cnblogs.com/tiger-xc/

blog.csdn.net/tiger_xc/

github.com/bayakala/

- 分享提纲

1、type class: ad-hoc and the higher kinded polimorphism
即兴多态及高阶类型参数多态

2、Scala.Future: traps,short-comes.Going to Monic.Task
编程中的陷阱及升级解决方案 Monic.Task

3、programming patterns: program integration
in distributed environment
集群环境下系统集成编程模式

- Functional programming my way

```
* 2+years - theories                理论
            abstractions            概念
            structures              数据结构
            combinator libraries    函数库
            …

* 80% - 在实际工作中没有得到利用
  goal: make use of existing tools and libraries like
        akka, spark …
  building on top of existing APIs

* 10% - used few abstractions like: Future,Traversable
       immutable construct operations like: for-comp,
       map,flatMap,fold-aggr, traverse, sequence …

* 10% - 用来了解开源软件源代码
       everything monad: point,flatMap

* in reality: patterns & tricks & caveats
              编程模式、技巧、避忌
```

- Polymorphism

  - 多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。
    就是允许方法重名，同一函数可以施用在不同类型的对象上。

  - 实现方式：

    1、overloading 重载

    2、inherentance 继承

    3、pattern-matching 模式匹配

    4、typeclass 即兴多态-ad-hoc polymorphism

- Polymorphism

  - 重载 overload

```
case class Color(scheme: String)
case class Person(name: String)
//…
object overloading {
  def tell(color: Color) = s"I'm Color ${color.scheme}"
  def tell(person: Person)=  s"I'm ${person.name}"
}
import overloading._
tell(Color("RED"))          //> res0: String = I'm Color RED
tell(Person("John"))        //> res1: String = I'm John
```

- Polymorphism

  - 继承 inheritance

```
object inheritance {
  trait AnyThing
  case class Color(scheme: String) extends AnyThing {
    def tell: String = s"I'm Color ${scheme}"
  }
  case class Person(name: String) extends AnyThing {
    def tell: String = s"I'm ${name}"
  }
}
import inheritance._
Color("RED").tell                    //> res0: String = I'm Color RED
Person("John").tell                  //> res1: String = I'm John
```

- Polymorphism

  - 模式匹配 pattern-matching

```scala
case class Color(scheme: String)
case class Person(name: String)
//…
object patternmatch {
  def tell(a: Any): String = a match {
      case Color(sch) =>  s"I'm Color ${sch}"
      case Person(nm) => s"I'm ${nm}"
      case i: Int => s"I'm a Integer with value $i"
  }
}
import patternmatch._
tell(3)                  //> res0: String = I'm a Integer with value 3
tell(Color("RED"))       //> res1: String = I'm Color RED
tell(Person("Jonh"))     //> res2: String = I'm Jonh
```

- Polymorphism

  - typeclass - 即兴多态

    ✓ a programming pattern:

      1. trait with type parameter
      2. function with implicit parameter
      3. implicit type instance
      4. call function without any type info

- Polymorphism

  - typeclass - 即兴多态

    1. trait with type parameter

```
trait Tellable[A] {
  def tell(a: A): String
}
```

    2. function with implicit parameter

```
 def tellAll[A](a: A)(implicit Teller: Tellable[A]): String =
                     Teller.tell(a)
```

    3. create implicit type instance and call the same function

```
  case class Color(scheme: String)
  case class Person(name: String)
//…
  implicit val colorTellable = new Tellable[Color] {
    def tell(c: Color): String = s"I'm Color ${c.scheme}"
  }
  tellAll(Color("Red"))        //> res2: String = I'm Color Red
```

```
  implicit val personTellable extends Tellable[Person] {
    def tell(p: Person): String = s"I'm $p.name"
  }
  tellAll(Person("John"))      //> res3: String = I'm Person(John).name
```

- Polymorphism

  - typeclass - 即兴多态 demo

```scala
trait Addable[A] {              //monoid
  val mzero: A
  def madd(x: A, y: A): A
}
case class Crew(names: List[String])
object Addable {
  implicit object intAddable extends Addable[Int] {
    def mzero = 0
    def madd(x: Int, y: Int) = x + y
  }
  implicit object strAddable extends Addable[String] {
    val mzero = ""
    def madd(x: String, y: String) = x + y
  }
  implicit object crewAddable extends Addable[Crew] {
    val mzero = Crew(List())
    def madd(x: Crew, y: Crew): Crew = Crew(x.names ++ y.names)
  }
  def apply[A](implicit M: Addable[A]): Addable[A] = M
}

def sum2[A](xa: List[A])(implicit M: Addable[A]): A = xa.foldLeft(M.mzero)(M.madd)

sum2(List(1,2,3))                                          //> res2: Int = 6
sum2(List("ab","c","def"))                                 //> res3: String = abcdef

sum2(List(Crew(List("john")), Crew(List(“susan","peter"))))
                                          //> res4: Crew = Crew(List(john, susan, peter))
```

