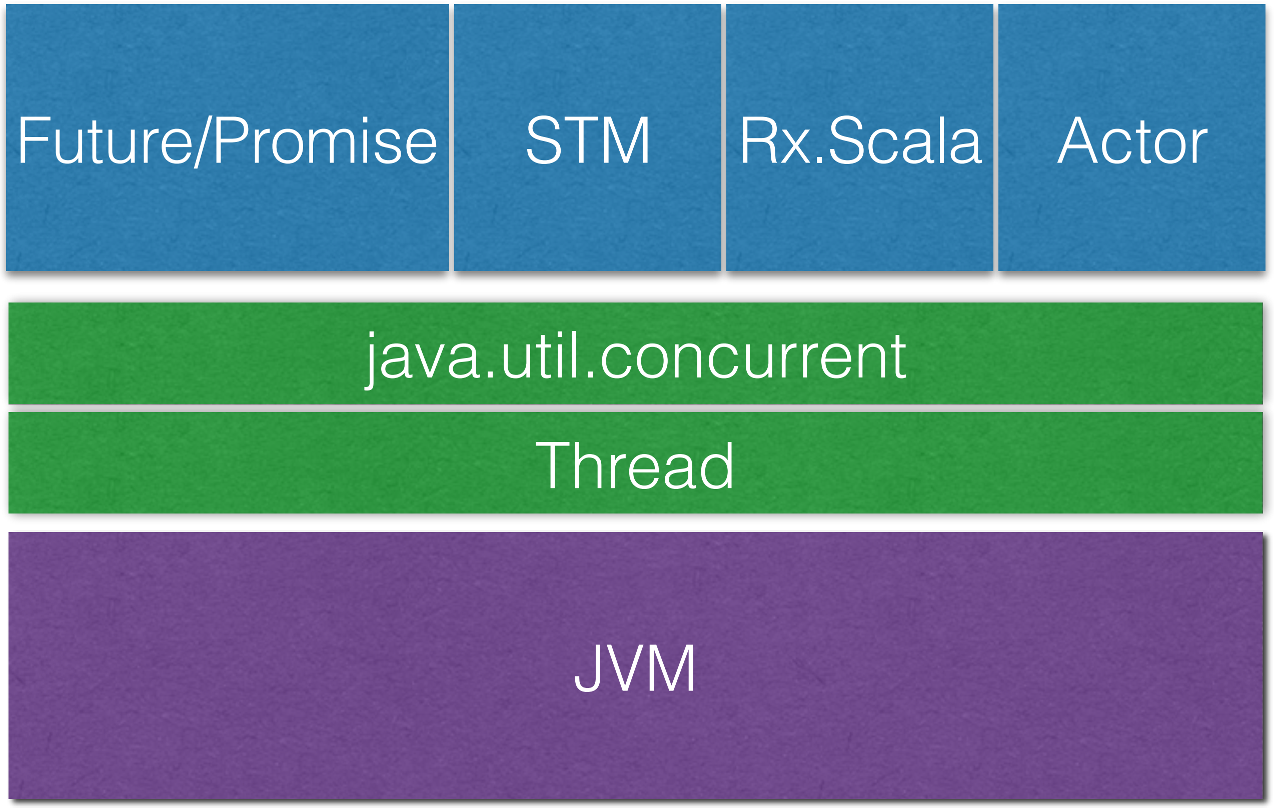
分布式编程Scala工具箱

吴雪峰@ThoughtWorks 2014.10

Scala 工具箱

- Futures and Promises
- STM
- Reactive Extensions
- Actor

In Scala

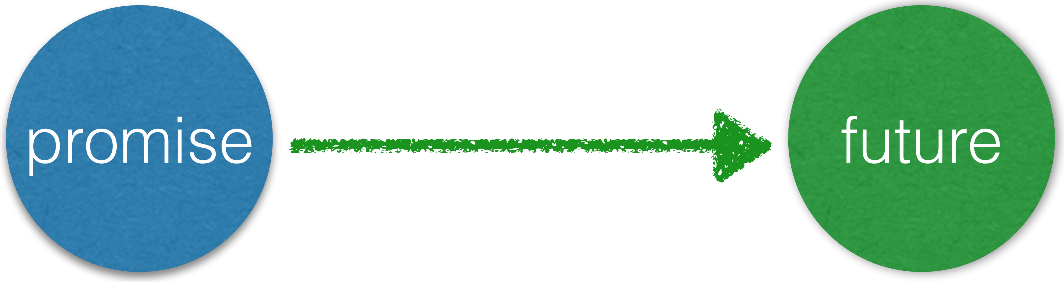

Future/Promise
STM
Rx.Scala
Actor
java.util.concurrent
Thread
JVM

Promises

一个对未知结果的承诺 proxy

Futures

对未来的读取



write once

read once

```
def complete(result: Try[T])
```

```
def onComplete[U](f: Try[T] => U)
```

```
Await.result(f, duration.Duration.Inf)
```

```
import scala.concurrent.{ Future, Promise }

val p = Promise[T]
val f = p.future
val producer = Future {
  val r = produceSomething()
  p success r
  continueDoingSomethingUnrelated()
}
val consumer = Future {
  startDoingSomething()
  f onSuccess {
