更新说明

此文档是 CacheKit 项目的整体文档，以此文档为准。

最新消息请及时关注：www.cachekit.com

更新日期：2020 年 6 月

前言

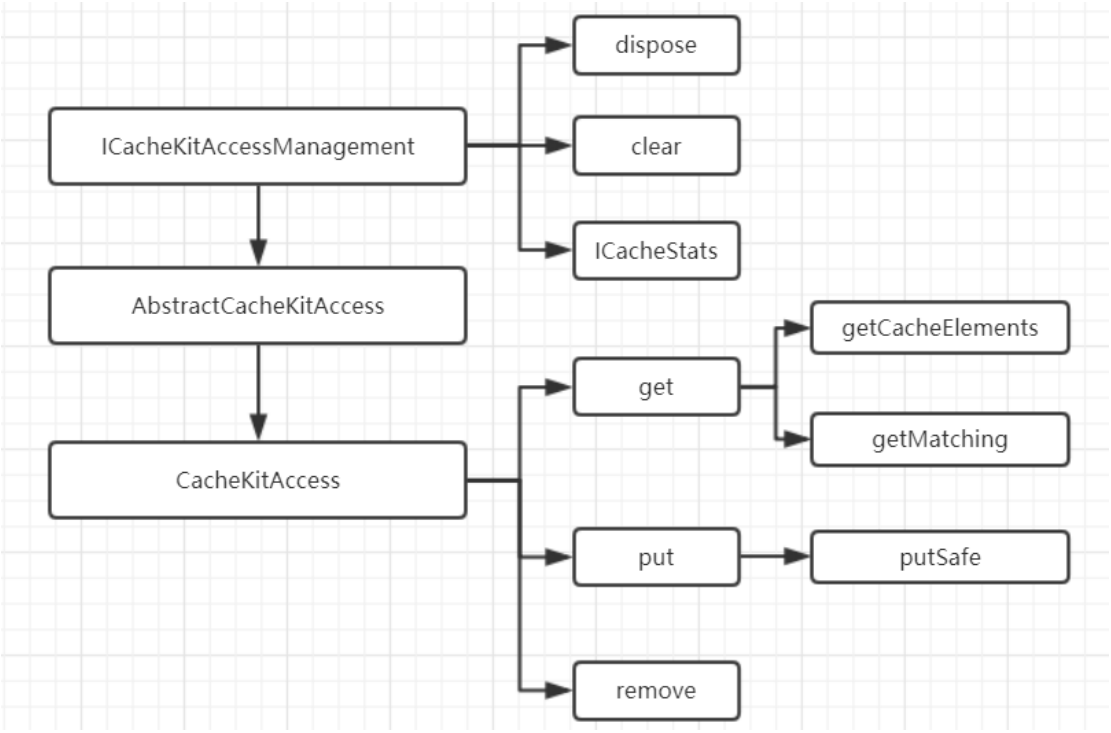
编码能力的提升，依照量变引发质变的规律。而量又分为两个层次：代码量和时间量。代码量包括几百个类，几十万行代码的级别；时间量包括一年又一年的温故知新，以年为单位的沉淀和反思。

古语有云：操千曲而后晓声，观千剑而后识器。对此，笔者深有感触，遥想十年前为了考六级，曾熟背过二百篇英语短文，至此才明白，英语是怎么回事。后来，机缘巧合，笔者潜心研究分布式和缓存领域，断断续续数年的光阴已过，越发认识到项目的长久价值。长久价值，就是因为长久才能产生的价值。

虽然 IT 技术日新月异，但是它不是你的内功，两耳不闻窗外，一心发展内力，这才是明智之举，长远之路。切勿人云己云，随波逐流。

1、CacheKit 访问层介绍

1.1、访问层架构图



1.2、写作手法鉴赏

AbstractCacheKitAccess<K， V>，虽然为抽象函数，但是类体并没有抽象方法，此处写作手法在于：避免使用此类创建对象。

2、函数调用变事件队列的异步处理的写作手法鉴赏

2.1、函数的深入分析

首先，我们要把函数当做一个高度复杂的生命体来看待，把它解剖一下，看看其内部的各个零件或者器官。以下面函数为例：

```
public class Printer
{
    public void print(String msg)
    {
        System.out.println(msg);
    }
}
```

其组成部分包括：

（1）函数的参数，即：String msg

（2）参数的名称，即：print

以上两部分可以包装成事件和事件类型，如下所示：

```
//事件类型
public enum EventType
{
    PRINT, //打印
    COPY //扫描
}

public class Event<T> extends EventObject implements Serializable
{
    private static final long serialVersionUID = 1L;

    private EventType eventType = EventType.PRINT;

    public Event(T param, EventType type)
    {//注意：param表示函数的参数
        super(param);
        this.eventType = type;
    }

    public EventType getEventType()
    {
        return eventType;
    }

    @SuppressWarnings("unchecked")

    public T getParam()
    {
        return (T) super.getSource();

    }
}
```

