

# State Schema Evolution for Apache Flink® Applications

## Apache Flink® 流式应用中状态的数据结构定义升级

公司: data Artisans

职位: Software Engineer

演讲者: 戴资力, Tzu-Li (Gordon) Tai



# Speaker

## 讲者资讯

- Software Engineer @ data Artisans
- PMC Member @ Apache Flink
- Flink connectors (Apache Kafka, AWS Kinesis, Elasticsearch ...)
- Upgradability of stateful Flink applications (e.g. State Schema Evolution)

# About data Artisans

关于 data Artisans



Original Creators of  
Apache Flink®

Apache Flink 原始作者



PLATFORM

Real-Time Stream Processing  
Enterprise Ready

企业级实时流式处理

# Agenda

## 1. Evolving Stateful Flink Streaming Applications

Flink 有状态流式应用升级的考虑要素

## 2. Schema Evolution for Flink Built-in Types

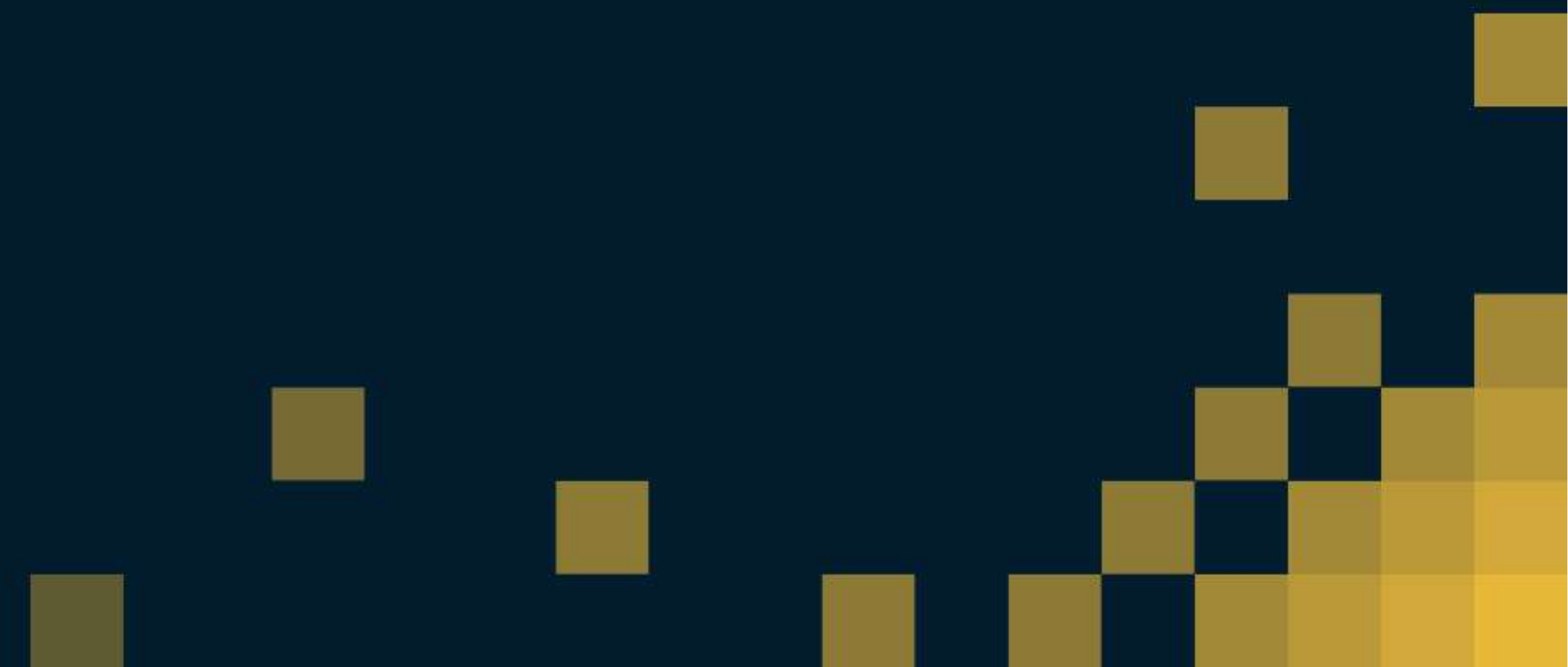
Flink 内建类别的数据结构定义更新

## 3. Implementing Custom State Serializers

自订状态序列化器的实现

# Evolving Stateful Flink Streaming Applications

Flink 有状态流式应用升级的考虑要素





# Anatomy of a Flink stream job upgrade

Flink 流式应用升级流程解析

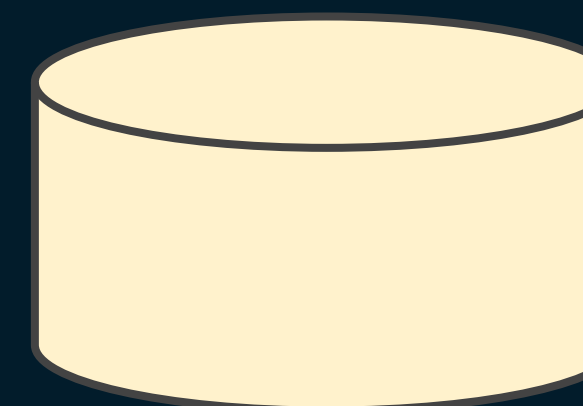
User code

使用者代码

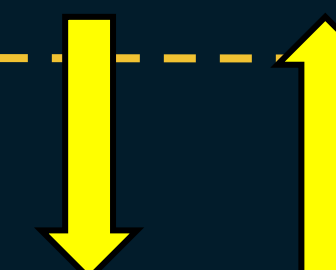


Local state  
backend

本地状态后端



local read / writes  
that manipulate state



Persisted  
savepoint

持久保存点



# Anatomy of a Flink stream job upgrade

Flink 流式应用升级流程解析

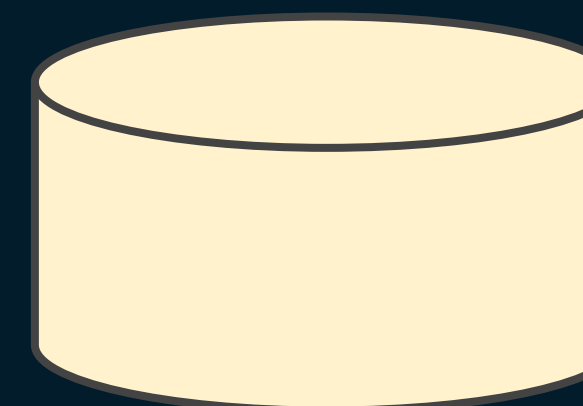
User code

使用者代码



Local state  
backend

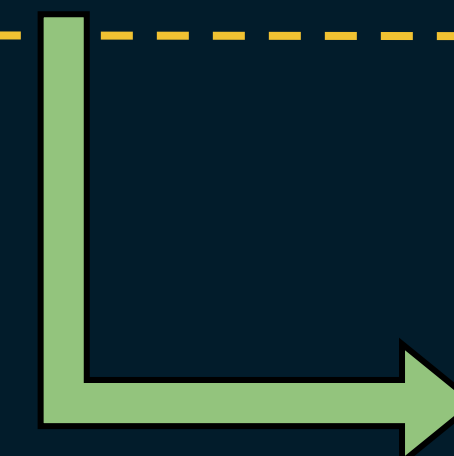
本地状态后端



local read / writes  
that manipulate state

Persisted  
savepoint

持久保存点



persist to DFS  
on savepoint

# Anatomy of a Flink stream job upgrade

Flink 流式应用升级流程解析

User code

使用者代码



upgrade application



Local state  
backend

本地状态后端

Persisted  
savepoint

持久保存点





# Anatomy of a Flink stream job upgrade

Flink 流式应用升级流程解析

User code

使用者代码



Local state  
backend

本地状态后端



Persisted  
savepoint

持久保存点

Restore state  
to state  
backends



# Anatomy of a Flink stream job upgrade

Flink 流式应用升级流程解析

User code

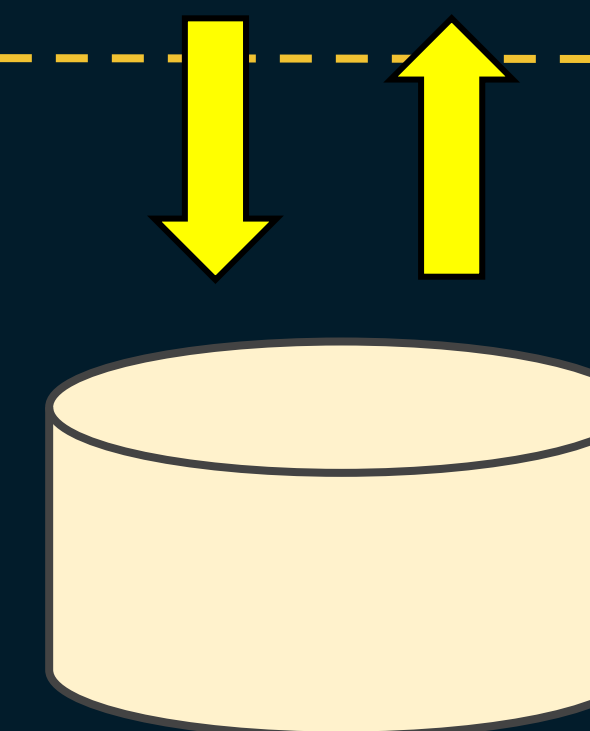
使用者代码



Local state  
backend

本地状态后端

continue to  
access state



Persisted  
savepoint

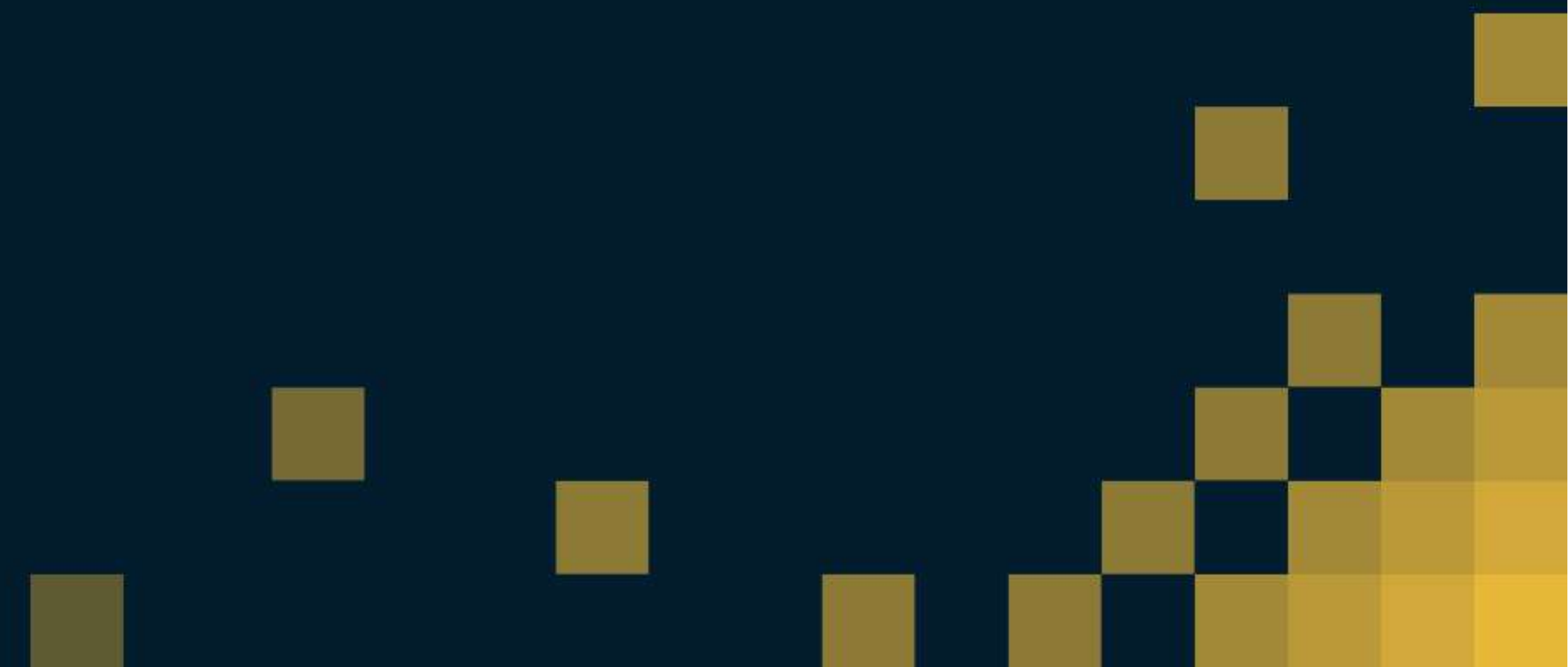
持久保存点





# Schema Evolution for Built-In Types

Flink 内建类别的数据结构定义更新



# State registration with built-in serialization

## 状态注册时使用内建序列化器

```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        MyStateType.class  
    );  
  
ValueState<MyStateType> state = getRuntimeContext().getState(desc);
```



# State registration with built-in serialization

## 状态注册时使用内建序列化器

```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        MyStateType.class  
    );
```

type information for state  
状态类别资讯

```