- Polymorphism

  - typeclass - 即兴多态 demo

```
trait FoldLeft[F[_]] {
  def foldLeft[A,B](fa: F[A])(b: B)(f: (B,A) => B): B
}
object FoldLeft {
  implicit object listFold extends FoldLeft[List] {
    def foldLeft[A,B](fa: List[A])(b: B)(f: (B,A) => B) = fa.foldLeft(b)(f)
  }
  def apply[F[_]](implicit F: FoldLeft[F]): FoldLeft[F] = F
}
FoldLeft[List].foldLeft(List(1,2,3))(0)(_ + _)    //> res7: Int = 6

def sum3[A: Addable,F[_]: FoldLeft](fa: F[A]): A = {
  val adder = implicitly[Addable[A]]
  val folder = implicitly[FoldLeft[F]]
  folder.foldLeft(fa)(adder.mzero)(adder.madd)
}
//> sum3: [A, F[_]](fa: F[A])(implicit evidence$1: Addable[A],
//                              implicit evidenc$2:FoldLeft[F])


sum3(List(Crew(List("john")), Crew(List("susan","peter"))))
            //> res8: Crew = Crew(List(john, susan, peter))
```

- Polymorphism

  - Method Injection 方法注入

```
class AddableOp[A](a: A)(implicit M: Addable[A]) {
  def |+|(y: A) = M.madd(a,y)
}
implicit def toAddableOp[A: Addable](a: A): AddableOp[A] = new
AddableOp[A](a)
3 |+| 2                              //> res9: Int = 5
("hello" |+| " ") |+| "world!"   //> res10: String = hello world!
```

```
implicit class addableOp[A: Addable](a: A) {
   def |+|(y: A) = implicitly[Addable[A]].madd(a,y)
}
3 |+| 2                                //> res9: Int = 5
("hello" |+| " ") |+| "world!"     //> res10: String = hello world!
```

- Polymorphism

  - Method Injection 方法注入

```
trait AddableOp[A] {
  val M: Addable[A]
  val x: A
  def |%| = M.mzero
  def |+|(y: A) = M.madd(x,y)
}
implicit def toAddableOps[A: Addable](a: A): AddableOp[A] =
  new AddableOp[A] {
     val M = implicitly[Addable[A]]
     val x = a
  }
3 |+| 2                              //> res9: Int = 5
("hello" |+| " ") |+| "world!" //> res10: String = hello world!
3.|%|                                //> res11: Int = 0
"hi".|%|                             //> res12: String = ""
```

- # Higher Kinded Polymorphism in scala

✓ first order parametric polymorphism = generics 泛型

```
trait iterable[T] {
  def filter(p: T => Boolean): Iterable[T]
  def remove(p: T => Boolean): Iterable[T] = filter(x => !p(x))
}
```

- T = type, abstraction over type = first order parametric polymorphism
- F[T] = type constructor, higher kind
- Abstraction over type constructor = higher kinded polymorphism

```
trait MyIterable[T, F[_]] {
  def filter(p: T => Boolean): F[T]
  def remove(p: T => Boolean): F[T] = filter(x => !p(x))
}
```

- Higher Kinded Polymorphism in scala

✓ duplicated code reduction - 减少重复代码

- first order parametric polymorphism

```
trait iterable[T] {
  def filter(p: T => Boolean): Iterable[T]
  def remove(p: T => Boolean): Iterable[T] = filter(x => !p(x))
}

trait List[T] extends Iterable[T] {
  def filter(p: T => Boolean): List[T]
  def remove(p: T => Boolean): List[T] = filter(x => !p(x))
}
```

- higher kinded polymorphism

```
trait MyIterable[T, F[_]] {
  def filter(p: T => Boolean): F[T]
  def remove(p: T => Boolean): F[T] = filter(x => !p(x))
}

trait MyList[T] extends MyIterable[T, List]
```

- Polymorphism

  - typeclass - 即兴多态 demo

```
trait FoldLeft[F[_]] {
  def foldLeft[A,B](fa: F[A])(b: B)(f: (B,A) => B): B
}
object FoldLeft {
  implicit object listFold extends FoldLeft[List] {
    def foldLeft[A,B](fa: List[A])(b: B)(f: (B,A) => B) = fa.foldLeft(b)(f)
  }
  def apply[F[_]](implicit F: FoldLeft[F]): FoldLeft[F] = F
}
FoldLeft[List].foldLeft(List(1,2,3))(0)(_ + _)    //> res7: Int = 6

def sum3[A: Addable,F[_]: FoldLeft](fa: F[A]): A = {
  val adder = implicitly[Addable[A]]
  val folder = implicitly[FoldLeft[F]]
  folder.foldLeft(fa)(adder.mzero)(adder.madd)
}
//> sum3: [A, F[_]](fa: F[A])(implicit evidence$1: Addable[A],
//                              implicit evidenc$2:FoldLeft[F])


sum3(List(Crew(List("john")), Crew(List("susan","peter"))))
            //> res8: Crew = Crew(List(john, susan, peter))
```