（3）函数的执行体，可以包装成事件处理器：

```
//事件处理器接口
public interface IEventHandler
{
```

```
    <T> void handleEvent(Event<T> event);
}


//具体的事件处理器实现类
public class PrintEventHandler implements IEventHandler
{

    private Printer printer = new Printer();

    public <T> void handleEvent(Event<T> event)
    {
        String param = (String) event.getParam();

        printer.print(param);
    }
}
```

（4）函数的调用过程：函数参数+函数执行，可以包装成事件运行器：

```
//事件运行器
public class EventRunner implements Runnable
{
    private final IEventHandler handler;

    private final Event<?> event;

    EventRunner(IEventHandler hand, Event<?> event)
    {
        this.handler = hand;
        this.event = event;
    }

    public void run()
    {
        handler.handleEvent(event);
    }
}
```

（5）函数的附着体，原先是 Printer，而现在变成了事件队列：

```
//事件队列
```

```
public class EventQueue
{
    private static final String THREAD_PREFIX = "EventQueue";

    private boolean destroyed = false;

    private LinkedBlockingQueue<Runnable> queue;

    private ThreadPoolExecutor queueProcessor;

    public EventQueue()
    {
        queue = new LinkedBlockingQueue<Runnable>();

        queueProcessor = new ThreadPoolExecutor(1, 1, 0L,
TimeUnit.MILLISECONDS, queue,
                new EventThreadFactory(THREAD_PREFIX));

    }

    public void addEventRunner(EventRunner runner)
    {

        if (!destroyed)
        {
            queueProcessor.execute(runner);
        }
    }

    public void dispose()
    {
        if (!destroyed)
        {
            destroyed = true;

            queueProcessor.shutdownNow();
            queueProcessor = null;
        }
    }
}
```

（6）补充：线程工厂类

```
import java.util.concurrent.ThreadFactory;

public class EventThreadFactory implements ThreadFactory
{
    private String prefix;
    private boolean threadIsDaemon = true;
    private int threadPriority = Thread.NORM_PRIORITY;

    public EventThreadFactory(String prefix)
    {
        this(prefix, Thread.NORM_PRIORITY);
    }

    public EventThreadFactory(String prefix, int threadPriority)
    {
        this.prefix = prefix;
        this.threadPriority = threadPriority;
    }

    @Override
    public Thread newThread(Runnable runner)
    {
        Thread thread = new Thread(runner);
        String name = thread.getName();
        thread.setName(prefix + name);
        thread.setDaemon(threadIsDaemon);
        thread.setPriority(threadPriority);
        return thread;
    }
}
```

2.2、函数的用法对比

在未改造之前，也就是未将函数变事件改造之前，函数的用法是这样的：

```
public class Client
{
    public static void main(String[] args)
    {
        Printer printer = new Printer();
        printer.print("hell world!");
```

```
    }
}
```

而改造后，函数变成了事件，异步处理的过程是这样的：

```
public class ClientEvent
{
    public static void main(String[] args)
    {

        EventQueue eventQueue = new EventQueue();

        IEventHandler handler = new PrintEventHandler();

        Event<String> event = new Event<String>("hello world",
EventType.PRINT);

        EventRunner runner = new EventRunner(handler, event);

        eventQueue.addEventRunner(runner);

        eventQueue.dispose();
    }
}
```

2.3、小作坊编码思想与工业化编码思想

虽然经过上面的改造，代码变得更繁琐了，但是获得了思想上的跃迁：从小作坊编码思想变成了工业化编程思想。

我们在工作中的代码，往往都是小作坊思想搞出来的代码，其营养价值很低。而立足于工业化思想所完成的功能开发，能让人体会编码艺术之美。

具体什么是小作坊编码思想，什么是工业化编码思想，后文将会有更多的介绍，敬请期待。

3、统计部分

```
public class StatElement<V> implements IStatElement<V>
{

    private static final long serialVersionUID = 1L;
```

```
    private String name = null;

    private V data = null;

    public StatElement(String name, V data)
    {
        super();
        this.name = name;
        this.data = data;
    }
}

public class Stats implements IStats
{
    private static final long serialVersionUID = 1L;

    private List<IStatElement<?>> stats = null;

    private String typeName = null;

}
```

4、内存组件

4.1、内存组件操作接口：三大操作

```
void update(ICacheElement<K, V> ce) throws IOException;

void update(ICacheElement<K, V> ce) throws IOException;

boolean remove(K key) throws IOException;
```