ValueState<MyStateType> state = getRuntimeContext().getState(desc);
```

# State registration with built-in serialization

## 状态注册时使用内建序列化器

```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        MyStateType.class
    );
```

type information for state  
状态类别资讯

```
ValueState<MyStateType> state = getRuntimeContext().getState(desc);
```

- Flink infers information about the type and creates a serializer for it
  - Primitive types: IntSerializer, DoubleSerializer, LongArraySerializer, etc.
  - Tuples: TupleSerializer
  - POJOs / Scala case classes: PojoSerializer, CaseClassSerializer
  - Apache Avro types: AvroSerializer
  - Fallback is Kryo: KryoSerializer



# Evolving state schema for Apache Avro types

以 Apache Avro 进行状态数据结构定义进化

- Can evolve schema according to Avro specifications\*  
可依据 Avro 规范\* 进化状态的数据结构定义
- Can swap between GenericRecord and code generated SpecificRecords  
可交替使用 GenericRecord 与代码生成的 SpecificRecord 类别
- Cannot change namespace of generated SpecificRecord classes  
不可更动 SpecificRecord 类别的命名空间

---

\*Avro specifications: <http://avro.apache.org/docs/1.7.7/spec.html#Schema+Resolution>

# Status quo of schema evolution support

## 内建型别的数据结构定义升级支援度现况

- Avro types are the only built-in types that support schema evolution (as of 1.7)  
目前仅有 Avro 型别有支援数据结构定义升级 (Flink 1.7 现况)
- More is planned for 1.8+: POJOs, Scala case classes, Rows (for Flink Tables)  
社群有规划支援 POJOs, Scala case class, Rows 等类别的数据结构定义升级
- Avoid using Kryo if you want evolvable schema for state  
若希望支援数据结构定义升级, 请避免使用 KryoSerializer



# Implementing Custom State Serializers

自订状态序列化器的实现



# State registration with custom serializers

## 状态注册时使用自订序列化器

```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new MyStateTypeSerializer();  
    );
```

```
class MyStateTypeSerializer extends TypeSerializer<MyStateType> { ... }
```

```
ValueState<MyStateType> state = getRuntimeContext().getState(desc);
```

# State Schema and Serialization

## 状态的数据结构定义和序列化

- The terms *data schema* and *serialization format* are interchangeable here  
在此, 「数据结构定义」与「序列化格式」两词可交互替换
- Evolving state's data schema requires evolving the state's serializer  
欲升级状态的数据结构定义则必须升级状态的序列化器
- Depending on serialization behaviour of state backends (heap v.s. off-heap)  
state migration may be required  
基于不同状态后端 (内存 / 非内存) 的序列化模式, 可能需要进行状态迁移



# State Serialization for Heap Backends

内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV1()  
    );
```

Local state  
backend

本地状态后端



Persisted  
savepoint

持久保存点



# State Serialization for Heap Backends

内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV1()  
    );
```