- scala. Future

```
def runMongo(q: query): Future[Int] = ???
def runCassandra(q: query): Future[Int] = ???
def runJdbc(q: query): Future[Int] = ???

runMongo(qryTotalCredit).onComplete {
  case Success(credit) => //do something ...
  case Failure(err) => //
}
//continue doing next things   ...
```

```
val totCredit: Future[Int] = for {
  mt <- runMongo(qryTotalCredit)
  ct <- runCassandra(qryTotalCredit)
  jc <- runJdbc(qryTotalCredit)
} yield (mt + ct + jc)

println(Await.result(totCredit,3 seconds))
```

```
List(runJdbc(qrypart1),runJdbc(qrypart2),runMongo(qry),runCassandra(qry))
  .sequence
  .map {list => list.fold(0)(_ + _)}
  .onComplete {
    case Success(sum) => println(s"total credit: $sum")
    case Failure(e) => println(e.getMessage)
  }
```

```
def sequence(F[G[B]): G[F[B]]  =>  List[Future[Int]]  => Future[List[Int]]
```

- scala.Future

```scala
val futureA = Future { ... }
val futureB = Future { ... }
// at this point both computation are executed
val result = for {
  a <- futureA
  b <- futureB
} yield ()
```

```scala
for {
  a <- Future { ... }
  // only a's computation is executed
  b <- Future { ... }
  // b's computation is executed only if a succeed
} yield ()
```

- Error ? Exception ?

```scala
val incrementResult: Future[Unit] =
  Future.traverse(basket.content) { product =>
    productRepo.incrementProductSells(product.id, product.quantity)
  }
```

- scala.Future

```
val total = Future.successful {
  println("Computing 2 + 2")
  2 + 2
}
println(total)
println(total)
```

```
Computing 2 + 2
Future(Success(4))
Future(Success(4))
```

- no referencial transparency，side-effect，impure 无法实现函数组合

```
val progA: Future[A] = for {
  b <- readFromB
  _ <- writeToLocationA(a)
  r <- getResult
} yield r

/* location A content updated */

... /* later */

val progB: Future[B] = for {
  a <- readFromA
  _ <- updateLocationA
  c <- getResult
}
...
```

```
val program: Future[Unit] = for {
  _ <- progA
  _ <- progB
} yield()
```

- scala.Future

- Require ExecutionConext for compilation

```
def run(query: DBQuery)(implicit executor: ExecutionContext): Future[ResultSet]

implicit val executor = scala.concurrent.ExecutionContext.global

val result: Future[ResultSet] = db.run(query)
```

```
trait ProductRepository {
  def findProduct(
                  productId: ProductId
                 )(implicit executor: ExecutionContext): Future[Option[Product]]

  def saveProduct(
                  product: Product
                 )(implicit executor: ExecutionContext): Future[Unit]

  def incrementProductSells(
                            productId: ProductId,
                            quantity: Int
                           )(implicit executor: ExecutionContext): Future[Unit]

  // More methods ...
}
```

- monix.Task

```
val taskA = Task {
  debug("Starting taskA"); Thread.sleep(1000); debug("Finished taskA")
}
import monix.execution.Scheduler.Implicits.global
val futureA = taskA.runAsync
```

```
trait ProductRepository {
  def findProduct(productId: ProductId): Task[Option[Product]]
  def saveProduct(product: Product): Task[Unit]
  def incrementProductSells(productId: ProductId, quantity: Int): Task[Unit]
  // More methods ...
}
```

```
val incrementResult: Task[Unit] =
  Task.traverse(basket.content) {  product =>
    productRepo.incrementProductSells(product.id, product.quantity)
  }
// nothing is executed yet
// we need to explicitly call runAsync to trigger the execution
incrementResult.runAsync
```

```
  val lt: List[Task[Int]] = List(Task(1),Task(2),Task(3))
  val tl: CancelableFuture[Int]= Task.gather(lt).runAsync
                       .map {list => list.fold(0)(_ + _)}
  tl.onComplete(println)   //6
  // If we change our mind...
  tl.cancel()
```

- monix.Task

```
/* ------ taskNow ----*/
val taskNow = Task.now { println("Effect"); "Hello!" }
//=> Effect
// taskNow: monix.eval.Task[String] = Delay(Now(Hello!))

/* --------taskDelay possible another on thread ------*/
val taskDelay = Task { println("Effect"); "Hello!" }
// taskDelay: monix.eval.Task[String] = Delay(Always(<function0>))

taskDelay.runAsync.foreach(println)
//=> Effect
//=> Hello!

// The evaluation (and thus all contained side effects)
// gets triggered on each runAsync:
taskDelay.runAsync.foreach(println)
//=> Effect
//=> Hello!
```

```
object Task {
  // ...
  def apply[A](f: => A): Task[A] = fork(eval(f))
  def def eval[A](a: => A): Task[A] = Eval(a _)
  def fork[A](fa: Task[A]): Task[A] = Task.forkedUnit.flatMap(_ => fa)
  // ...
  private final val forkedUnit = Async[Unit] { (context, cb) =>
    context.scheduler.executeAsync(() => cb.onSuccess(()))
  }
}
```