4.2、抽象类 AbstractMemoryCache

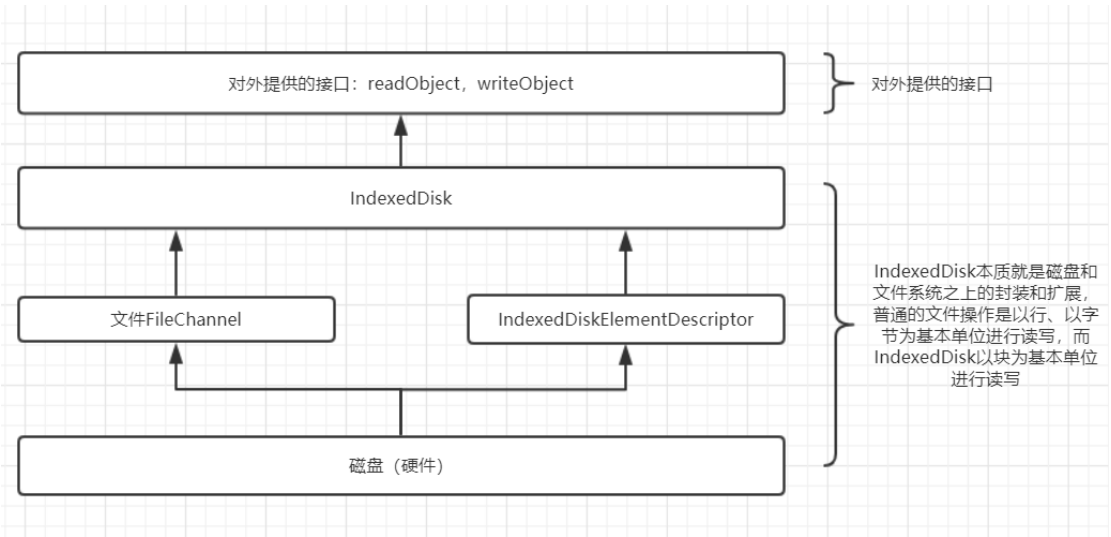
5、线程工厂

线程与异常有点类似，我们可控的东西并不多。即便如此，我们仍然可以给线程打上烙印，以表明它是从哪个工厂产生的。

需要注意一点：虽然从同一工厂产生的线程，但是它们并不是属于同一线程组。

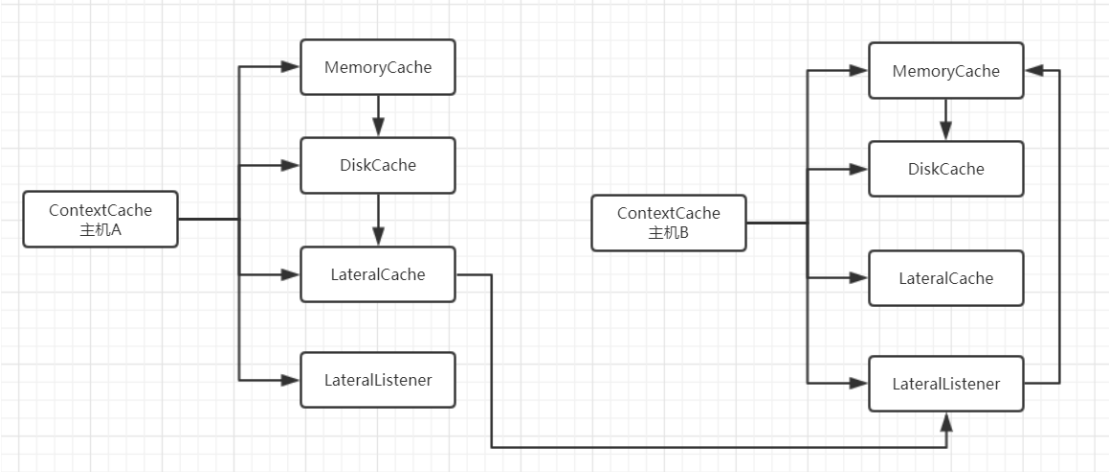
```
public class CacheKitThreadFactory implements ThreadFactory
{
    private String prefix;
    private boolean threadIsDaemon = true;
    private int threadPriority = Thread.NORM_PRIORITY;

    public CacheKitThreadFactory(String prefix)
    {
        this(prefix, Thread.NORM_PRIORITY);
    }

    public CacheKitThreadFactory(String prefix, int threadPriority)
    {
        this.prefix = prefix;
        this.threadPriority = threadPriority;
    }

    @Override
    public Thread newThread(Runnable runner)
    {
        Thread thread = new Thread(runner);
        String threadName = thread.getName();
        thread.setName(prefix + threadName);
        thread.setDaemon(threadIsDaemon);
        thread.setPriority(threadPriority);
        return thread;
    }
}
```