    case r => doSomethingWithResult()
  }
}
```

Future

```
scala.concurrent.Future

def getFollowers(url: String): String
def extractWeiboFollowers(json: String): Int
def extractWechatFriends(xml: String): Int

for {
  weiboJson <- Future(getFollowers("http://api.weibo.com/u/123/followers"))
  wechatXml <- Future(getFollowers("http://api.wechat.com/user/123/friends"))
  weboFollower = extractWeiboFollowers(weiboJson)
  wechatFriends = extractWechatFriends(wechatXml)
} yield weboFollower + wechatFriends * 5

influence: Future[Int]

Await.result(influence, 10 seconds)
```

STM

Software transactional memory

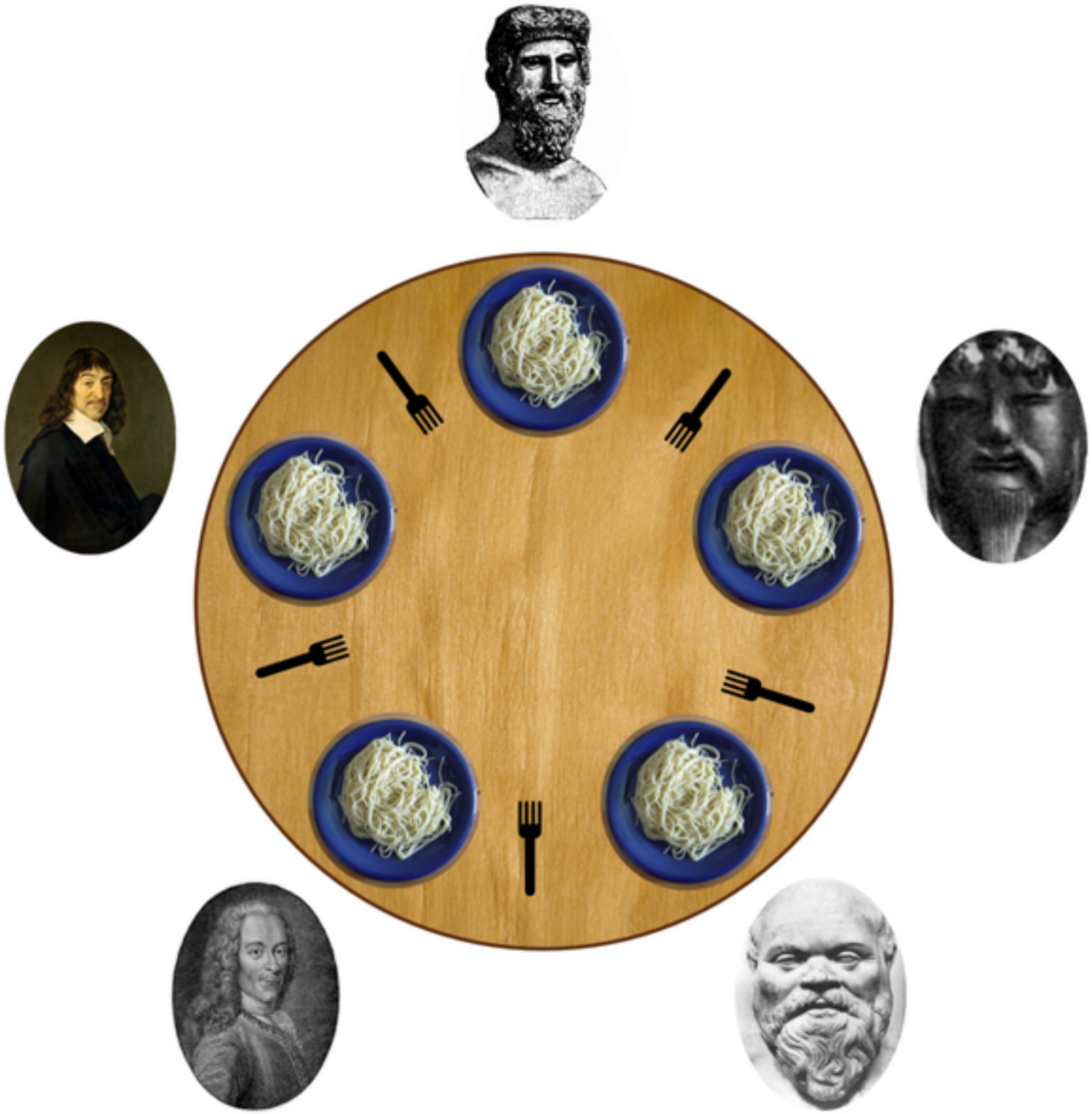
STM是一个类似于数据库事务并发控制的，
用来控制共享内存访问、并行计算的机制。
这是一个基于锁的同步的替代品。