- monix.Task

```
import monix.execution.Scheduler.Implicits.global
val total = Task {
  println("Computing 2 + 2”) ; 2 + 2
}
println(total.runSyncUnsafe(1.second))
println(total.runSyncUnsafe(1.second))
```

```
Computing 2 + 2
4
Computing 2 + 2
4
```

```
val incrementResult: Task[Unit] =
  Task.traverse(basket.content) {  product =>
    productRepo.incrementProductSells(product.id, product.quantity)
  }
// nothing is executed yet
// we need to explicitly call runAsync to trigger the execution
incrementResult.runAsync
```

```
val lt: List[Task[Int]] = List(Task(1),Task(2),Task(3))
val tl: CancelableFuture[Int]= Task.gather(lt).runAsync
                       .map {list => list.fold(0)(_ + _)}
tl.onComplete(println)   //6
// If we change our mind...
tl.cancel()
```

- monix.Task

```scala
def runAsync(implicit s: Scheduler): CancelableFuture[A] =
def runAsync(cb: Callback[A])(implicit s: Scheduler): Cancelable =
def runAsyncOpt(implicit s: Scheduler, opts: Options): CancelableFuture[A] =
def runAsyncOpt(cb: Callback[A])(implicit s: Scheduler, opts: Options): Cancelable =
final def runSyncMaybe(implicit s: Scheduler): Either[CancelableFuture[A], A] =
final def runSyncMaybeOpt(implicit s: Scheduler, opts: Options): Either[CancelableFuture[A], A] =
final def runSyncUnsafe(timeout: Duration)(implicit s: Scheduler, permit: CanBlock): A =
final def runSyncUnsafeOpt(timeout: Duration)
                  (implicit s: Scheduler, opts: Options, permit: CanBlock): A =
final def runOnComplete(f: Try[A] => Unit)(implicit s: Scheduler): Cancelable =
```

```scala
final def doOnFinish(f: Option[Throwable] => Task[Unit]): Task[A] =
final def doOnCancel(callback: Task[Unit]): Task[A] =
final def onCancelRaiseError(e: Throwable): Task[A] =
final def onErrorRecoverWith[B >: A](pf: PartialFunction[Throwable, Task[B]]): Task[B] =
final def onErrorHandleWith[B >: A](f: Throwable => Task[B]): Task[B] =
final def onErrorFallbackTo[B >: A](that: Task[B]): Task[B] =
final def restartUntil(p: (A) => Boolean): Task[A] =
final def onErrorRestart(maxRetries: Long): Task[A] =
final def onErrorRestartIf(p: Throwable => Boolean): Task[A] =
final def onErrorRestartLoop[S, B >: A](initial: S)
                       (f: (Throwable, S, S => Task[B]) => Task[B]): Task[B] =
final def onErrorHandle[U >: A](f: Throwable => U): Task[U] =
final def onErrorRecover[U >: A](pf: PartialFunction[Throwable, U]): Task[U] =
```

```scala
final class FutureToTask[A](x: => Future[A]) {
  def asTask: Task[A] = Task.deferFuture[A(x)
}

final class TaskToFuture[A](x: => Task[A]) {
  def asFuture: Future[A] = x.runAsync
}
```