6、磁盘组件

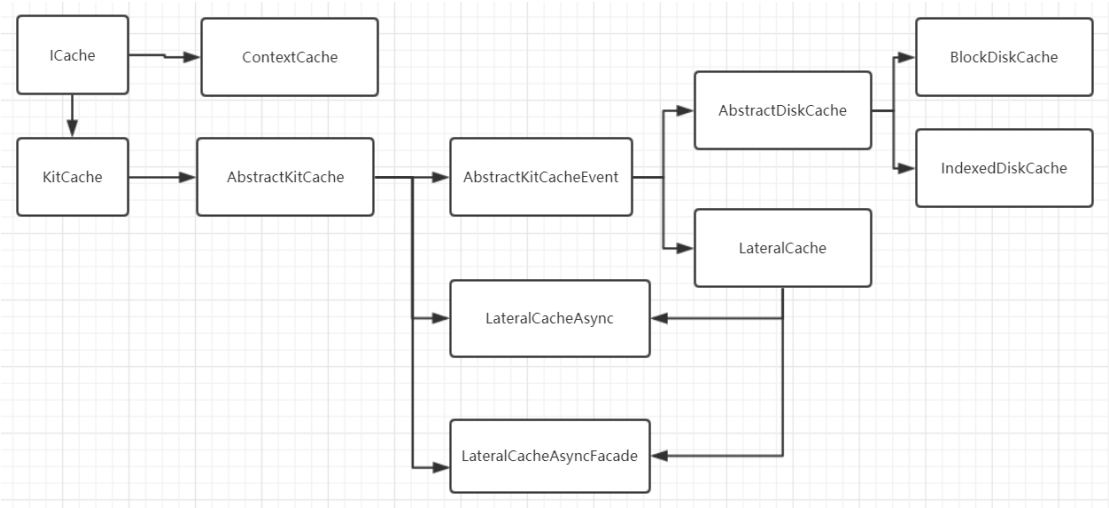


7、线性组件

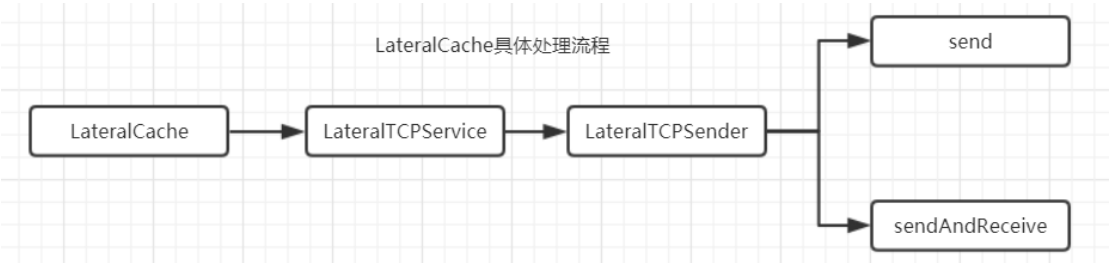
7.1 组件之间通信

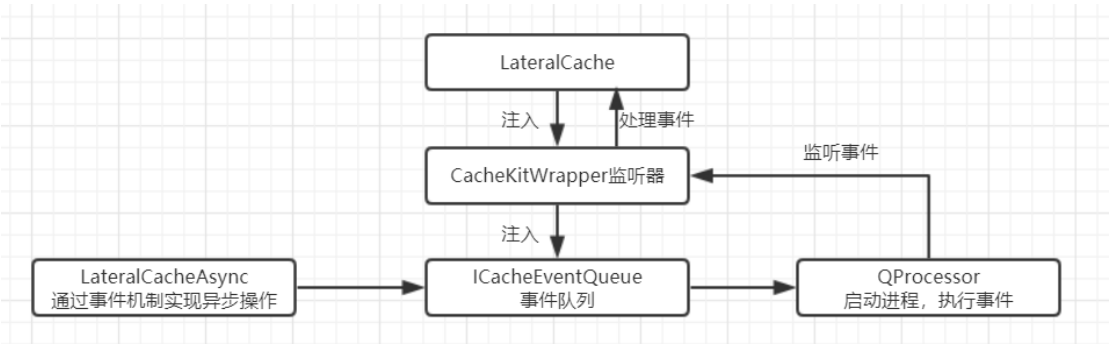


7.2 异步组件所处的位置：层次延伸与层次推进

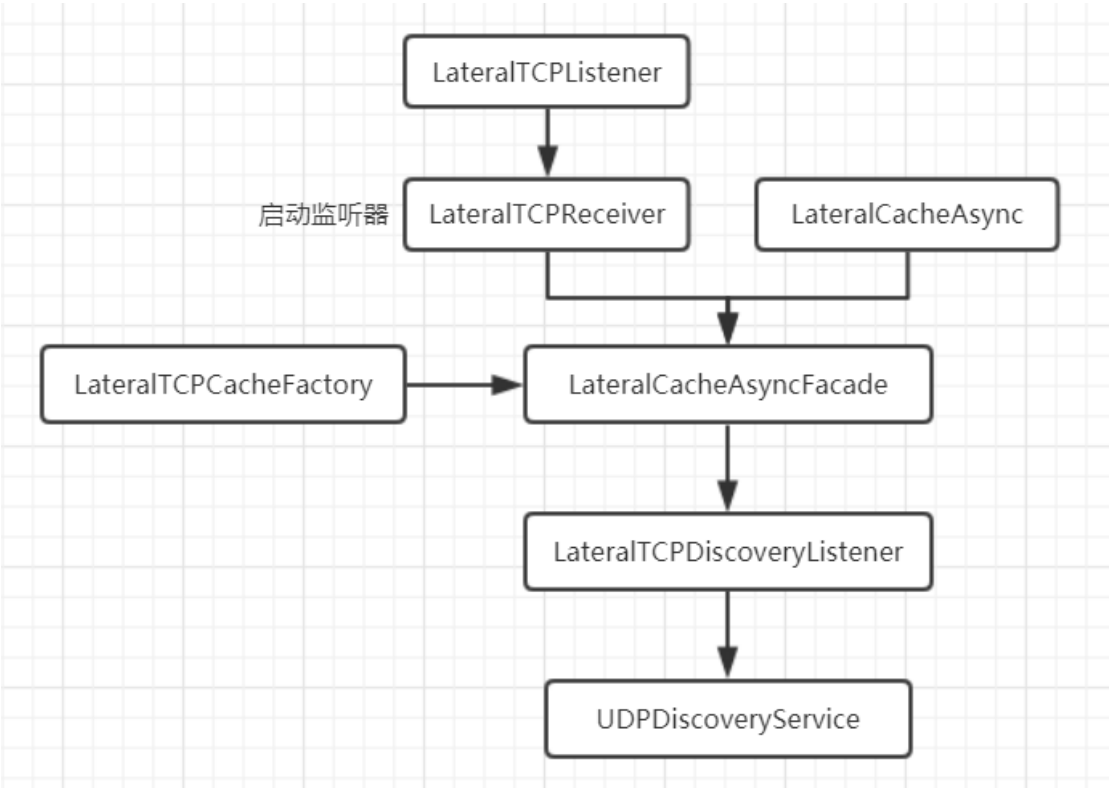


7.3 异步处理机制流程细节





7.4 线性组件的启动入口



8、广播的技术要点

8.1 UDPDiscoverySender 类的技术要点

MulticastSocket 提供了如下三个构造器：
public MulticastSocket() 使用本机默认地址、随机端口来创建 MulticastSocket 对象。
public MulticastSocket(int portNumber) 用本机默认地址，指定端口来创建 MulticastSocket 对象。
public MulticastSocket(SocketAddress bindaddr) 用指定 IP 地址，指定端口来创建 MulticastSocket 对象。

DatagramPacket 类用来表示数据包， DatagramPacket 类的构造函数有：
DatagramPacket(byte[]buf, intlength) 创建 DatagramPacket 对象，指定了数据包的内存空间和大小

DatagramPacket(byte[]buf, intlength，InetAddressaddress，intport) 不仅指定了数据包的内存空间和大小，而且指定了数据包的目标地址和端口。