在此的“事务”指的是一段执行代码：
一系列的读和写共享内存操作。这些读写操作逻辑上在一个时间点完成，其中间状态别的成功事务不可见。

数据库事务

脏读	READ_UNCOMMITTED
不可重复读	READ_COMMITTED
幻读	REPEATABLE_READ

哲学家就餐



```
 class Fork {
   val inUse = Ref(false)
 }

 class PhilosopherThread(meals: Int, left: Fork, right: Fork) extends Thread {
   override def run() {
     for (m <- 0 until meals) {
       // THINK
       pickUpBothForks()
       // EAT
       putDown(left)
       putDown(right)
     }
   }

   def pickUpBothForks() {
     atomic { implicit txn =>
       if (left.inUse() || right.inUse())
         retry
       left.inUse() = true
       right.inUse() = true
     }
   }

   def putDown(f: Fork) {
     f.inUse.single() = false
   }
 }

 def time(tableSize: Int, meals: Int): Long = {
   val forks = Array.tabulate(tableSize) { _ => new Fork }
   val threads = Array.tabulate(tableSize) { i => new PhilosopherThread(meals, forks(i), forks((i + 1) % tableSize)) }
   val start = System.currentTimeMillis
   for (t <- threads) t.start()
   for (t <- threads) t.join()
   System.currentTimeMillis - start
 }

 def main(args: Array[String]) {
   val meals = 100000
   for (p <- 0 until 3) {
     val elapsed = time(5, meals)
     printf("%3.1f usec/meal\n", (elapsed * 1000.0) / meals)
   }
 }
}
```

Reactive Extensions

iterator pattern + observer pattern

event	Iterable (pull)	Observable (push)
retrieve data	T next()	onNext(T)
discover error	throws Exception	onError(Exception)
complete	returns	onCompleted()

	single items	**multiple items**
synchronous	T getData()	Iterable<T> getData()
asynchronous	Future<T> getData()	Observable<T> getData()