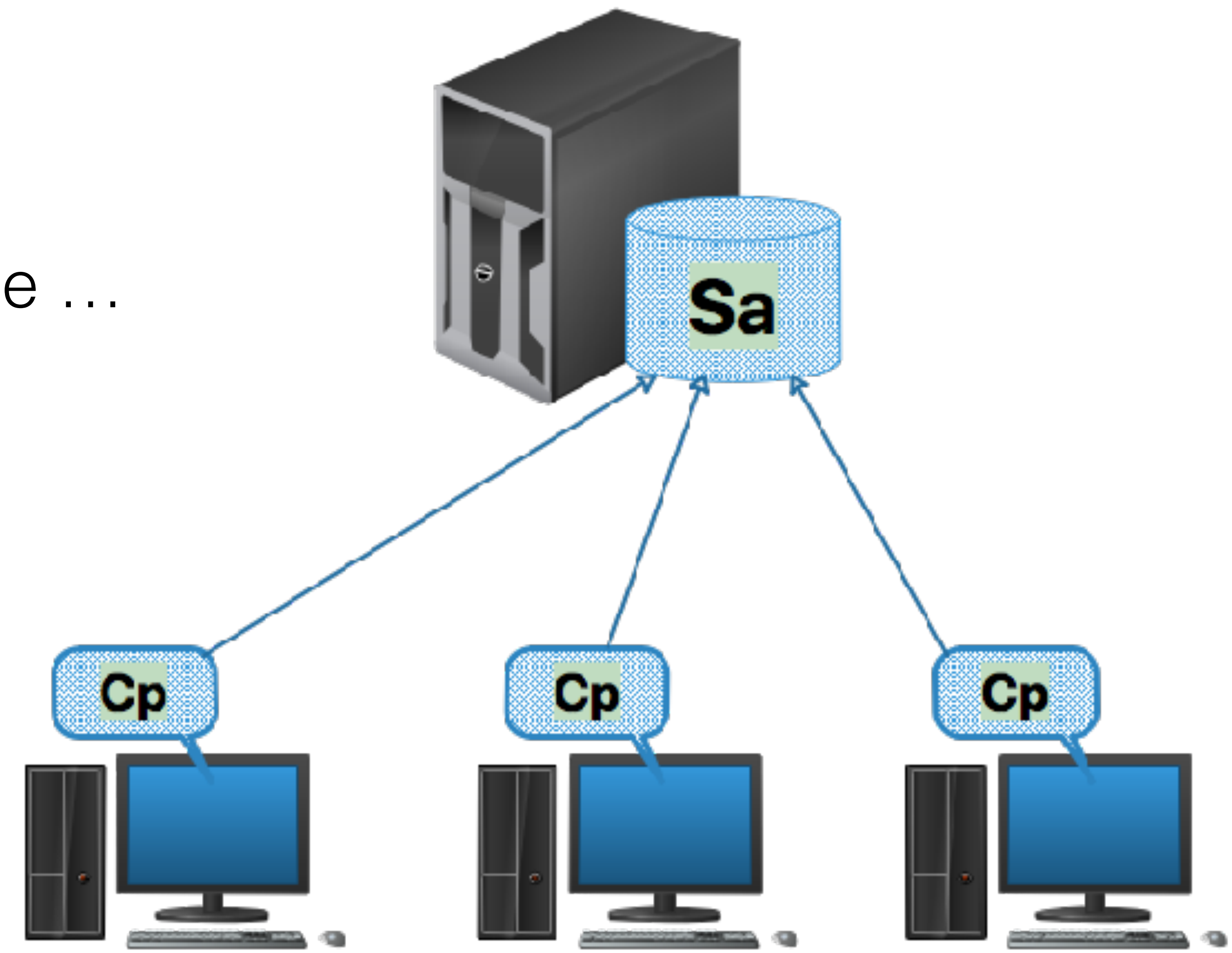
```scala
val taskA = Task.fromFuture(myFuture)

// don't want to execute the future right now
val taskB = Task.deferFuture(Future { ... })
// which is the same as
val taskC = Task.defer(Task.fromFuture(Future
{ ... }))
```

