutEnd) {//超时结束时，当前线程已经是线程池里的线程，直接调用callback
callback.onResult(getResult());
} else {//非超时结束时，当前线程为Netty线程池，要异步执行callback
mPushClient.getPushRequestBus().asyncCall(this);//会执行run方法
}
netty线程调用GatewayOKHandler#handler
}
}
LOGGER.info("push request {} end, {}, {}, {}", status, userId, location, timeLine);
}
/**
* run方法会有两个地方的线程调用
* 1. 任务超时时会调用，见PushRequestBus.I.put(sessionId, PushRequest.this);
* 2. 异步执行callback的时候，见PushRequestBus.I.asyncCall(this);
*/
@Override
public void run() {
超时任务到点触发run()
//判断任务是否超时，如果超时了此时状态是init，否则应该是其他状态，因为从submit方法过来的状态都不是init
if (status.get() == Status.init) {
timeout();
} else {