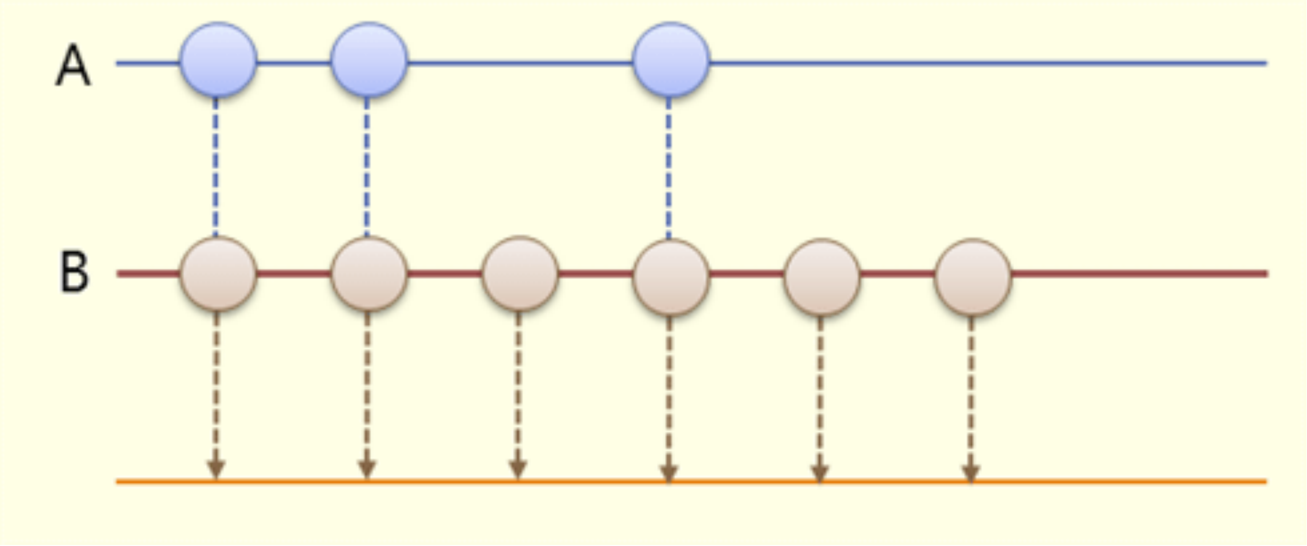
```scala
import java.net.URL
import java.util.Scanner

import rx.lang.scala.Observable

object AsyncWiki extends App {
  /*
   * Fetch a list of Wikipedia articles asynchronously.
   */
  def fetchWikipediaArticleAsynchronously(wikipediaArticleNames: String*): Observable[String] = {
    Observable(subscriber => {
      new Thread(new Runnable() {
        def run() {
          for (articleName <- wikipediaArticleNames) {
            if (subscriber.isUnsubscribed) {
              return
            }
            val url = "http://en.wikipedia.org/wiki/" + articleName
            val art = new Scanner(new URL(url).openStream()).useDelimiter("\\A").next()
            subscriber.onNext(art)
          }
          if (!subscriber.isUnsubscribed) {
            subscriber.onCompleted()
          }
        }
      }).start()
    })
  }

  fetchWikipediaArticleAsynchronously("Tiger", "Elephant")
    .subscribe(art => println("--- Article ---\n" + art.substring(0, 125)))
}
```