在发送数据时，必须指定接收方的 Socket 地址和端口号，因此使用第 2 个构造函数可创建发送数据的 DatagramPacket 对象。

编码细节：

UDPDiscoverySender(String host, int port)，目标方的 IP 和端口，其获取自属性文件，而属性文件映射自 xml 配置文件：
getUdpDiscoveryAttributes().getUdpDiscoveryAddr(),
getUdpDiscoveryAttributes().getUdpDiscoveryPort()

protected void passiveBroadcast(String host, int port,
ArrayList<String> cacheNames, long listenerId)，发送方的IP和端口，取自：
getUdpDiscoveryAttributes().getServiceAddress(),
getUdpDiscoveryAttributes().getServicePort()

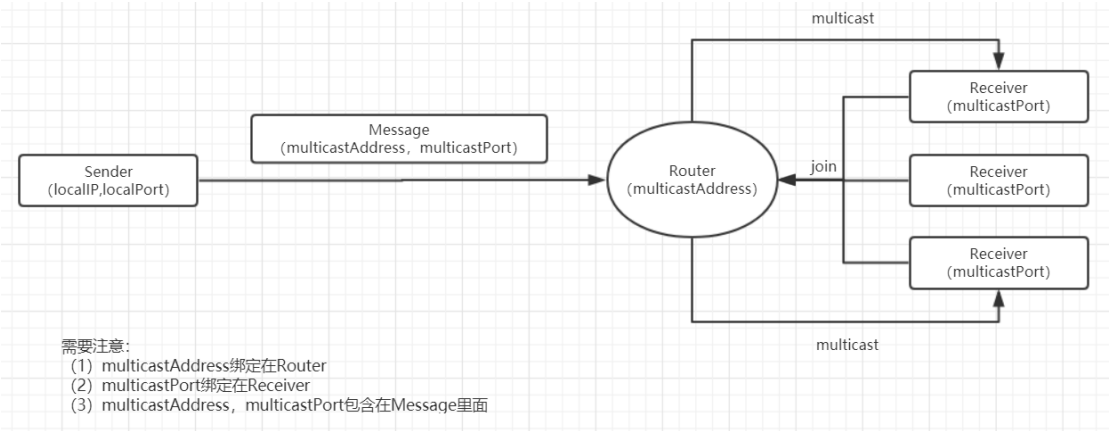
8.2 UDPDiscoveryReceiver 的技术要点

创建 MulticastSocket 对象后，还需要将 MulticastSocket 加入到指定的多点广播地址。MulticastSocket 使用 joinGroup()方法加入指定组，使用 leaveGroup()方法脱离一个组。如下所示：

joinGroup(InetAddress multicastAddr) 将该 MulticastSocket 加入到指定的多点广播地址
leaveGroup(InetAddress multicastAddr) 将该 MulticastSocket 离开指定的多点广播地址