Local state  
backend

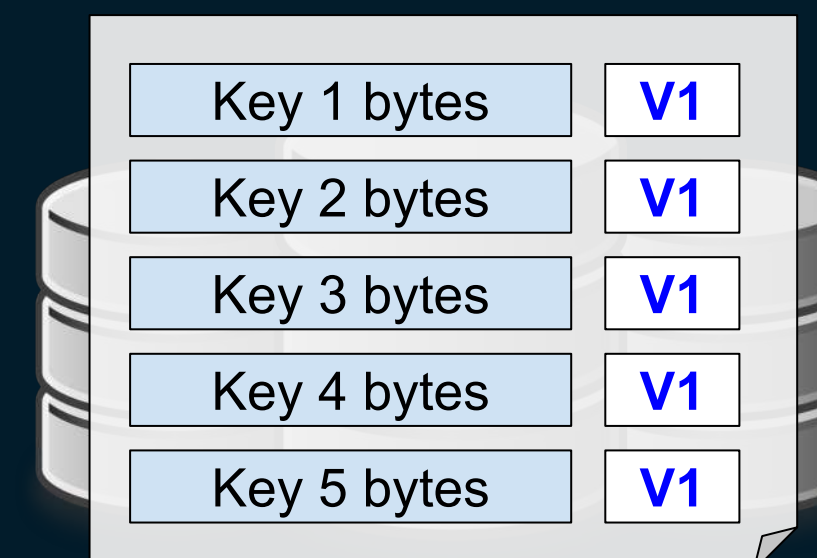
本地状态后端



Persisted  
savepoint

持久保存点

Serialized by  
V1 serializer



# State Serialization for Heap Backends

内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```



Local state  
backend

本地状态后端

Persisted  
savepoint

持久保存点

Key 1 bytes	V1
Key 2 bytes	V1
Key 3 bytes	V1
Key 4 bytes	V1
Key 5 bytes	V1



# State Serialization for Heap Backends

内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```



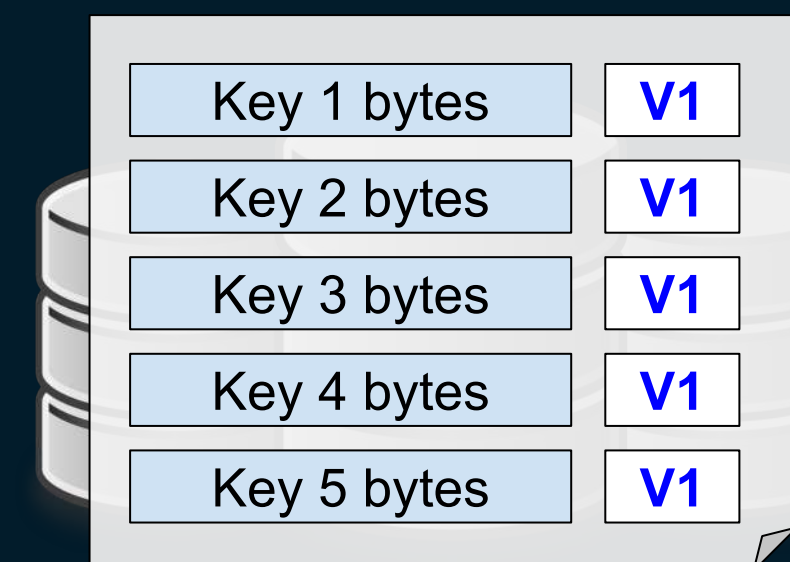
Local state  
backend

本地状态后端

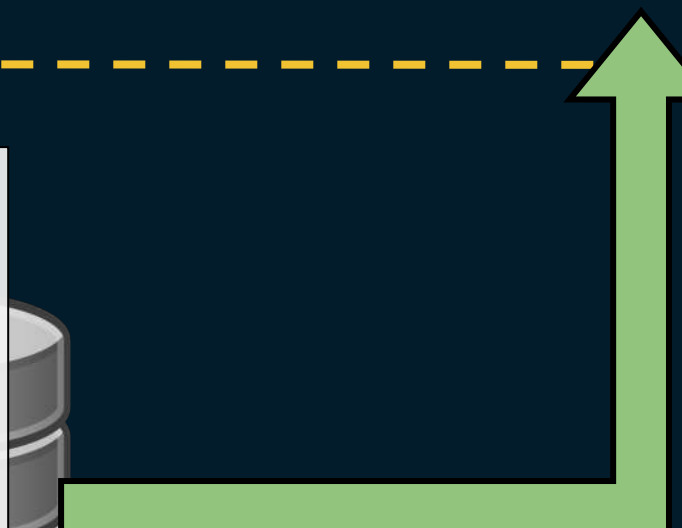


Persisted  
savepoint

持久保存点



Requires  
V1 serializer  
for restore



# State Serialization for Heap Backends

内存式后端的状态序列化模式

User code

使用者代码

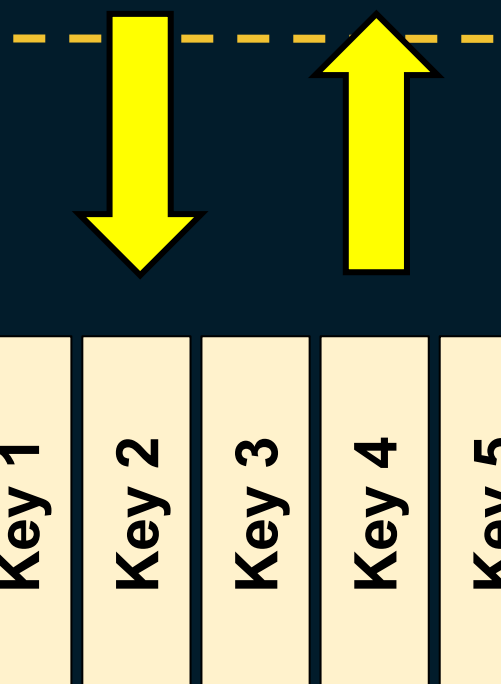


```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV2()  
    );
```



Local state  
backend

本地状态后端



Persisted  
savepoint

持久保存点

Key 1 bytes	V1
Key 2 bytes	V1
Key 3 bytes	V1
Key 4 bytes	V1
Key 5 bytes	V1

# State Serialization for Heap Backends

内存式后端的状态序列化模式

User code

使用者代码

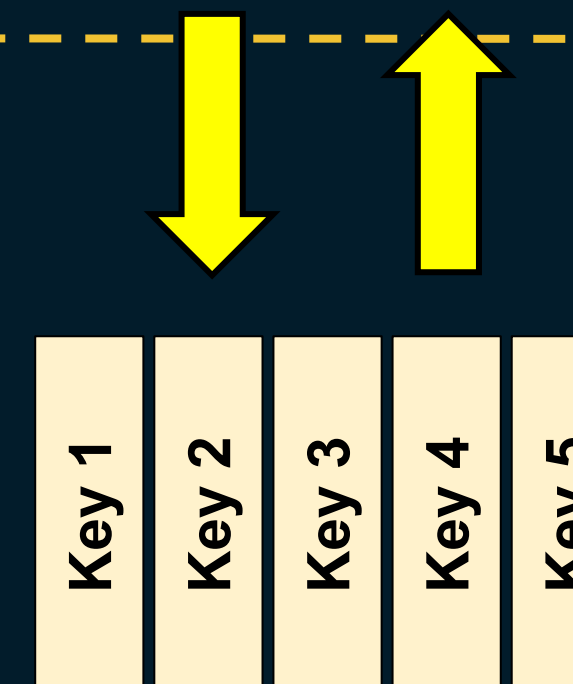


```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```



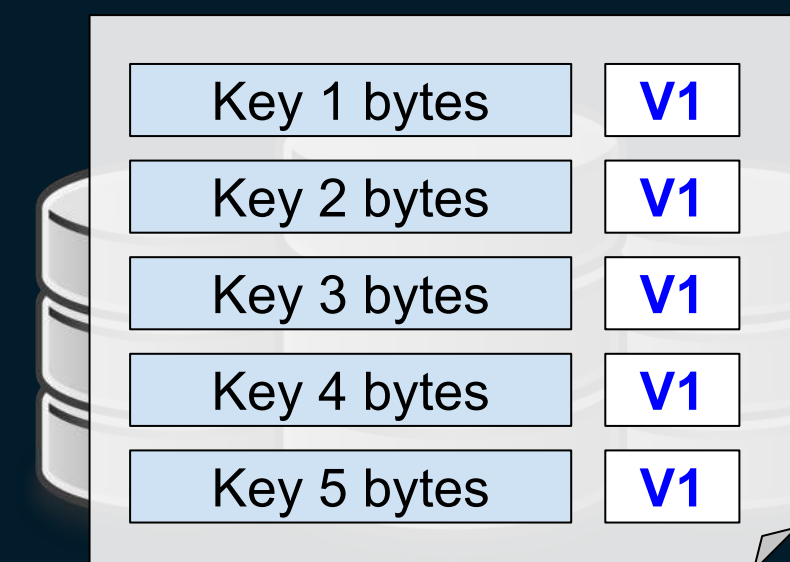
Local state  
backend

本地状态后端

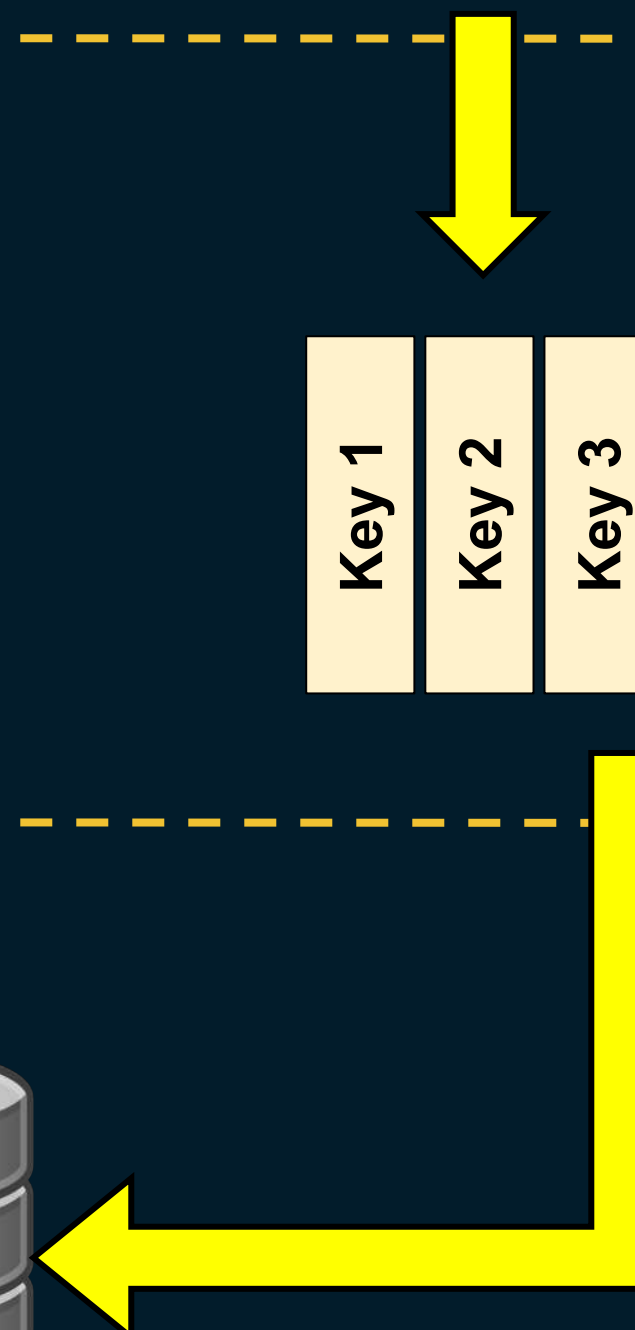


Persisted  
savepoint

持久保存点



Serialized by  
V2 serializer





# State Serialization for Heap Backends

## 内存式后端的状态序列化模式

- Serialization happens on restore + snapshot:  
*lazy* serialization, *eager* deserialization  
反序列化发生于状态恢复阶段、序列化发生于状态的保存点生成
- By nature, restoring + snapshotting state is already a state migration process  
状态的恢复与保存点生成本质上就是一个状态迁移的过程
- Requires a written form of the previous serializer in the snapshot  
需要状态之前的序列化器被写入于保存点中

# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

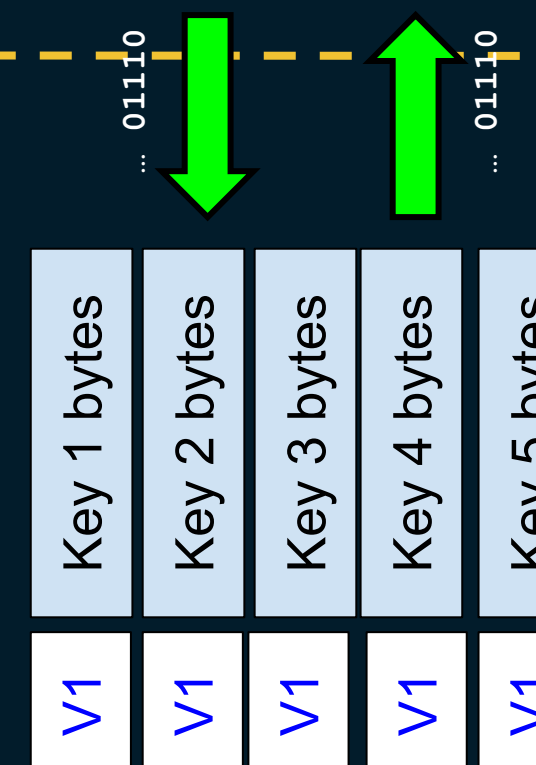
使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV1()  
    );
```

