Kotlin 中文开发者大会

Ktorm: 让你的数据库操作更具 Kotlin 风味的 ORM 框架

刘文俊

Developed by @vincentlauvlwj

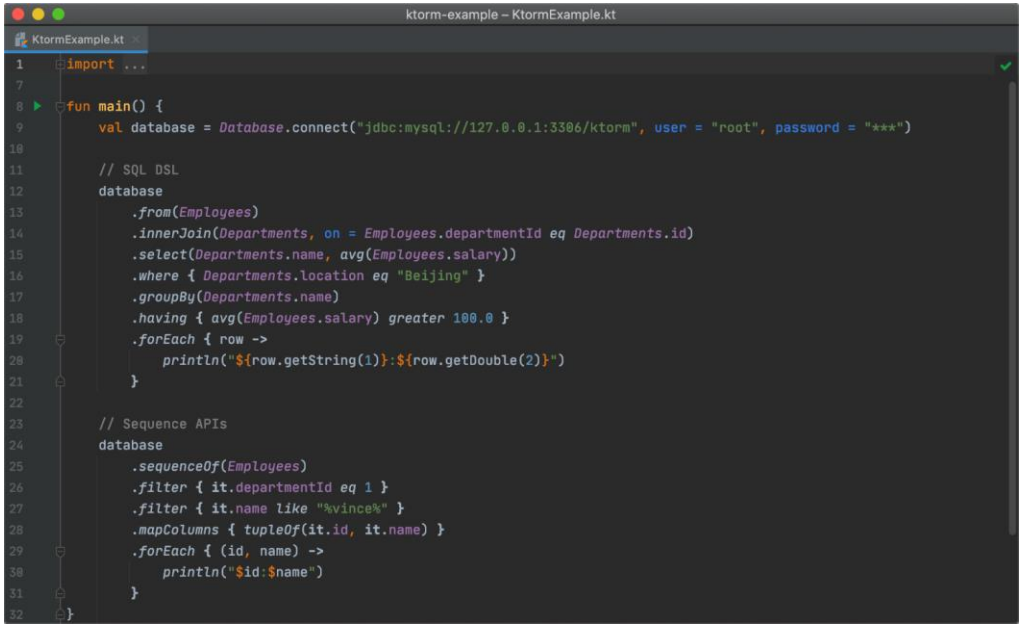
Ktorm 是什么?

为 Kotlin 设计的 ORM 框架

1. 基于 JDBC，面向服务端
2. Sequence API，与 Kotlin 原生集合类似的使用体验
3. 强类型 SQL DSL
4. 无配置文件，无三方依赖，轻量级，高效简洁

官网：https://www.ktorm.org

GitHub：https://github.com/kotlin-orm/ktorm



快速上手

- 添加依赖（Gradle）
- 添加依赖（Maven）
- Hello, World!

添加依赖 (Gradle)

```
plugins {
    kotlin("jvm") version "1.9.23"
    id("com.google.devtools.ksp") version "1.9.23-1.0.20"
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib"))
    implementation(kotlin("reflect"))
    implementation("org.ktorm:ktorm-core:4.1.1")
    implementation("org.ktorm:ktorm-support-mysql:4.1.1")
    implementation("org.ktorm:ktorm-ksp-annotations:4.1.1")
    ksp("org.ktorm:ktorm-ksp-compiler:4.1.1")
}
```

Ktorm 核心库与 MySQL 方言支持

KSP 注解与代码生成插件

添加依赖 (Maven)

```xml
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-reflect</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
    <dependency>
        <groupId>org.ktorm</groupId>
        <artifactId>ktorm-core</artifactId>
        <version>${ktorm.version}</version>
    </dependency>
    <dependency>
        <groupId>org.ktorm</groupId>
        <artifactId>ktorm-support-mysql</artifactId>
        <version>${ktorm.version}</version>
    </dependency>
    <dependency>
        <groupId>org.ktorm</groupId>
        <artifactId>ktorm-ksp-annotations</artifactId>
        <version>${ktorm.version}</version>
    </dependency>
</dependencies>
```

Ktorm 核心库

MySQL 方言支持

KSP 注解支持

```xml
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <extensions>true</extensions>
            <configuration>
                <compilerPlugins>
                    <compilerPlugin>ksp</compilerPlugin>
                </compilerPlugins>
            </configuration>
            <dependencies>
                <dependency>
                    <groupId>org.ktorm</groupId>
                    <artifactId>ktorm-ksp-compiler-maven-plugin</artifactId>
                    <version>${ktorm.version}</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

KSP 代码生成编译器插件

Hello, World!

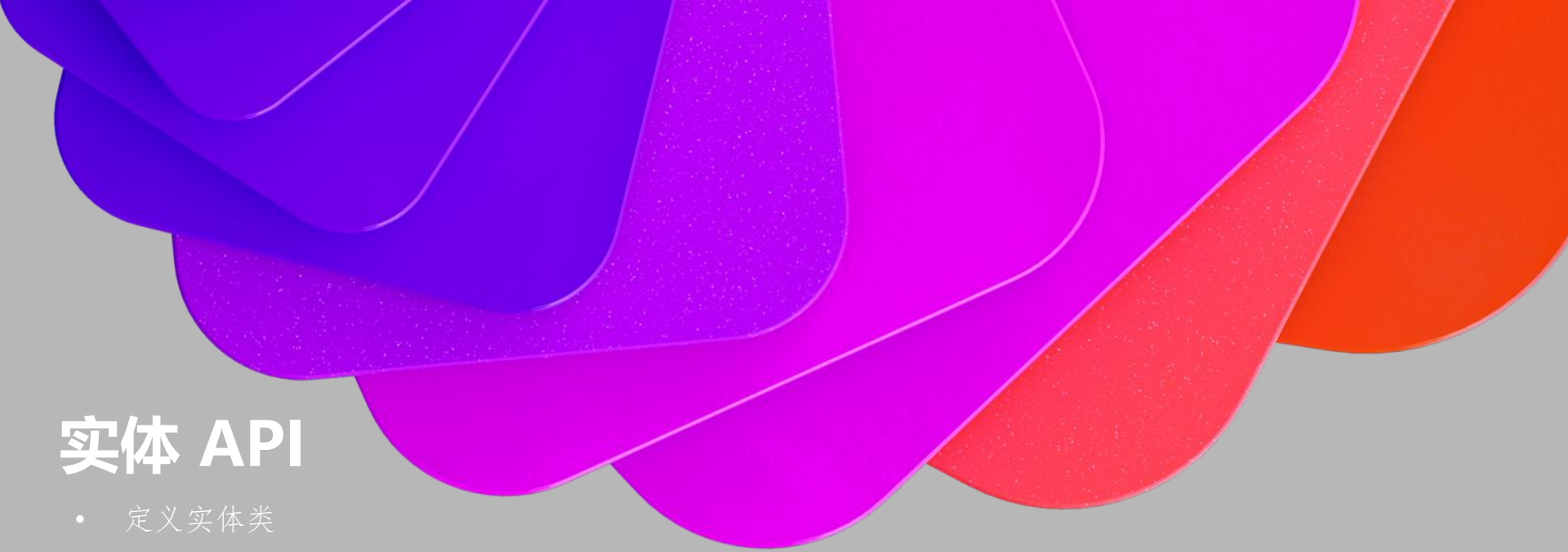
连接数据库