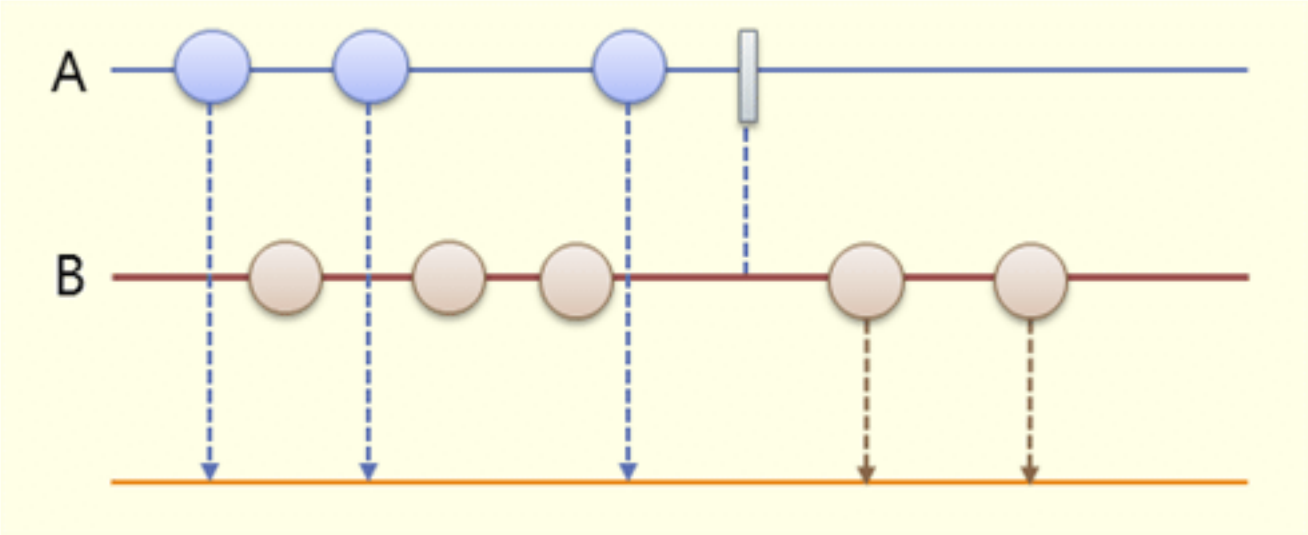
SelectMany



根据 A 序列的值，后续用 B 序列的值进行插入替换

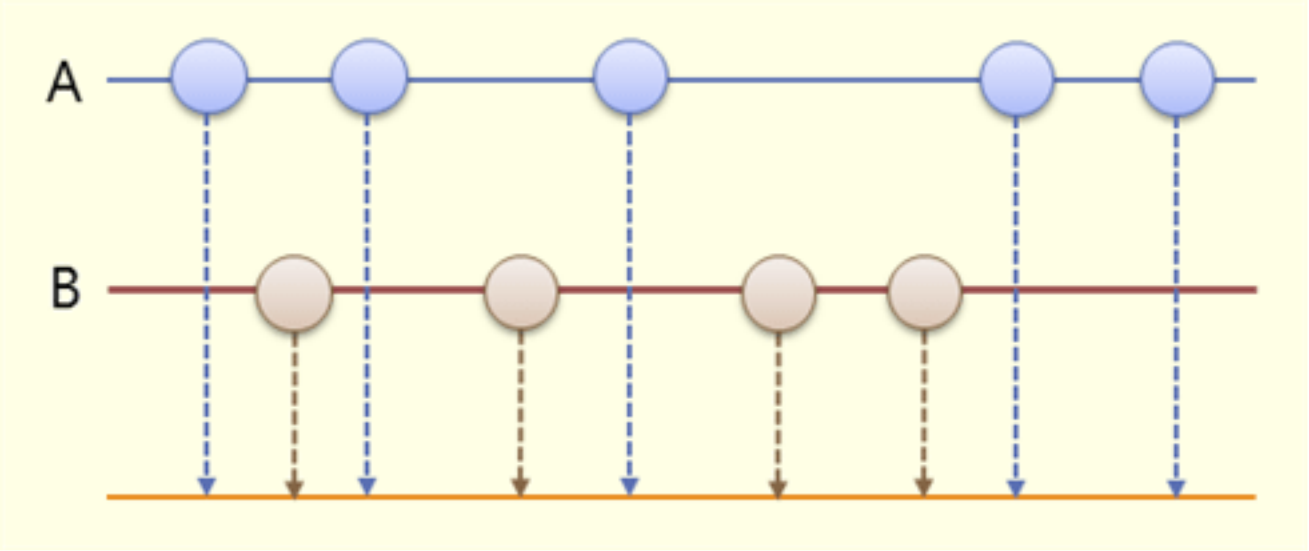
例如将鼠标移动事件插入鼠标按下事件中，
甚至对于序列自身的修改替换

Concat



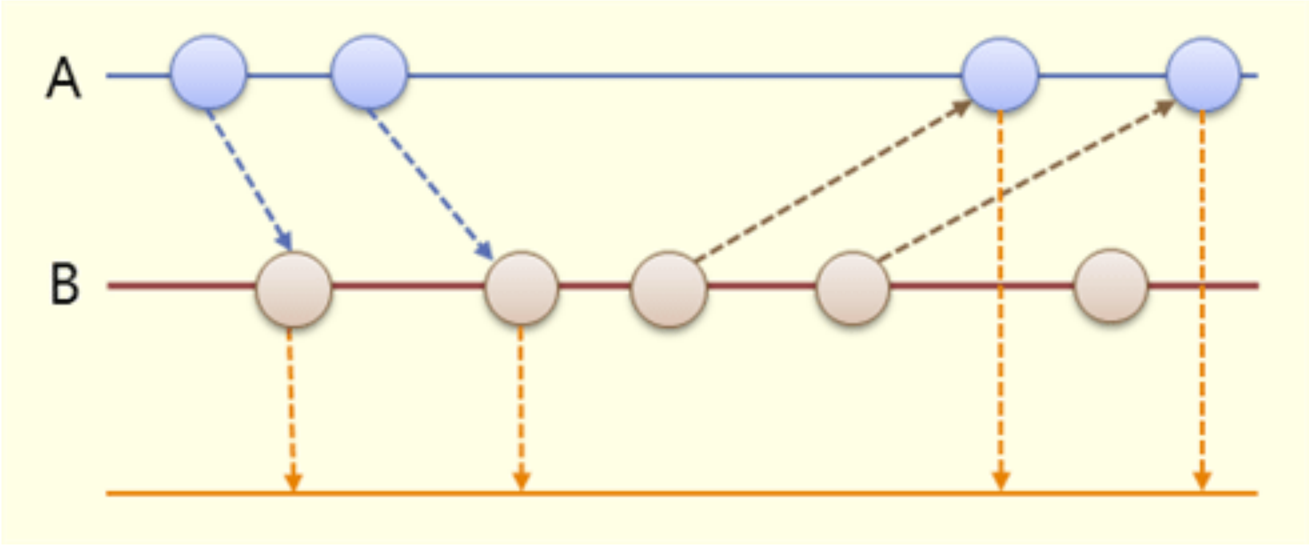
将2个序列进行连接。
这个时候，直到第一个序列终止前，
第二个序列的值就会被忽略掉。
我们可以理解是在第一个序列的结尾追加上另一个序列。

Merge