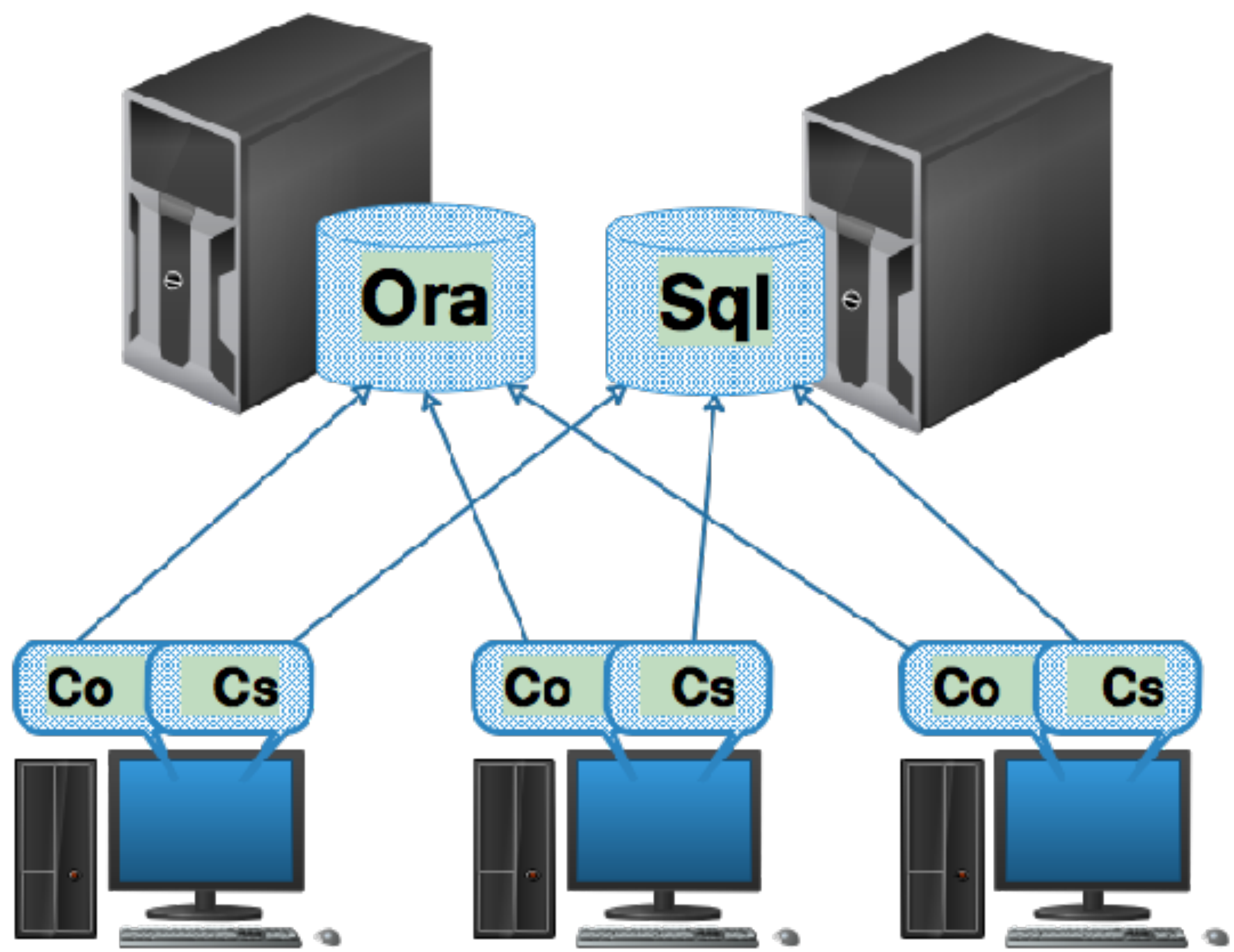
- Programming Model

  - client - server computing

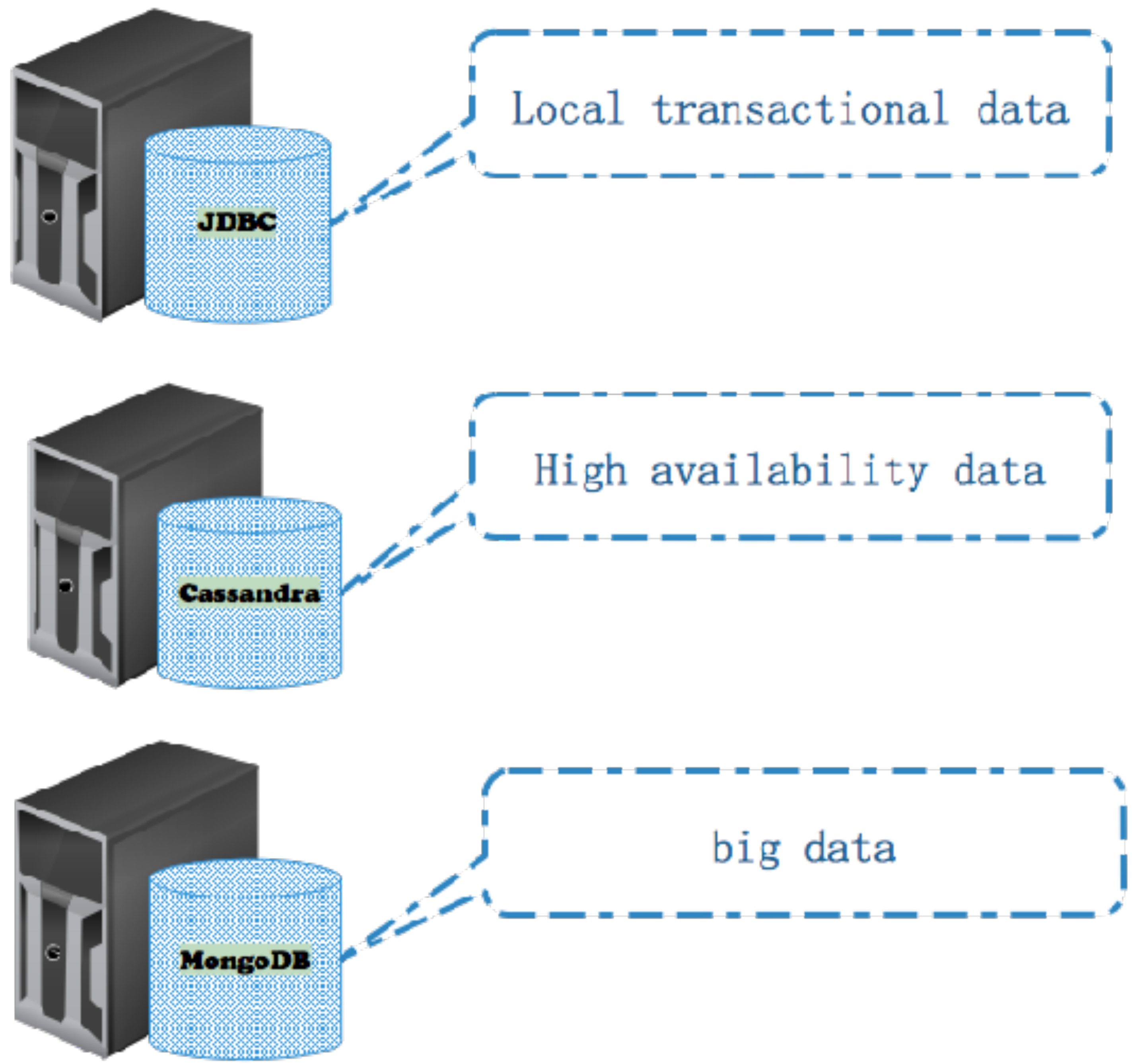
    - Sa: SQLServer, Oracle …
    - Cp: jdbc, ado.net …