```
fun main() {
    val database = Database.connect("jdbc:mysql://localhost:3306/ktorm", user = "root", password = "***")

    // select * from t_employee
    for (row in database.from(Employees).select()) {
        println(row[Employees.name])
    }
}

object Employees : Table<Nothing>("t_employee") {
    val id = int("id").primaryKey()
    val name = varchar("name")
    val job = varchar("job")
    val managerId = int("manager_id")
    val hireDate = date("hire_date")
    val salary = long("salary")
    val departmentId = int("department_id")
}
```

使用 DSL 创建查询的 Query 对象
Query 类提供了迭代器，因此可以直接使用 for 循环直接对查询结果进行迭代

查询结果 QueryRowSet 重载了索引运算符，因此可以用 [] 获取列的数据

使用 Kotlin object 定义表对象，继承 Table 类
使用 int, long, varchar, date 等函数定义列



实体 API

- 定义实体类
- 实体增删改操作
- 简单查询
- 聚合查询
- 分组聚合查询
- 运算符

定义实体类

```
@Table("t_department")
interface Department : Entity<Department> {
    @PrimaryKey
    var id: Int
    var name: String
    var location: String
}
```

常用注解：

- @Table：标注实体类，可指定表名，默认把实体类名转换成小写下划线分隔的形式作为表名
- @Column：标注实体类的字段，可指定列名，默认把字段名转换成小写下划线分隔的形式作为列名
- @PrimaryKey：标注主键字段
- @References：标注外键关联的实体，支持一对一和多对一关联

```
@Table("t_employee")
interface Employee : Entity<Employee> {
    @PrimaryKey
    var id: Int
    var name: String
    var job: String
    var managerId: Int?
    var hireDate: LocalDate
    var salary: Long
    @References
    var department: Department
}
```

KSP 生成的代码

```kotlin
/**
 * Table t_department.
 */
public open class Departments(alias: String?) : Table<Department>("t_department", alias) {
    /**
     * Column id.
     */
    public val id: Column<Int> = int("id").primaryKey().bindTo { it.id }

    /**
     * Column name.
     */
    public val name: Column<String> = varchar("name").bindTo { it.name }

    /**
     * Column location.
     */
    public val location: Column<String> = varchar("location").bindTo { it.location }

    /**
     * Return a new-created table object with all properties (including the table name and columns and
     * so on) being copied from this table, but applying a new alias given by the parameter.
     */
    public override fun aliased(alias: String): Departments = Departments(alias)

    /**
     * The default table object of t_department.
     */
    public companion object : Departments(alias = null)
}

/**
 * Return the default entity sequence of [Departments].
 */
public val Database.departments: EntitySequence<Department, Departments> get() = this.sequenceOf(Departments)
```

```kotlin
/**
 * Create an entity of [Department] and specify the initial values for each property, properties
 * that doesn't have an initial value will leave unassigned.
 */
public fun Department(
    id: Int? = Undefined.of(),
    name: String? = Undefined.of(),
    location: String? = Undefined.of()
): Department {...}

/**
 * Return a deep copy of this entity (which has the same property values and tracked statuses), and
 * alter the specified property values.
 */
public fun Department.copy(
    id: Int? = Undefined.of(),
    name: String? = Undefined.of(),
    location: String? = Undefined.of()
): Department {...}

/**
 * Return the value of [Department.id].
 */
public operator fun Department.component1(): Int = this.id

/**
 * Return the value of [Department.name].
 */
public operator fun Department.component2(): String = this.name

/**
 * Return the value of [Department.location].
 */
public operator fun Department.component3(): String = this.location
```

Why interface?

使用 interface 定义实体类，Ktorm 得以跟踪实体对象内部的状态变化，也能给实体对象注入一些数据操作的方法，从而实现 active record 模式的对象操作，例如：

```
// select * from t_employee where id = ?
// update t_employee set job = ?, salary = ? where id = ?
val e1 = database.employees.find { it.id eq 666 }
if (e1 != null) {
    e1.job = "CEO"
    e1.salary += 100_000_000
    e1.flushChanges()
}

// select * from t_employee where id = ?
// delete from t_employee where id = ?
val e2 = database.employees.find { it.id eq 777 }
e2?.delete()
```

那么，要怎样创建实体对象呢？