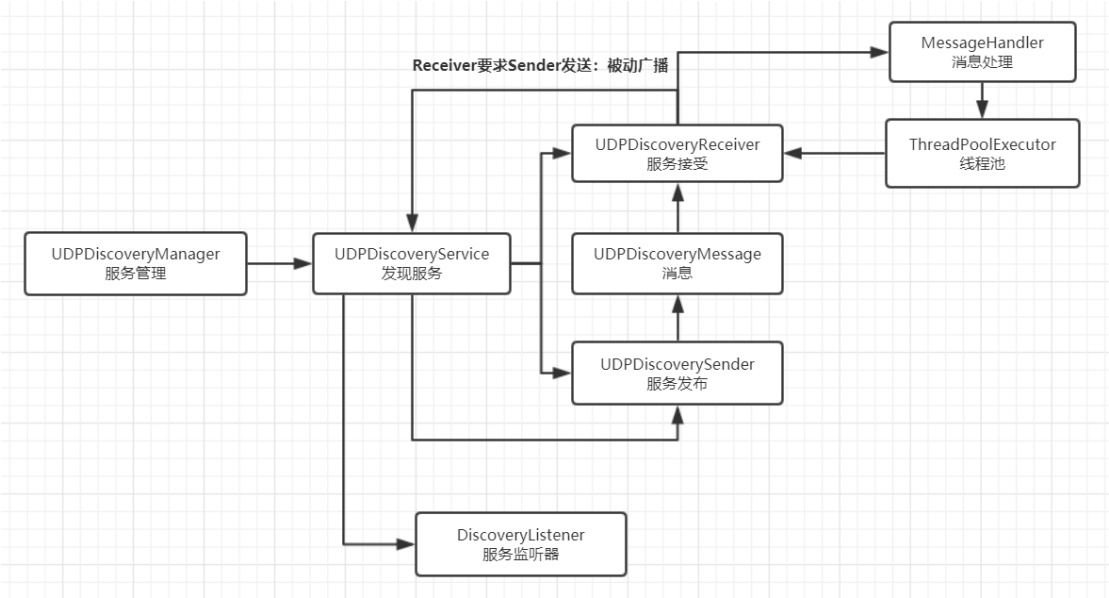
组播地址：
组播地址是组播组的一组主机所共享的地址。注意，它是共享地址，而不是绑定到本机的地址。
应用程序只将数据报包发送给组播地址，路由器将确保包被发送到改组播组中的所有主机。
组播地址的范围在 224.0.0.0~239.255.255.255 之间，都为 D 类地址 1110 开头。

8.3 广播示意图

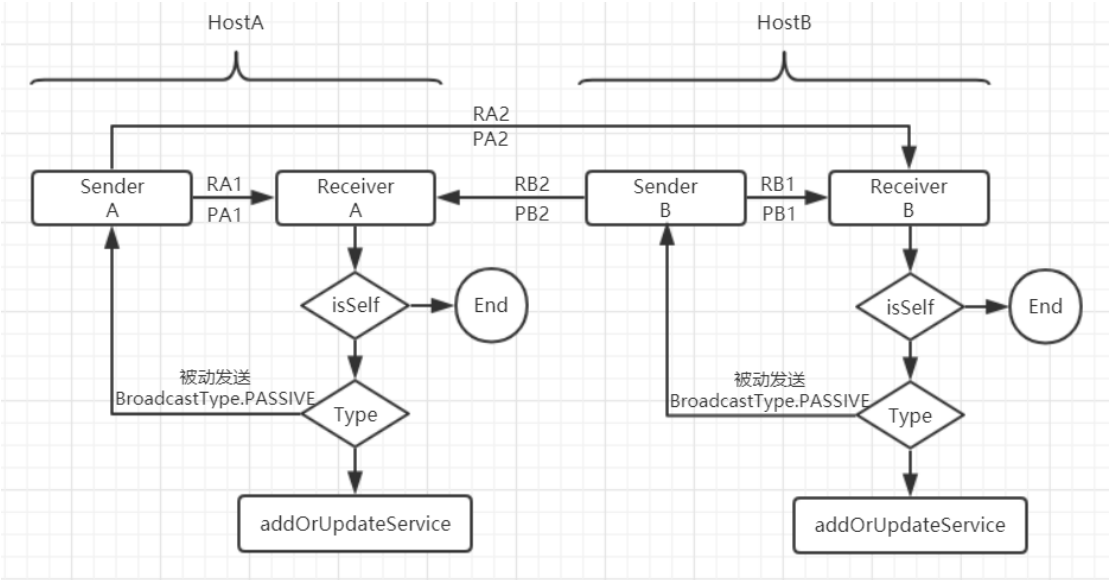


9、服务发现、服务注册

9.1 服务发现、服务注册（上）：架构图部分



9.2 服务发现，服务注册（下）：流程图部分



说明：

RA1：HostA 发送的 BroadcastType.REQUEST 报文 1

RA2：HostA 发送的 BroadcastType.REQUEST 报文 2

PA1：HostA 发送的 BroadcastType.PASSIVE 报文 1

PA2：HostA 发送的 BroadcastType.PASSIVE 报文 2

RB1：HostB 发送的 BroadcastType.REQUEST 报文 1

RB2：HostB 发送的 BroadcastType.REQUEST 报文 2

PB1：HostB 发送的 BroadcastType.PASSIVE 报文 1

PB2：HostB 发送的 BroadcastType.PASSIVE 报文 2

10、门面模式：动态增删的逻辑

10.1 接收器

```
ILateralCacheListener<K, V> listener = createListener(cacheattr,
cacheMgr);
```