Local state backend

本地状态后端



Persisted savepoint

持久保存点



# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

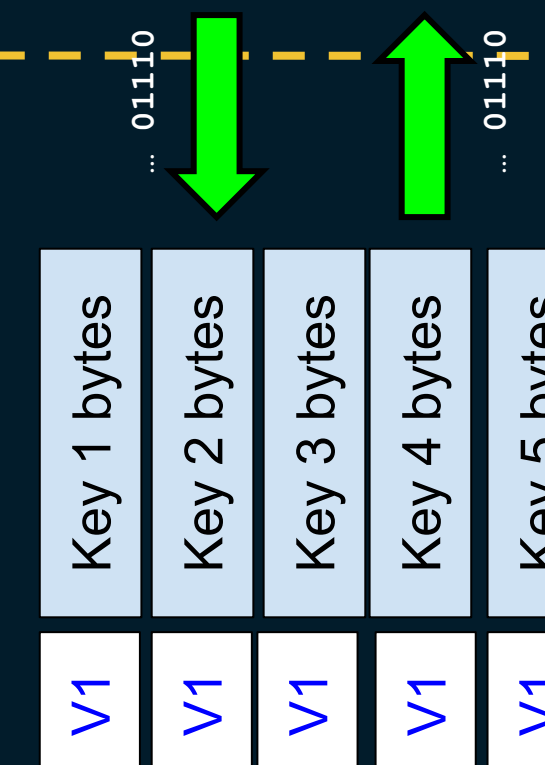
使用者代码



```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV1()
    );
```

Local state backend

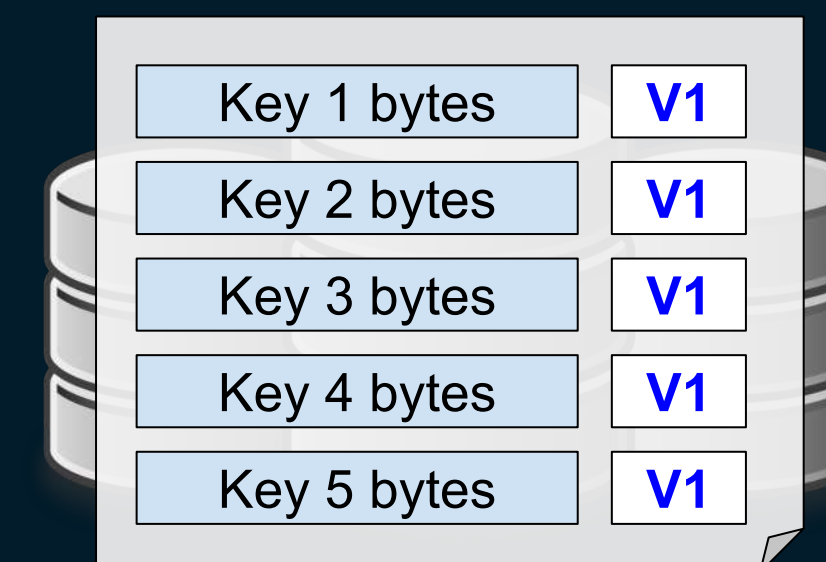
本地状态后端



Persisted savepoint

持久保存点

File transfer





# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV2()  
    );
```

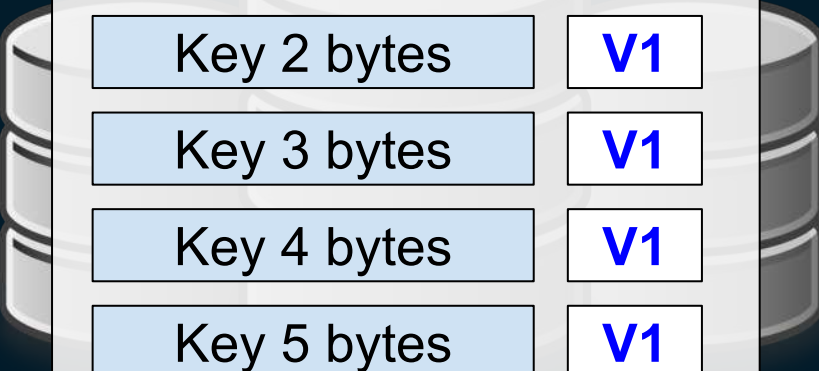


Local state  
backend

本地状态后端

Persisted  
savepoint

持久保存点



Key 1 bytes	V1
Key 2 bytes	V1
Key 3 bytes	V1
Key 4 bytes	V1
Key 5 bytes	V1

# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV2()  
    );
```



Local state backend

本地状态后端

Key 1 bytes	Key 2 bytes	Key 3 bytes	Key 4 bytes	Key 5 bytes
V1	V1	V1	V1	V1

Persisted savepoint

持久保存点

Key 1 bytes	V1
Key 2 bytes	V1
Key 3 bytes	V1
Key 4 bytes	V1
Key 5 bytes	V1

File transfer



# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码



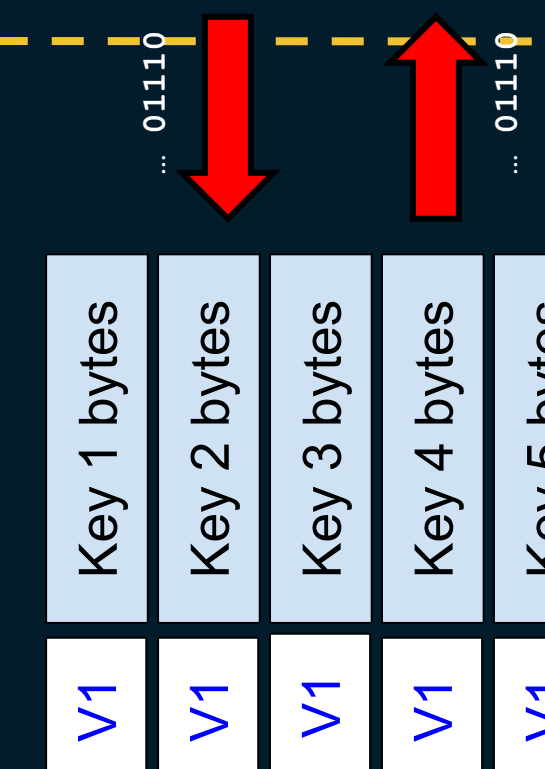
```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV2()  
    );
```



Local state backend

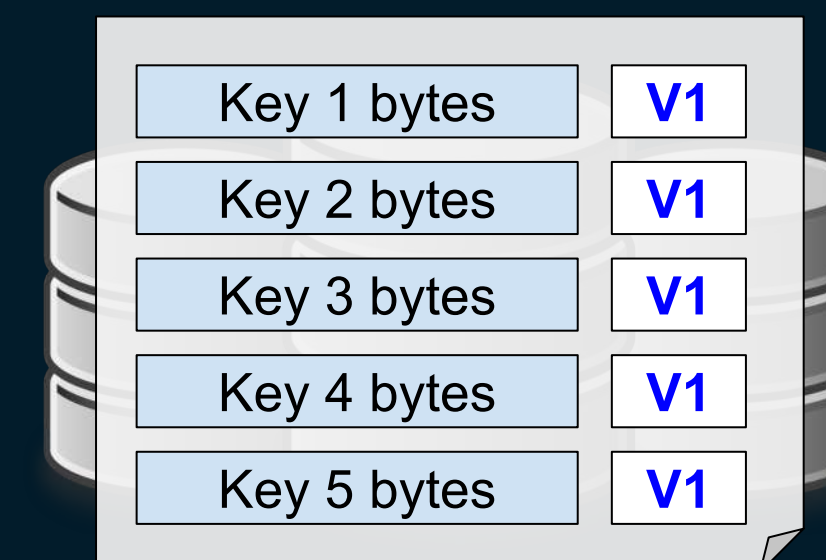
本地状态后端

state access with  
V2 serializer?