```
// 创建实体对象并保存到数据库
// insert into t_employee (name, job, hire_date) values (?, ?, ?)
val e3 = Employee(name = "vince", job = "engineer", hireDate = LocalDate.now())
database.employees.add(e3)

// 复制实体对象并修改其 job 字段
// 输出：Employee(id=123, name=vince, job=tech evangelist, hireDate=2024-11-02)
val e4 = e3.copy(job = "tech evangelist")
println(e4)

// 实体对象解构语法
// 输出：id=123, name=vince, job=engineer
val (id, name, job) = e3
println("id=$id, name=$name, job=$job")
```

简单查询

```
// 查询所有员工数据并输出
// select *
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
for (employee in database.employees) {
    println(employee)
}
```

```
// 查询所有员工数据，返回 List<Employee>
// select *
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
val employees = database.employees.toList()
```

```
// 增加筛选条件，查询 ID 为 1 的部门的所有员工，返回 List<Employee>
// select *
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// where t_employee.department_id = ?
val employees = database.employees.filter { it.departmentId eq 1 }.toList()
```

```
// 引用关联表中的字段进行条件过滤，查询技术部门的所有员工，返回 List<Employee>
// select *
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// where _ref0.name = ?
val employees = database.employees
    .filter { it.refs.department.name eq "tech" }
    .toList()
```

```
// 增加多个筛选条件，filter 函数可以重复使用
// select *
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// where (t_employee.department_id = ?) and (t_employee.manager_id is not null)
val employees = database.employees
    .filter { it.departmentId eq 1 }
    .filter { it.managerId.isNotNull() }
    .toList()
```

简单查询

```
// 使用 mapColumns 定制查询返回的字段，返回类型为 List<Tuple3<Int?, String?, Int?>>
// select t_employee.id, t_employee.name, datediff(?, t_employee.hire_date)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// where t_employee.department_id = ?
database.employees
    .filter { it.departmentId eq 1 }
    .mapColumns { tupleOf(it.id, it.name, dateDiff(LocalDate.now(), it.hireDate)) }
    .forEach { (id, name, days) ->
        println("$id:$name:$days")
    }


// 使用 sortedBy 函数对查询结果排序
// select *
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// order by t_employee.salary
val employees = database.employees.sortedBy { it.salary }.toList()


// 支持指定多个排序字段
// select *
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// order by t_employee.salary desc, t_employee.hire_date
val employees = database.employees
    .sortedBy({ it.salary.desc() }, { it.hireDate.asc() })
    .toList()
```

```
// 使用 drop 和 take 分页，并计算页数
val pageNo = 1
val pageSize = 10
val query = database.employees
    .filter { it.departmentId eq 1 }
    .drop((pageNo - 1) * pageSize)
    .take(pageSize)

// select *
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// where t_employee.department_id = ?
// limit ?, ?
val employees = query.toList()

// select count(*)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// where t_employee.department_id = ?
val pageCount =
    if (query.totalRecordsInAllPages % pageSize == 0) {
        query.totalRecordsInAllPages / pageSize
    } else {
        query.totalRecordsInAllPages / pageSize + 1
    }
```

聚合查询

```
// 统计工资超过 10000 的人数
// select count(*)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// where t_employee.salary > ?
val cnt = database.employees.count { it.salary gt 10000 }

// 判断是否存在工资超过 10000 的员工
// count { it.salary gt 10000 } > 0
val any = database.employees.any { it.salary gt 10000 }

// 判断是否不存在工资超过 10000 的员工
// count { it.salary gt 10000 } == 0
val none = database.employees.none { it.salary gt 10000 }

// 判断是否所有员工的工资都超过 10000
// count { it.salary lte 1000 } == 0
val all = database.employees.all { it.salary gt 10000 }

// 计算所有员工的工资之和
// select sum(t_employee.salary)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
val sum = database.employees.sumBy { it.salary }
```

```
// 计算所有员工的最高工资
// select max(t_employee.salary)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
val max = database.employees.maxBy { it.salary }

// 计算所有员工的最低工资
// select min(t_employee.salary)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
val min = database.employees.minBy { it.salary }

// 计算所有员工的平均工资
// select avg(t_employee.salary)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
val avg = database.employees.averageBy { it.salary }

// 同时计算所有员工的平均工资和极差
// select avg(t_employee.salary), max(t_employee.salary) - min(t_employee.salary)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
val (average, range) = database.employees
    .aggregateColumns { tupleOf(avg(it.salary), max(it.salary) - min(it.salary)) }
```

分组聚合查询

```
// 计算每个部门的人数，返回 Map<Int, Int>
// select t_employee.department_id, count(*)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// group by t_employee.department_id
val countMap = database.employees.groupingBy { it.departmentId }.eachCount()
```

```
// 计算每个部门的工资总和，返回 Map<Int, Long>
// select t_employee.department_id, sum(t_employee.salary)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// group by t_employee.department_id
val sumMap = database.employees.groupingBy { it.departmentId }.eachSumBy { it.salary }
```

```
// 计算每个部门的最高工资，返回 Map<Int, Long>
// select t_employee.department_id, max(t_employee.salary)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// group by t_employee.department_id
val maxMap = database.employees.groupingBy { it.departmentId }.eachMaxBy { it.salary }
```

```
// 计算每个部门的最低工资，返回 Map<Int, Long>
// select t_employee.department_id, min(t_employee.salary)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// group by t_employee.department_id
val minMap = database.employees.groupingBy { it.departmentId }.eachMinBy { it.salary }
```

```
// 计算每个部门的平均工资，返回 Map<Int, Double>
// select t_employee.department_id, avg(t_employee.salary)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// group by t_employee.department_id
val avgMap = database.employees.groupingBy { it.departmentId }.eachAverageBy { it.salary }
```