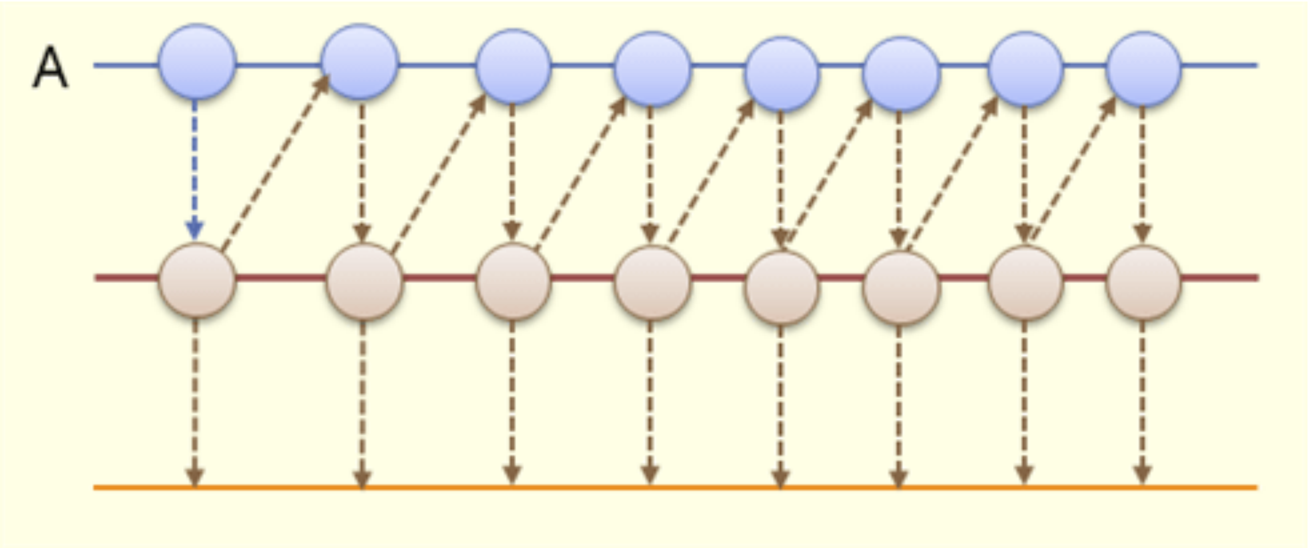
将所有的值都会合并进来。
不只是2个对象的连接，也可以进行多个对象的连接。
如果要对应多个控件的共通处理的话，使用Merge是很方便的。

Zip



Zip方法是A和B中各取1个值为一组（2个值）进行配对处理。一边的值如果发生偏移，那么Zip会直到取到2个值为止才输出。

Scan



是一个集计

Scan方法是1个前面的“结果”和现在的“值”进行合成输出的

A序列中，1个前面的“结果”(中间褐色的横线)
 和当前的“值”(上面蓝色的横线）进行合成。

生产者消费者问题 NETFLIX

Actor