Persisted savepoint

持久保存点





# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV2()  
    );
```



Local state backend

本地状态后端

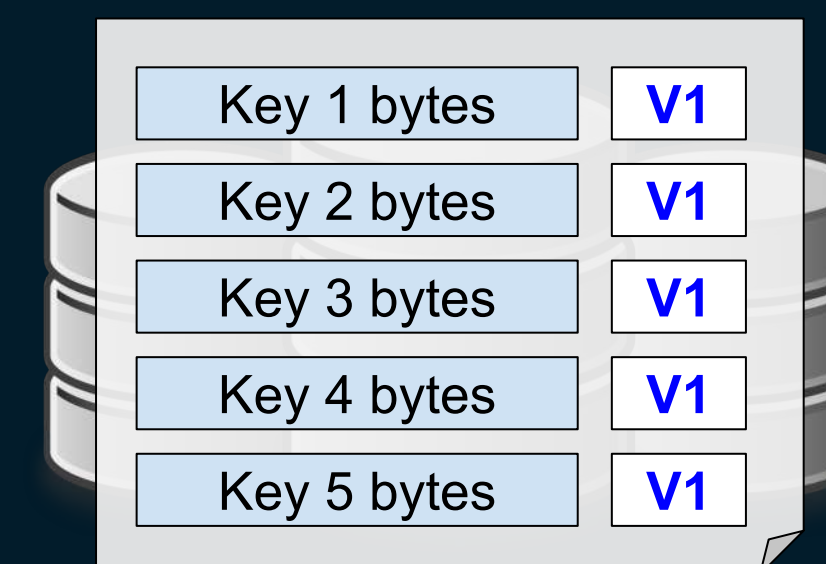
state access with  
V2 serializer?

Requires Migration!



Persisted savepoint

持久保存点



# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码

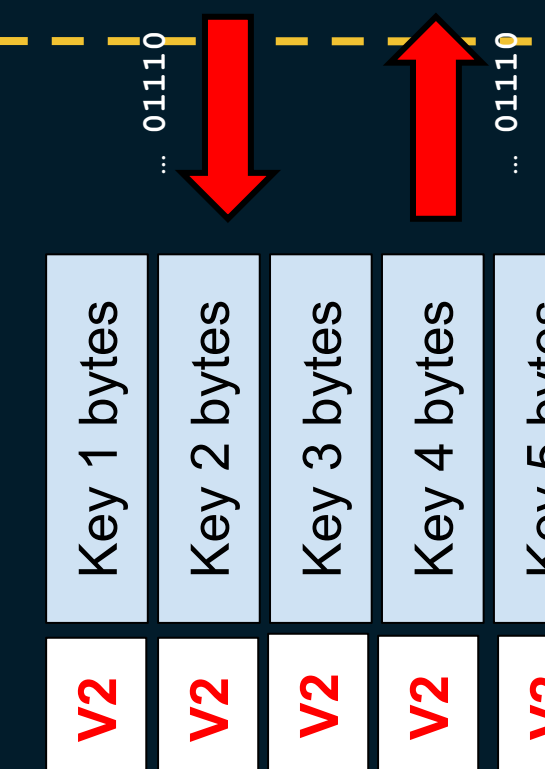


```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```



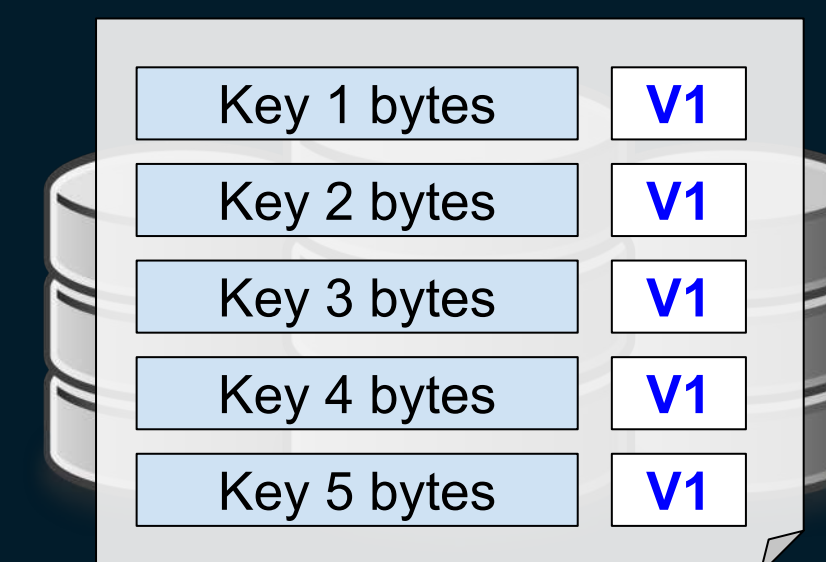
Local state backend

本地状态后端



Persisted savepoint

持久保存点



# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码

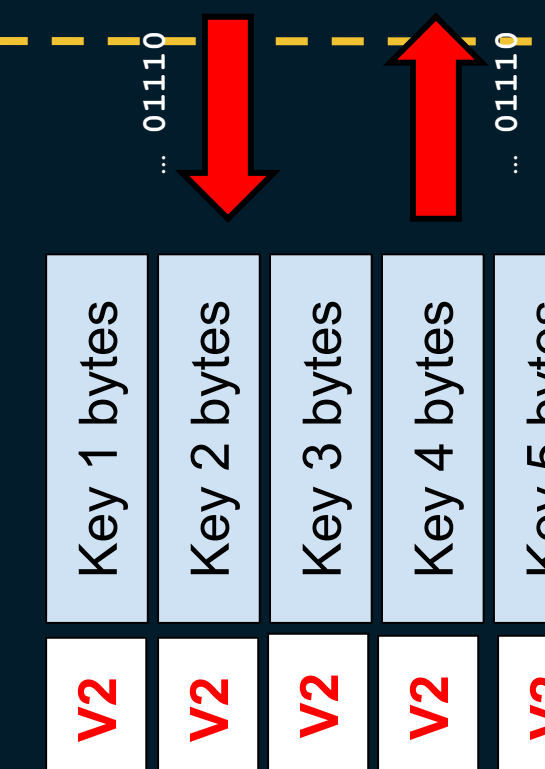


```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```



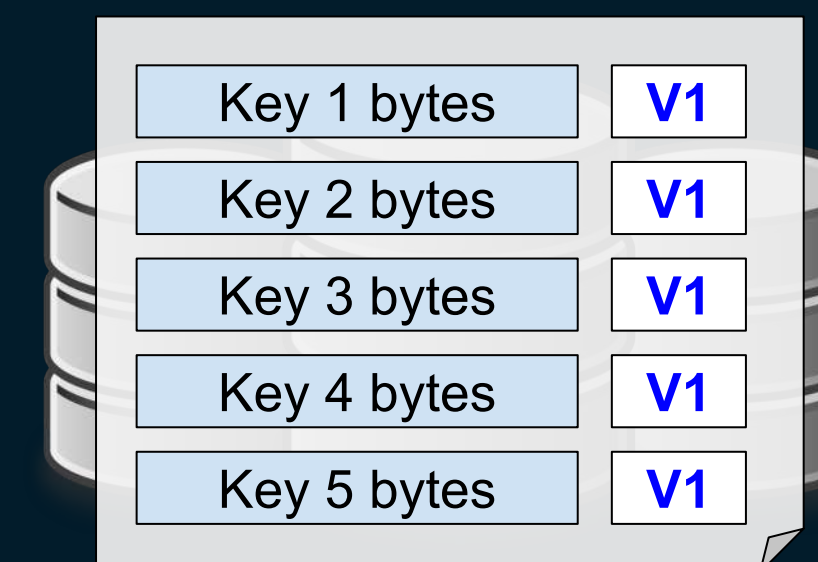
Local state backend

本地状态后端



Persisted savepoint

持久保存点



File transfer





# State Serialization for Out-of-Core Backends

## 非内存式后端的状态序列化模式

- Serialization happens on every state access:  
*Eager* serialization, *lazy* deserialization  
序列化、反序列化会发生于每一次状态的读写
- After restore, state migration occurs on first access if schema has changed  
状态恢复后，第一次的状态注册即视需求进行发生状态迁移
- The previous serializer is required if state migration occurs  
若需要进行状态迁移，则需要使用到状态的前一个序列化器

# Main abstraction: TypeSerializerSnapshot

编程抽象类 : TypeSerializerSnapshot

```
interface TypeSerializerSnapshot<T> {  
    int getCurrentVersion();  
    void writeSnapshot(DataOutputView out);  
    void readSnapshot(int readVersion, DataInputView in, ClassLoader userCodeClassLoader);  
    TypeSerializer<T> restoreSerializer();  
    TypeSerializerSchemaCompatibility<T> resolveSchemaCompatibility(TypeSerializer<T> newSerializer);  
}
```

# Main abstraction: TypeSerializerSnapshot

编程抽象类 : TypeSerializerSnapshot

```
interface TypeSerializerSnapshot<T> {  
    int getCurrentVersion();  
    void writeSnapshot(DataOutputView out);  
    void readSnapshot(int readVersion, DataInputView in, ClassLoader userCodeClassLoader);  
    TypeSerializer<T> restoreSerializer();  
    TypeSerializerSchemaCompatibility<T> resolveSchemaCompatibility(TypeSerializer<T> newSerializer);  
}
```

- Represents the written form of a state's serializer, written to snapshots  
代表着写入于保存点中状态的序列化器
- Encodes information about the state's written schema + serializer configuration  
拥有关于状态被序列化的格式以及序列化器的设定相关资讯
- Serves as a factory for the previous serializer  
可用于建构状态被写入时所使用的序列化器



# State Serialization for Heap Backends

内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV1()  
    );
```