```
// 同时计算每个部门的平均工资和极差，返回 Map<Int, Tuple2<Double, Long>>
// select
//   t_employee.department_id,
//   avg(t_employee.salary),
//   max(t_employee.salary) - min(t_employee.salary)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// group by t_employee.department_id
database.employees
    .groupingBy { it.departmentId }
    .aggregateColumns { tupleOf(avg(it.salary), max(it.salary) - min(it.salary)) }
    .forEach { departmentId, (average, range) ->
        println("departmentId=$departmentId, average=$average, range=$range")
    }
```

运算符

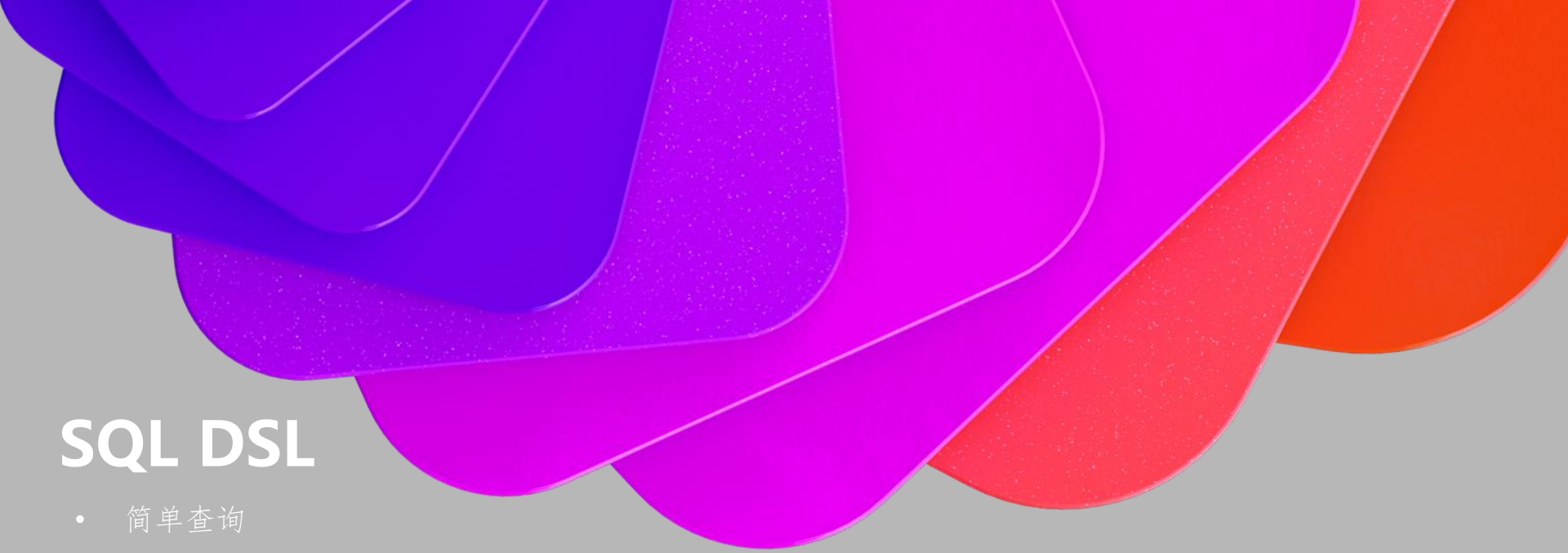
Kotlin 函数名	SQL 关键字/符号	使用示例
isNull	is null	Ktorm: Employees.name.isNull() SQL: t_employee.name is null
isNotNull	is not null	Ktorm: Employees.name.isNotNull() SQL: t_employee.name is not null
unaryMinus(-)	-	Ktorm: -Employees.salary SQL: -t_employee.salary
unaryPlus(+)	+	Ktorm: +Employees.salary SQL: +t_employee.salary
not(!)	not	Ktorm: !Employees.name.isNull() SQL: not (t_employee.name is null)
plus(+)	+	Ktorm: Employees.salary + Employees.salary SQL: t_employee.salary + t_employee.salary
minus(-)	-	Ktorm: Employees.salary - Employees.salary SQL: t_employee.salary - t_employee.salary
times(*)	*	Ktorm: Employees.salary * 2 SQL: t_employee.salary * 2
div(/)	/	Ktorm: Employees.salary / 2 SQL: t_employee.salary / 2
rem(%)	%	Ktorm: Employees.id % 2 SQL: t_employee.id % 2

运算符

Kotlin 函数名	SQL 关键字/符号	使用示例
like	like	Ktorm: Employees.name like “%vince%” SQL: t_employee.name like ‘%vince%’
notLike	not like	Ktorm: Employees.name notLike “%vince%” SQL: t_employee.name not like ‘%vince%’
and	and	Ktorm: Employees.name.isNotNull() and (Employees.name like “%vince%”) SQL: t_employee.name is not null and t_employee.name like ‘%vince%’
or	or	Ktorm: Employees.name.isNull() or (Employees.name notLike “%vince%”) SQL: t_employee.name is null or t_employee.name not like ‘%vince%’
xor	xor	Ktorm: Employees.name.isNotNull() xor (Employees.name notLike “%vince%”) SQL: t_employee.name is not null xor t_employee.name not like ‘%vince%’
lt / less	<	Ktorm: Employees.salary lt 1000 SQL: t_employee.salary < 1000
lte / lessEq	<=	Ktorm: Employees.salary lte 1000 SQL: t_employee.salary <= 1000
gt / greater	>	Ktorm: Employees.salary gt 1000 SQL: t_employee.salary > 1000
gte / greaterEq	>=	Ktorm: Employees.salary gte 1000 SQL: t_employee.salary >= 1000
eq	=	Ktorm: Employees.id eq 1 SQL: t_employee.id = 1
neq / notEq	<>	Ktorm: Employees.id neq 1 SQL: t_employee.id <> 1

运算符

Kotlin 函数名	SQL 关键字/符号	使用示例
between	between	Ktorm: Employees.id between 1…3 SQL: t_employee.id between 1 and 3
notBetween	not between	Ktorm: Employees.id notBetween 1…3 SQL: t_employee.id not between 1 and 3
inList	in	Ktorm: Employees.departmentId.inList(1, 2, 3) SQL: t_employee.department_id in (1, 2, 3)
notInList	not in	Ktorm: Employees.departmentId.notInList(1, 2, 3) SQL: t_employee.department_id not in (1, 2, 3)
exists	exists	Ktorm: exists(db.from(Employees).select()) SQL: exists (select * from t_employee)
notExists	not exists	Ktorm: notExists(db.from(Employees).select()) SQL: not exists (select * from t_employee)



SQL DSL

- 简单查询
- 连接查询
- 增删改

简单查询

