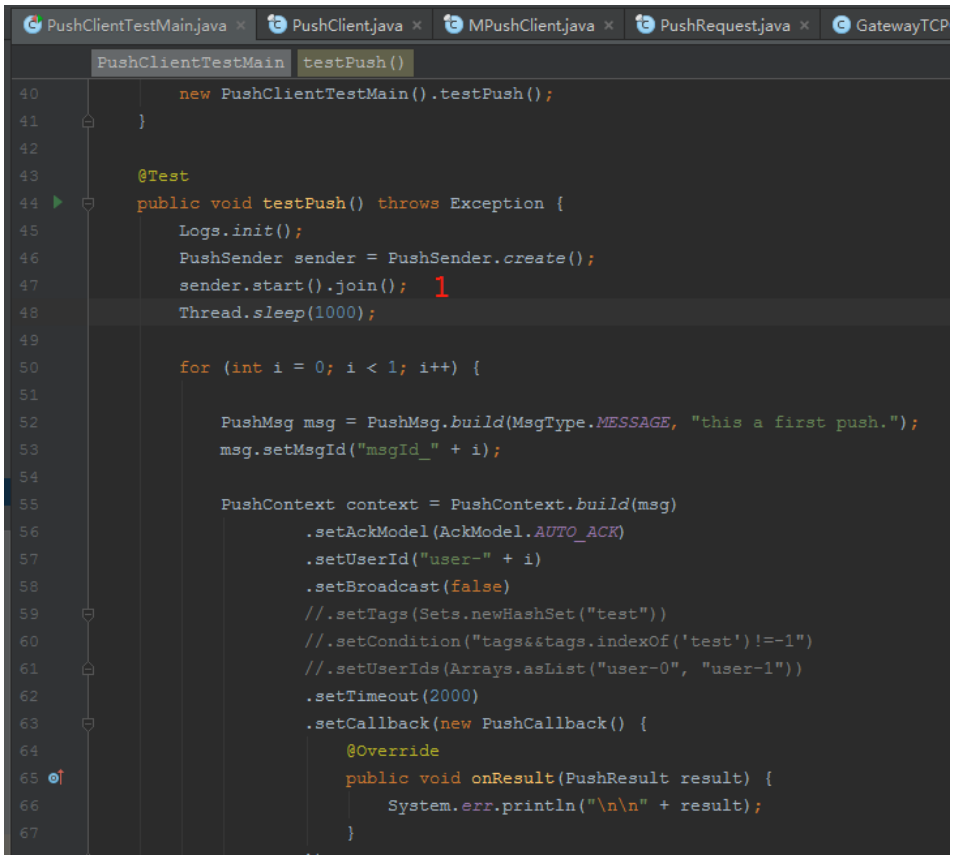
PushClientTestMain.java




1、start()方法进入PushClient的父类BaseService#start()方法，BaseService#start(listener)方法，子类PushClient#doStart()方法；

**BaseService#start()方法:**

初始化FutureListener，继承JDK中的CompletableFuture类，主要是防止重复启动多个服务，并利用.join()等待所有的(PushClient#doStart()方法内的)启动动作都完成；

```
public final CompletableFuture<Boolean> start() {
    FutureListener listener = new FutureListener(started);
    start(listener);
    return listener;
}
```

**PushClient#doStart()方法:**

```java
@Override
protected void doStart(Listener listener) throws Throwable {
    if (mPushClient == null) {
        mPushClient = new MPushClient();  1
    }

    pushRequestBus = mPushClient.getPushRequestBus(); 2
    cachedRemoteRouterManager = mPushClient.getCachedRemoteRouterManager();3
    gatewayConnectionFactory = mPushClient.getGatewayConnectionFactory(); 4

    ServiceDiscoveryFactory.create().syncStart(); 5
    CacheManagerFactory.create().init();  6
    pushRequestBus.syncStart();  7
    gatewayConnectionFactory.start(listener);8
}
```

```java
public MPushClient() {
    monitorService = new MonitorService(); 1.1

    EventBus.create(monitorService.getThreadPoolManager().getEventBusExecutor());
                                                                        1.2
    pushRequestBus = new PushRequestBus(this); 1.3

    cachedRemoteRouterManager = new CachedRemoteRouterManager(); 1.4

    gatewayConnectionFactory = GatewayConnectionFactory.create(this); 1.5
}
```

1、初始化MPushClient

1.1 初始化监控服务MonitorService

初始化并不启动ThreadPoolManager连接池工厂(集中管理连接池)、初始化并不启动监控收集器ResultCollector(JVM/连接池)