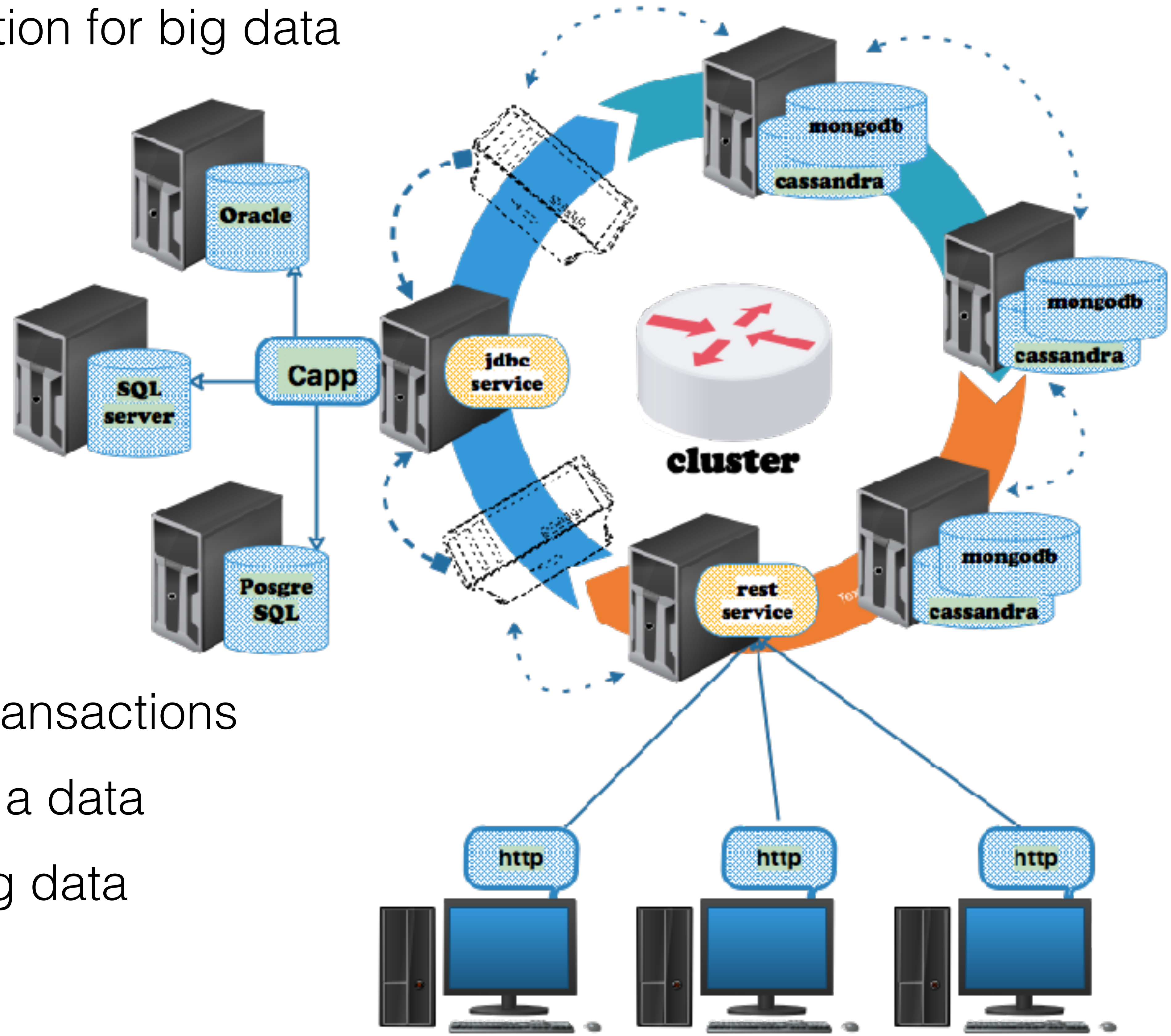
- Programming Model
  - client - side programming
    - ✓ traditional transactional recording model
    - ✓ db - for transactional data



- Data Model
  - I.T system data requirements
    - ✓ JDBC
    - ✓ Cassandra
    - ✓ MongoDB