Local state  
backend

本地状态后端

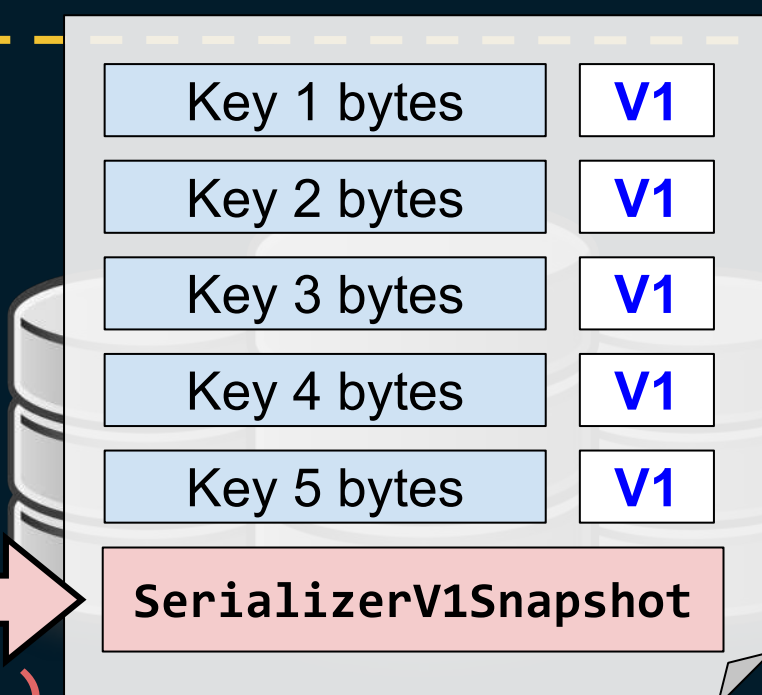


Persisted  
savepoint

持久保存点

Serialized by  
SerializerV1

SerializerV1  
.snapshotConfiguration.write(...)



# State Serialization for Heap Backends

内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```



Local state  
backend

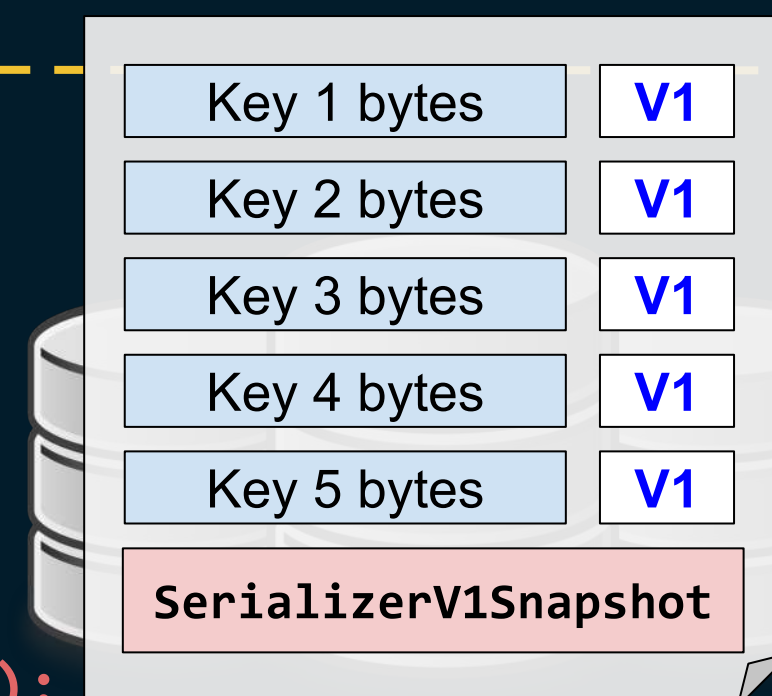
本地状态后端



Persisted  
savepoint

持久保存点

```
SerializerV1Snapshot
    .restoreSerializer();
```



SerializerV1



# State Serialization for Heap Backends

内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```

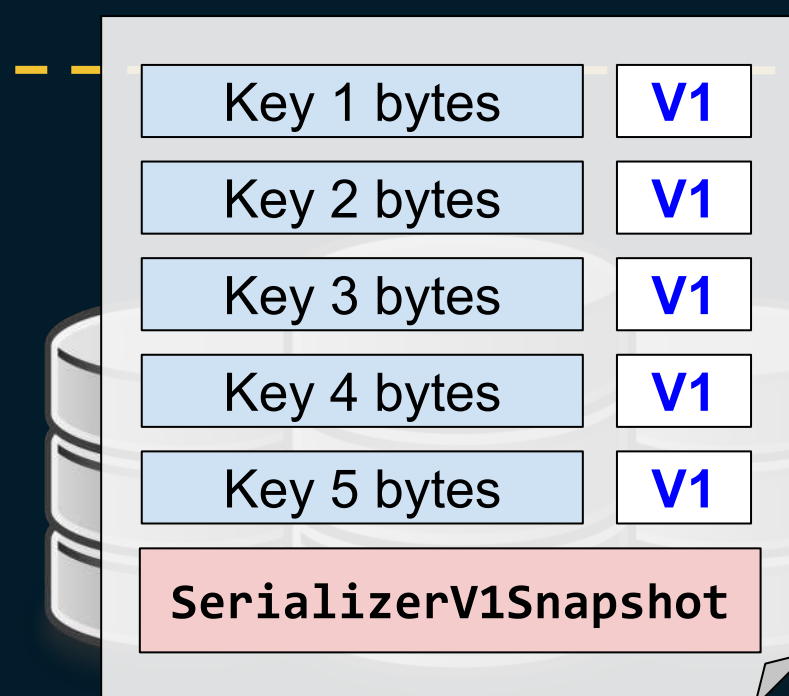


Local state  
backend

本地状态后端

Persisted  
savepoint

持久保存点



SerializerV1

Deserialized by  
SerializerV1



# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码



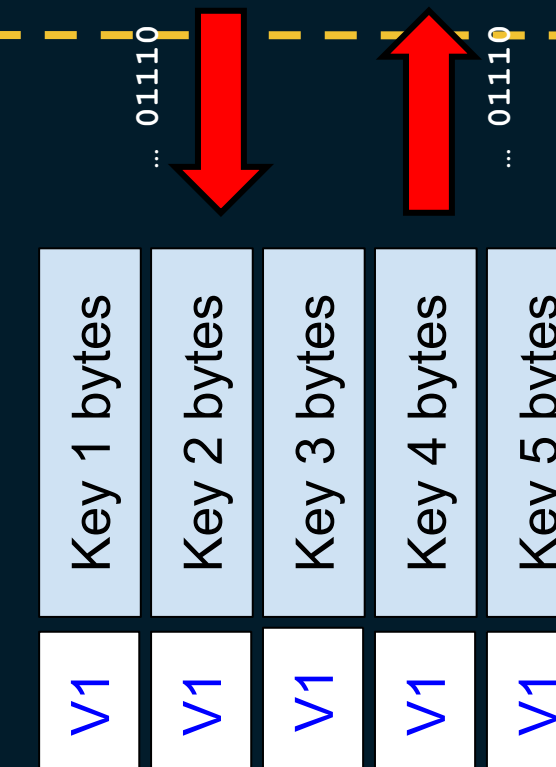
```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV2()  
    );
```



Local state backend

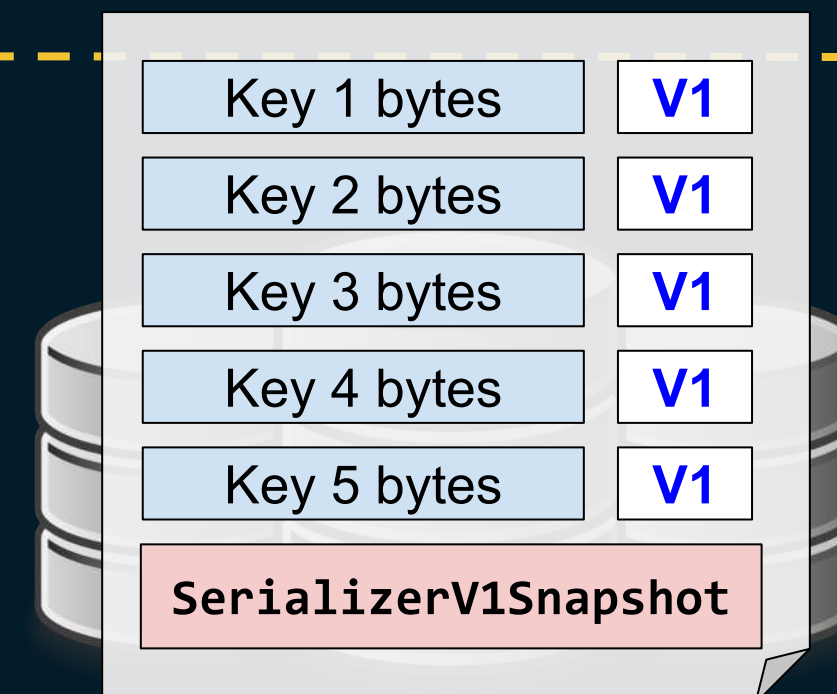
本地状态后端

state access with  
V2 serializer?