1.2 用初始化好的连接池对象，创建guava的EventBus实例AsyncEventBus

1.3 初始化PushRequestBus服务

主要用于发送消息的超时控制、异步回调

1.4 初始化cachedRemoteRouterManager对象

主要是查找远程路由信息，并缓存一份到本地内存中(guava的cache)；

1.5 创建网关连接

主要是根据配置创建GatewayTCPConnectionFactory或者GatewayUDPConnectionFactory连接工厂，实现获取连接、普通消息/广播消息发送；

2、获取已经初始化好的PushRequestBus对象

3、获取已经初始化好的cachedRemoteRouterManager对象

4、获取已经初始化好的gatewayConnectionFactory对象

5、初始化服务发现实现类，并调用syncStart启动

利用SPI机制，找到ServiceDiscoveryFactory接口实现SimpleDiscoveryFactory或者ZKDiscoveryFactory，得到FileSrd(本地cache.dat)或者ZKServiceRegistryAndDiscovery实例;

调用FileSrd或者ZKServiceRegistryAndDiscovery的doStart(Listener listener)方法启动（实在搞不懂为什么这么设计，有的有doStart，有的没有，这种抽象有点蛋疼）；

6、初始化缓存实现类，并调用init()

利用SPI机制，找到CacheManagerFactory接口实现SimpleCacheMangerFactory或者RedisCacheManagerFactory，得到FileCacheManger或者RedisManager实例，该实例都是些缓存的存取操作；

调用FileCacheManger或者RedisManager的init()方法进行缓存相关始化；

7、启动PushRequestBus服务

调用doStart()方法，启动监听、获得pushClient连接池对象；

8、启动gatewayConnectionFactory网关服务

调用GatewayTCPConnectionFactory或者GatewayUDPConnectionFactory的start(listener)

这里的设计也有些奇怪，为什么不是直接调用start()，而是start(Listener)

GatewayTCPConnectionFactory doStart()

```
private final AttributeKey<String> attrKey = AttributeKey.valueOf("host_port");
private final Map<String, List<Connection>> connections = Maps.newConcurrentMap();

private ServiceDiscovery discovery;
private GatewayClient gatewayClient;

private MPushClient mPushClient;

public GatewayTCPConnectionFactory(MPushClient mPushClient) {
    this.mPushClient = mPushClient;
}

@Override
protected void doStart(Listener listener) throws Throwable {
    EventBus.register(this);    8.1

    gatewayClient = new GatewayClient(mPushClient);8.2
    gatewayClient.start().join();8.3
    discovery = ServiceDiscoveryFactory.create();    8.4
    discovery.subscribe(GATEWAY_SERVER, this);8.5
    discovery.lookup(GATEWAY_SERVER).forEach(this::syncAddConnection);8.6
    listener.onSuccess();8.7
}
```

8.1、注册本对象到EventBus，同时@Subscribe订阅ConnectionConnectEvent事件

8.2、client实例化


8.2.1 消息转发类初始化

MessageDispatcher#register注册消息命令和对应的消息处理类；

MessageDispatcher#onReceive消息处理，根据接收到的消息命令，得到相应的消息处理类方法进行处理

8.2.2 注册消息发送成功处理类GatewayOKHandler

发送成功有两种情况：成功(返回结果)、成功且超时(不做处理)

8.2.3 注册消息发送失败处理类GatewayErrorHandler

发送失败有几种情况：超时(不做处理)、用户离线、发送失败、用户路由信息更改

8.2.4 初始化Netty连接池管理

存放connection到map中，并提供对conn的查找、添加、删除操作；

8.2.5 初始化Netty的channel处理类

该类标注@ChannelHandler.Sharable，是一个共享的处理器；

**channelActive事件方法**：建立连接成功，初始化NettyConnection并加入到连接池、EventBus推送 ConnectionConnectEvent事件到GatewayTCPConnectionFactory#on()方法;

**channelRead事件方法**：接收到(解码后的)消息，调用MessageDispatcher#onReceive做消息处理

**exceptionCaught方法**：打印conn错误日志，关闭ChannelHandlerContext上下文

**channelInactive事件方法**：连接关闭，删除连接池中的NettyConnection、EventBus推送ConnectionCloseEvent事件到LocalRouterManager#on()、RemoteRouterManager#on()