- Programming Model
  - distributed solution for big data

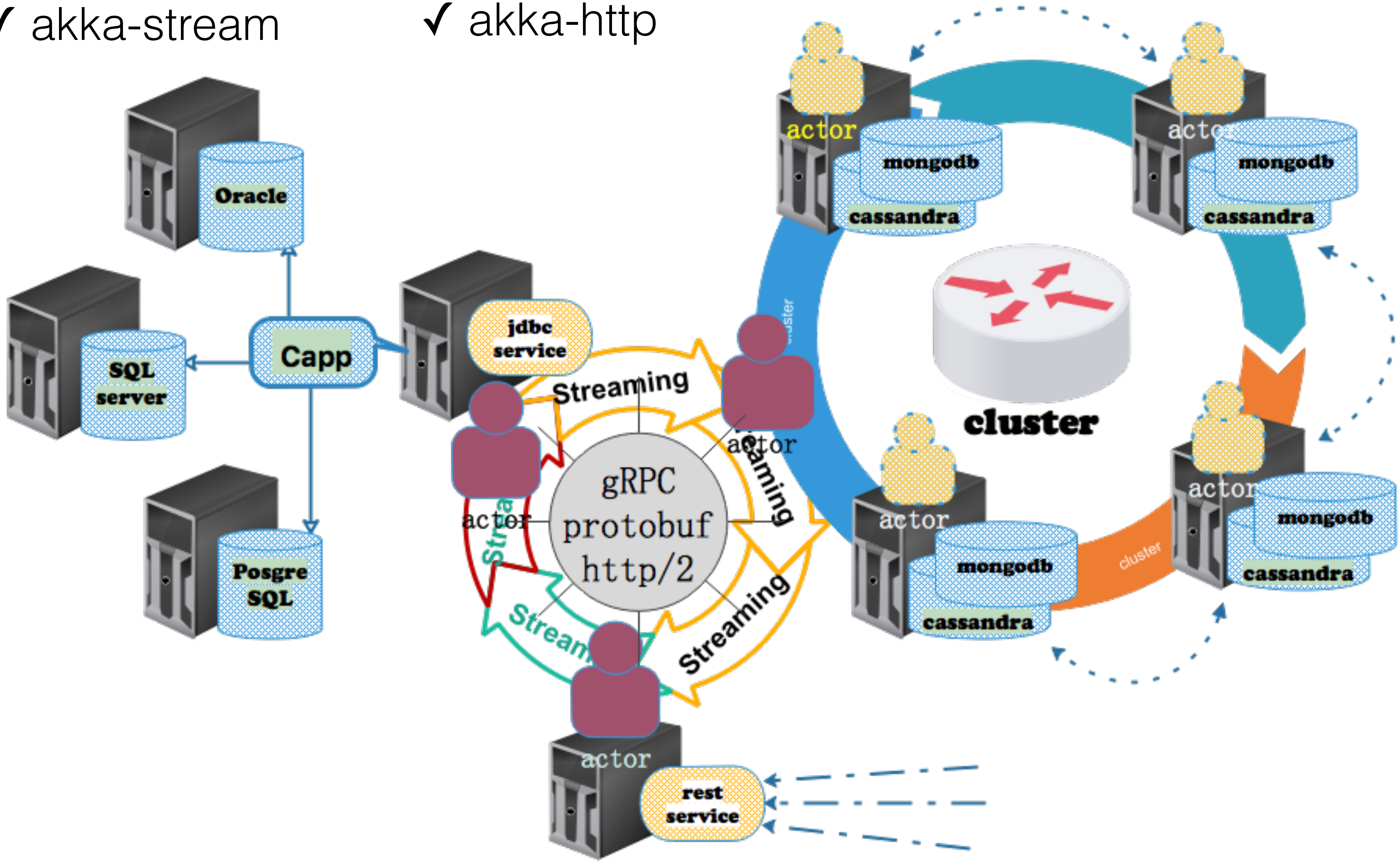


✓ JDBC - local transactions

✓ Cassandra - h.a data

✓ MongoDB - big data

- Distributed programming Model
  - ✓ actor-model
  - ✓ akka-cluster
  - ✓ akka-stream
  - ✓ gRPC
  - ✓ protobuf
  - ✓ akka-http



• gRPC and protobuf

- ✓ binary encoded msg
- ✓ on top of http/2
- ✓ bi-directional streaming
- ✓ akka-streams integration
- ✓ reactive-streams
- ✓ auto-gen boiler code

- IDL service description

```
syntax = "proto3";
package grpc.akka.stream.services;
message Num {
  int32 num = 1;
}
message SumResult {
  int32 result = 1;
}
service SumNumbers {
  rpc KeepAdding(stream Num) returns (stream SumResult) {}
}
```

- service-side implementation code

```
override def keepAdding: Flow[Num, SumResult, NotUsed] = {
  Flow[Num].scan(SumResult(0)) {
    case (a,b) => SumResult(b.num + a.result)
  }
}
```

- client-side implementation code

```
def ContSum(nums: Seq[Int]): Source[String,NotUsed] = {
  Source(nums.map(Num(_)).to[collection.immutable.Iterable])
    .throttle(1, 500.millis, 1, ThrottleMode.shaping)
    .via(stub.keepAdding)
    .map(r => s"current sum = ${r.result}")
}
```

Thank you !

谢谢！

github.com/bayakala/scala-meetup-20180527