```
// 查询 ID 为 1 的部门的所有工程师的名字和工资
// select t_employee.name, t_employee.salary
// from t_employee
// where (t_employee.department_id = ?) and (t_employee.job = ?)
database
    .from(Employees)
    .select(Employees.name, Employees.salary)
    .where((Employees.departmentId eq 1) and (Employees.job eq "engineer"))
    .forEach { row ->
        println("name=${row[Employees.name]}, salary=${row[Employees.salary]}")
    }
```

```
// 查询平均工资大于 10000 的部门，返回他们的部门 id 以及平均工资
// select t.department_id as t_department_id, avg(t.salary)
// from t_employee t
// group by t.department_id
// having avg(t.salary) > ?
val t = Employees.aliased("t")
database
    .from(t)
    .select(t.departmentId, avg(t.salary))
    .groupBy(t.departmentId)
    .having { avg(t.salary) gt 10000.0 }
    .forEach { row ->
        println("id=${row[t.departmentId]}, avgSalary=${row.getDouble(2)}")
    }
```

```
// 获取每个部门的 ID 和部门内员工的平均工资，并按平均工资从高到低排序，取前 10 条
// select t.department_id as t_department_id, avg(t.salary)
// from t_employee t
// group by t.department_id
// order by avg(t.salary) desc
// limit ?, ?
val t = Employees.aliased("t")
database
    .from(t)
    .select(t.departmentId, avg(t.salary))
    .groupBy(t.departmentId)
    .orderBy(avg(t.salary).desc())
    .limit(0, 10)
    .map { row ->
        tupleOf(row[t.departmentId], row.getDouble(2))
    }
```

简单查询

```
// 支持窗函数，查询每个员工和他的工资在其所在部门的排名
// select
//   t.name as t_name,
//   t.salary as t_salary,
//   t.department_id as t_department_id,
//   rank() over (
//     partition by t.department_id order by t.salary desc
//   )
// from t_employee t
val t = Employees.aliased("t")
database
    .from(t)
    .select(
        t.name,
        t.salary,
        t.departmentId,
        rank().over {
            partitionBy(t.departmentId).orderBy(t.salary.desc())
        }
    )
```

```
// 支持 case-when 语法，查询所有员工的 ID 和名字，并根据名字判断性别
// select
//    t.id as t_id,
//    t.name as t_name,
//    case t.name when ? then ? when ? then ? else ? end
// from t_employee t
val t = Employees.aliased("t")
database
    .from(t)
    .select(
        t.id,
        t.name,
        CASE(t.name)
            .WHEN("vince").THEN("male")
            .WHEN("mary").THEN("female")
            .ELSE("unknown")
            .END()
    )
```

连接查询

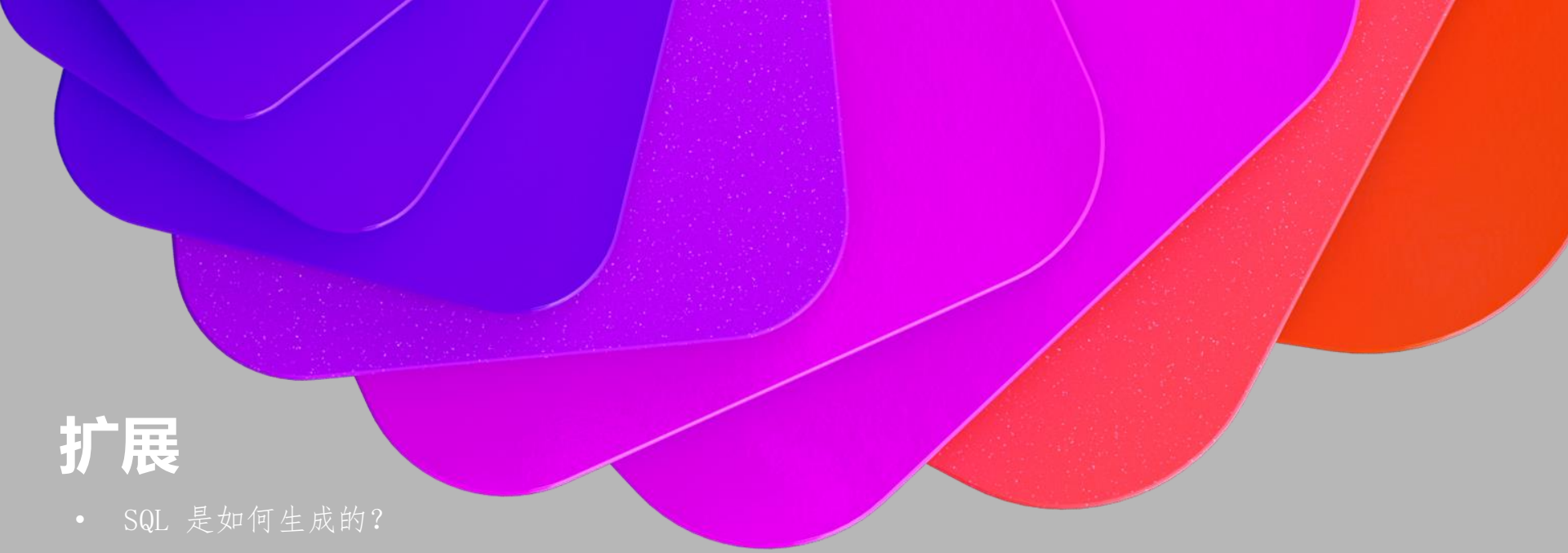
```
// 查询工资大于 10000 的员工名字和他所在的部门的名字
// select
//   t_employee.name as t_employee_name,
//   t_department.name as t_department_name
// from t_employee
// left join t_department on t_employee.department_id = t_department.id
// where t_employee.salary > ?
database
    .from(Employees)
    .leftJoin(Departments, on = Employees.departmentId eq Departments.id)
    .select(Employees.name, Departments.name)
    .where(Employees.salary gt 100L)
    .map { row ->
        tupleOf(row[Employees.name], row[Departments.name])
    }
```