Persisted savepoint

持久保存点



# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV2()  
    );
```



Local state backend

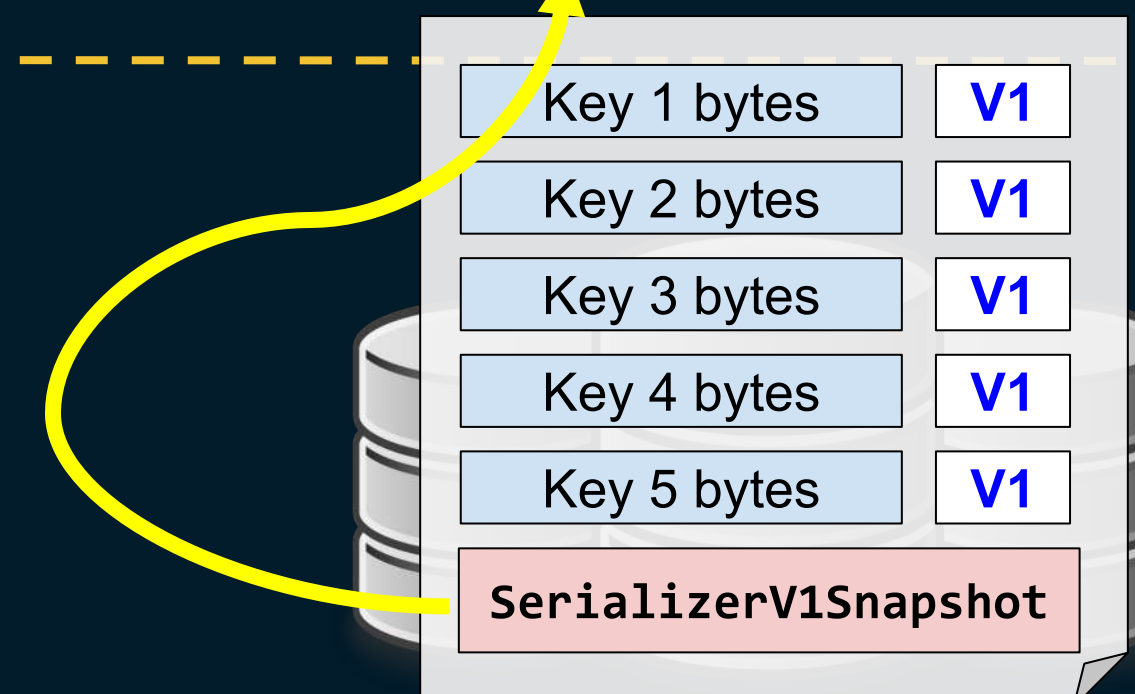
本地状态后端

```
TypeSerializerSchemaCompatibility<T> compat =  
    serializerV1Snapshot  
        .resolveSchemaCompatibility(serializerV2)  
  
if (compat.isCompatibleAfterMigration()) {  
    // migrate the state schema  
}
```

Key 1 bytes	Key 2 bytes	Key 3 bytes	Key 4 bytes	Key 5 bytes
V1	V1	V1	V1	V1

Persisted savepoint

持久保存点





# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```



Local state backend

本地状态后端

SerializerV1

Key 1 bytes	Key 2 bytes	Key 3 bytes	Key 4 bytes	Key 5 bytes
V1	V1	V1	V1	V1

Persisted savepoint

持久保存点

```
SerializerV1Snapshot
    .restoreSerializer();
```

Key 1 bytes	V1
Key 2 bytes	V1
Key 3 bytes	V1
Key 4 bytes	V1
Key 5 bytes	V1
SerializerV1Snapshot	



# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV2()  
    );
```



Local state backend

本地状态后端

State object

SerializerV1

read

Key 1 bytes	Key 2 bytes	Key 3 bytes	Key 4 bytes	Key 5 bytes
V1	V1	V1	V1	V1

Persisted savepoint

持久保存点

Key 1 bytes	V1
Key 2 bytes	V1
Key 3 bytes	V1
Key 4 bytes	V1
Key 5 bytes	V1
SerializerV1Snapshot	

# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV2()  
    );
```



Local state backend

本地状态后端

State object

SerializerV2

SerializerV1

Key 1 bytes	Key 2 bytes	Key 3 bytes	Key 4 bytes	Key 5 bytes
V1	V1	V1	V1	V1

Persisted savepoint

持久保存点

Key 1 bytes	V1
Key 2 bytes	V1
Key 3 bytes	V1
Key 4 bytes	V1
Key 5 bytes	V1
SerializerV1Snapshot	



# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码



```
ValueStateDescriptor<MyStateType> desc =  
    new ValueStateDescriptor<>(  
        "my-value-state",  
        new SerializerV2()  
    );
```



Local state backend

本地状态后端

State object

SerializerV2

write

SerializerV1

Key 1 bytes	Key 2 bytes	Key 3 bytes	Key 4 bytes	Key 5 bytes
V1	V1	V1	V1	V1

Persisted savepoint

持久保存点

Key 1 bytes	V1
Key 2 bytes	V1
Key 3 bytes	V1
Key 4 bytes	V1
Key 5 bytes	V1
SerializerV1Snapshot	



# State Serialization for Out-of-Core Backends

非内存式后端的状态序列化模式

User code

使用者代码

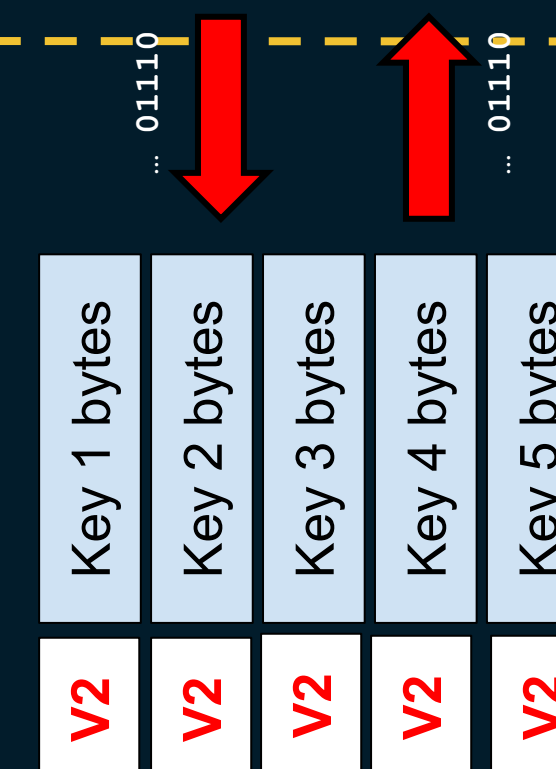


```
ValueStateDescriptor<MyStateType> desc =
    new ValueStateDescriptor<>(
        "my-value-state",
        new SerializerV2()
    );
```



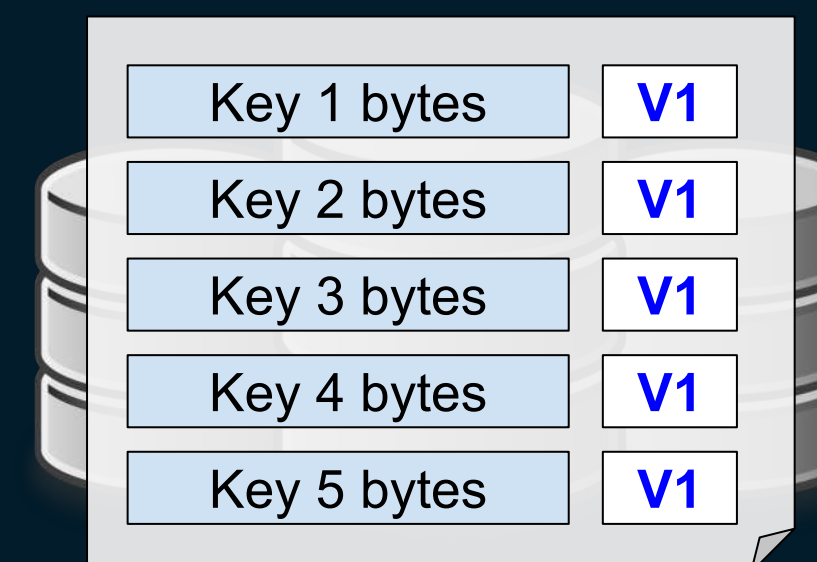
Local state backend

本地状态后端



Persisted savepoint

持久保存点



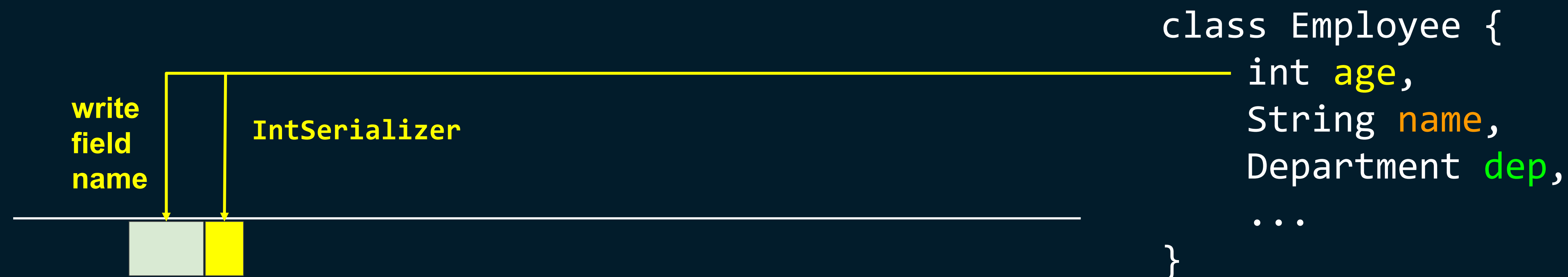
## Example: Evolvable PojoSerializer [\[FLINK-10987\]](#)

范例：可进化序列化格式的 PojoSerializer