8.2.6 初始化流浪整形线程池

8.2.7 初始化Netty全局流量整形处理类GlobalChannelTrafficShapingHandler

针对所有的Channel, 通过参数设置：报文的接收速率、报文的发送速率、整形周期

8.3、client启动（Netty客户端启动）

因GatewayClient 继承NettyTCPClient，并且未重写父类的doStart()方法，所以这里的gatewayClient.start()其实是调用NettyTCPClient类中的doStart()方法

```
@Override
protected void doStart(Listener listener) throws Throwable {
    if (useNettyEpoll()) {
        createEpollClient(listener);
    } else {
        createNioClient(listener);
    }
}
```

```
private void createNioClient(Listener listener) {
    NioEventLoopGroup workerGroup = new NioEventLoopGroup(
            getWorkThreadNum(), new DefaultThreadFactory(ThreadNames.T_TCP_CLIENT), getSelectorProvider()
    );
    workerGroup.setIoRatio(getIoRate());
    createClient(listener, workerGroup, getChannelFactory());
}

private void createEpollClient(Listener listener) {
    EpollEventLoopGroup workerGroup = new EpollEventLoopGroup(
            getWorkThreadNum(), new DefaultThreadFactory(ThreadNames.T_TCP_CLIENT)
    );
    workerGroup.setIoRatio(getIoRate());
    createClient(listener, workerGroup, EpollSocketChannel::new);
}
```

根据配置中心的设置，创建NioEventLoopGroup或者EpollEventLoopGroup，EventLoop主要工作是注册channel，并负责channel中读写等事件，这就涉及到不同的监听方式，linux下有3种事件监听方式：select/poll/epoll

而Netty呢则使用如下：

- NioEventLoop：采用的是jdk Selector接口（使用PollSelectorImpl的poll方式）来实现对Channel的事件检测
- EpollEventLoop：没有采用jdk Selector的接口实现EPollSelectorImpl，而是Netty自己实现的epoll方式来实现对Channel的事件检测，所以在EpollEventLoop中就不存在jdk的Selector。

### 2.2.1 NioEventLoop介绍

对于NioEventLoopGroup的功能，NioEventLoop都要做实际的实现，NioEventLoop既要实现注册功能，又要实现运行Runnable任务

对于注册Channel：NioEventLoop将Channel注册到NioEventLoop内部的PollSelectorImpl上，来监听该Channel的读写事件

对于运行Runnable任务：NioEventLoop的父类的父类SingleThreadEventExecutor实现了运行Runnable任务，在SingleThreadEventExecutor中，有一个任务队列还有一个分配的线程

```
private final Queue<Runnable> taskQueue;
private volatile Thread thread;
```

NioEventLoop在该线程中不仅要执行Selector带来的IO事件，还要不断的从上述taskQueue中取出任务来执行这些非IO事件。下面我们来详细看下这个过程

https://my.oschina.net/pingpangkuangmo/blog/742929

**关于ioRatio:** (默认50，io线程50% 业务线程50%，100代表100%IO线程)
用户为了简化线程处理模型，把所有的业务任务封装成Task，丢到Nett用的I/O线程NioEventLoop中执行。为了防止过多的业务任务阻塞I/O线程的网络读写操作，NioEventLoop提供了设置I/O任务和非I/O任务的处理比例，通过合理的调整处理比例，来保证更合理的资源调度。
关于IO线程与业务线程的精细控制，可以参考：
https://mp.weixin.qq.com/s/Qia4LzaWLB_qIXlOek9DJg 蚂蚁金服通信框架SOFABolt解析 | 序列化机制(Serializer)

```
private void createClient(Listener listener, EventLoopGroup workerGroup, ChannelFactory<? extends Channel> channelFactory) {
    this.workerGroup = workerGroup;
    this.bootstrap = new Bootstrap();
    bootstrap.group(workerGroup)//
            .option(ChannelOption.SO_REUSEADDR, true)//
            .option(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)//
            .channelFactory(channelFactory);
    bootstrap.handler(new ChannelInitializer<Channel>() { // (4)
        @Override
        public void initChannel(Channel ch) throws Exception {
            initPipeline(ch.pipeline());
        }
    });
    initOptions(bootstrap);
    listener.onSuccess();
}

public ChannelFuture connect(String host, int port) {
    return bootstrap.connect(new InetSocketAddress(host, port));
}

public ChannelFuture connect(String host, int port, Listener listener) {
    return bootstrap.connect(new InetSocketAddress(host, port)).addListener(f -> {
        if (f.isSuccess()) {
            if (listener != null) listener.onSuccess(port);
            LOGGER.info("start netty client success, host={}, port={}", host, port);
        } else {
            if (listener != null) listener.onFailure(f.cause());
            LOGGER.error("start netty client failure, host={}, port={}", host, port, f.cause());
        }
    });
```

8.4 初始化服务发现实例

根据SPI指定的实现类初始化FileSrd或者ZKServiceRegistryAndDiscovery

8.5 订阅注册的服务节点信息变化

调用ZKServiceRegistryAndDiscovery#subscribe方法，订阅节点添加、删除、修改事件；

8.6 查找注册的服务，并针对每个服务节点创建多个conn连接