```
// 查询每个员工的名字、他直属上司的名字、他所在部门的名字
// select emp.name as emp_name, mgr.name as mgr_name, dept.name as dept_name
// from t_employee emp
// left join t_employee mgr on emp.manager_id = mgr.id
// left join t_department dept on emp.department_id = dept.id
// order by emp.id
val emp = Employees.aliased("emp")
val mgr = Employees.aliased("mgr")
val dept = Departments.aliased("dept")
database
    .from(emp)
    .leftJoin(mgr, on = emp.managerId eq mgr.id)
    .leftJoin(dept, on = emp.departmentId eq dept.id)
    .select(emp.name, mgr.name, dept.name)
    .orderBy(emp.id.asc())
    .map { row ->
        tupleOf(row[emp.name], row[mgr.name], row[dept.name])
    }
```

增删改

```
// 插入
// insert into t_employee (
//   name, job, manager_id,
//   hire_date, salary, department_id
// )
// values (
//   ?, ?, ?, ?, ?, ?
// )
database.insert(Employees) {
    set(it.name, "jerry")
    set(it.job, "trainee")
    set(it.managerId, 1)
    set(it.hireDate, LocalDate.now())
    set(it.salary, 50)
    set(it.departmentId, 1)
}
```

```
// 更新
// update t_employee
//   set job = ?,
//   manager_id = ?,
//   salary = ?
// where id = ?
database.update(Employees) {
    set(it.job, "engineer")
    set(it.managerId, null)
    set(it.salary, 100)
    where {
        it.id eq 2
    }
}

// 删除
// delete from t_employee where id = ?
database.delete(Employees) { it.id eq 4 }
```

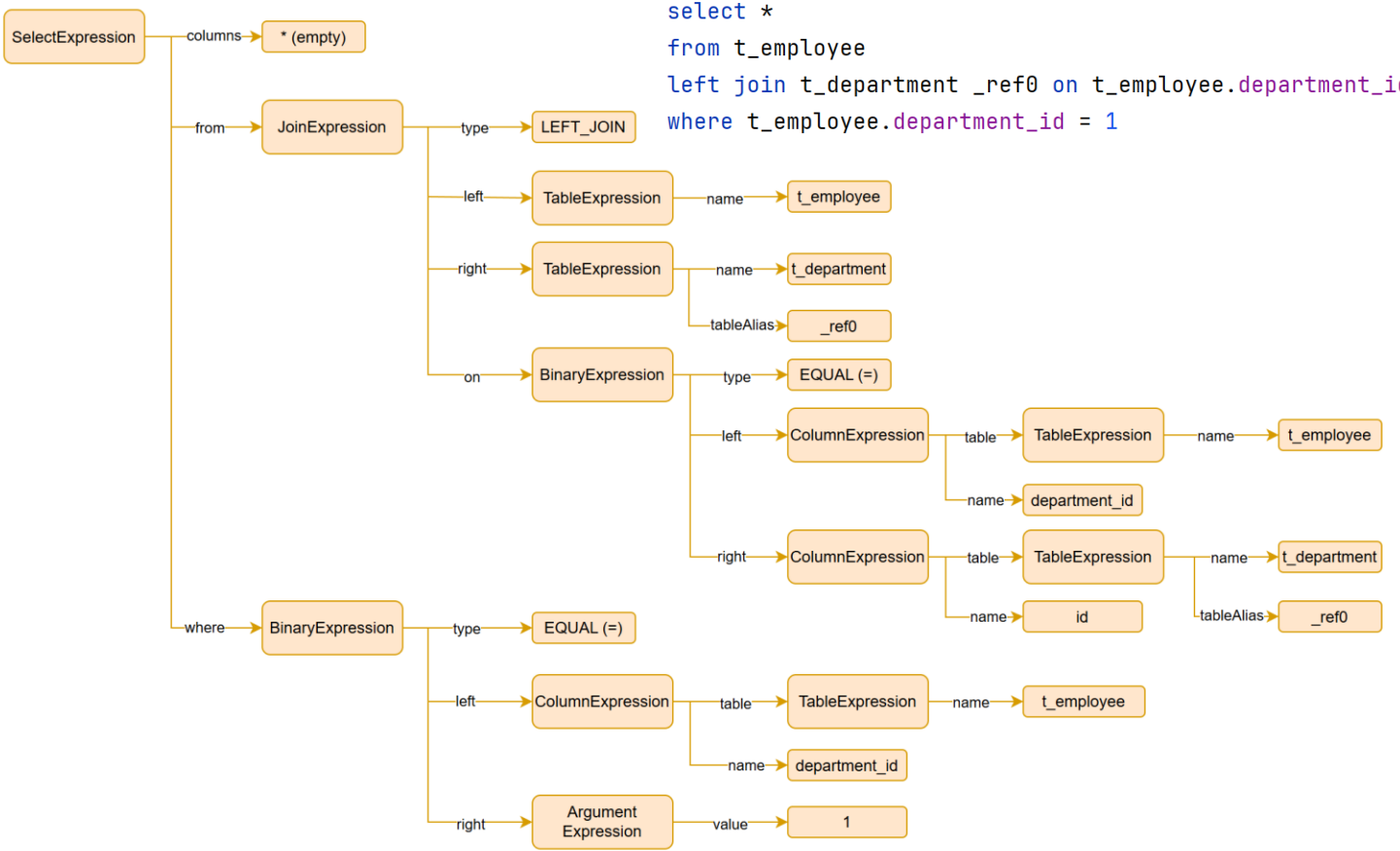


扩展

- SQL 是如何生成的？
- 自定义运算符
- 自定义 SQL 函数
- 自定义 SQL 类型
- 方言支持

SQL 是如何生成的? (AST)