```
class Employee {  
    int age,  
    String name,  
    Department dep,  
    ...  
}
```

# Example: Evolvable PojoSerializer [\[FLINK-10987\]](#)

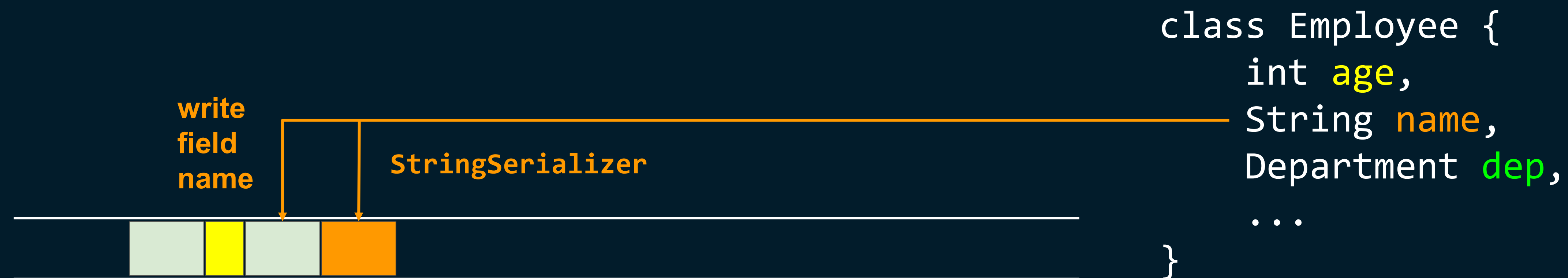
范例：可进化序列化格式的 PojoSerializer





# Example: Evolvable PojoSerializer [\[FLINK-10987\]](#)

范例：可进化序列化格式的 PojoSerializer



# Example: Evolvable PojoSerializer [\[FLINK-10987\]](#)

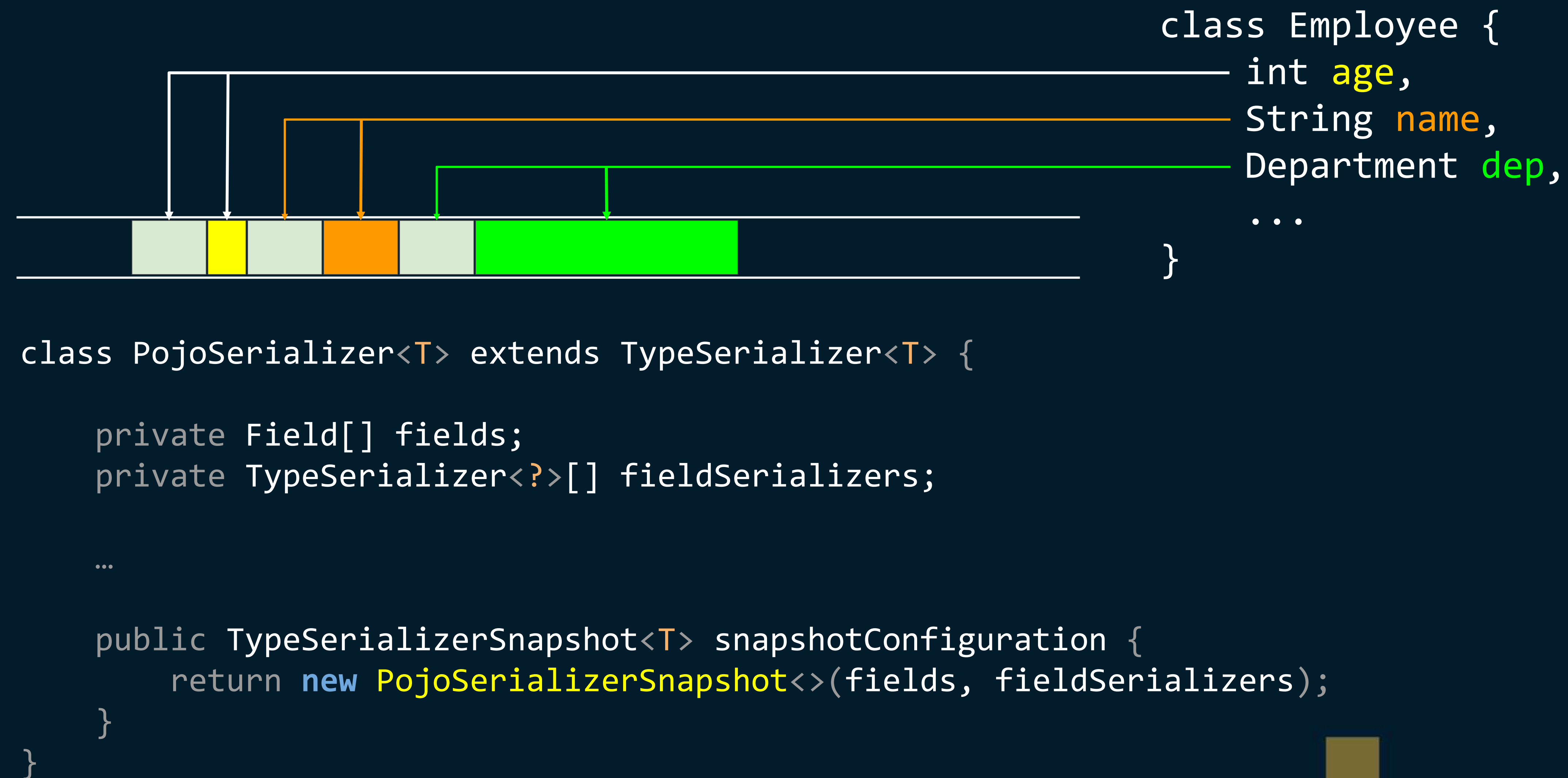
范例：可进化序列化格式的 PojoSerializer



```
class Employee {
    int age,
    String name,
    Department dep,
    ...
}
```

# Example: Evolvable PojoSerializer [\[FLINK-10987\]](#)

范例：可进化序列化格式的 PojoSerializer





# Example: Evolvable PojoSerializer [\[FLINK-10987\]](#)

范例：可进化序列化格式的 PojoSerializer

```
class PojoSerializerSnapshot<T> implements TypeSerializerSnapshot<T> {

    private Field[] fields;
    private TypeSerializer<?>[] fieldSerializers;

    /**
     * Constructor for instantiating the snapshot when reading.
     */
    public PojoSerializerSnapshot() {}

    /**
     * Constructor to create a snapshot for writing.
     */
    public PojoSerializerSnapshot(Field[] fields, TypeSerializer<?>[]
fieldSerializers) {
        this.fields = fields;
        this.fieldSerializers = fieldSerializers;
    }

    ...
}
```

# Example: Evolvable PojoSerializer [\[FLINK-10987\]](#)

范例：可进化序列化格式的 PojoSerializer

```
class PojoSerializerSnapshot<T> implements TypeSerializerSnapshot<T> {

    ...

    public TypeSerializerSchemaCompatibility<T> resolveSchemaCompatibility(TypeSerializer<T> newSerializer) {
        if (newSerializer instanceof PojoSerializer) {
            Field[] newFields = ((PojoSerializer<T>) newSerializer).getFields();

            if (hasDifferentTypedFields(this.fields, newFields)) {
                return TypeSerializerSchemaCompatibility.incompatible();
            } else if (hasNewFields(this.fields, newFields) || hasRemovedFields(this.fields, newFields)) {
                return TypeSerializerSchemaCompatibility.compatibleAfterMigration();
            }

            return TypeSerializerSchemaCompatibility.compatibleAsIs();
        }

        return TypeSerializerSchemaCompatibility.incompatible();
    }
}
```

# Example: Evolvable PojoSerializer [\[FLINK-10987\]](#)

范例：可进化序列化格式的 PojoSerializer

```
class PojoSerializerSnapshot<T> implements TypeSerializerSnapshot<T> {  
  
    ...  
  
    public TypeSerializer<T> restoreSerializer() {  
        return new PojoSerializer<>(fields, fieldSerializers);  
    }  
}
```



## Miscellaneous Best Practices

实现最佳守则

- Avoid classname changes to the serializer snapshot class  
避免 `TypeSerializerSnapshot` 实现类名被更动
  - Classname is the entrypoint to reading a serializer snapshot  
类名为读取 `TypeSerializerSnapshot` 的入口点
  - Avoid using anonymous or nested classes for snapshot classes  
避免使用匿名类或巢状类作为 `TypeSerializerSnapshot` 的实现
- Use `CompositeSerializerSnapshot` to handle nested `TypeSerializers`  
可利用 `CompositeSerializerSnapshot` 类应付巢状的 `TypeSerializer`

# Conclusion

总结

- Flink 1.7 now supports state schema evolution

自 Flink 1.7 开始支援状态的数据结构定义升级

- Avro schema evolution is supported; more support is on the radar

支援 Avro 数据结构定义升级；支援其他原生类别的数据结构定义升级将会在未来持续增加

- Covered details on implementing custom state serializers with evolve-able schema

针对可升级数据结构定义的状态序列化器的实现方法进行解析



**THANKS**







# Flink Forward San Francisco 2019

## The Apache Flink® Conference

Stream Processing | Event Driven | Real Time

San Francisco | April 1-2, 2019

Register



Organized by dataArtisans

#flinkforward

<https://sf-2019.flink-forward.org/register>