10.2 发送器（数组）

```
LateralCacheAsync<K, V>[] asyncArray = asyncs.toArray(new
LateralCacheAsync[0]);
```

10.3 门面

```
LateralCacheAsyncFacade<K, V> lcaf = new LateralCacheAsyncFacade<K,
V>(listener, asyncArray, cacheattr);
```

10.4 监听器（将门面添加到监听器里面）

```
LateralTCPDiscoveryListener discoveryListener =
getDiscoveryListener(cacheattr, cacheMgr);
            discoveryListener.addAsyncFacade(cacheattr.getCacheName(),
lcaf);
```

10.5 发现器

```
discovery.addDiscoveryListener(discoveryListener);
```

10.6 接受消息，动态增删发送器

11、分布式一致性

首先解释一下何为一致性问题，考虑如下的场景：有一组进程 p1，p2，.....，pn，一个变量 v。一致性问题就是：如何让这组进程就变量 v 的值达成一致。什么是“达成一致”，请考虑如下情形：
p1 令 v=a，p2 令 v=b，显然，进程 p1 和 p2 就 v 的值没有达成一致。如果 p2 令 v=a，那么认为 p1，p2 就 v 的值达成一致。让各个进程对变量 v 的值达成一致需要两个过程：

（1）给变量 v 选择一个值，假设是 c
（2）让各个进程都认为 v=c，即认为就 c 达成一致，这时我们认为变量 v 的值被决定。

接着介绍一个性质，称之为法定集合性质。我们将一个超过半数的集合称之为法定集合，比如数字 1，2，3，4，5 共 5 个元素，{1，2，3}有三个元素就是法定集合。法定集合性质：任意两个法定集合，必定存在一个公共的成员。

实际上我们不能等所有进程都认为 v 是同个值，才认为 v 的值被决定。这样，一旦有一个进程挂了，v 的值永远就不被决定。这里我们直接给出 v 的值被决定的定义：当法定集合的进程令 v 为某个相同的值，比如都是 c，那么称 v 的值被决定为 c。一旦变量 v 的值被决定为 c，那么不会再有另外一个不为 c 的值被决定。

12、锁与信号的艺术：Semaphore 用法

Semaphore，读[ˈseməfɔ:(r)]，是计数信号量。Semaphore 管理一系列许可证。每个 acquire 方法阻塞，直到有一个许可证可以获得然后拿走一个许可证；每个 release 方法增加一个许可证，这可能会释放一个阻塞的 acquire 方法。简单的用法如下所示：

```
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

public class SemaphoreDemo
{
    private Semaphore smp = new Semaphore(3, true); //公平策略
    private Random rnd = new Random();

    class Task implements Runnable
    {
        private String id;
```

```
        Task(String id)
        {
            this.id = id;
        }

        public void run()
        {
            try
            {
                smp.acquire();
                //smp.acquire(int permits);//使用有参数方法可以使用permits
个许可
                System.out.println("Thread " + id + " is working");
                System.out.println("Thread waiting count:" +
smp.getQueueLength());
                work();
                System.out.println("Thread " + id + " is over");
            }
            catch (InterruptedException e)
            {
            }
            finally
            {
                smp.release();
            }
        }

        public void work()
        {//假装在工作，实际在睡觉
            int worktime = rnd.nextInt(1000);
            System.out.println("Thread " + id + " worktime  is " +
worktime);
            try
            {
                Thread.sleep(worktime);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args)
```

```
    {
        SemaphoreDemo semaphoreDemo = new SemaphoreDemo();
        ExecutorService se = Executors.newCachedThreadPool();
        se.submit(semaphoreDemo.new Task("a"));
        se.submit(semaphoreDemo.new Task("b"));
        se.submit(semaphoreDemo.new Task("c"));
        se.submit(semaphoreDemo.new Task("d"));
        se.submit(semaphoreDemo.new Task("e"));
        se.submit(semaphoreDemo.new Task("f"));
        se.shutdown();
    }
}
```