```sql
select *
from t_employee
left join t_department _ref0 on t_employee.department_id = _ref0.id
where t_employee.department_id = 1
```



SQL 是如何生成的? (Visitor Pattern)

```
public abstract class SqlFormatter() : SqlExpressionVisitor {

    override fun visitSelect(expr: SelectExpression): SelectExpression {
        writeKeyword("select ")
        if (expr.isDistinct) {
            writeKeyword("distinct ")
        }
        if (expr.columns.isNotEmpty()) {
            visitExpressionList(expr.columns) { visitColumnDeclaringAtSelectClause(it) }
        } else {
            write("* ")
        }

        writeKeyword("from ")
        visitQuerySource(expr.from)

        if (expr.where != null) {
            writeKeyword("where ")
            visit(expr.where)
        }
        if (expr.groupBy.isNotEmpty()) {
            writeKeyword("group by ")
            visitExpressionList(expr.groupBy)
        }
        if (expr.having != null) {
            writeKeyword("having ")
            visit(expr.having)
        }
        if (expr.orderBy.isNotEmpty()) {
            writeKeyword("order by ")
            visitExpressionList(expr.orderBy)
        }
        if (expr.offset != null || expr.limit != null) {
            writePagination(expr)
        }
        return expr
    }
```

```
override fun visitTable(expr: TableExpression): TableExpression {
    write("${expr.name.quoted} ")
    if (!expr.tableAlias.isNullOrBlank()) {
        write("${expr.tableAlias.quoted} ")
    }

    return expr
}

override fun <T : Any> visitColumn(expr: ColumnExpression<T>): ColumnExpression<T> {
    if (expr.table != null) {
        if (!expr.table.tableAlias.isNullOrBlank()) {
            write("${expr.table.tableAlias.quoted}.")
        } else {
            write("${expr.table.name.quoted}.")
        }
    }

    write("${expr.name.quoted} ")
    return expr
}

override fun <T : Any> visitArgument(expr: ArgumentExpression<T>): ArgumentExpression<T> {
    write("? ")
    _parameters += expr
    return expr
}
```

自定义运算符

```
// 使用 PostgreSQL ilike 运算符模糊搜索名字为 vince 的员工，忽略大小写
// select *
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// where t_employee.name ilike ?
val employees = database.employees
    .filter { it.name ilike "%vince%" }
    .toList()
```

```
/**
 * 支持 PostgreSQL ilike 运算符，忽略大小写的模糊匹配
 */
infix fun ColumnDeclaring<*>.ilike(argument: String) =
    ILikeExpression(
        left = this.asExpression(),
        right = ArgumentExpression(argument, VarcharSqlType)
    )
```

```
/**
 * ilike 表达式类
 */
data class ILikeExpression(
    val left: ScalarExpression<*>,
    val right: ScalarExpression<*>,
    override val sqlType: SqlType<Boolean> = BooleanSqlType,
    override val isLeafNode: Boolean = false,
    override val extraProperties: Map<String, Any> = emptyMap()
) : ScalarExpression<Boolean>()
```

```
/**
 * 在 SqlFormater 处理 ILikeExpression 的 SQL 拼接
 */
override fun visitILike(expr: ILikeExpression): ILikeExpression {
    visit(expr.left)
    writeKeyword("ilike ")
    visit(expr.right)
    return expr
}
```

自定义 SQL 函数

```
// 使用 MySQL datediff 函数计算每个员工的入职天数
// select t_employee.id, datediff(?, t_employee.hire_date)
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// where t_employee.department_id = ?
database.employees
    .filter { it.departmentId eq 1 }
    .mapColumns { tupleOf(it.id, dateDiff(LocalDate.now(), it.hireDate)) }
    .forEach { (id, days) ->
        println("$id:$days")
    }
```

```
/**
 * 支持 MySQL datediff 函数，比较两个 date 类型的字段间隔的天数
 */
fun dateDiff(left: ColumnDeclaring<LocalDate>, right: ColumnDeclaring<LocalDate>) =
    FunctionExpression(
        functionName = "datediff",
        arguments = listOf(left.asExpression(), right.asExpression()),
        sqlType = IntSqlType
    )
```

```
/**
 * 重载支持直接传入 LocalDate，允许表字段和 SQL 入参比较
 */
fun dateDiff(left: ColumnDeclaring<LocalDate>, right: LocalDate) =
    dateDiff(left, left.wrapArgument(right))
```

```
/**
 * 重载支持直接传入 LocalDate，允许表字段和 SQL 入参比较
 */
fun dateDiff(left: LocalDate, right: ColumnDeclaring<LocalDate>) =
    dateDiff(right.wrapArgument(left), right)
```

自定义 SQL 类型

```kotlin
@Table("t_employee")
interface Employee : Entity<Employee> {
    @PrimaryKey
    var id: Int
    var name: String
    var job: String
    var managerId: Int?
    var hireDate: LocalDate
    var salary: Long
    @References
    var department: Department
    @Column(sqlType = JsonSqlType::class)
    var hobbies: List<String>
}
```