13、Paxos 组件

Paxos 组件内容庞大，处理逻辑复杂，故想出一种划分主线的学习方法。下面有多条主线，每条主线能串联起若干内容，这样更容易理解。

13.1 节点主线

分布式是由多个节点组成的，以节点为切入，并贯彻其中，称之为节点主线。

（1）节点的 IP 和 Port 被封装成 Member。
（2）每个节点又划分为几个角色，角色是用来承担功能的。每个节点都有四种角色：UDPMessenger（信使），Leader，Acceptor，FailureDetector
（3）Leader 角色初始化的时候，系统指定某个节点为 leader，这个时候还未经过投票选举的，还未成为真正的 leader。Leader 初始化的时候，还需要安排一名助理 assistants。助理的作用是帮 leader 向外传递指令，并等待指令的反馈结果。
（4）一朝天子一朝臣，每个新皇帝登基之后都需要设置自己的年后，leader 也不例外，其年号为：viewNumber。年号是怎么呢？是根据 leader 在候选人中的排序而定的。如果发生意外，leader 被迫下野，等日后被复立，其年号将增大一轮。
（5）leader 拥有助理，并设置年号之后，还需要兼顾监听其他竞争对手动态实情，当朝 leader 下野，潜龙就能东山再起。监听功能为：implements FailureListener
（6）acceptor 角色初始化的时候，节点那就不客气了，直接默认自己就是 leader。
（7）FailureDetector 监听各个节点的存活情况，发现异常则告知 leader。

13.2 UDPMessenger 信使主线

分布式系统中，每个节点都有一个信使，信使用来发送和接受信息的。同时，信使还不断地发出定时 tick 消息。每隔 100 毫秒发出一个 tick 消息。tick 消息驱动着此节点中各种角色的运转。首先看 leader 角色的行为。

leader 接受到 tick 消息之后，主要有两个事情要做：
（1）更新自己的时钟
（2）检测助理手上的指令是否需要发送或者重发，一般都是安排在发送时间或者重发时间之后的一秒之后才开始检查。

接着再看一下 FailureDetector 角色的行为。

FailureDetector 接受 tick 消息之后，如果 time > lastHearbeat + *INTERVAL* 则发送心跳消息，接受到心跳消息之后，存放在 lastHeardFrom，可以探知有谁没有回复。

13.3 投票确定 leader 主线

分布式系统指定某个节点为 leader 之后，还需要经过投票才能最终确定下来。假设系统总共有三个节点，投票确定 leader 的主线是这样的：

第一步：系统有三个节点，每个节点有四种角色：信使，leader，acceptor，FailureDetector

第二步：在这三个节点中，系统指定 leader 节点，这个过程在 leader 角色中完成。

第三步：leader 节点知道自己是 leader，但是另外两个节点不知道谁是 leader，必须给他们发送消息，当 leader 节点收到半数以上的确认，这个时候才能真正的成为 leader 节点。

第四步：leader 节点发送的消息是：Election，需要注意：viewNumber，表示选举的次序。消息准备好之后，等待信使发来的 tick，再发送出去。

第五步：leader 节点发送 Election 消息之后，不仅自己要接受这个消息，其他的两个节点也要接受并处理这个消息。这个消息要经过各个节点的四个角色：信使，leader，acceptor，FailureDetector

第六步：leader 节点接受自己发来的 Election 消息。leader 角色不处理，acceptor 角色更新 leader 和 viewnumber，然后发送 ViewAccepted 消息。
new ViewAccepted(viewNumber, accepted, me)，一朝天子一朝臣，viewNumber 就是 accepted 表示在这个年号里面收到的消息

第七步：其他节点接受 leader 节点发来的 Election 消息。leader 角色更新 viewNumber，acceptor 角色更新 leader 和 viewnumber，然后发送 ViewAccepted 消息。

第八步：leader 节点接受自己发来的 ViewAccepted。leader 角色通过 assistant 来处理，acceptor 角色不处理

第九步：leader 节点接受其他节点发来的 ViewAccepted。leader 角色通过 assistant 来处理，acceptor 角色不处理。当收到第二个节点的 ViewAccepted 消息后，达到半数的时候，正式确立自己的位置为主，当收到第三个节点的 ViewAccepted 消息后，将请求处理完毕

13.3 广播消息

第一步：任意节点可以广播消息，消息首先需要发送给 leader 节点

第二步：leader 节点的 leader 角色接受广播消息，并生产 Proposal，然后生产 MultiAccept<Accept, Accepted>，最后发出来的。发送的是 Accept。

第三步：leader 节点收到自己发来的 Accept，处理过程如下：

leader 角色收到了 Accept，它可以处理 Accepted，但是无法处理 Accept，所以不处理；

acceptor 角色收到了 Accept，然后回复了 Accepted，

第四步：非 leader 节点收到了 leader 发送的 Accept。处理过程同上。

第五步：leader 节点收到了自己发送的 Accepted，以及其他节点发送的 Accepted，当达到半数的时候，则回复 MultiSuccess，实际上发送的是 Success 消息。

第六步：leader 节点收到 Success 消息后，leader 角色不处理，acceptor 角色，完成 Receiver 的处理逻辑，并回复 SuccessAck 消息。其他非 leader 节点，处理过程同上。

注意：MultiRequest<T extends Serializable, R extends MessageWithSender>，是个发送器和接收器。

14、编码语境

cachekit 是专有名词，cachekit.default 表示 cache 的 ID。

cachekit.kit 是专有名词，cachekit.kit.kit1 表示某个 kit 组件的 ID。

CacheEvent 是专门做日志处理的事件。