```kotlin
/**
 * 自定义数据类型，支持保存 json 数据；注意，自定义 SqlType 必须满足以下任一条件
 * 1. Kotlin 单例对象
 * 2. 普通 class，提供一个接受 TypeReference 类型的构造函数
 */
class JsonSqlType<T : Any>(typeRef: TypeReference<T>) : SqlType<T>(Types.VARCHAR, "json") {
    private val objectMapper = ObjectMapper()
    private val javaType = objectMapper.constructType(typeRef.referencedType)

    override fun doSetParameter(ps: PreparedStatement, index: Int, parameter: T) {
        ps.setString(index, objectMapper.writeValueAsString(parameter))
    }

    override fun doGetResult(rs: ResultSet, index: Int): T? {
        val json = rs.getString(index)
        if (json.isNullOrBlank()) {
            return null
        } else {
            return objectMapper.readValue(json, javaType)
        }
    }
}
```

方言支持

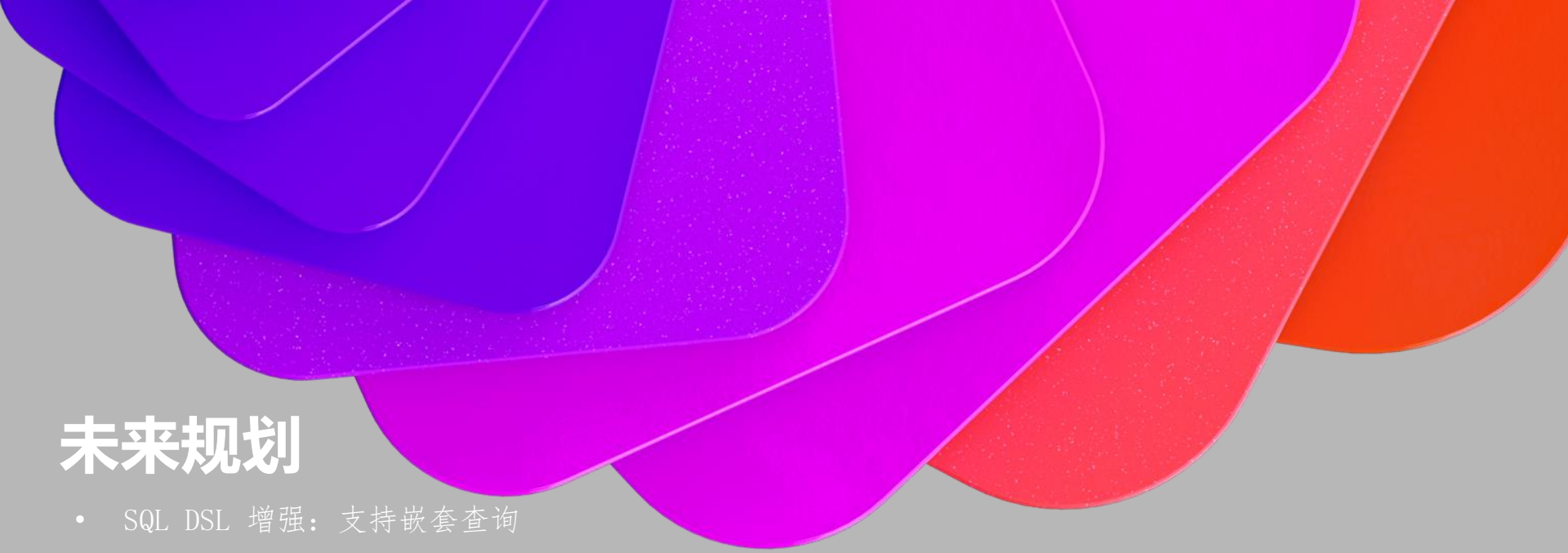
数据库类型	模块名	SqlDialect 实现类
MySQL	ktorm-support-mysql	org.ktorm.support.mysql.MySqlDialect
PostgreSQL	ktorm-support-postgresql	org.ktorm.support.postgresql.PostgreSqlDialect
SQLite	ktorm-support-sqlite	org.ktorm.support.sqlite.SQLiteDialect
SqlServer	ktorm-support-sqlserver	org.ktorm.support.sqlserver.SqlServerDialect
Oracle	ktorm-support-oracle	org.ktorm.support.oracle.OracleDialect

```xml
<!-- Maven -->
<dependency>
    <groupId>org.ktorm</groupId>
    <artifactId>ktorm-support-mysql</artifactId>
    <version>${ktorm.version}</version>
</dependency>
```

```
// Gradle
implementation("org.ktorm:ktorm-support-mysql:4.1.1")
```

```
// 插入一条数据，主键冲突时，更新 salary 字段
// insert into t_employee (id, name, job, salary, hire_date, department_id)
// values (?, ?, ?, ?, ?, ?)
// on duplicate key update salary = salary + ?
database.insertOrUpdate(Employees) {
    set(it.id, 1)
    set(it.name, "vince")
    set(it.job, "engineer")
    set(it.salary, 1000)
    set(it.hireDate, LocalDate.now())
    set(it.departmentId, 1)
    onDuplicateKey {
        set(it.salary, it.salary + 900)
    }
}
```

```
// 使用 MySQL 全文搜索
// select *
// from t_employee
// left join t_department _ref0 on t_employee.department_id = _ref0.id
// where match (t_employee.name, t_employee.job) against (?)
val engineers = database.employees
    .filter { match(it.name, it.job).against("engineer") }
    .toList()
```



未来规划

- SQL DSL 增强：支持嵌套查询
- 编译器插件：支持 >、<、== 等 Kotlin 内置运算符

Thanks!
Have a
Nice Kotlin

Kotlin 中文开发者大会

问答环节

Kotlin 中文开发者大会